ELSEVIER

Next-generation DNA sequencing techniques

Wilhelm J. Ansorge

Ecole Polytechnique Federal Lausanne, EPFL, Switzerland

Next-generation high-throughput DNA sequencing techniques are opening fascinating opportunities in the life sciences. Novel fields and applications in biology and medicine are becoming a reality, beyond the genomic sequencing which was original development goal and application. Serving as examples are: personal genomics with detailed analysis of individual genome stretches; precise analysis of RNA transcripts for gene expression, surpassing and replacing in several respects analysis by various microarray platforms, for instance in reliable and precise quantification of transcripts and as a tool for identification and analysis of DNA regions interacting with regulatory proteins in functional regulation of gene expression. The next-generation sequencing technologies offer novel and rapid ways for genome-wide characterisation and profiling of mRNAs, small RNAs, transcription factor regions, structure of chromatin and DNA methylation patterns, microbiology and metagenomics. In this article, development of commercial sequencing devices is reviewed and some European contributions to the field are mentioned. Presently commercially available very high-throughput DNA sequencing platforms, as well as techniques under development, are described and their applications in bio-medical fields discussed.

Introduction

Next-generation high-throughput DNA sequencing techniques, which are opening fascinating new opportunities in biomedicine, were selected by *Nature Methods* as the method of the year in 2007 [1]. However, the path to gaining acceptance of the novel technology was not an easy one. Until a few years ago the methods used for the sequencing were the Sanger enzymatic dideoxy technique first described in 1977 [2] and the Maxam and Gilbert chemical degradation method described in the same year [3], which was used in sequence cases which could not easily be resolved with the Sanger technique. The two laboratories where the first automated DNA sequencers were produced, simultaneously, were those of Leroy Hood at Caltech [4], commercialised by Applied Biosystems, and Wilhelm Ansorge at the European Molecular Biology Laboratory EMBL [5,6] and commercialised by Pharmacia-Amersham, later General Electric (GE) Healthcare. The Sanger method was used in the first automated fluorescent project for sequencing of a genome region, in which sequence determination of the complete gene locus for the HPRT gene was performed using the EMBL technique; in that project the important concept of paired-end sequencing was also introduced for the first time [7]. The achievement of successful and unambiguous sequencing of a real genomic DNA region, loaded with many sequence pitfalls like Alu sequences in both directions of the HPRT gene locus, demonstrated the feasibility of using an automated fluorescence-based technique for the sequencing of entire genomes, and in principle the feasibility of the technical sequencing part of the Human Genome project.

When the international community decided on determination of the whole human genome sequence, the goal triggered the development of techniques allowing higher sequencing throughput. In Japan, the work on fluorescent DNA sequencing technology by the team of H. Kambara (http://www.hitachi.com/rd/fellow_kambara.html) in the Hitachi laboratories resulted in the development after 1996 of a high-throughput capillary array DNA sequencer. Two

E-mail address: wilhelm.ansorge@epfl.ch.

1871-6784/$ - see front matter © 2009 Published by Elsevier B.V. doi:10.1016/j.nbt.2008.12.009

companies, ABI (commercialising the Kambara system) and Amersham (taking over and developing further the system set up in the US by the Molecular Dynamics company), commercialised automated sequencing using parallel analysis in systems of up to 384 capillaries at that time. Together with partial miniaturisation of the robotic sample preparation, large efforts in automation of laboratory processes and advances in new enzymes and biochemicals, the Sanger technique made possible the determination of the sequence of the human genome by two consortia working in parallel. It was the unique method used for DNA sequencing, with innumerable applications in biology and medicine.

As the users and developers of the DNA sequencing techniques realised, the great limitations of the Sanger sequencing protocols for even larger sequence output were the need for gels or polymers used as sieving separation media for the fluorescently labelled DNA fragments, the relatively low number of samples which could be analysed in parallel and the difficulty of total automation of the sample preparation methods. These limitations initiated efforts to develop techniques without gels, which would allow sequence determination on very large numbers (i.e. millions) of samples in parallel. One of the first developments of such a technique was at the EMBL (at that time one of the two world leaders in DNA sequencing technology) from 1988 to 1990. A patent application by EMBL [8] described a large-scale DNA sequencing technique without gels, extending primers in 'sequencing-by-synthesis, addition and detection of the incorporated base', proposing and describing the use of the so-called 'reversible terminators' for speed and efficiency [8]. The first step of the technique consisted in detecting the next added fluorescently labelled base (reversible terminator) in the growing DNA chain by means of a sensitive CCD camera. This was performed on a large number of DNA samples in parallel, attached either to a planar support or to beads, on DNA chips, minimising reaction volumes in a miniaturised microsystem. In the next step the terminator was converted to a standard nucleotide and the dye removed from it. This cycle and the process were repeated to determine the next base in the sequence. The principle described in the patent application is in part very similar to that used today in the so-called next-generation devices, with many additional original developments commercialised by Illumina-Solexa, Helicos and other companies.

Since 2000, focused developments have continued in several groups. Various institutions, particularly European laboratories, considered the capillary systems as the high point and in a less visionary decision ceased developments of even the most promising novel sequencing techniques, turning their attention exclusively to arrays. By contrast, in the US, funding for development and testing of novel, non-gel-based high-throughput sequencing technologies were provided by the large granting agencies and private companies. Efforts to bring the platforms to maturity were under way. The resulting devices and platforms available on the market in mid-2008, as well as some interesting parallel developments, are described in more detail below. The EU has recently initiated significant support for the development of novel high-throughput DNA sequencing technologies, among others the READNA initiative (www.cng.fr/READNA).

Next-generation DNA sequencing platforms

Novel DNA sequencing techniques provide high speed and throughput, such that genome sequencing projects that took several years with the Sanger technique can now be completed in a matter of weeks. The advantage of these platforms is the determination of the sequence data from amplified single DNA fragments, avoiding the need for cloning of DNA fragments. A limiting factor of the new technology remains the overall high cost for generating the sequence with very high-throughput, even though compared with Sanger sequencing the cost per base is lower by several orders of magnitude. Reduction of sequencing errors is another factor; in this respect the Sanger sequencing technique remains competitive in the immediate future. Other limitations in some applications are short read lengths, non-uniform confidence in base calling in sequence reads, particularly deteriorating 3′-sequence quality in technologies with short read lengths and generally lower reading accuracy in homopolar stretches of identical bases. The huge amount of data generated by these systems (over a gigabase per run) in the form of short reads presents another challenge to developers of software and more efficient computer algorithms.

The 454 GenomeSequencer FLX instrument (Roche Applied Science)

The principle of pyrophosphate detection, the basis of this device, was described in 1985 [9], and a system using this principle in a new method for DNA sequencing was reported in 1988 [10]. The technique was further developed into a routinely functioning method by the teams of M. Ronaghi, M. Uhlen, and P. Nyren in Stockholm [11], leading to a technique commercialised for the analysis of 96 samples in parallel in a microtiter plate.

The GS instrument was introduced in 2005, developed by 454 Life Sciences, as the first next-generation system on the market. In this system (Fig. 1), DNA fragments are ligated with specific adapters that cause the binding of one fragment to a bead. Emulsion PCR is carried out for fragment amplification, with water droplets containing one bead and PCR reagents immersed in oil. The amplification is necessary to obtain sufficient light signal intensity for reliable detection in the sequencing-by-synthesis reaction steps. When PCR amplification cycles are completed and after denaturation, each bead with its one amplified fragment is placed at the top end of an etched fibre in an optical fibre chip, created from glass fibre bundles. The individual glass fibres are excellent light guides, with the other end facing a sensitive CCD camera, enabling positional detection of emitted light. Each bead thus sits on an addressable position in the light guide chip, containing several hundred thousand fibres with attached beads. In the next step polymerase enzyme and primer are added to the beads, and one unlabelled nucleotide only is supplied to the reaction mixture to all beads on the chip, so that synthesis of the complementary strand can start. Incorporation of a following base by the polymerase enzyme in the growing chain releases a pyrophosphate group, which can be detected as emitted light. Knowing the identity of the nucleotide supplied in each step, the presence of a light signal indicates the next base incorporated into the sequence of the growing DNA strand.

The method has recently increased the achieved reading length to the 400–500 base range, with paired-end reads, and as such is being applied to genome (bacterial, animal, human) sequencing. One spectacular application of the system was the identification of the culprit in the recent honey-bee disease epidemics (see company web pages below). A relatively high cost of operation

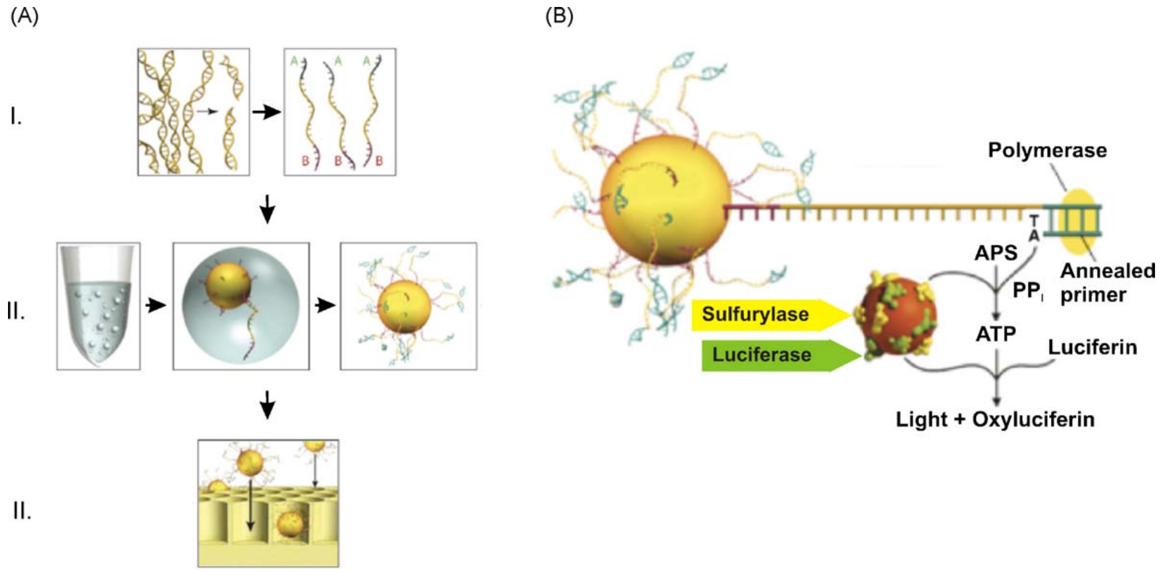


FIGURE 1

(A) Outline of the GS 454 DNA sequencer workflow. Library construction (I) ligates 454-specific adapters to DNA fragments (indicated as A and B) and couples amplification beads with DNA in an emulsion PCR to amplify fragments before sequencing (II). The beads are loaded into the picotiter plate (III). **(B)** Schematic illustration of the pyrosequencing reaction which occurs on nucleotide incorporation to report sequencing-by-synthesis. (Adapted from http://www.454.com.)

and generally lower reading accuracy in homopolar stretches of identical bases are mentioned presently as the few drawbacks of the method. The next upgrade 454 FLX Titanium will quintuple the data output from 100 Mb to about 500 Mb, and the new picotiter plate in the device uses smaller beads about 1 μm diameter. The device, schema of operation, its further developments and list of publications with applications can be found at http://www.454.com/index.asp and in [1].

The Illumina (Solexa) Genome Analyzer

The Solexa sequencing platform was commercialised in 2006, with Illumina acquiring Solexa in early 2007. The principle (Fig. 2) is on the basis of sequencing-by-synthesis chemistry, with novel reversible terminator nucleotides for the four bases each labelled with a different fluorescent dye, and a special DNA polymerase enzyme able to incorporate them. DNA fragments are ligated at both ends to adapters and, after denaturation, immobilised at one end on a solid support. The surface of the support is coated densely with the adapters and the complementary adapters. Each single-stranded fragment, immobilised at one end on the surface, creates a 'bridge' structure by hybridising with its free end to the complementary adapter on the surface of the support. In the mixture containing the PCR amplification reagents, the adapters on the surface act as primers for the following PCR amplification. Again, amplification is needed to obtain sufficient light signal intensity for reliable detection of the added bases. After several PCR cycles, random clusters of about 1000 copies of single-stranded DNA fragments (termed DNA 'polonies', resembling cell colonies after polymerase amplification) are created on the surface. The reaction mixture for the sequencing reactions and DNA synthesis is supplied onto the surface and contains primers, four reversible terminator nucleotides each labelled with a different fluorescent dye and the DNA polymerase. After incorporation into the DNA strand, the terminator nucleotide, as well as its position on the support surface, is detected and identified via its fluorescent dye by the CCD camera. The terminator group at the 3′-end of the base and the fluorescent dye are then removed from the base and the synthesis cycle is repeated. The sequence read length achieved in the repetitive reactions is about 35 nucleotides. The sequence of at least 40 million polonies can be simultaneously determined in parallel, resulting in a very high sequence throughput, on the order of Gigabases per support.

In 2008 Illumina introduced an upgrade, the Genome Analyzer II that triples output compared to the previous Genome Analyzer instrument. A paired-end module for the sequencer was introduced, and with new optics and camera components that allow the system to image DNA clusters more efficiently over larger areas, the new instrument triples the output per paired-end run from 1 to 3 Gb. The system generates at least 1.5 Gb of single-read data per run, at least 3 Gb of data in a paired-end run, recording data from more than 50 million reads per flow cell. The run time for a 36-cycle run was decreased to two days for a single-read run, and four days for a paired-end run. Information on the Genome Analyzer system can be found at http://www.solexa.com/ and in [1].

The Applied Biosystems ABI SOLiD system

The ABI SOLiD sequencing system, a platform using chemistry based upon ligation, was introduced in Autumn 2007. The generation of a DNA fragment library and the sequencing process by subsequent ligation steps are shown schematically in Figs 3,4. In this technique, DNA fragments are ligated to adapters then bound to beads. A water droplet in oil emulsion contains the amplification reagents and only one fragment bound per bead; DNA fragments on the beads are amplified by the emulsion PCR. After DNA denaturation, the beads are deposited onto a glass support surface. In a first step, a primer is hybridised to the adapter. Next, a mixture of oligonucleotide octamers is also hybridised to the DNA fragments and ligation mixture added. In these octamers, the doublet

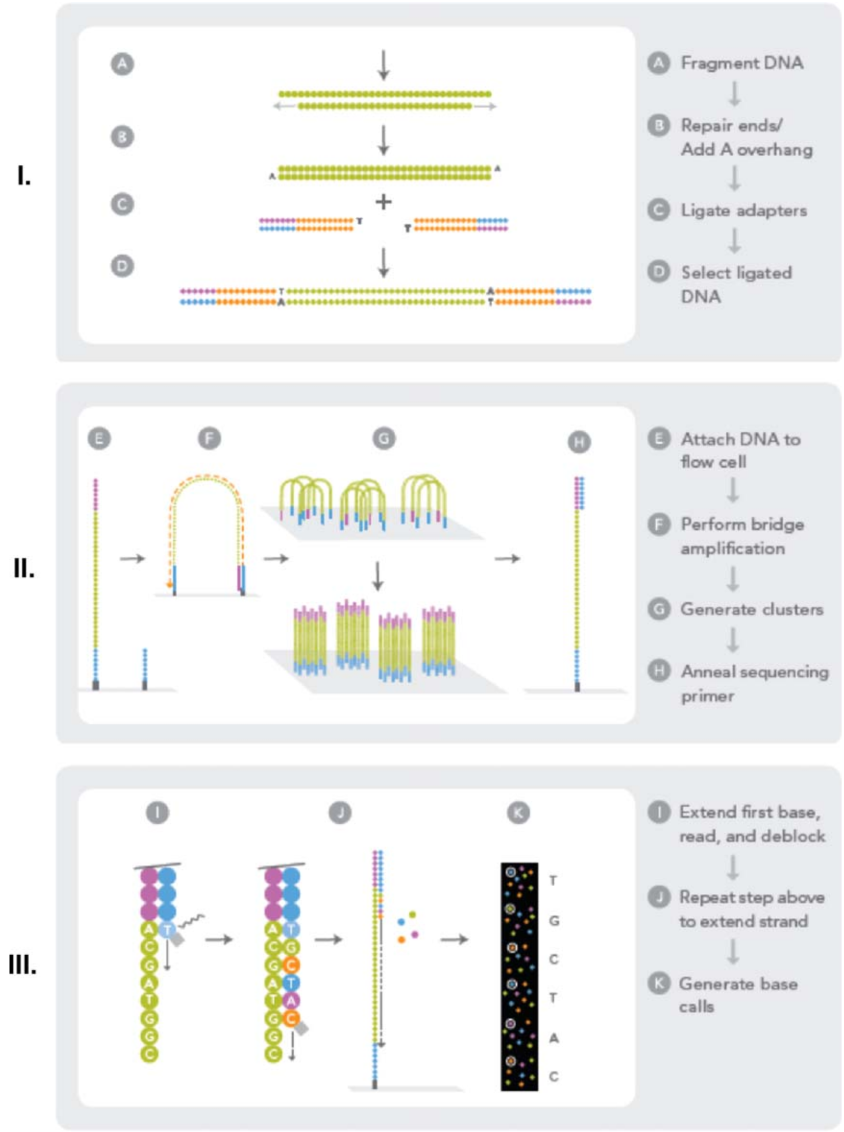

I.
A Fragment DNA
B Repair ends/ Add A overhang
C Ligate adapters
D Select ligated DNA
II.
E Attach DNA to flow cell
F Perform bridge amplification
G Generate clusters
H Anneal sequencing primer
III.
I Extend first base, read, and deblock
J Repeat step above to extend strand
K Generate base calls
T
G
C
T
A
C

FIGURE 2

Outline of the Illumina Genome Analyzer workflow. Similar fragmentation and adapter ligation steps take place (I), before applying the library onto the solid surface of a flow cell. Attached DNA fragments form 'bridge' molecules which are subsequently amplified via an isothermal amplification process, leading to a cluster of identical fragments that are subsequently denatured for sequencing primer annealing (II). Amplified DNA fragments are subjected to sequencing-by-synthesis using 3′ blocked labelled nucleotides (III). (Adapted from the Genome Analyzer brochure, http://www.solexa.com.)

of fourth and fifth bases is characterised by one of four fluorescent labels at the end of the octamer. After the detection of the fluorescence from the label, bases 4 and 5 in the sequence are thus determined. The ligated octamer oligonucleotides are cleaved off after the fifth base, removing the fluorescent label, then hybridisation and ligation cycles are repeated, this time determining bases 9 and 10 in the sequence; in the subsequent cycle bases 14 and 15 are determined, and so on. The sequencing process may be continued in the same way with another primer, shorter by one base than the previous one, allowing one to determine, in the successive cycles, bases 3 and 4, 8 and 9, 13 and 14. The achieved sequence reading length is at present about 35 bases. Because each base is determined with a different fluorescent label, error rate is reduced. Sequences can be determined in parallel for more than 50 million bead clusters, resulting in a very high throughput of the order of Gigabases per run.

Applied Biosystems produced an updated version in 2008, the SOLiD 2.0 platform, which may increase the output of the instrument from 3 to 10 Gb per run. This change will reduce the overall run time of a fragment library on the new system to 4.5 days from 8.5 days on the existing machine. For further information see www3.appliedbiosystems.com/index.htm, and in [1]

The Helicos single-molecule sequencing device, HeliScope

The systems discussed above require the emulsion PCR amplification step of DNA fragments, to make the light signal strong enough for reliable base detection by the CCD cameras. PCR amplification has revolutionised DNA analysis, but in some instances it may introduce base sequence errors into the copied DNA strands, or favour certain sequences over others, thus changing the relative frequency and abundance of various DNA fragments that existed before amplification. Ultimate miniaturisation

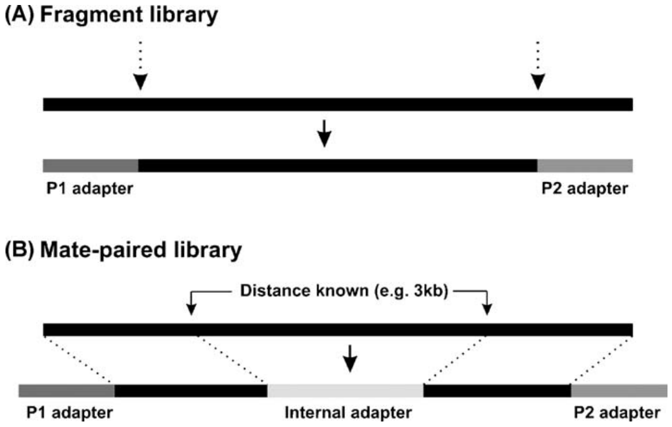


FIGURE 3

Library preparation for DNA sequencing using the SOLiD DNA sequencing platform. **(A)** *Fragment library*: After whole genome DNA is randomly fragmented (indicated by the dashed arrows), two different 25 bp DNA adapters (P1 and P2) are ligated at the 5′- and 3′-ends of the DNA fragments generated. **(B)** *Mate-paired library*: In this case, DNA fragments that are separated from another DNA fragment of known length (e.g. 3 kb for this example) are ligated such that they encompass an internal adapter. Subsequently, two different DNA adapters are ligated at the 5′- and 3′-ends, similarly to (A). (Adapted and modified from http://www.appliedbiosystems.com.)

into the nanoscale, and the minimal use of biochemicals, would be achieved if the sequence could be determined directly from a single DNA molecule, without the need for PCR amplification and its potential for distortion of abundance levels. This requires a very sensitive light detection system and a physical arrangement capable of detecting and identifying light from a single dye molecule. Techniques for the detection and analysis of single molecules have been under intensive development over past decades, and several very sensitive systems for single photon detection have been produced and tested. One of the first techniques for sequencing from a single DNA molecule was described by the team of S. Quake [12], and licensed by Helicos Biosciences.

Helicos introduced the first commercial single-molecule DNA sequencing system in 2007. The nucleic acid fragments are hybridised to primers covalently anchored in random positions on a glass cover slip in a flow cell. The primer, polymerase enzyme and labelled nucleotides are added to the glass support. The next base incorporated into the synthesised strand is determined by analysis of the emitted light signal, in the sequencing-by-synthesis technique (similar to Fig. 2, but on only one DNA fragment, without amplification). This system also analyses many millions of single DNA fragments simultaneously, resulting in sequence throughput in the Gigabase range. Although still in the first years of operation, the system has been tested and validated in several applications with promising results, for example in the pre-natal trisomy-21 (Down Syndrome) test, using only the maternal blood sample, potentially replacing the standard test which is associated with some risk to the foetus [13].

When the Helicos system was used to sequence the genome of M13 phage, read lengths averaged about 23 bases. There were still some limitations in the single-molecule technology, on the basis of the first generation of the chemistry. In the homopolar regions, multiple fluorophore incorporations could decrease emissions, sometimes below the level of detection; when errors did occur, most were deletions. Helicos announced that it has recently developed a new generation of 'one-base-at-a-time' nucleotides which allow more accurate homopolymer sequencing, and lower overall error rates. For further information, see http://www.helicosbio.com/

Novel DNA sequencing techniques in development

Developments of novel DNA sequencing techniques are taking place in many groups worldwide. In the laboratory of Church [14] a technique similar to the sequencing-by-synthesis method above has been developed, with multiplex polony technology. Several hundred sequencing templates are deposited onto thin agarose layers, and sequences are determined in parallel. This presents increase of several orders of magnitudes in the number of samples which can be analysed simultaneously. A further advantage is the large reduction of reaction volumes, the smaller amounts of reagents needed and the resulting lower cost. The group continues development of their platform and offers this technique to academic laboratories using off-the-shelf optics, hardware and reagents.

Another promising approach, attempting to use real-time single-molecule DNA sequence determination, is being developed by VisiGen Biotechnologies http://visigenbio.com/. They have produced a specially engineered DNA polymerase (acting as a 'real-time sensor' for modified nucleotides) with a donor fluorescent dye incorporated close to the active site involved in selection of the nucleotides during synthesis. All four nucleotides to be integrated have been modified, each with a different acceptor dye. During the synthesis, when the correct nucleotide is found, selected and enters the active site of the enzyme, the donor dye label in the polymerase comes into close proximity with the acceptor dye on the nucleotides and energy is transferred from donor to acceptor dye giving rise to a fluorescent resonant energy transfer (FRET) light signal. The frequency of this signal varies depending on the label incorporated in the nucleotides, so that by recording frequencies of emitted FRET signals it will be possible to determine base sequences, at the speed at which the polymerase can integrate the nucleotides during the synthesis process (usually a few hundred per second). The acceptor fluorophore is removed during nucleotide incorporation, which ensures that there are no DNA modifications that might slow down the polymerase during synthesis. VisiGen plans to offer a service on the basis of its real-time single-molecule nanosequencing technology by end 2009, followed by the launch of equipment and reagents 18 months later. The technology could eventually enable researchers to sequence an entire human genome in less than a day for under $1000. The company is currently working on its first version of the instrument, which can generate around 4 Gb of data per day. The single-molecule approach requires no cloning and no amplification, which eliminates a large part of the cost relative to current technologies. In addition, read lengths for the instrument are expected to be around 1 kb, longer than any current platform.

Another US company, Pacific Biosciences (http://www.pacific-biosciences.com/index.php), announced recently that it is working on a next-generation DNA sequencing instrument that will eventually be able to produce 100 Gb of sequence data per hour, or a diploid human genome at onefold coverage in about 4 min. They plan to sell their first systems during 2010. The company's single-molecule real-time (SMRT) technology is based on zero mode waveguides (ZMWs) which were originally developed at Cornell

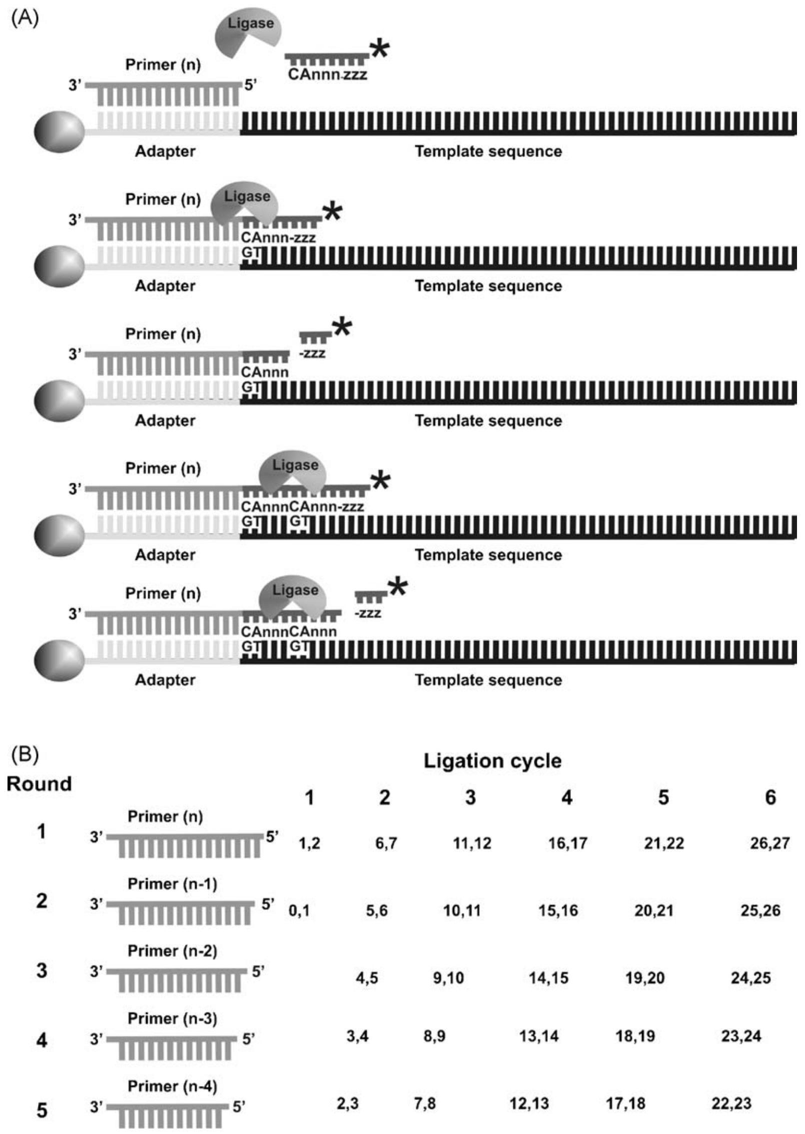

(A)
Ligase
Primer (n)
CAnnn-zzz
3'
5'
Adapter
Template sequence
CAnnn-zzz
GT
-zzz
CAnnn
CAnnnCAnnn-zzz
CAnnnCAnnn
(B)
Round
Ligation cycle
1 2 3 4 5 6
1 Primer (n) 1,2 6,7 11,12 16,17 21,22 26,27
2 Primer (n-1) 0,1 5,6 10,11 15,16 20,21 25,26
3 Primer (n-2) 4,5 9,10 14,15 19,20 24,25
4 Primer (n-3) 3,4 8,9 13,14 18,19 23,24
5 Primer (n-4) 2,3 7,8 12,13 17,18 22,23

FIGURE 4

Sequencing-by-ligation, using the SOLiD DNA sequencing platform. **(A)** Primers hybridise to the P1 adapter within the library template. A set of four fluorescence-labelled di-base probes competes for ligation to the sequencing primer. These probes have partly degenerated DNA sequence (indicated by *n* and *z*) and for simplicity only one probe is shown (labelling is denoted by asterisk). Specificity of the di-base probe is achieved by interrogating the first and second base in each ligation reaction (CA in this case for the complementary strand). Following ligation, the fluorescent label is enzymatically removed together with the three last bases of the octamer. **(B)** Sequence determination by the SOLiD DNA sequencing platform is performed in multiple ligation cycles, using different primers, each one shorter from the previous one by a single base. The number of ligation cycles (six for this example) determines the eventual read length, whilst for each sequence tag, six rounds of primer reset occur [from primer (*n*) to primer ($n-4$)]. The dinucleotide positions on the template sequence that are interrogated each time, are depicted underneath each ligation cycle and are separated by 5-bp from the dinucleotide position interrogated in the subsequent ligation cycle. (Adapted and modified from http://www.appliedbiosystems.com.)

University Nanobiotechnology Center. ZMWs are nanometre-scale aperture chambers in a 100 nm metal film deposited on a clear substrate. Owing to the behaviour of light aimed at such a small chamber, the observation volume is only 20 zeptolitres, enabling researchers to measure the fluorescence of nucleotides incorporated by a single DNA polymerase enzyme into a growing DNA strand in real time. The developers have so far observed read lengths of about 1500 bases and a rate of 10 bases/s, and have been able to analyse up to 3000 ZMWs in parallel.

Another single-molecule sequencing technique may develop from studies on translocation of DNA through various artificial nanopores. The work in this field was pioneered at Harvard by D. Branton, G. Church and J. Golovchenko, at UC Santa Cruz by D. Deamer and M. Akeson and at the NI Standards and Technology by J. Kasianowicz. The approach is based on the modulation of the ionic current through the pore as a DNA molecule traverses it, revealing characteristics and parameters (diameter, length and conformation) of the molecule. Recent study in this direction is the work of Trepanier and colleagues [15], which contains references to previous studies on this subject. In their work [15] they analyze several limitations of the method. One limitation to single-base resolution in nanopore-based DNA sequencing approaches is the insufficient control of the translocation speed of the molecule, for example during electrophoresis of the DNA molecules through the nanopore. This was overcome by the integration of an optical trapping system, which enables control and lowering of the translocation speed by several hundred-fold. For the demonstration, a known DNA fragment was used, attached via streptavidin–biotin to a polystyrene bead with a diameter of 10 μm. The bead was placed into the optical trap, and translocation speed was reduced about 200-fold, giving more time for analysis of the DNA molecule passing through the nanopore. It was also possible to control the motion of the molecule and return it back to its starting point before the translocation, thus making possible repeated measurements and analysis.

Another of these approaches, studied by several teams collaborating as part of an EU consortium on nano-DNA-sequencing coordinated by R. Zikic from Belgrade, L. Forro and A. Radenovic (EPFL Lausanne), is the development of a nano-electronic device for high-throughput single-molecule DNA sequencing, with the potential to determine long genomic sequences. This is on the basis of the electrical characterisation of individual nucleotides, whilst DNA passes through a nanopore (similar to [15]), with integrated nanotube side-electrodes developed at EPFL, Lausanne. A lithographically fabricated nanogap is produced with single-nanometre precision and allows characterisation of the tunnelling conductance across DNA bases and the electrical response of DNA molecule translocation between two carbon nanotube electrodes. The translocation rate of DNA through the nanopore will be varied by an optical tweezers system (in addition to standard techniques of applied voltage, viscosity change and DNA charge at various pH), aiming to achieve single-base resolution. Further improvements and modifications of the technique, increasing the number of parameters measured during the translocation of the DNA enabling single-base resolution, could lead to a rapid nanopore-based DNA sequencing technique.

Sequenom (http://www.sequenom.com) has licensed technology from Harvard University, to develop a nanopore-based sequencing platform that will be faster and cheaper than currently available technologies. In the near term they plan to use it for large-scale genotyping applications, RNA and epigenetic analyses. In the long term it has the potential to provide a commercially viable, rapid, sub-1000 dollar human genome sequencing solution. The technology has also been licensed by Oxford Nanopore Technologies, UK (http://www.nanoporetech.com/).

BioNanomatrix and Complete Genomics (http://bionanomatrix.com/) announced in 2007 the formation of a joint venture to develop technology to sequence a human genome in eight hours for less than $100. The proposed platform will use Complete Genomics's sequencing chemistry and BioNanomatrix's nanofluidic technology. They plan to adapt DNA sequencing chemistry with linearised nanoscale DNA imaging to create a system that can read DNA sequences greater than 100,000 bases. With their design and price they target the possible sequencing of many genomes. Complete Genomics company (http://www.completegenomics.com) presented recently a new method, using rolling circle PCR amplification resulting in DNA nanoballs, and a modified ligation technique, for fast and non expensive sequencing of human genomes.

A very different approach to single-molecule DNA sequencing, using RNA polymerase (RNAP), has been presented recently [16]. In the planned method, RNAP is attached to one polystyrene bead, whilst the distal end of a DNA fragment is attached to another bead. Each bead is placed in an optical trap and the pair of optical traps levitates the beads. The RNAP interacts with the DNA fragment and the transcriptional motion of RNAP along the template changes the length of the DNA between the two beads. This leads to displacement of the two beads that can be registered with precision in the Angstrom range, resulting in single-base resolution on a single DNA molecule. By aligning four displacement records, each with a lower concentration of one of the four nucleotides, in a role analogous to the primers used in Sanger sequencing, and using for calibration the known sequences flanking the unknown fragment to be sequenced, it is possible to deduce the sequence information. Thirty out of 32 bases were correctly identified in about 2 min. The technique demonstrates that the movement of a nucleic acid enzyme, and the very sensitive optical trap method, may allow extraction of sequence information directly from a single DNA molecule.

Applications of high-throughput DNA sequencing

Novel fields and applications in biology and medicine are becoming a reality, beyond genomic sequencing as the original development goal and application. Examples include personal genomics with detailed analysis of individual genomic stretches; precise analysis of RNA transcripts for gene expression, surpassing and replacing in several aspects analysis carried out by various microarray platforms, for example in reliable and precise transcript quantification; and as a tool for identification and analysis of DNA regions that interact with regulatory proteins in functional regulation of gene expression. Next-generation sequencing technologies offer novel, rapid ways for genome-wide characterisation and profiling of mRNAs, small RNAs, transcription factor regions, chromatin structure and DNA methylation patterns, in microbiology and metagenomics.

Personal genomics, project human diversity in 1000 genomes

The cost of genome sequencing, an important factor in future studies, is becoming low enough to make personal genomics a close reality. Reduction of cost by two orders of magnitude is needed to be able to realise the potential of personal genomics, for which the goal of $1000 for a human genome sequence has been set. The impressive results obtained so far in various projects with the new technology are very convincing and will lead to lower cost. The analysis of the first two available human genomes [17,18] has demonstrated, how difficult it still is to draw medically or biologically relevant conclusions from individual sequences. More genomes need to be sequenced, to learn how genotype correlates with phenotype. A plan for a project to sequence 1000 human genomes has been prepared, which will allow creation of a reference standard for the analysis of human genomic variations that is expected to contribute to studies of disease (http://www.1000genomes.org/). Illumina, Roche 454 Life Sciences and Applied Biosystems will take part in the project and generate the equivalent of 25 human genomes each per year over a period of three years. This significant sequence contribution will enable the team to analyse the human genome with deeper sequencing and shorten its completion time. The 1000 Genomes Project will identify variants present at a frequency of 1% over most of the genome, and as low as 0.5% within genes.

The immediate applications and relevance of next-generation sequencing techniques in the medical field have been demonstrated already, by the ability to detect cancer alleles with deep sequencing of genomic DNA in cancerous tissues (carefully isolated by laser microdissection and capture techniques), which would have presented a very tedious task for the Sanger technique.

RNA sequencing, analysis of gene expression

The high throughput of next-generation sequencing technology, rapidly producing huge numbers of short sequencing reads, made possible the analysis of a complex sample containing a mixture of a large number of nucleic acids, by sequencing simultaneously the entire sample content. This is now possible without the tedious and time-consuming bacterial cloning, avoiding associated disadvantages. It may also be applied to the characterisation of mRNAs, methylated DNA, DNA or RNA regions bound by certain proteins and other DNA or RNA regions involved in gene expression and regulation. The original SAGE technique [19] demonstrated novelty and powerful analysis, but was limited in applications because of the need for difficult ligation of a huge number of short DNA transcripts, subsequent cloning and Sanger sequencing. Using next-generation technology, the concept of the SAGE method now allows the analysis of RNA transcripts in a biological sample by obtaining short sequence tags, 20–35 bases long, directly from each transcript in the sample. With this technique, transcripts are characterised through their sequence [20], in contrast to the probe hybridisation employed in DNA chip techniques, with their inherent difficulties of cross-hybridisation and quantitation. Owing to the huge number of samples analysed simultaneously, sequence-based techniques can detect low abundance RNAs, small RNAs, or the presence of rare cells contained in the sample. Another advantage of this approach is that it does not require prior knowledge of the genome sequence. The technique has been applied recently to transcriptome profiling in stem cells [21] and to RNA-Seq study into alternative splicing in human cells [22].

Chromatin immunoprecipitation, ChIP-Seq technique

Next-generation sequencing technology allowed replacement of microarrays in the mapping step with high-throughput sequencing of DNA binding sites, and their direct mapping to a reference genome in the database [23]. The sequence of the binding site is mapped with high resolution to regions shorter than 40 bases, a resolution not achievable by microarray mapping. Moreover, the ChiP-Seq technique is not biased and allows the identification of unknown protein binding sites, which is not the case with the ChiP-on-chip approach, where the sequence of the DNA fragments on the microarray is pre-determined, e.g. in promoter arrays, exon arrays, etc.

Prospects for future DNA sequencing technology and applications

The availability of ultra-deep sequencing of genomic DNA will transform the biological and medical fields in the near future, in analysis of the causes of disease, development of new drugs and diagnostics. It may become a promising tool in the analysis of mental and developmental disorders such as schizophrenia and autism [24–26]. It is anticipated that DNA sequencing of whole genomes for clinical purposes using these new technologies will probably occur in the next couple of decades. Some of the most recent applications can be found in the proceedings of the AGBT conference (Advances in Genome Biology and Technology), February 2009.

The novel sequencing technologies will be also useful in microbial genomics, for example in the metagenomics measuring the genetic diversity encoded by microbial life in organisms inhabiting a common environment [27]. Many microbial sequencing projects have been already completed or are being prepared and several comparative genome analyses are under way to link genotype and phenotype at the genomic level. The proposed Human Microbiome Project (also called The Second Human Genome Project), analysing the collection of microbes in and on the human body, will contribute to understanding human health and disease [28]. Changes in microbial communities in the body have been generally linked to immune system function, obesity and cancer. In future, each individual's microbiome could eventually become a medical biometric.

An important application is planned by the US DOE Joint Genome Institute, JGI (http://www.jgi.doe.gov/), which will focus its sequencing efforts on new plant and microbial targets that may be of use in the development of alternative energies. The JGI plans to sequence the genome of the marine red alga, which may play an important environmental role in removing carbon dioxide from the atmosphere.

Genomics, proteomics and medical research all benefit from recent advances and novel techniques for high-throughput analysis (e.g. DNA and protein microarrays, quantitative PCR, mass spectrometry, novel DNA sequencing techniques and others). Devices with short DNA sequence reads (25–50 bases) have already found many applications, but for genomic sequencing, and for analysis of the ever more important structural genetic variations in genomes, such as copy number variations, chromosomal translocations, inversions, large deletions, insertions and duplications, it would be a great advantage if sequence read length on the original single DNA molecule could be increased to several 1000 bases and more per second. Ideally, the goal would be the sequence determination of a whole chromosome from a single original DNA molecule. Hopes for future in this direction may provide novel developments in several physical techniques (e.g. various advanced AFM methods, electron microscopy, soft X-rays, various spectroscopic techniques, nanopores and nano-edges), with many improvements needed and under intense development.

Acknowledgement

I am grateful to Dr George Patrinos for discussions and help with preparation of the figures.

References

1 Schuster, S.C. *et al.* (2008) Method of the year, next-generation DNA sequencing. Functional genomics and medical applications. *Nat. Methods* 5, 11–21

2 Sanger, F. *et al.* (1977) DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. U. S. A.* 74, 5463–5467

3 Maxam, A.M. and Gilbert, W. (1977) A new method for sequencing DNA. *Proc. Natl. Acad. Sci. U. S. A.* 74, 560–564

4 Smith, L.M. *et al.* (1986) Fluorescence detection in automated DNA sequence analysis. *Nature* 321, 674–679

5 Ansorge, W. *et al.* (1986) A non-radioactive automated method for DNA sequence determination. *J. Biochem. Biophys. Methods* 13, 315–323

6 Ansorge, W. *et al.* (1987) Automated DNA sequencing: ultrasensitive detection of fluorescent bands during electrophoresis. *Nucleic Acids Res.* 15, 4593–4602

7 Edwards, A. *et al.* (1990) Automated DNA sequencing of the human HPRT locus. *Genomics* 6, 593–608

8 Ansorge, W., EMBL Heidelberg (1991), Process for sequencing nucleic acids without gel sieving media on solid support and DNA chips (Verfahren zur Sequenzierung von Nukleinsauren ohne Gele). German Patent Application DE 41 41 178 A1 and Corresponding Worldwide Patent Applications.

9 Nyren, P. and Lundin, A. (1985) Enzymatic method for continuous monitoring of inorganic pyrophosphate synthesis. *Anal. Biochem.* 151, 504–509

10 Hyman, E.D. (1988) A new method of sequencing DNA. *Anal. Biochem.* 174, 423–436

11 Ronaghi, M. *et al.* (1996) Real-time DNA sequencing using detection of pyrophosphate release. *Anal. Biochem.* 242, 84–89

12 Braslavsky, I. *et al.* (2003) Sequence information can be obtained from single DNA molecule. *Proc. Natl. Acad. Sci. U. S. A.* 100, 3960–3964

13 Quake, S.R. *et al.* (2008) Pre-natal trisomia-21-test from maternal blood sample. *Proc. Natl. Acad. Sci. U. S. A.* 105, 16266–16271

14 Shendure, J. *et al.* (2005) *Science* 309, 1728–1732

15 Trepagnier, E.H. *et al.* (2007) Controlling DNA capture and propagation through artificial nanopores. *Nano Lett.* 7, 2824–2830

16 Greenleaf, W.J. and Block, S.M. (2006) Single-molecule, motion-based DNA sequencing using RNA polymerase. *Science* 313, 801

17 Wheeler, D.A. *et al.* (2008) The complete genome of an individual by massively parallel DNA sequencing. *Nature* 452, 872–876

18 Levy, S. *et al.* (2007) The diploid genome sequence of an individual human. *PloS* 5, e254

19 Velculescu, V.E. *et al.* (1995) Serial analysis of gene expression. *Science* 270, 484–487

20 Mortazavi, A. *et al.* (2008) Mapping and quantifying mammalian transcriptomes by RNA-Seq. *Nat. Methods* 5, 621–628

21 Cloonan, N. *et al.* (2008) Stem cell transcriptome profiling via massive-scale mRNA sequencing. *Nat. Methods* 5, 613–619

22 Sultan, M. *et al.* (2008) A global view of gene activity and alternative splicing by deep sequencing of the human transcriptome. *Science* 321, 956–960

23 Robertson, G. *et al.* (2007) ChIP-Seq techniques. *Nat. Methods* 4, 651–657

24 Morrow, E.M. *et al.* (2008) Identifying autism loci and genes by tracing recent shared ancestry. *Science* 321, 218–223

25 Geschwind, D.H. (2008) Autism – family connections. *Nature* 454, 838–839

26 Sutcliffe, J.S. (2008) Insights into the pathogenesis of autism. *Science* 321, 208–209

27 Hugenholtz, P. and Tyson, G.W. (2008) Metagenomics. *Nature* 455, 481–483

28 Turnbaugh, P.J. *et al.* (2007) The human microbiome project. *Nature* 449, 804–810

What is next generation sequencing?

Sam Behjati,[1,2] Patrick S Tarpey[1]

[1]Cancer Genome Project, Wellcome Trust Sanger Institute, Wellcome Trust Genome Campus, Hinxton, Cambridgeshire, UK
[2]Department of Paediatrics, University of Cambridge, Cambridge, UK

Correspondence to
Dr Sam Behjati, Cancer Genome Project, Wellcome Trust Sanger Institute, Wellcome Trust Genome Campus, Hinxton, Cambridgeshire CB10 1SA, UK; sam.behjati@gmail.com

Received 31 July 2013
Accepted 2 August 2013
Published Online First
28 August 2013

To cite: Behjati S, Tarpey PS. *Arch Dis Child Educ Pract Ed* 2013;**98**:236–238.

ABSTRACT

Next generation sequencing (NGS), massively parallel or deep sequencing are related terms that describe a DNA sequencing technology which has revolutionised genomic research. Using NGS an entire human genome can be sequenced within a single day. In contrast, the previous Sanger sequencing technology, used to decipher the human genome, required over a decade to deliver the final draft. Although in genome research NGS has mostly superseded conventional Sanger sequencing, it has not yet translated into routine clinical practice. The aim of this article is to review the potential applications of NGS in paediatrics.

INTRODUCTION

There are a number of different NGS platforms using different sequencing technologies, a detailed discussion of which is beyond the scope of this article. However, all NGS platforms perform sequencing of millions of small fragments of DNA in parallel. Bioinformatics analyses are used to piece together these fragments by mapping the individual reads to the human reference genome. Each of the three billion bases in the human genome is sequenced multiple times, providing high depth to deliver accurate data and an insight into unexpected DNA variation (figure 1). NGS can be used to sequence entire genomes or constrained to specific areas of interest, including all 22 000 coding genes (a whole exome) or small numbers of individual genes.

POTENTIAL USES OF NGS IN CLINICAL PRACTICE

Clinical genetics

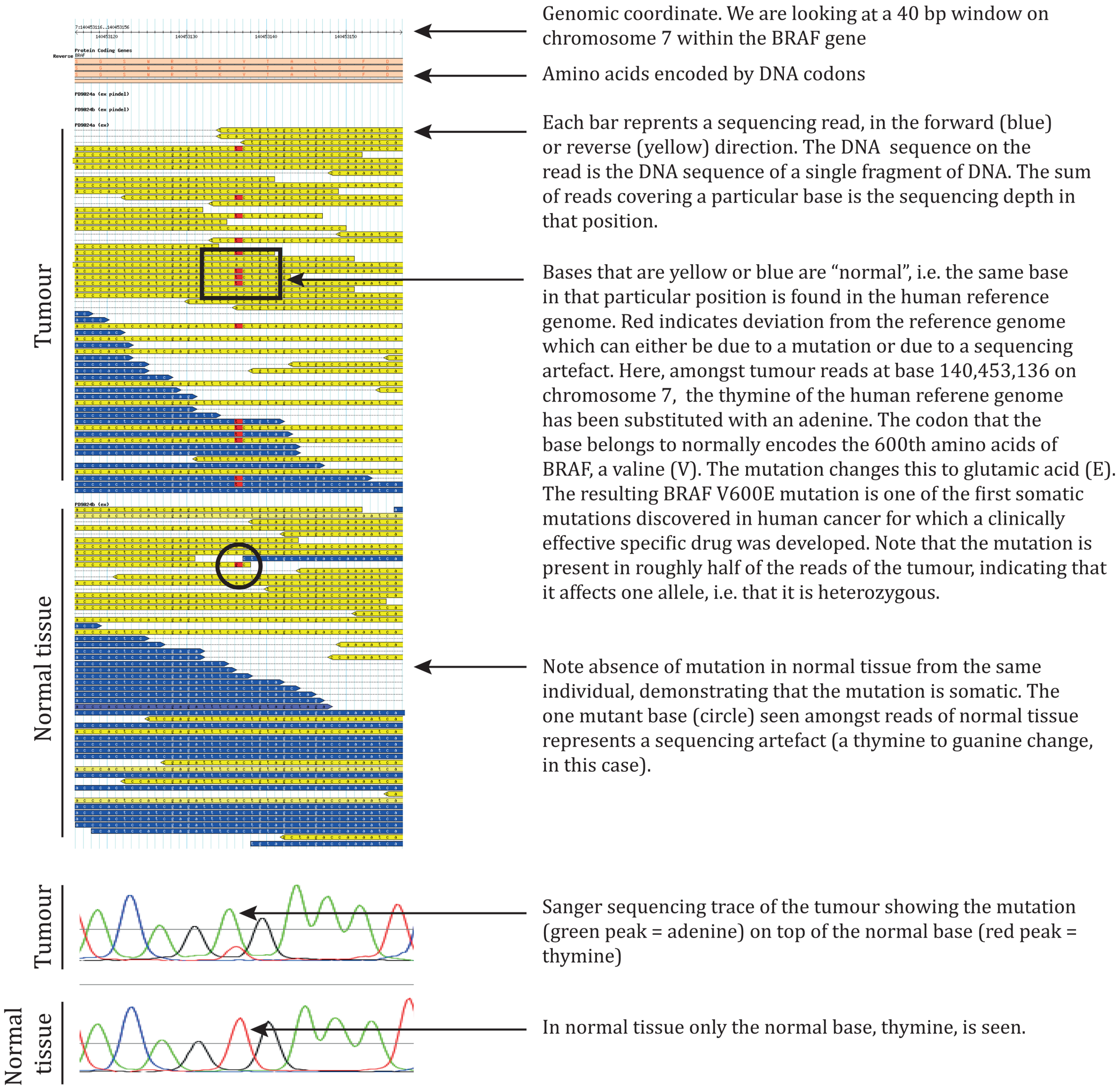
There are numerous opportunities to use NGS in clinical practice to improve patient care, including:

NGS captures a broader spectrum of mutations than Sanger sequencing

The spectrum of DNA variation in a human genome comprises small base changes (substitutions), insertions and deletions of DNA, large genomic deletions of exons or whole genes and rearrangements such as inversions and translocations. Traditional Sanger sequencing is restricted to the discovery of substitutions and small insertions and deletions. For the remaining mutations dedicated assays are frequently performed, such as fluorescence in situ hybridisation (FISH) for conventional karyotyping, or comparative genomic hybridisation (CGH) microarrays to detect submicroscopic chromosomal copy number changes such as microdeletions. However, these data can also be derived from NGS sequencing data directly, obviating the need for dedicated assays while harvesting the full spectrum of genomic variation in a single experiment. The only limitations reside in regions which sequence poorly or map erroneously due to extreme guanine/cytosine (GC) content or repeat architecture, for example, the repeat expansions underlying Fragile X syndrome, or Huntington's disease.

Genomes can be interrogated without bias

Capillary sequencing depends on preknowledge of the gene or locus under investigation. However, NGS is completely unselective and used to interrogate full genomes or exomes to discover entirely novel mutations and disease causing genes. In paediatrics, this could be exploited to unravel the genetic basis of unexplained syndromes. For example, a nationwide project, Deciphering Developmental Disorders,[1] running at the Wellcome Trust Sanger Institute in collaboration with NHS clinical genetics services aims to unravel the genetic basis of unexplained developmental delay by sequencing affected children and their parents to uncover deleterious de novo variants. Allying these molecular data with detailed clinical phenotypic information has been successful in identifying novel genes mutated in affected children with similar clinical features.



Genomic coordinate. We are looking at a 40 bp window on chromosome 7 within the BRAF gene

Amino acids encoded by DNA codons

Each bar reprents a sequencing read, in the forward (blue) or reverse (yellow) direction. The DNA sequence on the read is the DNA sequence of a single fragment of DNA. The sum of reads covering a particular base is the sequencing depth in that position.

Bases that are yellow or blue are "normal", i.e. the same base in that particular position is found in the human reference genome. Red indicates deviation from the reference genome which can either be due to a mutation or due to a sequencing artefact. Here, amongst tumour reads at base 140,453,136 on chromosome 7, the thymine of the human referene genome has been substituted with an adenine. The codon that the base belongs to normally encodes the 600th amino acids of BRAF, a valine (V). The mutation changes this to glutamic acid (E). The resulting BRAF V600E mutation is one of the first somatic mutations discovered in human cancer for which a clinically effective specific drug was developed. Note that the mutation is present in roughly half of the reads of the tumour, indicating that it affects one allele, i.e. that it is heterozygous.

Note absence of mutation in normal tissue from the same individual, demonstrating that the mutation is somatic. The one mutant base (circle) seen amongst reads of normal tissue represents a sequencing artefact (a thymine to guanine change, in this case).

Sanger sequencing trace of the tumour showing the mutation (green peak = adenine) on top of the normal base (red peak = thymine)

In normal tissue only the normal base, thymine, is seen.

Figure 1 Example of next generation sequencing (NGS) raw data-BRAF V600E mutation in melanoma. The mutation was found by our group in 2002 as part of several year-long efforts to define somatic mutations in human cancer using Sanger sequencing, prior to the advent of NGS.

The increased sensitivity of NGS allows detection of mosaic mutations

Mosaic mutations are acquired as a postfertilisation event and consequently they present at variable frequency within the cells and tissues of an individual. Capillary sequencing may miss these variants as they frequently present with a subtlety which falls below the sensitivity of the technology. NGS sequencing provides a far more sensitive read-out and can therefore be used to identify variants which reside in just a few per cent of the cells, including mosaic variation. In addition, the sensitivity of NGS sequencing can be increased further, simply by increasing sequencing depth. This has seen NGS employed for very sensitive investigations such as interrogating foetal DNA from maternal blood[2] or tracking the levels of tumour cells from the circulation of cancer patients.[3]

MICROBIOLOGY

The main utility of NGS in microbiology is to replace conventional characterisation of pathogens by morphology, staining properties and metabolic criteria with a genomic definition of pathogens. The genomes of pathogens define what they are, may harbour information about drug sensitivity and inform the relationship of different pathogens with each other which can be used to trace sources of infection outbreaks. The last recently received media attention, when NGS was used to reveal and trace an outbreak of methicillin-resistant Staphylococcus aureus (MRSA) on a neonatal intensive care unit in the UK.[4] What was most remarkable was that routine microbiological surveillance did not show that the cases of MRSA that occurred over several months were related. NGS of the pathogens, however, allowed precise characterisation of the MRSA isolates and revealed a protracted outbreak of MRSA which could be traced to a single member of staff.

ONCOLOGY

The fundamental premise of cancer genomics is that cancer is caused by somatically acquired mutations, and consequently it is a disease of the genome. Although capillary-based cancer sequencing has been ongoing for over a decade, these investigations were limited to relatively few samples and small numbers of candidate genes. With the advent of NGS, cancer genomes can now be systemically studied in their entirety, an endeavour ongoing via several large scale cancer genome projects around the world, including a dedicated paediatric cancer genome project.[5] For the child suffering from cancer this may provide many benefits including a more precise diagnosis and classification of the disease, more accurate prognosis, and potentially the identification of 'drug-able' causal mutations. Individual cancer sequencing may, therefore, provide the basis of personalised cancer management. Currently pilot projects are underway using NGS of cancer genomes in clinical practice, mainly aiming to identify mutations in tumours that can be targeted by mutation-specific drugs.

LIMITATIONS

The main disadvantage of NGS in the clinical setting is putting in place the required infrastructure, such as computer capacity and storage, and also the personnel expertise required to comprehensively analyse and interpret the subsequent data. In addition, the volume of data needs to be managed skilfully to extract the clinically important information in a clear and robust interface. The actual sequencing cost of NGS is negligible. For example, a state of the art NGS platform can generate approximately 150 000 000 reads for around £1000 whereas a single Sanger read typically costs less than £1. However, to make NGS cost effective one would have to run large batches of samples which may require supra-regional centralisation. Following the initial capital investment, the capacity of a NGS facility can provide a service on national scale likely offering economic benefits in addition to improvements in patient care.

Clinical bottom line

- ▸ NGS has huge potential but is presently used primarily for research.
- ▸ NGS will allow paediatricians to take genetic information to the bedside.

Funding SB receives a Wellcome Trust Clinical Fellowship.

Competing interests None.

Provenance and peer review Commissioned; internally peer reviewed.

REFERENCES

1 http://www.ddduk.org/

2 Harris SR, Cartwright EJ, Török ME, *et al*. Whole-genome sequencing for analysis of an outbreak of meticillin-resistant Staphylococcus aureus: a descriptive study. *Lancet Infect Dis* 2013;13:130–6.

3 Dawson SJ, Tsui DW, Murtaza M, *et al*. Analysis of circulating tumor DNA to monitor metastatic breast cancer. *N Engl J Med* 2013;368:1199–209.

4 Chiu RW, Chan KC, Gao Y, *et al*. Noninvasive prenatal diagnosis of fetal chromosomal aneuploidy by massively parallel genomic sequencing of DNA in maternal plasma. *Proc Natl Acad Sci USA* 2008;105:20458–63.

5 http://www.pediatriccancergenomeproject.org

An introduction to Next-Generation Sequencing Technology

www.illumina.com/technology/next-generation-sequencing.html

For Research Use Only. Not for use in diagnostic procedures.

Table of Contents

I. Welcome to Next-Generation Sequencing

a. The Evolution of Genomic Science

DNA sequencing has come a long way since the days of two-dimensional chromatography in the 1970s. With the advent of the Sanger chain termination method[1] in 1977, scientists gained the ability to sequence DNA in a reliable, reproducible manner. A decade later, Applied Biosystems introduced the first automated, capillary electrophoresis (CE)-based sequencing instruments, the AB370 in 1987 and the AB3730xl in 1998, instruments that became the primary workhorses for the NIH-led and Celera-led Human Genome Projects.[2] While these "first-generation" instruments were considered high throughput for their time, the Genome Analyzer emerged in 2005 and took sequencing runs from 84 kilobase (kb) per run to 1 gigabase (Gb) per run.[3] The short read, massively parallel sequencing technique was a fundamentally different approach that revolutionized sequencing capabilities and launched the "next generation" in genomic science. From that point forward, the data output of next-generation sequencing (NGS) has outpaced Moore's law, more than doubling each year (Figure 1).

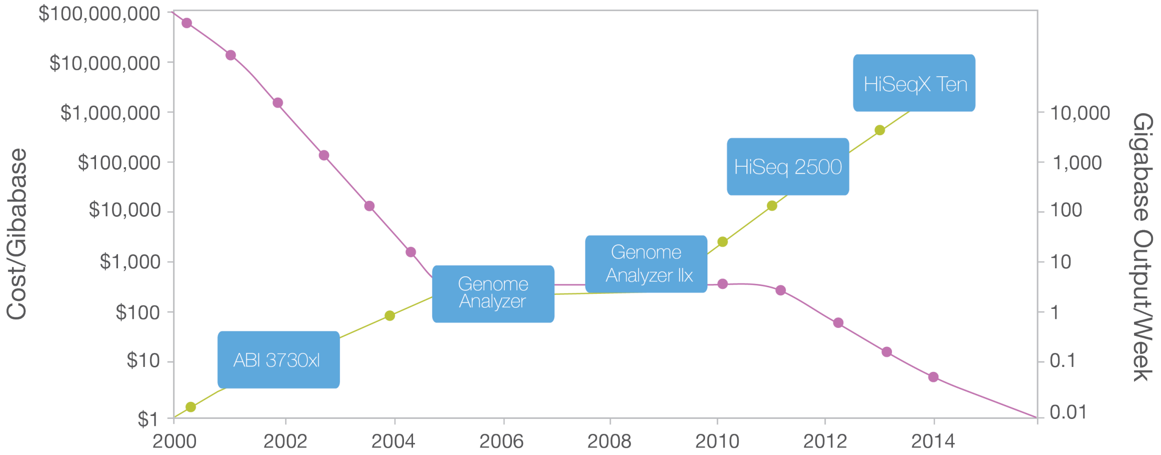


Figure 1: Sequencing Cost and Data Output Since 2000—The dramatic rise of data output and concurrent falling cost of sequencing since 2000. The Y-axes on both sides of the graph are logarithmic.

In 2005, a single run on the Genome Analyzer could produce roughly one gigabase of data. By 2014, the rate climbed to 1.8 terabases (Tb) of data in a single sequencing run, an astounding 1000× increase. It is remarkable to reflect on the fact that the first human genome, famously copublished in Science and Nature in 2001, required 15 years to sequence and cost nearly three billion dollars. In contrast, the HiSeq X® Ten System, released in 2014, can sequence over 45 human genomes in a single day for approximately $1000 each (Figure 2).[4]

Beyond the massive increase in data output, the introduction of NGS technology has transformed the way scientists think about genetic information. The $1000 dollar genome enables population-scale sequencing and establishes the foundation for personalized genomic medicine as part of standard medical care. Researchers can now analyze thousands to tens of thousands of samples in a single year. As Eric Lander, founding director of the Broad Institute of MIT and Harvard and principal leader of the Human Genome Project, states:

> "The rate of progress is stunning. As costs continue to come down, we are entering a period where we are going to be able to get the complete catalog of disease genes. This will allow us to look at thousands of people and see the differences among them, to discover critical genes that cause cancer, autism, heart disease, or schizophrenia."[5]

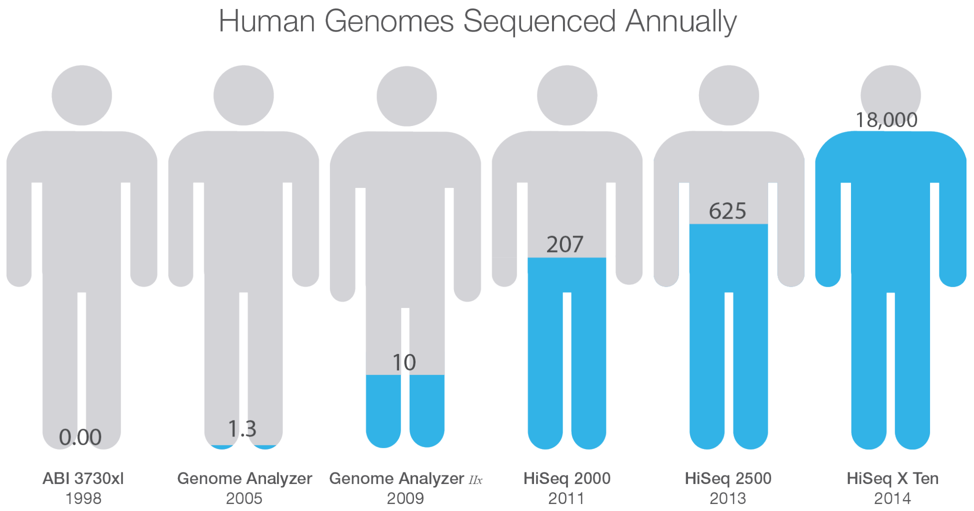


Figure 2: Human Genome Sequencing Over the Decades—The capacity to sequence all 3.2 billion bases of the human genome (at 30× coverage) has increased exponentially since the 1990s. In 2005, with the introduction of the Illumina Genome Analyzer System, 1.3 human genomes could be sequenced annually. Nearly 10 years later, with the Illumina HiSeq X Ten fleet of sequencing systems, the number has climbed to 18,000 human genomes a year.

b. The Basics of NGS Chemistry

In principle, the concept behind NGS technology is similar to CE sequencing. DNA polymerase catalyzes the incorporation of fluorescently labeled deoxyribonucleotide triphosphates (dNTPs) into a DNA template strand during sequential cycles of DNA synthesis. During each cycle, at the point of incorporation, the nucleotides are identified by fluorophore excitation. The critical difference is that, instead of sequencing a single DNA fragment, NGS extends this process across millions of fragments in a massively parallel fashion. More than 90% of the world's sequencing data are generated by Illumina sequencing by synthesis (SBS) chemistry.* It delivers high accuracy, a high yield of error-free reads, and a high percentage of base calls above Q30.[6–8]

Illumina NGS workflows include four basic steps:

1. **Library Preparation**—The sequencing library is prepared by random fragmentation of the DNA or cDNA sample, followed by 5′ and 3′ adapter ligation (Figure 3A). Alternatively, "tagmentation" combines the fragmentation and ligation reactions into a single step that greatly increases the efficiency of the library preparation process.[9] Adapter-ligated fragments are then PCR amplified and gel purified.
2. **Cluster Generation**—For cluster generation, the library is loaded into a flow cell where fragments are captured on a lawn of surface-bound oligos complementary to the library adapters. Each fragment is then amplified into distinct, clonal clusters through bridge amplification (Figure 3B). When cluster generation is complete, the templates are ready for sequencing.
3. **Sequencing**—Illumina SBS technology uses a proprietary reversible terminator–based method that detects single bases as they are incorporated into DNA template strands (Figure 3C). As all four reversible terminator–bound dNTPs are present during each sequencing cycle, natural competition minimizes incorporation bias and greatly reduces raw error rates compared to other technologies.[6,7] The result is highly accurate base-by-base sequencing that virtually eliminates sequence context–specific errors, even within repetitive sequence regions and homopolymers.
4. **Data Analysis**—During data analysis and alignment, the newly identified sequence reads are aligned to a reference genome (Figure 3D). Following alignment, many variations of analysis are possible, such as single nucleotide polymorphism (SNP) or insertion-deletion (indel) identification, read counting for RNA methods, phylogenetic or metagenomic analysis, and more.

A detailed animation of SBS chemistry is available at www.illumina.com/SBSvideo.

*Data calculations on file. Illumina, Inc., 2015.

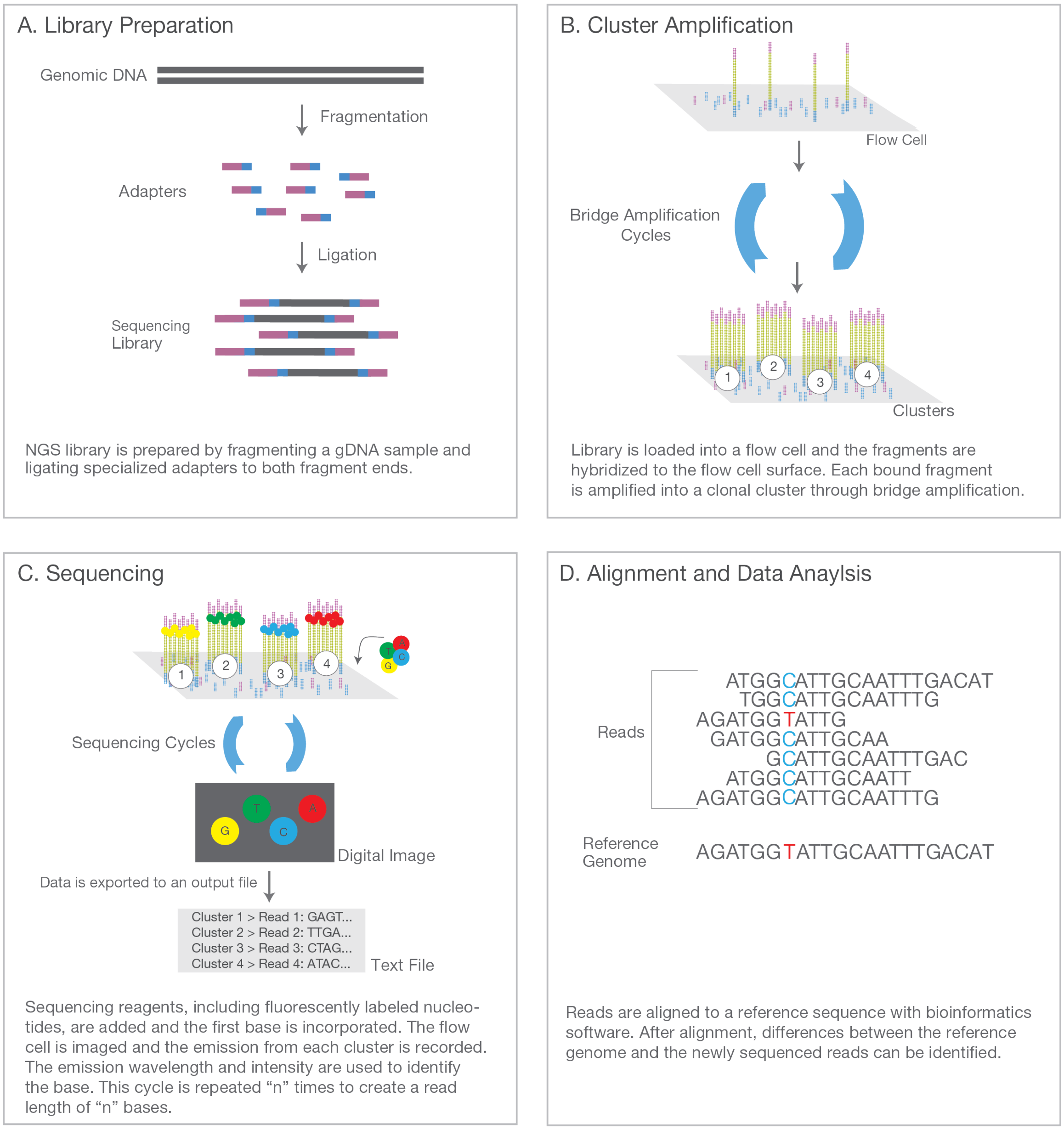

A. Library Preparation
Genomic DNA
Fragmentation
Adapters
Ligation
Sequencing Library
NGS library is prepared by fragmenting a gDNA sample and ligating specialized adapters to both fragment ends.
B. Cluster Amplification
Flow Cell
Bridge Amplification Cycles
1
2
3
4
Clusters
Library is loaded into a flow cell and the fragments are hybridized to the flow cell surface. Each bound fragment is amplified into a clonal cluster through bridge amplification.
C. Sequencing
1
2
3
4
T
A
C
G
Sequencing Cycles
T
A
G
C
Digital Image
Data is exported to an output file
Cluster 1 > Read 1: GAGT...
Cluster 2 > Read 2: TTGA...
Cluster 3 > Read 3: CTAG...
Cluster 4 > Read 4: ATAC...
Text File
Sequencing reagents, including fluorescently labeled nucleotides, are added and the first base is incorporated. The flow cell is imaged and the emission from each cluster is recorded. The emission wavelength and intensity are used to identify the base. This cycle is repeated "n" times to create a read length of "n" bases.
D. Alignment and Data Anaylsis
Reads
ATGGCATTGCAATTTGACAT
TGGCATTGCAATTTG
AGATGGTATTG
GATGGCATTGCAA
GCATTGCAATTTGAC
ATGGCATTGCAATT
AGATGGCATTGCAATTTG
Reference Genome
AGATGGTATTGCAATTTGACAT
Reads are aligned to a reference sequence with bioinformatics software. After alignment, differences between the reference genome and the newly sequenced reads can be identified.

Figure 3: Next-Generation Sequencing Chemistry Overview—Illumina NGS includes four steps: (A) library preparation, (B) cluster generation,(C) sequencing, and (D) alignment and data analysis.

c. Advances in Sequencing Technology

Paired-End Sequencing

A major advance in NGS technology occurred with the development of paired-end (PE) sequencing (Figure 4). PE sequencing involves sequencing both ends of the DNA fragments in a library and aligning the forward and reverse reads as read pairs. In addition to producing twice the number of reads for the same time and effort in library preparation, sequences aligned as read pairs enable more accurate read alignment and the ability to detect indels, which is not possible with single-read data.[8] Analysis of differential read-pair spacing also allows removal of PCR duplicates, a common artifact resulting from PCR amplification during library preparation. Furthermore, PE sequencing produces a higher number of SNV calls following read-pair alignment.[8,9] While some methods are best served by single-read sequencing, such as small RNA sequencing, most researchers currently use the paired-end approach.

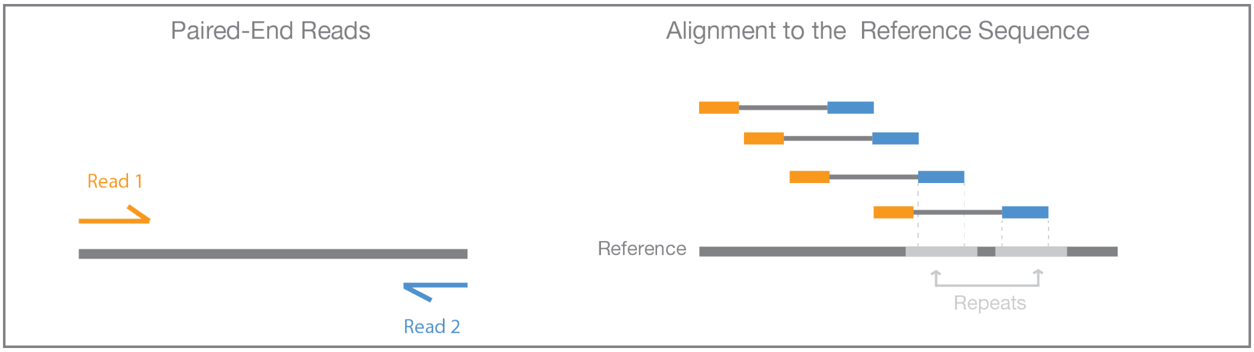


Figure 4: Paired-End Sequencing and Alignment—Paired-end sequencing enables both ends of the DNA fragment to be sequenced. Because the distance between each paired read is known, alignment algorithms can use this information to map the reads over repetitive regions more precisely. This results in better alignment of reads, especially across difficult-to-sequence, repetitive regions of the genome.

Tunable Coverage and Unlimited Dynamic Range

The digital nature of NGS allows a virtually unlimited dynamic range for read-counting methods, such as gene expression analysis. Microarrays measure continuous signal intensities and the detection range is limited by noise at the low end and signal saturation at the high end, while NGS quantifies discrete, digital sequencing read counts. By increasing or decreasing the number of sequencing reads, researchers can tune the sensitivity of an experiment to accommodate various study objectives. Because the dynamic range with NGS is adjustable and nearly unlimited, researchers can quantify subtle gene expression changes with much greater sensitivity than traditional microarray-based methods. Sequencing runs can be tailored to zoom in with high resolution on particular regions of the genome, or provide a more expansive view with lower resolution.

The ability to easily tune the level of coverage offers several experimental design advantages. For instance, somatic mutations may only exist within a small proportion of cells in a given tissue sample. Using mixed tumor–normal cell samples, the region of DNA harboring the mutation must be sequenced at extremely high coverage, often upwards of 1000×, to detect these low-frequency mutations within the mixed cell population. On the other side of the coverage spectrum, a method like genome-wide variant discovery usually requires a much lower coverage level. In this case, the study design involves sequencing many samples (hundreds to thousands) at lower resolution, to achieve greater statistical power within a given population.

Advances in Library Preparation

With Illumina NGS, library preparation has undergone rapid improvements. The first NGS library prep protocols involved random fragmentation of the DNA or RNA sample, gel-based size selection, ligation of platform-specific oligonucleotides, PCR amplification, and several purification steps. While the 1–2 days required to generate these early NGS libraries were a great improvement over traditional cloning techniques, current NGS protocols, such as Nextera® XT DNA Library Preparation, have reduced the library prep time to less than 90 minutes.[10] PCR-free and gel-free kits are also available for sensitive sequencing methods. PCR-free library preparation kits result in superior coverage of traditionally challenging areas such as high AT/GC-rich regions, promoters, and homopolymeric regions.[11]

For a complete list of Illumina library preparation kits, visit www.illumina.com/products/by-type/sequencing-kits/library-prep-kits.html.

Multiplexing

In addition to the rise of data output per run, the sample throughput per run in NGS has also increased over time. Multiplexing allows large numbers of libraries to be pooled and sequenced simultaneously during a single sequencing run (Figure 5). With multiplexed libraries, unique index sequences are added to each DNA fragment during library preparation so that each read can be identified and sorted before final data analysis. With PE sequencing and multiplexing, NGS has dramatically reduced the time to data for multisample studies and enabled researchers to go from experiment to data quickly and easily.

Gains in throughput from multiplexing come with an added layer of complexity, as sequencing reads from pooled libraries need to be identified and sorted computationally in a process called demultiplexing before final data analysis (Figure 5). The phenomenon of index misassignment between multiplexed libraries is a known issue that has impacted NGS technologies from the time sample multiplexing was developed.[12] Index hopping is a specific cause of index misassignment that can result in incorrect assignment of libraries from the expected index to a different index in the pool, leading to misalignment and inaccurate sequencing results.

For more information regarding index hopping, including mechanisms by which it occurs, how Illumina measures index hopping, and best practices for mitigating the impact of index hopping on sequencing data quality, read the Effects of Index Misassignment on Multiplexing and Downstream Analysis White Paper.

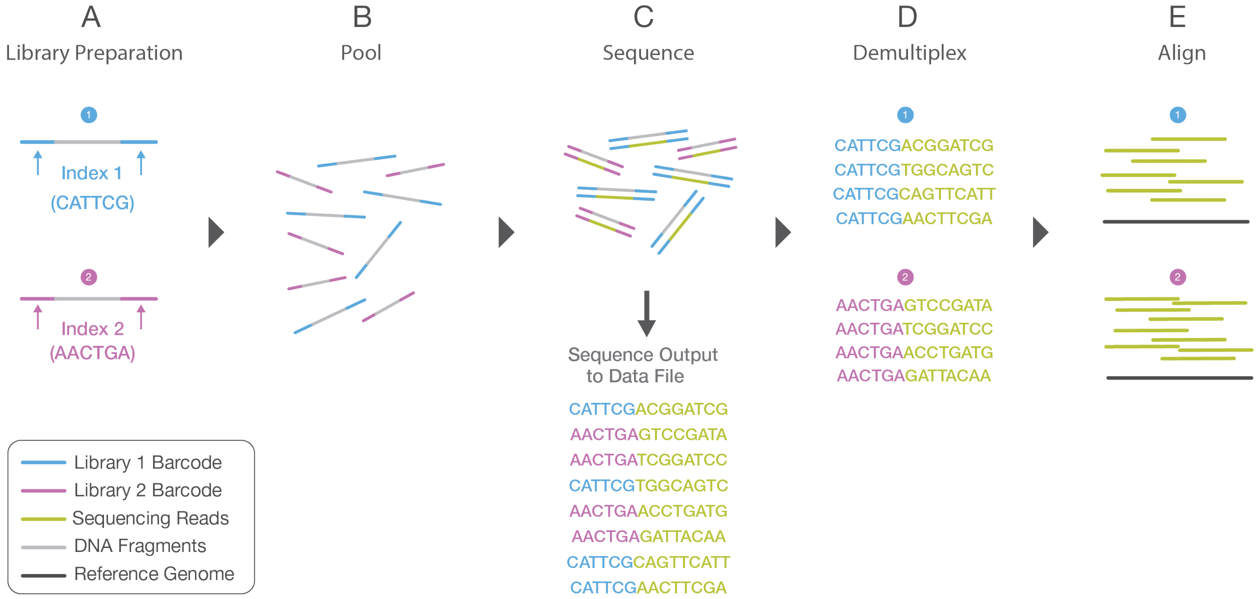


Figure 5: Library Multiplexing Overview—(A) Unique index sequences are added to two different libraries during library preparation. (B) Libraries are pooled together and loaded into the same flow cell lane. (C) Libraries are sequenced together during a single instrument run. All sequences are exported to a single output file. (D) A demultiplexing algorithm sorts the reads into different files according to their indexes. (E) Each set of reads is aligned to the appropriate reference sequence.

Flexible, Scalable Instrumentation

While the latest NGS platforms can produce massive data output, NGS technology is also highly flexible and scalable. Sequencing systems are available for every method and scale of study, from small laboratories to large genome centers (Figure 6). Illumina NGS instruments range from the benchtop MiniSeq™ System, with output ranging from 1.8–7.5 Gb for targeted sequencing studies, to the NovaSeq™ 6000 System, which can generate an impressive 6 Tb and 20 B reads in ~ 2 days† for population-scale studies.

Flexible run configurations are also engineered into the design of Illumina NGS sequencers. For example, the HiSeq® 2500 System offers two run modes and single or dual flow cell sequencing while the NextSeq® Series of Sequencing Systems offers two flow cell types to accommodate different throughput requirements. The HiSeq 3000/4000 Series uses the same patterned flow cell technology as the HiSeq X instruments for cost-effective production-scale sequencing. The new NovaSeq Series of systems unites the latest high-performance imaging with the next generation of Illumina patterned flow cell technology to deliver massive increases in throughput. This flexibility allows researchers to configure runs tailored to their specific study requirements, with the instrument of their choice.

For an in-depth comparison of Illumina platforms, visit www.illumina.com/systems/sequencing.html or explore the Sequencing Platform Comparison Tool at www.illumina.com/systems/sequencing-platforms/comparison-tool.html.

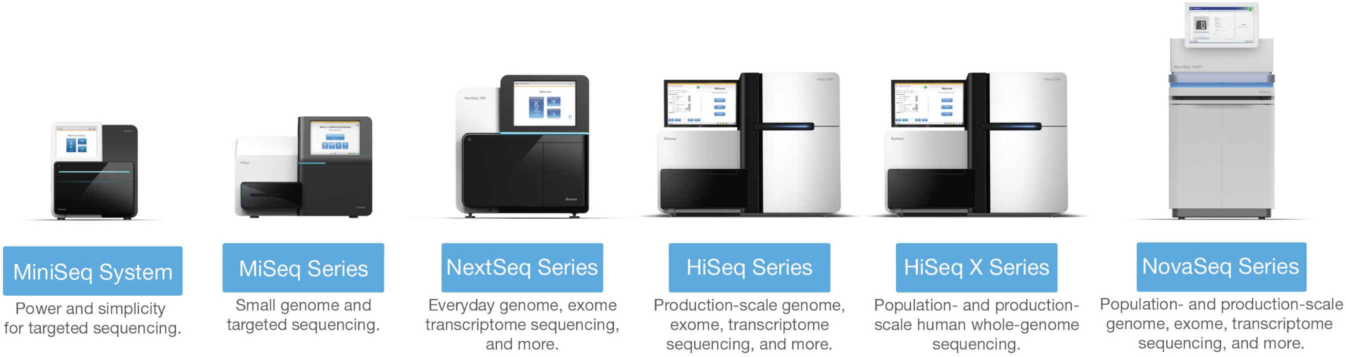

MiniSeq System
Power and simplicity for targeted sequencing.
MiSeq Series
Small genome and targeted sequencing.
NextSeq Series
Everyday genome, exome transcriptome sequencing, and more.
HiSeq Series
Production-scale genome, exome, transcriptome sequencing, and more.
HiSeq X Series
Population- and production-scale human whole-genome sequencing.
NovaSeq Series
Population- and production-scale genome, exome, transcriptome sequencing, and more.

Figure 6: Sequencing Systems for Virtually Every Scale—Illumina offers innovative NGS platforms that deliver exceptional data quality and accuracy over a wide scale, from small benchtop sequencers to production-scale sequencing systems.

II. NGS Methods

NGS platforms enable a wide variety of methods, allowing researchers to ask virtually any question related to the genome, transcriptome, or epigenome of any organism. Sequencing methods differ primarily by how the DNA or RNA samples are obtained (eg, organism, tissue type, normal vs. affected, experimental conditions, etc) and by the data analysis options used. After the sequencing libraries are prepared, the actual sequencing stage remains fundamentally the same, regardless of the method. There are various standard library preparation kits that offer protocols for whole-genome sequencing (WGS), RNA sequencing (RNA-Seq), targeted sequencing (such as exome sequencing or 16S sequencing), custom-selected regions, protein-binding regions, and more. Although the number of NGS methods is constantly growing, a brief overview of the most common methods is presented here.

a. Genomics

Whole-Genome Sequencing

Microarray-based, genome-wide association studies (GWAS) have been a common approach for identifying disease associations across the whole genome. While GWAS microarrays can interrogate over four million markers per sample, the most comprehensive method of interrogating the 3.2 billion bases of the human genome is WGS. The rapid drop in sequencing cost and the ability of WGS to produce large volumes of data rapidly make it a powerful tool for genomics research. While WGS is commonly associated with sequencing human genomes, the scalable, flexible nature of the method makes it equally useful for sequencing any species, such as agriculturally important livestock, plant genomes, or disease-related microbial genomes. This broad utility was demonstrated during the recent *E. coli* outbreak in Europe in 2011, which prompted a rapid scientific response. Using the latest NGS systems, researchers quickly sequenced the bacterial strain, enabling them to track the origins and transmission of the outbreak as well as identify genetic mutations conferring the increased virulence.[13]

Exome Sequencing

Exome sequencing is a widely-used targeted sequencing method. The exome represents less than 2% of the human genome, but contains most of the known disease-causing variants, making whole-exome sequencing (WES) a cost-effective alternative to WGS.[14] With WES, the protein-coding portion of the genome is selectively captured and sequenced. It can efficiently identify variants across a wide range of applications, including population genetics, genetic disease, and cancer studies.

[†] With dual flow cell mode enabled.

De novo Sequencing

De novo sequencing refers to sequencing a novel genome where there is no reference sequence available for alignment. Sequence reads are assembled as contigs and the coverage quality of *de novo* sequence data depends on the size and continuity of the contigs (ie, the number of gaps in the data). Another important factor in generating high-quality *de novo* sequences is the diversity of insert sizes included in the library. Combining short-insert paired-end and long-insert mate pair sequences is the most powerful approach for maximal coverage across the genome (Figure 7). The combination of insert sizes enables detection of the widest range of structural variant types and is essential for accurately identifying more complex rearrangements. The short-insert reads, sequenced at higher depths, can fill in gaps not covered by the long inserts, which are often sequenced at lower read depths. Therefore, using a combined approach results in higher quality assemblies. In parallel with NGS technology improvements, many algorithmic advances have emerged in sequence assemblers for short-read data. Researchers can perform high-quality *de novo* assembly using NGS reads and publicly available short-read assembly tools with existing computer resources in the laboratory.

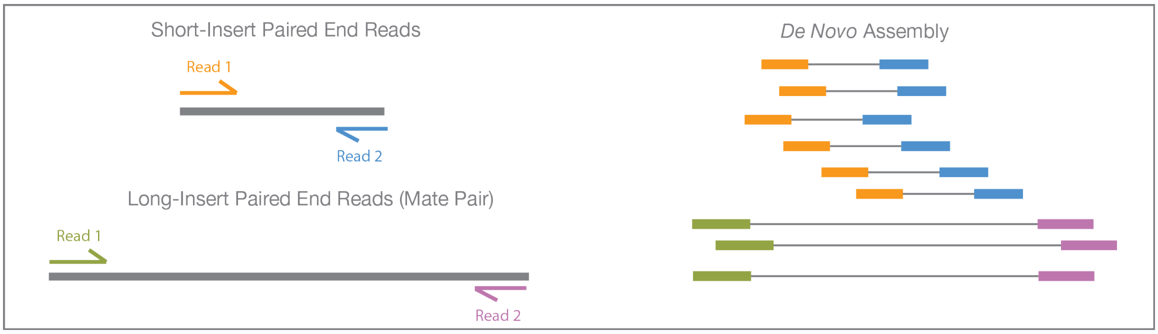

Short-Insert Paired End Reads
Read 1
Read 2
Long-Insert Paired End Reads (Mate Pair)
Read 1
Read 2
De Novo Assembly

Figure 7: Mate Pairs and *De novo* Assembly—Using a combination of short and long insert sizes with paired-end sequencing results in maximal coverage of the genome for *de novo* assembly.

Targeted Sequencing

With targeted sequencing, a subset of genes or regions of the genome are isolated and sequenced. Targeted sequencing allows researchers to focus time, expenses, and data analysis on specific areas of interest and enables sequencing at much higher coverage levels. For example, a typical WGS study achieves coverage levels of 30–50× per genome, while a targeted resequencing project can easily cover the target region at 500–1000× or higher. This higher coverage allows researchers to identify rare variants, variants that would be too rare and too expensive to identify with WGS or CE-based sequencing.

Targeted sequencing panels can be purchased with fixed, preselected content or can be custom designed. A wide variety of targeted sequencing library prep kits are available, including kits with probe sets focused on specific areas of interest such as cancer, cardiomyopathy, or autism. Custom probe sets are available through DesignStudio™ Software enabling researchers to target regions of the genome relevant to specific research interests. Custom targeted sequencing is ideal for examining genes in specific pathways, or for follow-up studies from GWAS or WGS. Illumina currently supports two methods for targeted sequencing, target enrichment and amplicon generation (Figure 8).

Target enrichment captures between 10 kb–62 Mb regions, depending on the library prep kit parameters. Amplicon sequencing allows researchers to sequence 16–1536 targets at a time, spanning 2.4–652.8 kb of total content, depending on the library prep kit used. This highly multiplexed approach enables a wide range of applications for discovery, validation, or screening of genetic variants. Amplicon sequencing is useful for discovery of rare somatic mutations in complex samples (eg, cancerous tumors mixed with germline DNA).[15,16] Another common amplicon application is sequencing the bacterial 16S rRNA gene across multiple species, a widely used method for phylogeny and taxonomy studies, particularly in diverse metagenomic samples.[17]

For more information on Illumina targeted, WGS, exome, or *de novo* sequencing solutions, visit www.illumina.com/applications/sequencing/dna_sequencing.html.

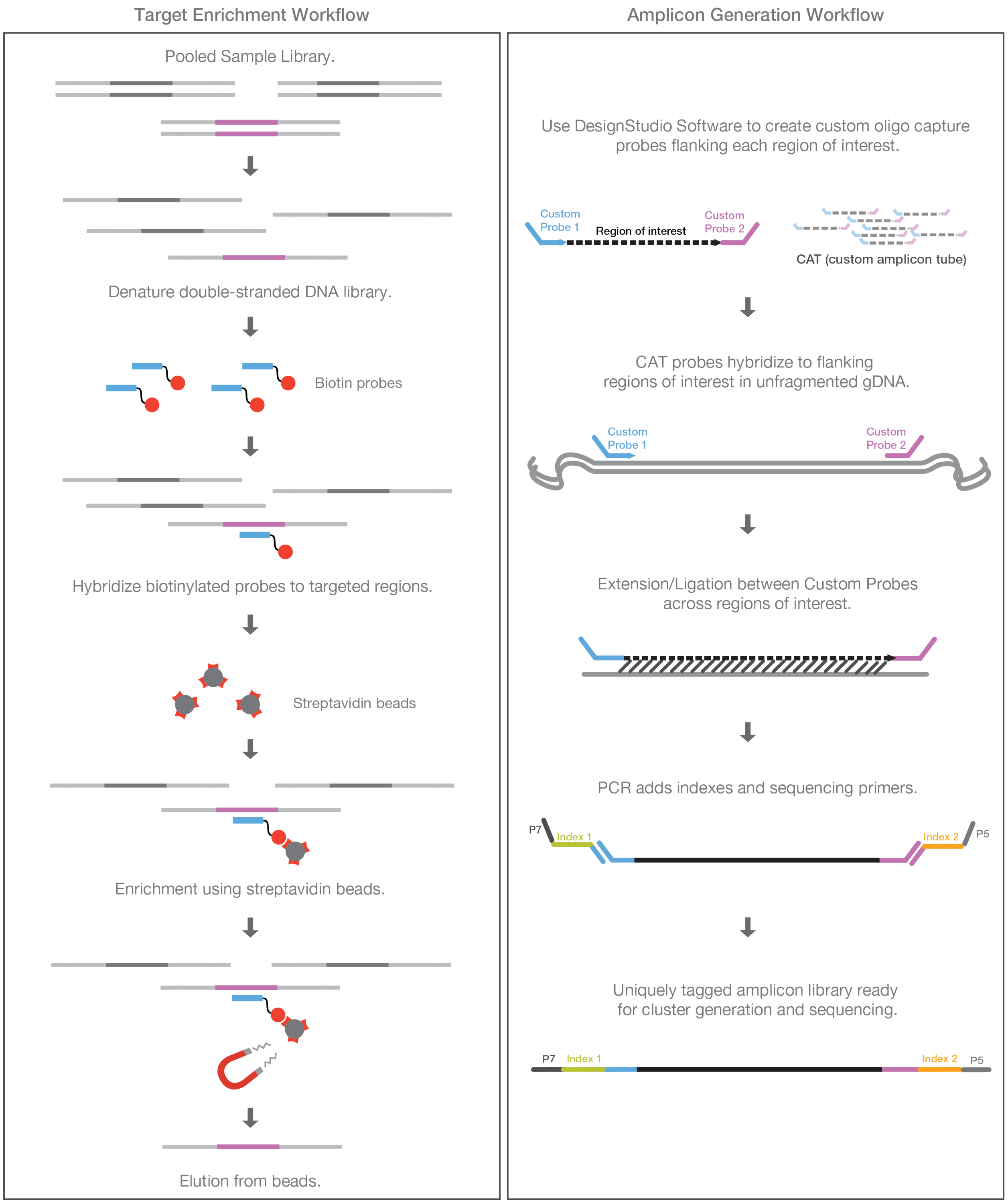

Target Enrichment Workflow
Pooled Sample Library.
Denature double-stranded DNA library.
Biotin probes
Hybridize biotinylated probes to targeted regions.
Streptavidin beads
Enrichment using streptavidin beads.
Elution from beads.
Amplicon Generation Workflow
Use DesignStudio Software to create custom oligo capture probes flanking each region of interest.
Custom Probe 1
Region of interest
Custom Probe 2
CAT (custom amplicon tube)
CAT probes hybridize to flanking regions of interest in unfragmented gDNA.
Custom Probe 1
Custom Probe 2
Extension/Ligation between Custom Probes across regions of interest.
PCR adds indexes and sequencing primers.
P7
Index 1
Index 2
P5
Uniquely tagged amplicon library ready for cluster generation and sequencing.
P7
Index 1
Index 2
P5

Figure 8: Target Enrichment and Amplicon Generation Workflows—With target enrichment, specific regions of interest are captured by hybridization to biotinylated probes, then isolated by magnetic pulldown. Amplicon sequencing involves the amplification and purification of regions of interest using highly multiplexed PCR oligos sets.

For Research Use Only. Not for use in diagnostic procedures.

b. Transcriptomics

Library preparation methods for RNA-Seq typically begin with total RNA sample preparation followed by a ribosome removal step. The total RNA sample is then converted to cDNA before standard NGS library preparation. RNA-Seq focused on mRNA, small RNA, noncoding RNA, or microRNAs can be achieved by including additional isolation or enrichment steps before cDNA synthesis (Figure 9).

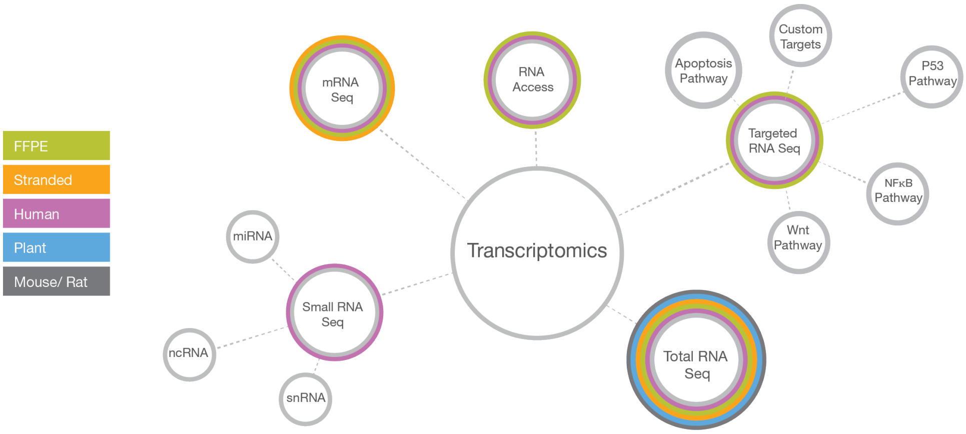


Figure 9: A Complete View of Transcriptomics with NGS—A broad range of methods for transcriptomics with NGS have emerged over the past 10 years including total RNA-Seq, mRNA-Seq, small RNA-Seq, and targeted RNA-Seq.

Total RNA and mRNA Sequencing

Transcriptome sequencing is a major advance in the study of gene expression because it allows a snapshot of the whole transcriptome rather than a predetermined subset of genes. Whole-transcriptome sequencing provides a comprehensive view of a cellular transcriptional profile at a given biological moment and greatly enhances the power of RNA discovery methods. As with any sequencing method, an almost unlimited dynamic range allows identification and quantification of both common and rare transcripts. Additional capabilities include aligning sequencing reads across splice junctions, and detection of isoforms, novel transcripts, and gene fusions. Library preparation kits that support precise detection of strand orientation are available for both total RNA-Seq and mRNA-Seq methods.

Targeted RNA Sequencing

Targeted RNA sequencing is a method for measuring transcripts of interest for detecting differential expression, allele-specific expression, detection of gene-fusions, isoforms, cSNPs, and splice junctions. Illumina TruSeq® Targeted RNA Sequencing Kits include preconfigured, experimentally validated panels focused on specific cellular pathways or disease states such as apoptosis, cardiotoxicity, NFκB pathway, and more. Custom content can be designed and ordered for analysis of specific genes of interest. Targeted RNA sequencing is a powerful method for the investigation of specific pathways of interest or for the validation of gene expression microarray or whole-transcriptome sequencing results.

Small RNA and Noncoding RNASequencing

Small, noncoding RNA, or microRNA s are short, 18–22 bp nucleotides that play a role in the regulation of gene expression often as gene repressors or silencers. The study of microRNAs has grown as their role in transcriptional and translational regulation has become more evident.[18,19]

For more information regarding Illumina solutions for small RNA (noncoding RNA), targeted RNA, total RNA, and mRNA sequencing, visit www.illumina.com/applications/sequencing/rna.html.

c. Epigenomics

While genomics involves the study of heritable or acquired alterations in the DNA sequence, epigenetics is the study of heritable changes in gene activity caused by mechanisms other than DNA sequence changes. Mechanisms of epigenetic activity include DNA methylation, small RNA–mediated regulation, DNA–protein interactions, histone modification, and more.

Methylation Sequencing

A critical focus in epigenetics is the study of cytosine methylation (5mC) states across specific areas of regulation, such as promotors or heterochromatin. Cytosine methylation can significantly modify temporal and spatial gene expression and chromatin remodeling.[20] While there are many methods for the study of genetic methylation, methylation sequencing leverages the advantages of NGS technology and genome-wide analysis while assessing methylation states at the single-nucleotide level. Two methylation sequencing methods are widely used: whole-genome bisulfite sequencing (WGBS) and reduced representation bisulfite sequencing (RRBS). With WGBS, sodium bisulfite chemistry converts nonmethylated cytosines to uracils, which are then converted to thymines in the sequence reads or data output. In RRBS, DNA is digested with MspI, a restriction enzyme unaffected by methylation status. Fragments in the 100–150 bp size range are isolated to enrich for CpG and promotor containing DNA regions. Sequencing libraries are then constructed using the standard NGS protocols.

For more information on methylation sequencing solutions, visit
www.illumina.com/techniques/sequencing/methylation-sequencing.html

ChIP Sequencing

Protein–DNA or protein–RNA interactions have a significant impact on many biological processes and disease states. These interactions can be surveyed with NGS by combining chromatin immunoprecipitation (ChIP) assays and NGS methods. ChIP-Seq protocols begin with the chromatin immunoprecipitation step (ChIP protocols vary widely as they must be specific to the species, tissue type, and experimental conditions).

For more information on ChIP-Seq, visit
www.illumina.com/techniques/sequencing/dna-sequencing/chip-seq.html.

Ribosome Profiling

Ribosome profiling is a method based on deep sequencing of ribosome protected–mRNA fragments. Purification and sequencing of these fragments provides a “snapshot” of all the ribosomes active in a cell at a specific time point. This information can determine what proteins are being actively translated in a cell, and can be useful for investigating translational control, measuring gene expression, determining the rate of protein synthesis, or predicting protein abundance. Ribosome profiling enables systematic monitoring of cellular translation processes and prediction of protein abundance. Determining what regions of a transcript are being translated can help define the proteome of complex organisms. With NGS, ribosome profiling allows detailed and accurate *in vivo* analysis of protein production.

To learn more about Illumina ribosome profiling, visit
www.illumina.com/applications/sequencing/rna.html.

III. Illumina DNA-to-Data NGS Solutions

a. The Illumina NGS Workflow

Illumina offers a comprehensive solution for the NGS workflow, from library preparation to data analysis (Figure 10). Library preparation kits are available for all NGS methods, including WGS, exome sequencing, targeted sequencing, RNA-Seq, and more. Illumina library preparation protocols can accommodate a range of throughput needs, from manual protocols for smaller laboratories to fully automated library preparation workstations for larger laboratories or genome centers. Likewise, Illumina offers a full portfolio of sequencing platforms, from the benchtop MiniSeq and MiSeq® Systems to the factory-scale HiSeq X and NovaSeq Series of Sequencing Systems that deliver the right level of speed, capacity, and cost for various laboratory or sequencing center requirements. For the last step in the NGS workflow, Illumina offers user-friendly bioinformatics tools that are easily accessible through the web, on instrument, or through onsite servers.

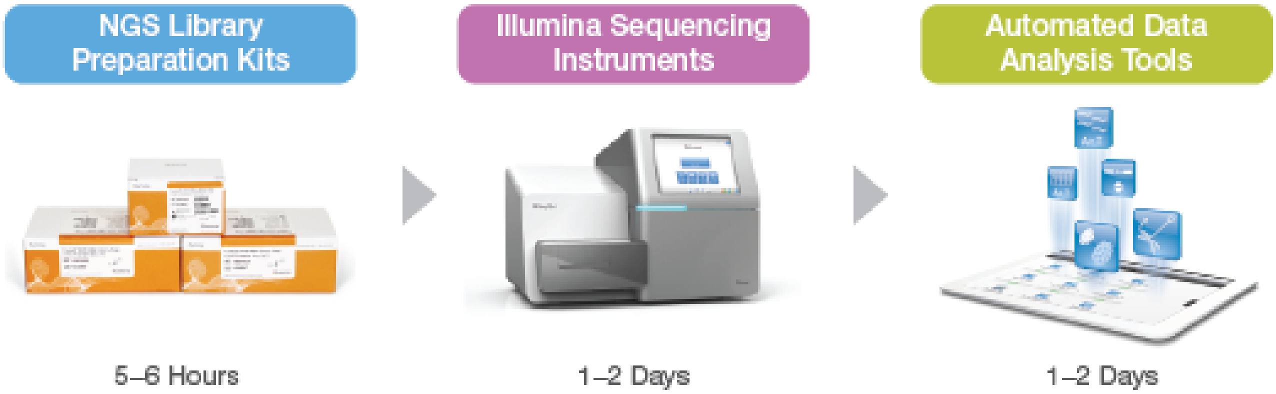

NGS Library Preparation Kits
Illumina Sequencing Instruments
Automated Data Analysis Tools
5–6 Hours
1–2 Days
1–2 Days

Figure 10: Illumina DNA-to-Data Solutions—Illumina provides fully integrated, DNA-to-data solutions, with technology and support for every step of the NGS workflow including library preparation, sequencing, and final data analysis.

b. Integrated Data Analysis

Data from any Illumina sequencing system can be streamed into BaseSpace® Sequence Hub, a user-friendly genomics cloud computing platform that offers simplified data management, analytical sequencing tools, and data storage. BaseSpace Sequence Hub is optimized to automate processing of the large volume of data generated. Researchers will find a rich ecosystem of commercial and open-source tools, from Illumina and third-party developers, for data analysis, including alignment and variant detection, annotation, visualization, interpretation, and somatic variant calling. BaseSpace Onsite Sequence Hub is a local version of BaseSpace Sequence Hub that enables data storage and analysis onsite through an installed local server. On-instrument access to BaseSpace Sequence Hub enables the integration of many workflow steps, including library prep planning with BaseSpace Prep,[‡] run set-up and chemistry validation, and real-time automatic data transfer to the BaseSpace computing environment.

The NGS workflow then proceeds seamlessly through alignment and subsequent data analysis steps with BaseSpace Apps. BaseSpace Apps offer a wide variety of analysis pipelines, including analysis for *de novo* assembly, SNP and indel variant analysis, RNA expression profiling, 16S metagenomics, tumor-normal comparisons, epigenetic/gene regulation analysis, and many more. Illumina collaborates closely with commercial and academic software developers to create a full ecosystem of data analysis tools that address the needs of various research objectives. In the final stages of the NGS workflow, data can be shared with collaborators or delivered instantly to customers around the world.[§]

To learn more about BaseSpace Sequence Hub, visit www.illumina.com/basespace.

[‡] Currently available with MiniSeq and NextSeq 500/550 Systems only. HiSeq and MiSeq Systems can use Illumina Experiment Manager (IEM) for the same planning and validation functions.

[§] Cloud-based environment only. BaseSpace Onsite Sequence Hub restricts data sharing to local users.

IV. Glossary

adapters: The oligos bound to the 5′ and 3′ end of each DNA fragment in a sequencing library. The adapters are complementary to the lawn of oligos present on the surface of Illumina sequencing flow cells.

bridge amplification: An amplification reaction that occurs on the surface of an Illumina flow cell. During flow cell manufacturing, the surface is coated with a lawn of two distinct oligonucleotides often referred to as "P5" and "P7." In the first step of bridge amplification, a single-stranded sequencing library (with complementary adapter ends) is loaded into the flow cell. Individual molecules in the library bind to complementary oligos as they "flow" across the oligo lawn. Priming occurs as the opposite end of a ligated fragment bends over and "bridges" to another complementary oligo on the surface. Repeated denaturation and extension cycles (similar to PCR) results in localized amplification of single molecules into millions of unique, clonal clusters across the flow cell. This process, also known as "clustering," occurs in an automated, flow cell instrument called a cBot System or in an onboard cluster module within an NGS instrument.

clusters: A clonal grouping of template DNA bound to the surface of a flow cell. Each cluster is seeded by a single template DNA strand and is clonally amplified through bridge amplification until the cluster has ~1000 copies. Each cluster on the flow cell produces a single sequencing read. For example, 10,000 clusters on the flow cell would produce 10,000 single reads and 20,000 paired-end reads.

contigs: A stretch of continuous sequence, *in silico*, generated by aligning overlapping sequencing reads.

coverage level: The average number of sequenced bases that align to each base of the reference DNA. For example, a whole genome sequenced at 30× coverage means that, on average, each base in the genome was sequenced 30 times.

flow cell: A glass slide with one, two, or eight physically separated lanes, depending on the instrument platform. Each lane is coated with a lawn of surface bound, adapter-complimentary oligos. A single library or a pool of up to 96 multiplexed libraries can be run per lane, depending on application parameters.

indexes/barcodes/tags: A unique DNA sequence ligated to fragments within a sequencing library for downstream *in silico* sorting and identification. Indexes are typically a component of adapters or PCR primers and are ligated to the library fragments during the sequencing library preparation stage. Illumina indexes are typically between 8–12 bp. Libraries with unique indexes can be pooled together, loaded into one lane of a sequencing flow cell, and sequenced in the same run. Reads are later identified and sorted via bioinformatic software. All together, this process is known as "multiplexing."

insert: During the library preparation stage, the sample DNA is fragmented, and the fragments of a specific size (typically 200–500 bp, but can be larger) are ligated or "inserted" in between two oligo adapters. The original sample DNA fragments are also referred to as "inserts."

mate pair library: A sequencing library with long inserts ranging in size from 2–5 kb typically run as paired-end libraries. The long gap length in between the sequence pairs is useful for building contigs in *de novo* sequencing, identification of indels, and other methods.

multiplexing: See "indexes/barcodes/tags."

read: NGS uses sophisticated instruments to determine the nucleotide sequence of a DNA or RNA sample. In general terms, a sequence "read" refers to the data string of A, T, C, and G bases corresponding to the sample DNA or RNA. With Illumina technology, millions of reads are generated in a single sequencing run.

reference genome: A reference genome is a fully sequenced and assembled genome that acts as a scaffold against which new sequence reads are aligned and compared. Typically, reads generated from a sequencing run are aligned to a reference genome as a first step in data analysis.

sequencing by synthesis (SBS): SBS technology uses four fluorescently labeled nucleotides to sequence the tens of millions of clusters on the flow cell surface in parallel. During each sequencing cycle, a single labeled dNTP is added to the nucleic acid chain. The nucleotide label serves as a "reversible terminator" for polymerization: after dNTP incorporation, the fluorescent dye is identified through laser excitation and imaging, then enzymatically cleaved to allow the next round of incorporation. Base calls are made directly from signal intensity measurements during each cycle.

V. References

1. Sanger F, Nicklen S, Coulson AR. DNA sequencing with chain-terminating inhibitors. *PNAS*. 1977;74(12):5463–5467.
2. Collins FS, Morgan M, Patrinos A. The human genome project: lessons from large-scale biology. *Science*. 2003;300(5617):286–290.
3. Davies K. 13 years ago, a beer summit in an English pub led to the birth of Solexa. *BioIT World*. September 28, 2010. (www.bio-itworld.com/2010/issues/sept-oct/solexa.html).
4. Illumina. HiSeq X Ten Series of Sequencing Systems. 2014. (www.illumina.com/documents/products/datasheets/datasheet-hiseq-x-ten.pdf).
5. Fallows J. When will genomics cure cancer?. *The Atlantic*. January 2014. (www.theatlantic.com/magazine/archive/2014/01/when-will-genomics-cure-cancer/355739/)
6. Ross MG, Russ C, Costello M, et al. Characterizing and measuring bias in sequence data. *Genome Biol*. 2013;14(5):R51.
7. Bentley DR, Balasubramanian S, Swerdlow HP, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nat*. 2008;456(7218):53–59.
8. Nakazato T, Ohta T, Bono H. Experimental design-based functional mining and characterization of high-throughput sequencing data in the sequence read archive. *PLoS One*. 2013;8(10):e77910.
9. Illumina. Nextera DNA Library Preparation Kits data sheet. 2014. (www.illumina.com/documents/products/datasheets/datasheet_nextera_dna_sample_prep.pdf).
10. Illumina. Nextera XT DNA Library Preparation Kit data sheet. 2014.(www.illumina.com/documents/products/datasheets/datasheet_nextera_xt_dna_sample_prep.pdf).
11. Illumina. TruSeq DNA PCR-Free Library Preparation Kit data sheet. 2013. (www.illumina.com/documents/products/datasheets/datasheet_truseq_dna_pcr_free_sample_prep.pdf).
12. Kircher M, Sawyer S, Meyer M. Double indexing overcomes inaccuracies in multiplex sequencing on the Illumina platform. *Nucleic Acids Res*. 2012:2513–2524.
13. Grad YH, Lipsitch M, Feldgarden M, et al. Genomic epidemiology of the Escherichia coli O104:H4 outbreaks in Europe, 2011. *PNAS*. 2012;109(8):3065–3070.
14. van Dijk EL, Auger H, Jaszczyszyn Y, Thermes C. Ten years of next-generation sequencing technology. Trends Genet. 2014;(9):418–426.
15. McEllistrem MC. Genetic diversity of the pneumococcal capsule: implications for molecular-based serotyping. *Future Microbiol*. 2009;4(7):857–865.
16. Lo YMD, Chiu RWK. Next-generation sequencing of plasma/serum DNA: an emerging research and molecular diagnostic tool. *Clin Chem*. 2009;55:607–608.
17. Ram JL, Karim AS, Sendler ED, Kato I. Strategy for microbiome analysis using 16S rRNA gene sequence analysis on the Illumina sequencing platform. *Syst Biol Reprod Med*. 2011;57(3):117–118.
18. Wang Y, Kim S, Kim IM. Regulation of metastasis by microRNAs in ovarian cancer. *Front Oncol*. 2014;10:143.
19. Dior Up, Kogan L, Chill HH, Eizenberg N, Simon A, Revel A. Emerging roles of microRNA in the embryo-endometrium cross talk. *Semin Reprod Med*. 2014;32 (5):402–409.
20. Phillips T. The role of methylation in gene expression. *Nat Education*. 2008;1(1):116.

Illumina, Inc. • 1.800.809.4566 toll-free (US) • +1.858.202.4566 tel • techsupport@illumina.com • www.illumina.com

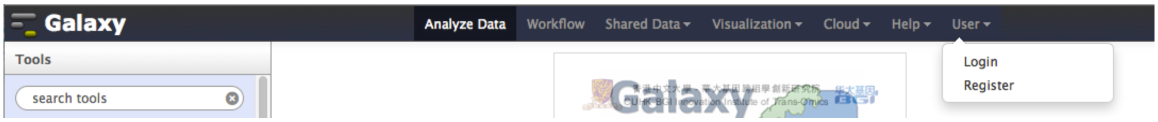
For Research Use Only. Not for use in diagnostic procedures.

Using Galaxy for NGS Analyses

Luce Skrabanek

Registering for a Galaxy account

Before we begin, first create an account on the main public Galaxy portal.
Go to:
https://main.g2.bx.psu.edu/
Under the User tab at the top of the page, select the Register link and follow the instructions on that page.



This will only take a moment, and will allow all the work that you do to persist between sessions and allow you to name, save, share, and publish Galaxy histories, workflows, datasets and pages.

Disk quotas

As a registered user on the Galaxy main server, you are now entitled to store up to 250GB of data on this public server. The little green bar at the top right of your Galaxy page will always show you how much of your allocated resources you are currently using. If you exceed your disk quota, no further jobs will be run until you have (permanently) deleted some of your datasets.

Note: Many of the examples throughout this document have been taken from published Galaxy pages. Thanks to users james, jeremy and aun1 for making these pages and the associated datasets available.

Importing data into Galaxy

The right hand panel of the Galaxy window is called the Tools panel. Here we can access all the tools that are available in this build of Galaxy. Which tools are available will depend on how the administrator of the Galaxy instance you are using has set it up.

The first tools we will look at will be the built-in tools for importing data into Galaxy. There are multiple ways of importing data into Galaxy. Under the Get Data tab, you will see a list of 15+ tools that you can use to import data into your history.

Upload from database queries

You can upload data from a number of different databases. For example, you can retrieve data from different sections of the UCSC Genome Browser, using the UCSC Main, UCSC Archaea and BX Main tools.

Retrieving data from UCSC

The Galaxy tool will open up the Table Browser from UCSC in the Galaxy window. At this point, you can use the Table Browser exactly as you would normally. For example, let's say we want to obtain a BED-formatted dataset of all RefSeq genes from platypus.

1. Open the Get Data → UCSC Main tool.
2. Select platypus from the Genome pulldown menu.
3. Change the Track pulldown menu to RefSeq genes.
4. Keep all other options fixed (including region: genome and output format: BED and the 'Send output to Galaxy' box checked).
5. Click on the 'Get Output' button.
6. Select the desired format (here we will choose the default one BED record per whole gene).
7. Click the 'Send query to Galaxy' button.

Similarly, you can access BioMart through the BioMart and Gramene tools. In these cases, the Galaxy window will be replaced by the query forms for these databases, but once you have retrieved your results, you will get the option to upload them to Galaxy.

Other database query systems that open up within Galaxy include:
FlyMine, MouseMine, RatMine, YeastMine, EuPathDB, EpiGRAPH, GenomeSpace.

The others will open up in their own window, replacing the Galaxy window, but will have an option to export the results from your query to Galaxy:
WormBase, modENCODE fly, modENCODE modMine, modENCODE worm.

Upload data from a File

The Get Data → Upload File tool is probably the easiest way of getting data into a history. Using this tool, you can import data from a file on your computer, or from a website or FTP site. Here we will import a file from the website associated with this Galaxy workshop.

1. Open the Get Data → Upload File tool.
2. Enter the following URL into the text-entry box (either by typing it or right-click the link on the course webpage, select 'Copy Link' and paste the link into the text-entry box.)
   http://chagall.med.cornell.edu/galaxy/rnaseq/GM12878_rnaseq1.fastqsanger
3. Change the File-Format pulldown menu from 'Auto-detect' to 'Fastqsanger'. Although Galaxy is usually quite good at recognizing formats, it is always safer, if you know the format of your file, not to leave its recognition to chance. This is especially important with FastQ formats, as they all look similar but will give different results if Galaxy assumes the wrong format (see FastQ quality scores section.)
4. Click Execute.

Galaxy will usually have some kind of help text below the parameters for any tool. However, the help text is not always up-to-date or covers all the options available. This is the case here, where the help text details many of the formats listed in the File Format pulldown menu, but not all of them.

As mentioned, we chose the fastqsanger format here. The FastQ format is a format that includes not only the base sequence, but also the quality scores for each position. You will see five possible different FastQ formats in the pulldown menu [fastq, fastqcssanger, fastqillumina, fastqsanger, fastqsolexa]. The fastq format is the "default" FastQ format. If your dataset becomes tagged with this format, it will have to be converted into one of the other named formats before you can begin to do analysis with it. The fastqcssanger format is for color space sequences and the other three are FastQ formats that encode quality scores into ASCII with different conversions.

History

The right hand panel of the Galaxy window is called the History panel. Histories are the way that datasets are stored in an organized fashion within Galaxy.

1. To change the name of your history, click on the Unnamed history text. This will highlight the history name window and allow you to type the new name of the history.

Histories can also be associated with tags and annotations which can help to identify the purpose of a history. As the number of histories in your Galaxy account grows, these tags/annotations become more and more important to help you keep track of what the function of each history is.

Every dataset that you create gets a new window in this history panel. Clicking on the name of the dataset will open the window up to a slightly larger preview version, which shows the first few lines of that dataset.

There are many icons associated with each dataset.

1. The eye icon will show the contents of that dataset in the main Galaxy panel.
2. The pencil icon will allow you to edit any attributes associated with the dataset.
3. The X icon deletes the dataset from the history. Note that a dataset is not permanently deleted unless you choose to make it so.
4. The disk icon allows you to download the dataset to your computer.
5. The "i" icon gives you details about how you obtained that dataset (i.e., if you downloaded it from somewhere, or if it is the result of running a job within the Galaxy framework.)
6. The scroll icon allows you to associate tags with the dataset. Similarly to the tags for the history itself, these tags can help to quickly identify particular datasets.
7. The post-it-note icon allows you associate annotations with a dataset. These annotations are also accessible via the Edit Attributes pencil icon.

All datasets belonging to a certain analysis are grouped together within one history. Histories are sharable, meaning that you can allow other users to access a specific history (including all the datasets in it). To make a history accessible to other users:

1. Click the gear icon at the top of the history panel that you want to share.
2. Choose the Share or Publish option.
   a. To make the history accessible via a specific URL, click the Make History Accessible via Link button.
   b. To publish the history on Galaxy's Published Histories section, click the Make History Accessible and Publish option. To see the other histories that have been made accessible at this site, click the Shared Data tab at the top of the page and select Published Histories.

To go back to your current Galaxy history, click the Analyze Data tab at the top of the page. To see a list of all of your current histories, click the gear icon at the top of the history panel and select Saved Histories.

All initial datasets used in this class, as well as prepared mapping runs, are available via two shared histories.
https://main.g2.bx.psu.edu/u/luce/h/workshopdatasets
https://main.g2.bx.psu.edu/u/luce/h/mappingresults

You can import these histories into your own account, using the green "+" icon at the top of the page. You can now either directly start using these histories, or copy any dataset from these histories into any other history you happen to be working with, by using the Copy Datasets command, accessible from the gear icon in the History panel.

Changing dataset formats, editing attributes

If Galaxy does not detect the format of your dataset correctly, you can always change the format of your dataset manually by editing the attributes of the dataset (the pen icon to the upper right of the dataset link in your history.) Specifically, there is a "Change data type" section, where you can select from a pulldown menu the correct format that you want associated with your dataset.

If not already included in the details section of a dataset, it can also be helpful to include other notes about a) where you got the data, b) when you got the data (although the dataset should have a Created attribute, visible from the "view details" icon, the "i"), c) if importing from a database, what query you used to select the data.

Exercises

Update the attributes of the GM12878 dataset that we previously downloaded.

1. Click the pencil icon for the GM12878 dataset.
2. Change the name of the dataset to GM12878 in the Name field.
3. Associate the dataset that we just downloaded with the human hg19 genome in the Database/Build field.
4. Add to the Notes field that we downloaded it from a website. Include the website URL.
5. Click Save.

Examining and Manipulating FastQ data

Quality Scores

The FastQ format provides a simple extension to the FastA format, and stores a simple numeric quality score with each base position. Despite being a "standard" format, FastQ has a number of variants, deriving from different ways of calculating the probability that a base has been called in error, to different ways of encoding that probability in ASCII, using one character per base position.

PHRED scores

Quality scores were originally derived from the PHRED program which was used to read DNA sequence trace files, and linked various sequencing metrics such as peak resolution and shape to known sequence accuracy. The PHRED program assigned quality scores to each base, according to the following formula:

$$Q\_PHRED = -10\ log10\ (Pe)$$

where Pe is the probability of erroneously calling a base. PHRED put all of these quality scores into another file called QUAL (which has a header line as in a FastA file, followed by whitespace-separated integers. The lower the integer, the higher the probability that the base has been called incorrectly.

PHRED Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90 %
20	1 in 100	99 %
30	1 in 1000	99.9 %
40	1 in 10000	99.99 %
50	1 in 100000	99.999 %

While scores of higher than 50 in raw reads are rare, with post-processing (such as read mapping or assembly), scores of as high as 90 are possible.

Quality scores for NGS data are generated in a similar way. Parameters relevant to a particular sequencing chemistry are analyzed for a large empirical data set of known accuracy. The resulting quality score lookup tables are then used to calculate a quality score for *de novo* next-generation sequencing data.

Solexa scores

The Solexa quality scores, which were used in the earlier Illumina pipelines, are calculated differently from the PHRED scores:

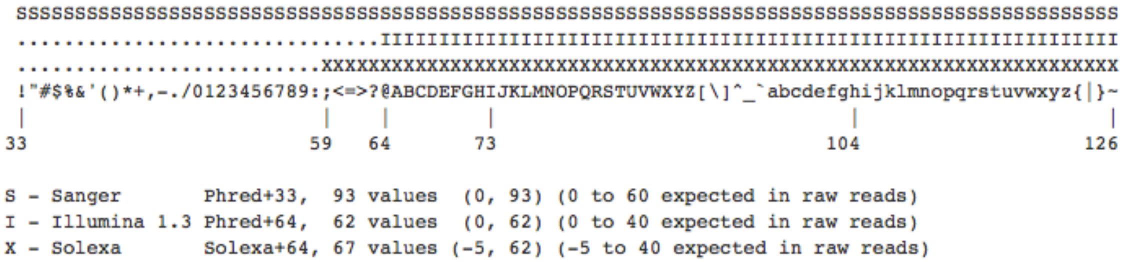
$$Q\_SOLEXA = -10\ log10\ (\frac{Pe}{1-Pe})$$

FastQ Conversion

Changing between FastQ formats

Galaxy is able to interchange between the different FastQ formats with a tool called FASTQ Groomer, found under the NGS: QC and manipulation tab. Since Galaxy tools are designed to work with the Sanger FastQ format, it is advisable to convert any FastQ datasets in another FastQ format to Sanger FastQ. The FASTQ Groomer tool takes as input any dataset designated as FastQ format and converts it according to the equations found in Cock et al. NAR 2009.

Description Galaxy format name	ASCII characters		Quality score	
	Range	Offset	Type	Range
Sanger standard/Illumina 1.7+ fastqsanger	33 to 126	33	PHRED	0 to 93
Solexa/early Illumina fastqsolexa	59 to 126	64	Solexa	-5 to 62
Illumina 1.3+ fastqillumina	64 to 126	64	PHRED	0 to 62



```
 SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
 ...............................IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
 ..........................XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
 |                         |    |        |                              |                     |
33                        59   64       73                            104                   126

S - Sanger        Phred+33,  93 values  (0, 93) (0 to 60 expected in raw reads)
I - Illumina 1.3 Phred+64,  62 values  (0, 62) (0 to 40 expected in raw reads)
X - Solexa        Solexa+64, 67 values (-5, 62) (-5 to 40 expected in raw reads)
```

FastQ Quality Control

There are two standard ways of examining FastQ data in Galaxy: using the Summary Statistics method and the FastQC method.

Summarizing and visualizing statistics about FastQ reads

A very important tool that Galaxy provides for FastQ dataset is the NGS: QC and manipulation → FASTQ Summary Statistics tool. For every column in a set of sequences, this tool will calculate the minimum, maximum, mean, median and first and third quartile quality scores, as well as an overall count of each type of base found for that column. This tool is especially useful for determining at which base sequences should be trimmed so that only high quality sequence is used in your NGS analysis.

The output from the Summary Statistics tool is designed to be used as input to the Graph/Display Data → Boxplot tool. This tool creates a boxplot graph from tabular data. For our purposes, its main function is to visualize the statistics from the Summary Statistics tool. Much of the output from the summary statistics tool is not used by the boxplot tool, since to draw a boxplot, you only need to specify the median, first and third quartiles, whiskers and outliers (if any). The output will be a PNG image viewed in GnuPlot.

Exercise

Run a quality control on the GM12878 dataset that we previously downloaded.

1. Open the NGS: QC and manipulation → FASTQ Summary Statistics tool. Make sure the GM12878 dataset is selected.
2. Click Execute.
3. Open the Graph/Display Data → Boxplot tool. Make sure the input dataset is the output from the Summary Statistics tool in the last step.
4. Change the X-axis label to "read position" and the Y-axis label to "quality score".
5. Click Execute.

FastQC quality control

Another way of looking at your data to determine the overall quality of your run and to warn you of any potential problems before beginning your analysis is to use the NGS: QC and manipulation → FastQC tool, from Babraham Bioinformatics. It takes as input either a FastQ dataset, or BAM or SAM datasets. FastQC bundles into one executable what many of the individual tools available in Galaxy do for specific FastQ formats.

Basic Statistics

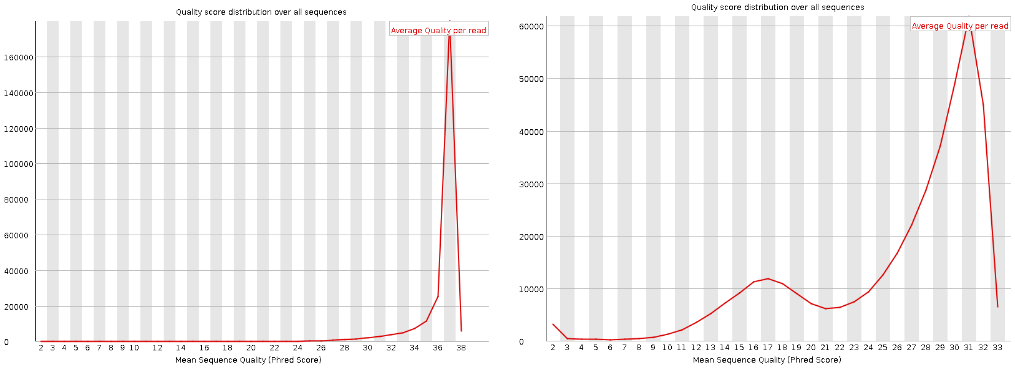
This section gives some simple composition statistics for the input dataset, including filename, filetype (base calls or colorspace), encoding (which FastQ format), total number of sequences, sequence length and %GC.

Per Base Sequence Quality

This plot shows the range of quality values over all bases at each position (similar to the boxplot from the BoxPlot tool.)

Per Sequence Quality Scores

This plot allows you to see if there is a subset of your sequences which has universally low scores (perhaps due to poor imaging). These should represent only a small number of the total sequences. See for comparison the "good" dataset and the "bad" dataset, below. All of the reads in the "good" dataset have a mean quality score of 37; the "bad" dataset has 10,000+ reads with a mean quality score of around 17.



(from http://www.bioinformatics.bbsrc.ac.uk/projects/fastqc/)

Per Base Sequence Content

This plot shows the proportion of each base at each position in the read. In a random library, you would expect that all frequencies are approximately equal at all positions, over all sequences. If you see strong biases which change for different bases then this usually indicates an overrepresented sequence which is contaminating your library. A bias which is consistent across all bases either indicates that the original library was sequence biased, or that there was a systematic problem during the sequencing of the library.

Per Base GC Content

This plots shows the GC content of each position in the run. In a random library, there should be minimal difference between positions, and the overall GC content should reflect the GC content of the genome under study. As in the Per Base Sequence Content plot, deviations across all positions could indicate an over-represented sequence. Similarly, if a bias is consistent across all bases, this could either indicate that the original library was sequence biased, or that there was a systematic problem during the sequencing of the library.

Per Sequence GC content

This plots the GC content across each sequence and compares it to a modeled GC content plot. In a random library, the plot should look normal and the peak should correspond to the overall GC of the genome under study. A non-normal distribution may indicate a contaminated library or biased subset.

Per Base N Content

This plots the percentage of base calls that were Ns (i.e., a base call could not be made with certainty) at every position. Ns more commonly appear towards the end of a run.

Sequence Length Distribution

This plots the distribution of all read lengths found.

Duplicate Sequences

This counts the number of times any sequence appears in the dataset and shows a plot of the relative number of sequences with different degrees of duplication. In a diverse library most sequences will occur only once in the final set. A low level of duplication may indicate a very high level of coverage of the target sequence, but a high level of duplication is more likely to indicate some kind of enrichment bias (e.g., PCR over-amplification).

Overrepresented Sequences

This creates a list of all the sequences which make up more than 0.1% of the total. A normal high-throughput library will contain a diverse set of sequences. Finding that a single sequence is very over-represented in the set either means that it is highly biologically significant, or indicates that the library is contaminated, or not as diverse as you expected.

Overrepresented Kmers

This counts the enrichment of every 5-mer within the sequence library. It calculates an expected level at which this k-mer should have been seen based on the base content of the library as a whole and then uses the actual count to calculate an observed/expected ratio for that k-mer. In addition to reporting a list of hits it will draw a graph for the top 6 hits to show the pattern of enrichment of that k-mer across the length of your reads. This will show if you have a general enrichment, or if there is a pattern of bias at different points over your read length.

Exercise

Run a quality control on the GM12878 dataset using the FastQC tool.

1. Open the NGS: QC and manipulation → FastQC tool.
2. Select the GM12878 dataset.
3. Click Execute.

FastQ Manipulation

Trimming the ends off reads

In general, we hope that most of the bases at the start of a run are of predominantly high quality. If, as the run progresses, the overall quality of the reads decreases, it may be wise to trim the reads before engaging in downstream analysis.

NGS: QC and manipulation → FastQ Trimmer [under Generic FASTQ manipulation]

This tool trims 3' and 5' ends from each read in a dataset. This is especially useful with Illumina data which can be of poorer quality towards the 3' end of a set of reads. For fixed-length reads such as Illumina and SOLiD data, the base offsets should be defined by the absolute number of bases that you want to take off either end, whereas for variable length reads like 454, the number of bases to be trimmed off the end is defined by the percentage of the entire length. Foe example, to take the last 20 bases off the end of each read, the offset from the 3' end is changed to 20.

Removing individual sequences

It is possible that you may want to get rid of some reads which contain one or more bases of low quality within the read. This is done using the Filter FastQ tool

NGS: QC and manipulation → Filter FastQ [under Generic FASTQ manipulation]

This tool allows the user to filter out reads that have some number of low quality bases and return only those reads of the highest quality. For example, if we wanted to remove from our dataset all reads where the quality of any base was less than 20, we change the Minimum quality box to 20. The "Maximum number of bases allowed outside of quality range" allows us to select how many bases per read must be below our threshold before we discard that read. If we leave this at the default setting of 0, all bases in a read must pass the threshold minimum quality score to be kept.

More complex manipulations

The Manipulate FastQ tool in Galaxy also allows much more complex manipulation of FastQ data, whether manipulating the read names, the sequence content or the quality score content. This tool also allows the removal of reads that match some given criteria.

NGS: QC and manipulation → Manipulate FASTQ [under Generic FASTQ manipulation]

This tool can work on all, or a subset of, reads in a dataset. The subset is selected using regular expressions on either the read name, sequence content, or quality score content. If no 'Match Reads' filter is added, the manipulation is performed on all reads. One of the more common manipulations used from this suite is the DNA to RNA manipulation on sequence content, which will have the effect of replacing all Ts in a read with Us.

Exercises

The GM12878 dataset seems to be of poor quality.

1. Trim the reads in the GM12878 dataset using the Generic FASTQ manipulation → FastQ Trimmer tool. Determine from the boxplot and FastQC figures where the quality of the reads begins to drop off sharply. Calculate how many bases have to be trimmed from the end and use that number as the Offset from 3' end.
2. Using the Generic FASTQ manipulation → Filter FastQ tool, filter out all sequences with any bases that have a quality less than 20. How many sequences do you have left in your dataset?
3. Run another QC (summary statistics and boxplot) on both of the new datasets from steps 1 and 2.
4. Modify the trimmed dataset from step 1 so that all Thymines are replaced by Uracils.

Operating on tabular/interval data

One of the advantages of Galaxy is that it makes available many database operations quite transparently. In a database setting, we would use those operations for "joining" tables on a common field, but another common usage is to join information from tables by finding overlapping genomic segments. In Galaxy, tables that include genomic intervals are said to be in interval format.

Joining two tables with interval data

In this exercise we will upload some TAF1 sites that have been detected using a ChIP-Seq experiment, and try to identify the genes that are associated with these sites.

1. Create a new history by clicking the gear icon and selecting Create New.
2. Name your new history to Simple database operations.
3. Upload the file from the following URL by using the Get Data → Upload File tool and entering http://galaxy.psu.edu/CPMB/TAF1_ChIP.txt into the URL/text entry box.
4. Click Execute.
5. Once the dataset has been downloaded, we can take a look at it. It is a tabular dataset where each line represents a location of a predicted TAF1 binding site in the genome. For this information to be meaningful, we need to tell Galaxy which genome build these genomic locations are referring to. In this case, this dataset was generated using the hg18 dataset.
    a. Change the attributes of the dataset by clicking the pencil icon.
    b. Move the URL from the Name field to the Annotation/Notes field.
    c. Change the name of the dataset to something manageable, like 'TAF1 Sites'.
    d. Change the Database/Build field to hg18 (rather than trying to scroll through the list in the pulldown menu, click in the pulldown menu box and start typing hg18; the list of options in the pulldown menu matching this keyword are quickly narrowed).
    e. Change the data type of the dataset (from the Datatype tab) from tabular to interval to tell Galaxy that this is a specific type of tabular data that contains genomic intervals.
6. Click on Save. Genomic interval data is defined by having a chromosome location, and start and stop positions. Galaxy will attempt to figure out which columns correspond to these data. In this case, the chromosome column is 2, the start column is 3, the end column is 4, and the name is located in column 5.

We now need to get the gene annotations from UCSC.

1. Open the Get Data → UCSC Main tool.
2. Change the assembly to March 2006 (hg18) in the assembly pull down menu.
3. Select RefSeq genes as the track to be queried.
4. Make sure the output format is set to BED and the Send Output to Galaxy box is checked.
5. Click 'get output'.
6. On the following page, make sure that you are downloading one BED record per whole gene.
7. Click the Send query to Galaxy button.
8. Use the Edit attributes function to rename the dataset to 'RefSeq genes', once this dataset of RefSeq genes has finished downloading.

Next we want to get a set of putative promoter sequences. Here, we are going to use the upstream sequence as a simple definition of a promoter sequence. (Note that, although in this example we are using a Galaxy tool to accomplish this, we could also have done this through the UCSC Main interface.)

1. Open the Operate on Genomic Intervals → Get Flanks tool to extract the upstream region of the RefSeq genes.
2. Change the length of the flanking region to 1000.
3. Click Execute.
4. Change the name of the resultant dataset to RefSeq promoters.

Now we will use the database join operation to join the original TAF1 binding site dataset with the promoter regions of the RefSeq genes.

1. Open the Operate on Genomic Intervals → Join tool. Note that this tool can only be used with interval datasets.
2. Select the promoters dataset as the first dataset and the TAF1 sites dataset as the second dataset to be joined.
3. Click Execute.

This operation returns a dataset of all those records where the genomic interval overlaps by at least one base, one pair per line. The UCSC "promoter" dataset contains a lot of extra exon information, so you will have to scroll all the way to the right to see the information about the TAF1 site that is being associated with each gene promoter region.

We can tidy this table up by selecting only a handful of columns to display in our final dataset.

1. Open the Text Manipulation → Cut tool.
2. Select c1,c2,c3,c4,c6,c15,c16,c17 as the columns to be cut. This will select only the columns with the chromosome information, start and stop positions of the promoter, the gene name, the strand the gene is coded on, the start and stop positions of the TAF1 site and the TAF1 site name.

Performing calculations on the result of interval data joining

The next example is slightly more complicated. Let's say we want to list all the exons on chromosome 22, sorted by the number of single nucleotide polymorphisms they contain. What genes do the five exons with the highest number of SNPs come from?

First, create a new history. Call it "SNPs in coding exons".

Next retrieve all coding exons from UCSC.

1. Use Get Data → UCSC Main tool.
2. Change the position to search to chr22.
3. Click the get Output button.
4. On the output format screen, change it to one BED record per coding exon.
5. Click Send query to Galaxy.
6. Rename this exon dataset to "Exons_22".

Now retrieve the SNP data, again from UCSC.

1. Use Get Data → UCSC Main tool.
2. This time, change the group to Variation and Repeats.
3. Change the position to chr22 again.
4. On the output format page, the one BED record per gene should again be selected. In this case, "gene" is actually "SNP" (or "feature").
5. Click the Send query to Galaxy button.
6. Rename this dataset to "SNPs_22".

To find the exon with the highest number of SNPs, we must first associate the SNPs with their exons. We do this using the database join command, which will associate with an exon any SNP whose given genomic interval overlaps that of any exon by at least one base (in this case, each SNP location is only one base in length).

1. Open the Operate on Genomics Intervals → Join tool.
2. Use the exons dataset as the first dataset and the SNPs dataset as the second dataset.
3. Click Execute.

Each line in this new dataset is the "join" of an exon and a SNP that is found within that exon (more exactly, whose genomic region overlaps that of the corresponding exon). The first six columns are those associated with the exon, the next six are those associated with the SNP.

Next, we want to count the number of SNPs associated with each exon.

1. Open the Join, Subtract, and Group → Group tool.
2. Change the Group by Column to c4 (i.e., the exon name).
3. Add a new operation.
4. Change the type of the new operation to Count and the column to c4. This groups the dataset on the exon name, and count how many exons of the same name are in each unique group.

The resulting dataset will have two columns: one is a list of all the distinct exon names, and the other is the number of times that exon name was associated with a SNP. We can now rearrange this dataset such that the exons with the highest number of SNPs are at the top.

1. Open the Filter and Sort → Sort tool.
2. Sort the dataset by column 2 (SNP count), in descending order (i.e., highest first).

We can see that the first exon (cds_0) of gene uc003bhh.3 has the highest number of SNPs with 26.

Finally, we want to select the five exons with the most SNPs.

1. Open the Text Manipulation → Select First tool.
2. Make sure the sorted dataset is selected.
3. Change the number of lines to select to 5.
4. Click Execute.

This dataset will simply be the first 5 lines of the previous dataset.

If we want to view these five exons in a genome browser, we will have to get their genomic coordinates, which we lost when we grouped the data. However, we still have this information in our original dataset, and we can re-associate this information with our exons of interest.

1. Open the Join, Subtract and Group → Compare two Datasets tool.
2. Choose c4 from our Exons_22 dataset and c1 from our final set of five exons (where c4 and c1 are the exon name column in each respective dataset).
3. Click Execute.

This extracts from our Exons_22 dataset any lines which match the exon name from our final dataset.
To visualize these regions, we can click on any of the visualization tools listed for that dataset (UCSC, GeneTrack, Ensembl or RViewer).

Converting to interval format

Not all datasets will be in interval format, and will have to be converted before they can be used in analyses like the above.

Download the variations from the ENSEMBL database for chromosome 22. We can access it through the BioMart portal in Galaxy.

1. Open the Get Data → BioMart tool. This will take us away from the Galaxy page and to the query interface to the BioMart databases.
2. Choose the Ensembl Variation 69 → Homo sapiens Somatic Variation (GRCh37.p8) database.
3. Click the Filters link. This is the section where you restrict the entries that will be retrieved.
    a. Open the Region section.
    b. In the chromosome pulldown menu, choose 22. The chromosome checkbox should be automatically checked as soon as you choose a chromosome.
4. Click the Attributes link. This is the section where you determine the fields that will be shown in the output.
    a. Check the Variation ID, Chromosome name and Position on Chromosome (bp) checkboxes, if they are not already checked.
5. Click the Results tab. The first 10 results will be shown.
6. In the Export results section, make sure that Galaxy and TSV are selected and click the Go button.

You will be redirected back to Galaxy and the dataset will be imported to your current history.

If for some reason the results are shown in HTML format, they will not be imported correctly into Galaxy. If this happens, reload the page and reselect the Results tab.

The variation data that we imported from BioMart is in tabular format, with the position of the variation being noted by a single column. The interval data format expects a start and an end position for every feature. To convert the tabular data to interval format:

1. Open the Text Manipulation → Compute tool.
2. Add the expression c3+1 as a new column to the current dataset, where c3 is our original position column.
3. Set Round Result to Yes to ensure that the new position column is an integer.
4. Click Execute.

This will add a new final column to the dataset. We now have to indicate to Galaxy that this dataset is in interval format, which we can do by clicking the pencil icon and editing the attributes.

1. Change the data type to interval.
2. Click Save.
3. Change the corresponding columns to name, chromosome, start and end. In this case, the name is in column 1, the chromosome is in column 2, the start in 3 and our newly computed end position is in column 4.
4. Click Save.

In the event that we had not included the chromosome field when importing from BioMart, we could add a chromosome column using the Text Manipulation → Add Column tool to add a new column, fully populated with a single number, the chromosome of interest.

Workflows

Workflows are great a great method to redo analyses automatically and simply on many datasets. They are also a good way to ensure reproducibility, especially if you share the workflows with your colleagues.

Extract Genomic Sequence Workflow

In this exercise, we are going to extract the genomic sequence flanking all the SNPs in the rhodopsin gene and then learn how to package those steps into a reusable workflow.

We will upload the data using the database query method.

1. Create a new history.
2. Open the Get Data → UCSC Main tool, which will bring up the query interface for the UCSC Genome Browser.
3. Making sure that the Mammal, Human and Feb 2009 options are selected, choose the Variation and Repeats dataset, which should change the track to the most recent SNP dataset (135).
4. Change the region radio button to position and enter uc003emt.3 in the textbox. Click the lookup button to automatically convert this to the assocaiated genomic region (uc003emt.3 is rhodopsin). Make sure that the position radio button is still checked.
5. Make sure the output format is set at BED and that the Send to Galaxy checkbox is checked.
6. Click the Get Output button. This takes you to a page which the information to be contained in the output. We want one BED record per gene which, in this case, is actually one record per SNP.
7. Click the Send query to Galaxy button.

Once the data has been downloaded, the history item will turn green. We can now edit the attributes for this dataset by clicking the pencil icon. Because this dataset is in BED format, it assumes that these are genomic regions, and has labeled the columns accordingly. In general, it is a good idea to make some notes in the Info field about where you got the information from (although some of this should automatically have been filled in the Database/Build field). We can also change the name of the dataset to something simpler, such as 'SNP locations'. Once you have entered your changes, click Save.

Now we want to get 50 bases of flanking sequence around each of these SNPs.

1. Select the Operate on Genomic Intervals → Get flanks tool.
2. Change the Location of the flanking regions to both (since we want flanking sequence on both sides of the SNP), and leave all other values as default.
3. Click Execute.
4. Change the annotations for this new dataset, by clicking the pencil button associated with this operation in the history. Change the name to 'SNP flank regions'.

This dataset looks very similar to our initial dataset, but has twice the number of lines: one for each 50 base flank on either side of every SNP.

Now we get the DNA sequence of each of these flanking regions.

1. Open the Fetch Sequences → Extract Genomic DNA tool.
2. Making sure the second dataset is selected, change the Output data type to interval, which returns a tab delimited dataset with the sequence, as well as chromosome, start and stop positions and associated SNPs.
3. Change the name of this dataset to 'Flanking Genomic Sequence' by editing the attributes with the pencil icon.

To make this series of steps into a reusable workflow, click the gear icon at the top of the history window and choose Extract Workflow. There should be three steps; ensure that all are checked for inclusion in the workflow. Rename the workflow to 'Retrieve flanking SNP sequence' and click Create Workflow. This workflow has now been added to the list of your workflows at the bottom of the leftmost column.

To use this workflow with another gene of interest, let us get some SNP data for another gene, say p53.

1. Create a new history by clicking on the gear icon and selecting Create New.
2. Go to the Get Data → UCSC Main tool.
3. Change the group to Variation and Repeats, input uc010vug.2 (p53) into the text entry box, and click lookup.
4. Making sure that the output is set at BED and is being sent to Galaxy, click the Get Output button.
5. On the following page, click the Send query to Galaxy button.
6. Once the dataset has been downloaded, change the name in the attributes panel by clicking on the pencil icon.

To run the workflow with this new dataset, choose the Workflow tab from the top of the screen. Locate your newly created workflow, and choose Run from the pulldown menu. The steps in the workflow are now listed. Step 1 requires you to input a dataset. Choose the p53 SNPs dataset that we just downloaded from UCSC, and click Run Workflow. The workflow will automatically run both the step that defined the flanking regions for each SNP, as well as the retrieval of the genomic sequence. Once the workflow is complete, you can click the eye icon of the final dataset in your history to view the interval-formatted genomic sequence for the flanking sequence for the 46 SNPs found in p53 (i.e., 92 flanking regions).

Filtering Exons by Feature Count Workflow

For an even more complicated workflow, let's use the "SNPs in coding exons" example from the previous section. Go to the coding exons history by finding it in your Saved Histories list. Convert that history to a workflow, by choosing Extract Workflow from the list of options accessible by clicking the gear icon. Rename the workflow to Coding Exon SNPs and click Create Workflow. Your workflow is now accessible from the Workflows tab at the top of the page.

We can now edit this workflow by clicking on its name and choosing Edit from the pulldown menu. Clicking on any of the boxes in this figure will bring up the list of options associated with that tool. There are a number of ways to use and edit workflows, but one of the most useful is the ability to hide certain intermediate steps so that when you re-use the workflow, you don't end up with multiple intermediate datasets shown in your history when all you really want is the end result. If we want to hide all intermediate steps except the last one, click the asterisk in the lower right corner of the box for the final step. Note that as soon as we do this, it turns darker orange in the overview window. We can also ensure that any datasets are named in a sensible manner within the editor. For example, we can rename each input dataset to indicate what kind of data that dataset should contain. In this case, we can rename one input dataset as Exons, and the other as SNPs, by simply clicking on the box representing that dataset and changing the name in the right hand panel. We can also rename the final dataset by clicking on its representation and choosing Rename Dataset in the Edit Step Actions section, and clicking the Create button. This opens a small text entry box where we can enter what we would like the dataset resulting from this workflow to be called. Call it Top 5 Exons. Within this editor, you can also change the value of any parameter in a tool. Once our changes are complete, choose Save from the menu (the cog) at the top right of the editor.

We can now run this new workflow.

1. Create a new history.
    a. Click the Analyze Data tab.
    b. Choose Create New from the gear icon.
2. Before we run the workflow, we need to download some data.
3. Retrieve the coding exons from chromosome 21 (remember to choose one BED record per coding exon) from the Get Data → UCSC Main tool.
4. Retrieve the common SNPs for chromosome 21.
5. Once the two datasets are downloaded, rename them to shorter names.
6. Start the workflow.
    a. Go to the Workflow menu.
    b. Select the Coding Exon SNPs workflow.
    c. Choose Run.
7. Choose the Exons dataset as the first dataset, the SNPs dataset as the second.
8. Click Run Workflow.

While waiting to be run, each dataset will be shown in the history, whether we marked it to be hidden or not. Once an operation is finished, if it was marked to be hidden, it will disappear from the history pane. When the workflow has finished running, we will be left with only the dataset that we did not hide, i.e., the list of the five exons with the highest number of SNPs.

Mapping Illumina data with BWA

Next Generation Sequencing

Next gen sequencing experiments result in millions of relatively small reads that must be mapped to a reference genome. Since using the standard alignment algorithms is unfeasible for such large numbers of reads, a lot of effort has been put into developing methods which are fast and are relatively memory-efficient.

Mappers

There are two mappers available in Galaxy: Bowtie and BWA. The crucial difference between these mappers is that BWA performs gapped alignments, whereas Bowtie does not (although there is a version of Bowtie available which does perform gapped alignments, it is not the one available in Galaxy). This, therefore, gives BWA greater power to detect indels and SNPs. Bowtie tends to be much faster, and have a smaller memory footprint, than BWA. BWA is generally used for DNA projects, whereas Bowtie is used for RNA-Seq projects since the exonic aligner TopHat uses Bowtie to do the initial mapping of reads.

Aligning reads with BWA [Burrows-Wheeler Alignment]

Create a new history by clicking the gear icon and selecting Create New. We will be working with the Blood-PCR1 dataset from the mtProjectDemo library. This is a dataset derived from the blood of a single individual, which has been enriched for mitochondrial sequence using PCR. Each eukaryotic cell contains many hundreds of mitochondria with many copies of mtDNA. Heteroplasmy is the (not uncommon) presence of multiple mtDNA variants within a single individual. We are going to search this dataset for heteroplasmic sites. This will use a similar methodology as if we were looking for SNPs, although the frequency of heteroplasmic sites will be much lower than would be seen for SNPs.

1. Click the Shared Data tab.
2. Choosing Data Libraries
3. Clicking on the mtProjectDemo link.
4. Check the box beside Blood-PCR1.
5. Make sure the Import to current history option is selected.
6. Click Go.

Return to your Galaxy analysis page by clicking the Analyze Data tab. As usual, with a new dataset:

1. Do a quality control check using the NGS: QC and manipulation → FASTQ Summary Statistics tool.
2. Click Execute.
3. Visualize these data using the Graph/Display Data → Boxplot tool.
4. Inspect the resulting graph by clicking on the eye icon associated with the result of this tool. We can see that the read length is 76 bases, and the quality ranges from 35 at the 5’ end to 18 at the 3’ end.

Since we will be using these reads to look for heteroplasmic sites, we will map these reads to the human genome using BWA. This action will take some time to complete, as we are mapping 500,000 reads to the complete human genome.

1. Select the NGS: Mapping → Map with BWA for Illumina tool.
2. Choose hg19 Full as the reference genome.
3. Click Execute.

Understanding some of the options for BWA

BWA in Galaxy is designed for short queries up to ~200bp with low error rate (<3%). It performs gapped global alignment with respect to reads, supports paired-end reads, and also visits suboptimal hits. This is probably the most widely used and least understood mapping algorithm.

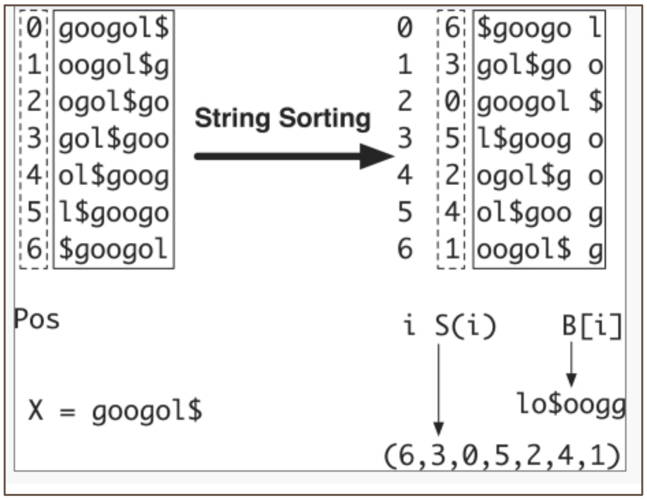
BWA is based on the Burrows-Wheeler transform, a reversible string transformation. This is a way of matching strings that has a small memory footprint and is able to count the number of matches to a string independent of the size of the genome.

The first step is to transform the genome. This is done via the index algorithm in BWA, and will usually take a few hours, but is already done in Galaxy for all the major genomes.

The Burrows-Wheeler transform essentially:

1. Adds a symbol to the end of the string to be transformed, which is lexicographically smaller than all the other symbols in the string.
2. Generates a list of strings, the same length as the original string (with added symbol), but where each letter is circularly moved forward one step.
3. Lexicographically sorts the generated strings.
4. We end up with four different variables to store:
    a. A suffix array for that string which lists the original indices of each of the sorted strings;
    b. The BWT string, made up of the last symbols of each of the newly ordered circulated strings;
    c. An indexed first column, where we have the start and end index of any given letter in the alphabet;
    d. A rank for each letter at each position, which tells us the number of occurrences of that letter above that row in the BWT.

Now, if we are trying to match a new string that is a substring of the original string, each occurrence of that substring will occur within an interval of the suffix array, because all the circulated strings that begin with that substring will have been sorted together. Once we find the suffix interval, we can deduce, from the suffix array, the position(s) of that substring in the original string.



Li and Durbin, Bioinformatics. 2009 Jul 15;25(14):1754-60. "Fast and accurate short read alignment with Burrows-Wheeler transform."

For exact matching, the interval is found by working backwards on your query string:

1. Find the start and end indices of the last character of your query in the indexed first column.
2. At the extremes of this range, find the rank for the next letter in the query string in the BWT.
3. Jump to these ranks of the next letter in the indexed first column.
4. Repeat until you have matched your whole query string. The positions of the final range in the index column will give the suffix array range for the query sequence, where the prefix of every row is our query. If at any time the ranks returned are the same, that means that the next character we want to find is not present in this range and our search stops.

For a very nice visual explanation of this algorithm, visit:
http://blog.avadis-ngs.com/2012/04/elegant-exact-string-match-using-bwt-2/

The 'aln' command finds the suffix array (SA) coordinates (i.e., the suffix interval) of good hits of each individual read, and the 'samse/sampe' command converts the SA coordinates to chromosomal coordinates, and pairs reads (for 'sampe') and generates the SAM-formatted alignments.

In the BWA tool in Galaxy, both the aln and samse/sampe commands are run to result in a SAM format output dataset.

For aln [default values]

-n NUM Maximum edit distance if the value is INT, or the fraction of missing alignments given 2% uniform base error rate if FLOAT. In the latter case, the maximum edit distance is automatically chosen for different read lengths. The maximum number of differences allowed is defined as: 15-37 bp reads: 2; 38-63: 3; 64-92: 4; 93-123: 5; 124-156: 6. [0.04]
-o INT Maximum number of gap opens. [1]
-e INT Maximum number of gap extensions, -1 for k-difference mode (disallowing long gaps) [-1]. This option is critical in allowing the discovery of indels.
-d INT Disallow a long deletion within INT bp towards the 3'-end. [16]
-i INT Disallow an indel within INT bp towards the ends. [5]
-l INT Take the first INT subsequence as seed. If INT is larger than the query sequence, seeding will be disabled. For long reads, this option is typically ranged from 25 to 35 for -k 2. [inf]
-k INT Maximum edit distance in the seed. [2]
-M INT Mismatch penalty. BWA will not search for suboptimal hits with a score lower than (bestScore-misMatchPenalty). [3]
-O INT Gap open penalty. [11]
-E INT Gap extension penalty. [4]
-R INT For paired-end reads only. Proceed with suboptimal alignments if there are no more than INT top hits. By default, BWA only searches for suboptimal alignments if the top hit is unique. Using this option has no effect on accuracy for single-end reads. It is mainly designed for improving the alignment accuracy of paired-end reads. However, the pairing procedure will be slowed down, especially for very short reads (~32bp).
-N Disable iterative search. All hits with fewer than the maximum allowed number of differences will be found. This mode is much slower than the default.

For samse/sampe:

-n INT (samse/sampe) Maximum number of alignments to output in the XA tag for reads paired properly. If a read has more than INT hits, the XA tag will not be written. [3] This is another critical parameter, and will determine whether suboptimal matches are returned.
-r STR (samse/sampe) Specify the read group, formatted as "@RG\tID:text\tSM:text". [null]
-a INT (sampe only) Maximum insert size for a read pair to be considered as being mapped properly. This option is only used when there are not enough good alignments to infer the distribution of insert sizes. [500]
-N INT (sampe only) Maximum number of alignments to output in the XA tag for disconcordant read pairs (excluding singletons). If a read has more than INT hits, the XA tag will not be written. [10]
-o INT (sampe only) Maximum occurrences of a read for pairing. A read with more occurrences will be treated as a single-end read. Reducing this parameter helps faster pairing. [100000]

[INT: integer; STR: string]

SAM [Sequence Alignment/Map] Format

The Sequence Alignment/Map (SAM) format is a generic nucleotide alignment format that describes the alignment of query sequences or sequencing reads to a reference sequence. It can store all the information about an alignment that is generated by most alignment programs. Importantly, it is a compact representation of the alignment, and can allow many of the operations on the alignment to be performed without loading the whole alignment into memory. The SAM format also allows the alignment to be indexed by reference sequence position to efficiently retrieve all reads aligning to a locus.

The SAM format consists of a header section and an alignment section.

Header section

The header section includes information about the alignment and the program that generated it. All lines in the header section are tab-delimited and begin with a "@" character, followed by tag:value pairs, where tag is a two-letter string that defines the content and the format of value.

There are five main sections to the header, each of which is optional:
@HD. The header line. If this is present, it must be the first line, and must include:
    VN: the format version.
@SQ. Includes information about the reference sequence(s). If this section is present, it must include two fields for each sequence:
    SN: the reference sequence name.
    LN: the reference sequence length.
@RG. Includes information about read groups. This can be used multiple times, once for each read group. If this section is present, each @RG section must include:
    ID: the read group identifier.
If an RG tag appears anywhere in the alignment section, there should be a single corresponding @RG line with matching ID tag in the header section.
@PG. Includes information about the program generating the alignment. Must include:
    ID: The program identifier.
If a PG tag appears anywhere in the alignment section, there should be a single corresponding @PG line with matching ID tag in the header section.
@CO. These are unstructured one-line comment lines which can be used multiple times.

Alignment section

Each alignment line has 11 mandatory fields and a variable number of optional fields. These fields always appear in the same order and must be present, but their values can be '0' or '*' (depending on the field) if the corresponding information is unavailable.

Mandatory Alignment Section Fields

Position	Field	Description
1	QNAME	Query template (or read) name
2	FLAG	Information about read mapping (see next section)
3	RNAME	Reference sequence name. This should match a @SQ line in the header.
4	POS	1-based leftmost mapping position of the first matching base. Set as 0 for an unmapped read without coordinate.
5	MAPQ	Mapping quality of the alignment. Based on base qualities of the mapped read.
6	CIGAR	Detailed information about the alignment (see relevant section).
7	RNEXT	Used for paired end reads. Reference sequence name of the next read. Set to "=" if the next segment has the same name.
8	PNEXT	Used for paired end reads. Position of the next read.
9	TLEN	Observed template length. Used for paired end reads and is defined by the length of the reference aligned to.
10	SEQ	The sequence of the aligned read.
11	QUAL	ASCII of base quality plus 33 (same as the quality string in the Sanger FASTQ format).
12	OPT	Optional fields (see relevant section).

FLAG field

The FLAG field includes information about the mapping of the individual read. It is a bitwise flag, which is a way of compactly storing multiple logical values as a short series of bits where each of the single bits can be addressed separately.

FLAG fields

Hex	Binary	Description
0x1	00000000001 (1)	The read is paired
0x2	00000000010 (2)	Both reads in a pair are mapped "properly" (i.e., in the correct orientation with respect to one another)
0x4	00000000100 (4)	The read itself is unmapped
0x8	00000001000 (8)	The mate read is unmapped
0x10	00000010000 (16)	The read has been reverse complemented
0x20	00000100000 (32)	The mate read has been reverse complemented
0x40	00001000000 (64)	The read is the first read in a pair
0x80	00010000000 (128)	The read is the second read in a pair
0x100	00100000000 (256)	The alignment is not primary (a read with split matches may have multiple primary alignment records)
0x200	01000000000 (512)	The read fails platform/vendor quality checks
0x400	10000000000 (1024)	PCR or optical duplicate

In a run with single reads, the only flags you will see are:
0 None of the bitwise flags have been set. This read has been mapped to the forward strand.
4 The read is unmapped.
16 The read is mapped to the reverse strand.

Some common flags that you may see in a paired experiment include:

69	1 + 4 + 64	The read is paired, is the first read in the pair, and is unmapped.
73	1 + 8 + 64	The read is paired, is the first read in the pair, and it is mapped while its mate is not.
77	1 + 4 + 8 + 64	The read is paired, is the first read in the pair, but both are unmapped.
133	1 + 4 + 128	The read is paired, is the second read in the pair, and it is unmapped.
137	1 + 8 + 128	The read is paired, is the second read in the pair, and it is mapped while its mate is not.
141	1 + 4 + 8 + 128	The read is paired, is the second read in the pair, but both are unmapped.

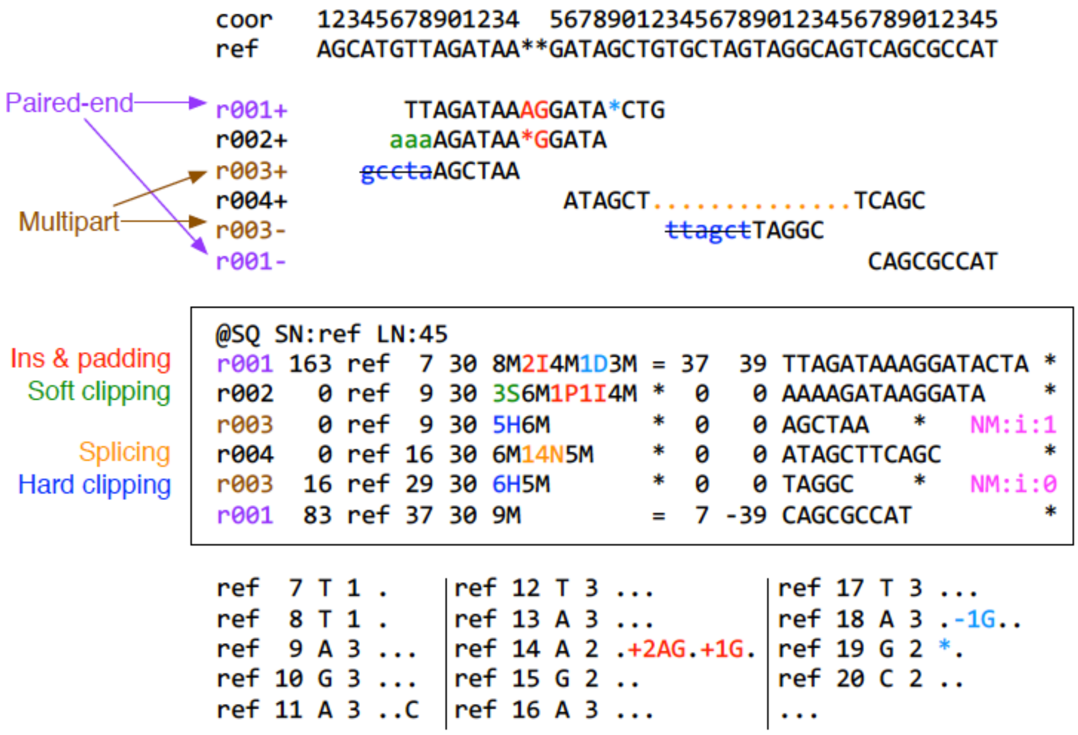
CIGAR [Concise Idiosyncratic Gapped Alignment Report] String

The CIGAR string describes the alignment of the read to the reference sequence. It is able to handle (soft- and hard-) clipped alignments, spliced alignments, multi-part alignments and padded alignments (as well as alignments in color space). The following operations are defined in CIGAR format:

CIGAR Format Operations

Operation	Description
M	Alignment match (can be a sequence match or mismatch)
I	Insertion to the reference
D	Deletion from the reference
N	Skipped region from the reference
S	Soft clipping (clipped sequences present in read)
H	Hard clipping (clipped sequences NOT present in alignment record)
P	Padding (silent deletion from padded reference)
=	Sequence match (not widely used)
X	Sequence mismatch (not widely used)

- H can only be present as the first and/or last operation.
- S may only have H operations between them and the ends of the CIGAR string.
- For mRNA-to-genome alignments, an N operation represents an intron. For other types of alignments, the interpretation of N is not defined.
- The sum of lengths of the M/I/S/=/X operations must equal the length of the read.



```
coor   12345678901234  5678901234567890123456789012345
ref    AGCATGTTAGATAA**GATAGCTGTGCTAGTAGGCAGTCAGCGCCAT

r001+        TTAGATAAAGGATA*CTG
r002+       aaaAGATAA*GGATA
r003+     gcctaAGCTAA
r004+                 ATAGCT..............TCAGC
r003-                        ttagctTAGGC
r001-                                      CAGCGCCAT
```

```
@SQ SN:ref LN:45
r001 163 ref  7 30 8M2I4M1D3M = 37  39 TTAGATAAAGGATACTA *
r002   0 ref  9 30 3S6M1P1I4M *  0   0 AAAAGATAAGGATA    *
r003   0 ref  9 30 5H6M       *  0   0 AGCTAA    *   NM:i:1
r004   0 ref 16 30 6M14N5M    *  0   0 ATAGCTTCAGC       *
r003  16 ref 29 30 6H5M       *  0   0 TAGGC     *   NM:i:0
r001  83 ref 37 30 9M         =  7 -39 CAGCGCCAT         *
```

```
ref  7 T 1 .     |ref 12 T 3 ...        |ref 17 T 3 ...
ref  8 T 1 .     |ref 13 A 3 ...        |ref 18 A 3 .-1G..
ref  9 A 3 ...   |ref 14 A 2 .+2AG.+1G. |ref 19 G 2 *.
ref 10 G 3 ...   |ref 15 G 2 ..         |ref 20 C 2 ..
ref 11 A 3 ..C   |ref 16 A 3 ...        |...
```

Li et al, Bioinformatics (2009) 25 (16): 2078-2079. "The Sequence Alignment/Map format and SAMtools"

OPT field

The optional fields are presented as key-value pairs in the format of TAG:TYPE:VALUE, where TYPE is one of:

- A Printable character
- I Signed 32-bin integer
- F Single-precision float number
- Z Printable string
- H Hex string

The information stored in these optional fields will vary widely with the mapper.

They can be used to store extra information from the platform or aligner. For example, the RG tag keeps the ‘read group’ information for each read, where a read group can be any set of reads that use the same protocol (sample/library/lane). In combination with the @RG header lines, this tag allows each read to be labeled with metadata about its origin, sequencing center and library.

Other commonly used optional tags include:

NM:i	Edit distance to the reference
MD:Z	Number matching positions/mismatching base
AS:i	Alignment score
BC:Z	Barcode sequence
X0:i	Number of best hits
X1:i	Number of suboptimal hits found by BWA
XN:i	Number of ambiguous bases in the reference
XM:i	Number of mismatches in the alignment
XO:i	Number of gap opens
XG:i	Number of gap extensions
XT:A	Type of match (Unique/Repeat/N/Mate-sw)
XA:Z	Alternative hits; format: (chr,pos,CIGAR,NM)
XS:i	Suboptimal alignment score
XF:	Support from forward/reverse alignment
XE:i	Number of supporting seeds

Thus, for example, we can use the NM:i:0 tag to select only those reads which map perfectly to the reference (i.e., have no mismatches). If we wanted to select only those reads which mapped uniquely to the genome, we could filter on the XT:A:U (where the U stands for “unique”).

Mapping example continued

Once the mapping is complete, we want to select only those reads that have mapped uniquely to the genome.
[Note: if your BWA mapping run hasn't finished yet, get a prepared result file from the shared history named MappingResults, using the Copy Datasets function]
Note that the header lines of our output include all the names of the reference sequences that our reads were being mapped to, i.e., every chromosome and its length.

In the final OPT column of the SAM output, BWA includes many additional pieces of information. For example, we can use the NM:i:0 tag to select only those reads which map perfectly to the reference (i.e., have no mismatches). In this case, we want to select only those reads which mapped uniquely to the genome. To do this, we filter on the XT:A:U (where the U stands for "unique").

1. Open the Filter and Sort → Select tool.
2. Input XT:A:U as the pattern to match.

Now that we have a filtered set of results, in SAM format, we want to convert them into BAM format (which is the binary indexed version of the SAM data).

1. Open the NGS SAM Tools → SAM to BAM tool. Make sure the filtered SAM dataset is selected.
2. Click Execute.

We can retrieve some simple statistics on our BAM file:

1. Open the NGS: SAM Tools → flagstat tool. This reads the bitwise flags from the SAM/BAM output and prints out their interpretation.
2. Click Execute.

We can see that 99.91% of our (filtered) reads were mapped to the reference genome.

The SAM and BAM formats are read-specific, in that every line in the file refers to a read. We want to convert the data to a format where every line represents a position in the genome instead, known as a pile-up.

1. Open the NGS SAM Tools → MPileup tool. The MPileup tool is a newer, more complex, version of the pileup tool which can handle multiple BAM files. It also introduces the concept of BAQ (base alignment quality) which takes into account local realignment of reads around putative SNP positions and will modify base qualities around these positions in an attempt to avoid calling false SNPs.
2. Set the reference genome to hg19.
3. Select the Advanced Options.
4. Change the coefficient for downgrading mapping quality for reads containing excessive mismatches to 50. This reduces the effect of reads with excessive mismatches and is a fix for overestimated mapping quality.
5. Click Execute.

Pileup Format

Each line in the pileup format includes information on:

1. The reference sequence (chromosome) name.
2. The position in the reference sequence.
3. The reference base at that position (uppercase indicates the positive strand; lowercase the negative strand). An asterisk marks an indel at that position.
4. The number of reads covering that position.
5. The read base at that position. This column also includes information about whether the reads match the reference position or not. A "." stands for a match to the reference base on the forward strand, a "," for a match on the reverse strand, "[ACGTN]" for a mismatch on the forward strand and "[acgtn]" for a mismatch on the reverse strand. A pattern "\[+-][0-9]+[ACGTNacgtn]+" indicates there is an insertion (+) or deletion (-) between this reference position and the next reference position. Finally, a "^" indicates the start of a new, or soft- or hard-clipped read, followed by the mapping quality of the read. The "$" symbol marks the end of a read segment.
6. The quality scores for each read covering that position.

For more detailed information on the pileup format, go to http://samtools.sourceforge.net/pileup.shtml

Analyze pileup

Since this is a heteroplasmic experiment, we reduce this dataset to ensure we only include the mitochondrial genome.

1. Use the Filter and Sort → Filter tool.
2. Set the search condition to c1 == 'chrM'. Make sure to include the quotes around the search term.

To see how well the various positions in the mitochondrial genome are covered

1. Open the Statistics → Summary Statistics tool.
2. Choose c4. This is the fourth column from the pileup (i.e., the column which shows the number of reads covering that position.)

In this case, we can see that the coverage is quite high (the mean is 1856 reads per base).

To extract positions that are likely to be heteroplasmic, we should include two pieces of information: the coverage (only keep those positions above a certain threshold of coverage) and the quality (only keep those base reads above a certain quality threshold, as they are more likely to be real heteroplasmic sites rather than sequencing errors). To filter out only those positions which pass these thresholds:

1. Open the NGS: SAM Tools → Filter pileup tool.
2. Make sure that the dataset that you are working on is the filtered pileup, not the summary statistics dataset.
3. Require that a base call must have a quality of at 30 to be considered.
4. Remove all bases covered by fewer than 200 reads.

5. Change the pulldown menus so that:
    a. the tool reports variants only;
    b. coordinates are converted to intervals (this makes it easier to join the results of this analysis with gene/promoter annotations);
    c. the total number of differences is reported;
    d. the quality and base strings are not returned (since these will be very large fields).

We can now remove any positions whose quality adjusted coverage does not meet our threshold of 200 using a secondary filtering step.

1. Open the Filter and Sort → Filter tool.
2. Use "c10>=200" as the filter condition.

We may also want to perform additional filtering steps, such as only keeping those sites which show at least a 0.15% frequency.

1. Open the Text Manipulation → Compute tool.
2. Enter (c11/c10) * 100.0 into the expression box. This will calculate the percentage of the quality adjusted coverage at each base that is represented by the total number of variants.
3. Do not round the result; keep the calculation as a floating point number.
4. Click Execute.
5. Open the Filter and Sort → Filter tool.
6. Enter c12 > 0.15 as the condition to filter on.
7. Click Execute.
8. Open the Filter and Sort → Sort tool.
9. Sort on c12 (the percentage column).
10. Sort in descending order.
11. Click Execute.

Note that this is a very simplistic way of filtering as it ignores the fact that some of the variant positions differ from the reference in every read (and will be seen here as 100% different) but show no heteroplasmy amongst themselves. Another possibility may be to filter out those sites which only have one difference from the reference.

Exercise

1. Make a workflow of this history. Leave out the summary statistics and boxplot steps. Name the workflow Mapping BWA Reads. Make sure all the steps connect correctly to one another.
Import the Blood-PCR2 dataset from the mtProjectDemo library into your history.
Rerun the workflow with this new dataset. (Run a quality control check on this dataset before running the workflow).

2. Associate the resulting heteroplasmies with gene / transcript / exon / intron information.

Filtering Reads by Mapping Quality Workflow

We can build a workflow that incorporates filtering on the SAM output.
Upload some PhiX reads and a PhiX genome from Emory using the following URLs:
http://bx.mathcs.emory.edu/outgoing/data/phiX174_genome.fa
http://bx.mathcs.emory.edu/outgoing/data/phiX174_reads.fastqsanger

1. Use the Get Data → Upload File tool.
2. Enter the URLs into the URL/Text entry box.
3. Click Execute.

The PhiX genome has been successfully identified as a FastA file. Change the name of this dataset to PhiX Genome. The reads have been identified as being in the FastQ format, but we need to specify which FastQ format. In the attributes panel, accessed by clicking the pencil icon for that dataset, change the name of this dataset to PhiX reads, and change the data type to fastqsanger.

We next align the reads to a PhiX genome. Although Galaxy has a built-in PhiX genome, we will use the one that we downloaded from Emory.

1. Select the NGS: Mapping → Map with BWA for Illumina tool.
2. Change the reference genome pulldown menu to say Use one from the history.
3. Make sure that the PhiX Genome dataset is selected in the second pulldown menu, and that the PhiX reads are selected as the FastQ file.
4. For this example, leave the settings at the default option.

This will generate a set of reads aligned to the PhiX genome in SAM format.

We can now mine the SAM data however we wish. Say we want to only select those reads that mapped perfectly to the genome. One of the optional pieces of information that is output by the BWA program is the edit distance to the reference (NM:i:x). If we want to select only those reads which matched the reference exactly, the number at the end of that tag should be zero.

1. Open the Filter and Sort → Select tool.
2. Input NM:i:0 as the string to be matched.
3. Click Execute.

Extract this workflow using the Extract Workflow option from the gear icon. Rename it to Subset Reads and save. Select the workflow from the Workflows tab. Clicking on the representation of the Select tool allows you to change the pattern that is being matched (for example, if we wanted to change it to select reads that were an edit distance of 1 away from the reference, we could change the pattern to NM:i:1. You can change the name of the output from this Select operation by choosing the Rename Dataset from the pulldown menu in the Edit Step Actions section and clicking Create. Make sure to save the updated workflow from the Options menu at the top right of the workflow editor.

RNA-Seq analysis with TopHat tools

RNA-Seq

RNA-Seq experiments are designed to identify the RNA content (or transcriptome) of a sample directly. These experiments allow the identification of relative levels of alleles, as well as detection of post-transcriptional mutations or detection of fusion genes. Most importantly, the RNA-Seq technique allows the comparison of the transcriptomes of different samples, most usually between tumor and normal tissue, to enable insight into the differential expression patterns seen in each state. RNA-Seq is the next generation version of other experimental techniques for describing transcriptomes, such as microarrays or EST sequencing, and can do so with fewer biases and at a higher resolution.

The alignment of RNA-Seq read data to a genome is complicated by the fact that the reads come from spliced transcripts and therefore there will be many intronic regions to deal with (i.e., regions that are present in the genome being aligned to, but not in the reads). One way of dealing with this problem is to align the reads against a set of (already spliced) known exonic sequences. The main drawback with this method is that if the set of known exonic sequences is incomplete, there will be many unalignable reads. Another approach is that taken by TopHat, which allows the identification of novel splice junctions.

Mapping reads to the transcriptome with TopHat

We are going to use two small datasets of under 100,000 single 75-bp reads from the ENCODE GM12878 cell line and ENCODE h1-hESC cell line, and compare the transcriptomes between the two cell lines.

1. Open the Get Data → Upload File tool.
2. Get the two RNA-Seq Analysis datasets by entering the following URLs in the text entry box.

http://chagall.med.cornell.edu/galaxy/rnaseq/GM12878_rnaseq1.fastqsanger
http://chagall.med.cornell.edu/galaxy/rnaseq/h1hESC_rnaseq2.fastqsanger

3. Change the File-Format pulldown menu from 'Auto-detect' to 'Fastqsanger'.
4. Do a quality control check on the data before beginning any analysis.
    a. Run the NGS: QC and manipulation → FASTQ Summary Statistics tool.
    b. Use the Graph/Display Data → Boxplot tool to visualize the quality score data summary.

In the case of the GM12878 dataset, there is a dramatic decrease in quality around base 60, so we want to trim off the last 16 bases from each read.

1. Open the NGS: QC and manipulation → FASTQ Trimmer tool.
2. Use 16 as offset from the 3' end.

In the case of the h1-hESC data, the data is mostly high quality with a single base in the middle with a median quality of 20, and tailing off to a median quality of around 22. These are all acceptable, so we will not trim the h1-hESC dataset at all.

The next step is to align the now filtered reads to the genome using TopHat.

1. Open the NGS: RNA Analysis → Tophat for Illumina tool.
2. Select the hg19 Full genome from the organism pull down menu.
3. Change the settings from defaults to display the full parameter list. In general, it is usually okay to keep most of the default parameters, but it is usually good practice to go down the list and make sure that they all look appropriate for the current analysis. Note that some of the options are only applicable to paired end reads (e.g., Library type and Use closure search).
4. Reduce the maximum intron length (both for initial (whole read) searches and split-segment searches) down to 100000.
5. Turn off indel search and coverage search, to speed up the analysis.
6. Do this for both datasets.

TopHat

TopHat is designed specifically to deal with junction mapping and overcomes the limitation of relying on annotation of known splice junctions. It does this by first aligning as many reads as it can to the genome, using the Bowtie aligner; those reads that align will be the ones that fit completely within an exonic region (any reads that are mapped non-contiguously are those that contain intronic regions). TopHat then tries to assemble the mapped reads into consensus sequences, using the reference sequence to determine consensus. To ensure that the edges of the exons are also covered, TopHat uses a small amount of flanking sequence from the reference on both sides to extend the consensus.

Once these alignment steps are complete, TopHat builds a database of possible splice junctions and tries to map reads against these junctions to confirm them. Specifically, TopHat tries to identify splice junctions with the known splice acceptor and donor sites GT-AG, GC-AG and AT-AC. TopHat has three ways in which it can define a potential junction. The first method of identifying/verifying potential junctions is when the short segments of a single read map far apart from each other on the same chromosome ("split-segment search"). For each splice junction, TopHat will search the initially unmapped reads to find any that can span that junction. The second method by which TopHat predicts junctions is called "coverage search", where TopHat tries to find possible introns within deeply sequenced islands. The third method is called "closure search", applicable only to paired end reads. If the two reads are mapped further apart from one another than the expected distance, TopHat assumes that they come from different exons and attempts to join them by looking for subsequences in the genomic interval between them that approximates the expected distance between them.

Options in TopHat

Mean inner distance between mate pairs. [PE only] If dealing with paired end reads, TopHat needs to be told the expected distance between those paired reads.

Library type. [PE only] Determines to which strand TopHat will attempt to align reads. fr-unstranded is the standard Illumina paired end situation where the left-most end of the read is mapped to the transcript strand and the right-most end is mapped to the other strand. fr-firststrand assumes that only the right-most end of a fragment is sequenced, whereas fr-secondstrand assumes that only the left-most end of the fragment is sequenced.

Anchor length. This is the minimum number of bases from aligned reads that have to be present on either side of a junction for it to be recognized by TopHat as a potential junction. Default: 8.

Maximum number of mismatches in anchor region. This defines the maximum number of mismatches allowed in the anchor region defined by the Anchor Length option. Default: 0.

Minimum intron length. This is the minimum amount of distance that can separate two exons (i.e., if two exons are closer than this, TopHat will not search for splice acceptor/donor sites between them and instead will assume that the exon has low coverage in the middle and attempt to merge it into one exon instead). Default: 70.

Maximum intron length. This is the maximum amount of distance that can separate two exons (i.e., if two exons are further apart than this, TopHat will not search for splice acceptor/donor sites between them, except in those cases where two shorter segments of a split-up read support such a distant pairing). Decreasing this distance will increase the speed of the search with a concomitant decrease in sensitivity. Default: 500000.

Allow indel search. Checking this option will allow TopHat to include insertions and deletions in your reads relative to the genome sequence. The length of the allowed insertions and deletions are determined by the two options that open up once this option is checked: Max insertion length and Max deletion length.

Minimum isoform fraction. For each junction, the average depth of read coverage is computed for the left and right flanking regions of the junction separately. The number of alignments crossing the junction is divided by the coverage of the more deeply covered side to obtain an estimate of the minor isoform frequency. The default value for this is set at 0.15, since Wang et al (2008) reported that 86% of the minor isoforms of alternative splicing events in humans were expressed at 15% or higher of the level of the major isoform.

Maximum number alignments. Discards from further analysis reads that map to an excessive number of different regions on the genome. This allows 'multireads' from genes with multiple copies to be reported, but tends to discard failed reads wich map to multiple low complexity regions. Default: 20.

Minimum intron length in split-segment search. The minimum intron length that may be found during split-segment search. Default: 50.

Maximum intron length in split-segment search. The maximum intron length that may be found during split-segment search. Default: 500000.

Number mismatches allowed in initial read mapping. The maximum number of mismatches that may appear in the initial (unsegmented) alignment. Default: 2.

Number of mismatches allowed in each segment alignment. The maximum number of mismatches that may appear in each segment of a segmented read. Default: 2.

Minimum length of read segments. The length of the segments that the reads are split into for the split-segment search.

Use own junctions. This allows you to give TopHat a set of junctions that it can add into its database of potential junctions. This is most commonly used when you have a mixed dataset (e.g., of paired end reads and single reads); run Tophat with one set of reads, save the potential junction database that TopHat produces, and then feed that database into the second run with the rest of original dataset.

Use gene annotation model. Available if supplying junctions to TopHat. TopHat uses the supplied exon records to build a set of known splice junctions for each gene and will add those junctions to its potential junction database.

Use raw junctions. Available if supplying junctions to TopHat. This allows you to supply TopHat with a list of raw junctions, usually from a previous TopHat run. Junctions are specified one per line, in a tab-delimited format. Records are defined as [chrom] [left] [right] [+/-], where left and right are zero-based coordinates, and specify the last character of the left sequence to be spliced to the first character of the right sequence, inclusive.

Only look for supplied junctions. Available if supplying junctions to TopHat. Forces TopHat to not add any more junctions from initial mapping results to its database and only use the junctions that you have supplied (either as gene annotations or as raw junctions.)

Use closure search. Enables the mate pair closure-based search for junctions. Closure-based search should only be used when the expected inner distance between mates is small (<= 50bp).

Use coverage search. Enables the coverage based search for junctions, so that TopHat can search for junctions within islands. Coverage search is disabled by default (such as for reads 75bp or longer), for maximum sensitivity. Enabling this will slow down the analysis dramatically.

Microexon search. With this option, the pipeline will attempt to map reads to microexons (very short exons, usually about 25 bases or less) by cutting up reads into smaller segments and attempting to align them to the genome. Works only for reads 50bp or longer.

Analysis continued

Each TopHat run will result in four files: a list of accepted mapped reads in BAM format , and three BED files: one for raw splice junctions (which can then be fed into a subsequent TopHat analysis), one each for detected insertions and deletions (although these will be empty if the indel search was disabled.)

The splice junctions file is formatted as a BED file, using all optional columns. This will enable the visualization of splice junctions as a pair of boxes joined by a thin line. The column headings are:

1. Chromosome name
2. Start position of junction representation
3. End position of junction representation
4. Name of the junction
5. Junction score (how many reads support the junction)
6. Strand
7. Position at which to start drawing the first box
8. Position at which to end drawing the last box
9. Color with which to paint the junction representation
10. Number of thick boxes (almost always 2)
11. Size of each of the thick boxes
12. Distance of the start position of each box from the start position of the junction representation (first number always 0)

The accepted hits file is a standard BAM file (with the standard SAM information). To verify the contents of this file, we can convert it to a human-readable SAM formatted file with the NGS: SAM Tools → BAM-to-SAM tool.

Because Bowtie allows reads to map to multiple places, and will return all found matches, there are a few tags that you will see in the SAM file that were not used in the BWA output.

NH:i	Number of hits (i.e., number of times the read mapped)
HI:i	Hit index (a way to refer to each separate mapping)
CC:Z	Reference name of the next hit
CP:i	Leftmost coordinate of the next hit

Assembly

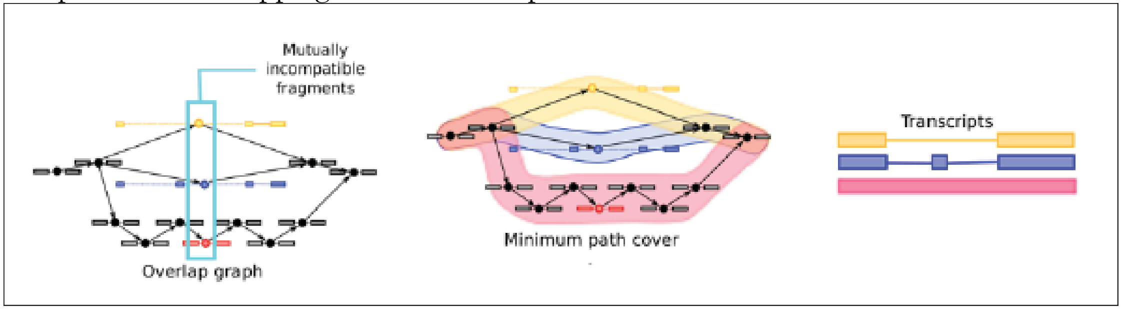
Once the reads have been mapped, we want to assemble the reads into complete transcripts which can then be analyzed for differential splicing events, or differential expression.

This is done using CuffLinks.

1. Run NGS: RNA Analysis → Cufflinks on the two accepted_hits datasets produced by our earlier Tophat analysis (one for each of our two original FastQ input datasets).
2. Change the max intron length to 100000.
3. Click Execute.

CuffLinks

CuffLinks uses a directed acyclic graph algorithm to identify the minimum set of independent transcripts that can explain the reads observed in an RNA-Seq experiment. It does this by grouping reads into clusters that all map to the same region of the genome and then identifying "incompatible" reads, which cannot possibly have come from the same transcript. Once the minimum number of possible transcripts has been identified, it then assigns each read in the cluster to one or more of those transcripts, depending on compatibility. Abundance for each transcript is estimated based on the number of compatible reads mapping to each transcript.



Trapnell et al, Nat Biotechnol (2010) 28(5): 511-515. "Transcript assembly and abundance estimation from RNA-Seq reveals thousands of new transcripts and switching among isoforms"

It is important to note that the input to CuffLinks must be a SAM/BAM file sorted by reference position. If your input is from TopHat, it is probably already in the correct format, but if this is a SAM file of some other provenance, it should be sorted. Below is a sample workflow to sort your SAM data. If you are starting with a BAM file, convert it to a SAM file first, and then back to BAM format after sorting.

1. Open the Filter and Sort → Select tool.
2. Use the pattern ^@ as the criterion for selecting lines. This will select all the SAM header lines.
3. Click Execute.
4. Open the Filter and Sort → Select tool again.
5. Use the same pattern as before, but this time change the pulldown menu to say NOT matching. This ignores all the header lines and selects all the alignment lines that now need to be sorted by reference position.
6. Open the Filter and Sort → Sort tool.
7. Sort the alignment lines file on column 3 (the reference chromosome number), alphabetically in ascending order.
8. Add a second column selection and sort on column 4 (the chromosome position), numerically, in ascending order.
9. Open the Text Manipulation → Concatenate datasets tool.
10. Ensure the first dataset selected is the SAM headers dataset from step 2.
11. Click the Add a new dataset button.
12. Add the sorted alignment lines dataset from step 8.

Options in CuffLinks

Maximum intron length. CuffLinks will not report transcripts with introns longer than this. Default: 300,000.

Minimum isoform fraction. After calculating isoform abundance for a gene, CuffLinks filters out transcripts that it believes are very low abundance, because isoforms expressed at extremely low levels often cannot reliably be assembled, and may even be artifacts of incompletely spliced precursors of processed transcripts. This parameter is also used to filter out introns that have very few spliced alignments supporting them. Default: 0.1 (i.e., 10% of the major isoform).

Pre MRNA fraction. Some RNA-Seq protocols produce a significant amount of reads that originate from incompletely spliced transcripts, and these reads can confound the assembly of fully spliced mRNAs. CuffLinks uses this parameter to filter out alignments that lie within the intronic intervals implied by the spliced alignments. The minimum depth of coverage in the intronic region covered by the alignment is divided by the number of spliced reads, and if the result is lower than this parameter value, the intronic alignments are ignored. Default: 0.15.

Perform quartile normalization. In some cases, a small number of abundant, differentially expressed genes can create the (incorrect) impression that less abundant genes are also differentially expressed. This option allows CuffLinks to exclude the contribution of the top 25 percent most highly expressed genes from the number of mapped fragments used in the FPKM denominator, improving robustness of differential expression calls for less abundant genes and transcripts.

Use reference annotation. Tells CuffLinks to use the supplied reference annotation to estimate isoform expression. It will not assemble novel transcripts, and the program will ignore alignments not structurally compatible with any reference transcript.

Perform bias correction. Requires a reference sequence file. This option forces CuffLinks to detect sequences which are overrepresented due to library preparation or sequencing bias and correct for this. Bias detection and correction can significantly improve accuracy of transcript abundance estimates.

Mean inner distance between mate pairs. [PE only] This is the expected (mean) inner distance between mate pairs. For, example, for paired end runs with fragments selected at 300bp, where each end is 50bp, you should set -r to be 200. The default is 20bp.

Standard deviation for inner distance between mate pairs. [PE only] The standard deviation for the distribution on inner distances between mate pairs. The default is 20bp.

The output from CuffLinks consists of three datasets: a GTF formatted dataset listing the assembled isoforms detected by CuffLinks, and two datasets separating out the coverage data from the GTF datasets for transcripts and for genes.

The GTF dataset contains the following information:

1. Chromosome name
2. Source (always Cufflinks)
3. Feature type (always either 'transcript' or 'exon')
4. Start position of the feature
5. End position of the feature
6. Score (the most abundant isoform for each gene is assigned a score of 1000. Minor isoforms are scored by the ratio (minor FPKM/major FPKM))
7. Strand of isoform
8. Frame (not used)
9. Attributes
   a. gene_id: Cufflinks gene id
   b. transcript_id: Cufflinks transcript id
   c. exon_number: Exon position in isoform. Only used if feature type is exon
   d. FPKM: Relative abundance of isoform
   e. frac (not used)
   f. conf_lo: Lower bound of the 95% CI for the FPKM
   g. conf_hi: Upper bound of the 95% CI for the FPKM
   h. cov: Depth of read coverage across the isoform

RPKM [Reads Per Kilobase per Million reads mapped]

RPKM is a measurement of transcript reads that has been normalized both for transcript length and for the total number of mappable reads from an experiment. This normalized number helps in the comparison of transcript levels both within and between samples. Normalizing by the total number of mapped reads allows comparison between experiments (since you may get more mapped reads in one experiment), whereas normalizing by the length of the transcript allows the direct comparison of expression level between differently sized transcripts (since longer transcripts are more likely to have more reads mapped to them than shorter ones).

$$RPKM = \frac{totalTranscriptReads}{mappedReads(millions)\ x\ transcriptLength(Kb)}$$

where mappedReads is the number of mapped reads in that experiment.

You will also see the term FPKM, where the F stands for Fragments. This is a similar measure to RPKM used for paired end experiments, where a fragment is a pair of reads.

Note that in our example, the RPKM numbers will be far higher than normally seen since the total number of mapped reads in our dataset is small (because our input dataset was a subset selected to map to a small region of chromosome 19).

Note that RPKM may not be the best method of quantifying differential expression. Other methods include DESeq and TMM from the Bioconductor package.

Comparison with reference annotations

We need to download a set of reference annotations against which to compare the transcripts assembled from the RNA-Seq experiments.

1. Open the Get Data → UCSC Main tool.
2. Select the hg19 assembly.
3. Make sure the Genes and Gene Prediction Tracks group is selected, choose the RefSeq genes track.
4. Change the region from the genome radio button to the position button, and enter chr19 (since all our reads map to chromosome 19).
5. Change the output format to GTF (gene transfer format). Make sure the Send to Galaxy button is checked and click the Get Output button.

To compare the assembled transcripts against the RefSeq data:

1. Open the NGS: RNA Analysis → Cuffcompare tool.
2. Select the CuffLinks assembled transcripts GTF-formatted dataset for the h1-hESC data.
3. Add a new Additional GTF Input File and select the GM12878 assembled transcripts.
4. Set the Use Reference Annotation to Yes and choose the GTF-formatted RefSeq Genes datasets.
5. Since we will be looking at only a small section of chromosome 19, check the box for Ignore reference transcripts that are not overlapped by any transcript in input files.
6. Set Use Sequence Data to Yes.
7. Click Execute.

CuffCompare

CuffCompare compares the assembled transcripts to a reference annotation and details the identities and differences between them.

Options in CuffCompare

Use Reference Annotation. An optional "reference" annotation GTF. Each sample is matched against this file, and sample isoforms are tagged as overlapping, matching, or novel where appropriate.

Ignore reference transcripts that are not overlapped by any transcript in input files. Causes CuffCompare to ignore reference transcripts that are not overlapped by any transcript in your assembled transcripts datasets. Useful for ignoring annotated transcripts that are not present in your RNA-Seq samples and thus adjusting the "sensitivity" calculation in the accuracy report written in the transcripts_accuracy file

Use Sequence Data. Use sequence data for some optional classification functions, including the addition of the p_id attribute required by CuffDiff, which is the identifier for the coding sequence contained by this transcript. Set to Yes if comparing multiple experiments.

Running CuffCompare results in a number of different datasets.

The transcript accuracy dataset calculates the accuracy of each of the transcripts as compared to the reference at various levels (nucleotide, exon, intron, transcript, gene), e.g., how often an exon that was predicted by the output from CuffLinks was actually seen in the reference. The Sn and Sp columns calculate sensitivity (the proportion of exons, for example, that have been correctly identified) and specificity (the proportion of predicted exons that are annotated as such in the reference) at each level, while the fSn and fSp columns are "fuzzy" variants of these same accuracy calculations, allowing for a very small variation in exon boundaries to still be counted as a match.

There are two tmap datasets, one for each of the input assembled transcripts dataset. The tmap dataset lists the most closely matching reference transcript for each transcript identified by CuffLinks for that dataset. Each row in the dataset contains the following information:

1. ref_gene_id: Reference gene name, derived from the gene_name attribute from the reference GTF record, if present, else the gene_id.
2. ref_id: Reference transcript id, derived from the transcript_id attribute from the reference GTF record.
3. class_code: Relationship between the CuffLinks transcript and the reference transcript.
4. cuff_gene_id: The gene_id from the CuffLinks assembled transcripts dataset.
5. cuff_id: The transcript_id from the CuffLinks assembled transcripts dataset.
6. FMI: Expression level of transcript expressed as fraction of major isoform. Ranges from 1 to 100.
7. FPKM: Expression of this transcript.
8. FPKM_conf_lo: The lower limit of the 95% FPKM CI.
9. FPKM_conf_hi: The upper limit of the 95% FPKM CI.
10. cov: The estimated average depth of read coverage across the transcript.
11. len: The length of the transcript.
12. major_iso_id: The CuffLinks transcript_id of the major isoform for this transcript's gene.
13. ref_match_len: Length of the matching reference gene.

The class codes are defined as follows:

=	Match.
c	Contained.
j	New isoform.
e	A single exon transcript overlapping a reference exon and at least 10 bp of a reference intron, indicating a possible pre-mRNA fragment.
i	A single exon transcript falling entirely with a reference intron.
r	Repeat. Currently determined by looking at the reference sequence and applied to transcripts where at least 50% of the bases are lower case.
p	Possible polymerase run-on fragment.
u	Unknown, intergenic transcript.
o	Unknown, generic overlap with reference.
x	Exonic overlap with reference on the opposite strand.

Using these codes, we could now, for example, extract all new isoforms from the tmap dataset by using the Filter and Sort → Filter and using c3 == 'j' as our match criterion.

There are two refmap datasets, one for each input assembled transcripts dataset. The refmap dataset lists, for each reference transcript, all the transcripts identified by CuffLinks for that dataset that at least partially match it. Each row in the dataset contains the following information:

1. ref_gene_id: Reference gene name, derived from the gene_name attribute from the reference GTF record, if present, else the gene_id.
2. ref_id: Reference transcript id, derived from the transcript_id attribute from the reference GTF record.
3. class_code: Relationship between the CuffLinks transcript and the reference transcript. (Can only be either = (full match) or c (partial match)).
4. cuff_id_list: A list of all CuffLinks transcript_ids matching the reference transcript.

The transcript tracking dataset is created when multiple assembled transcript datasets are given to CuffCompare as input. The tracking file matches transcripts up between samples, assigning each (unified) transcript its own internal transfrag and locus id.

The last dataset is a GTF-formatted combined transcripts dataset which contains one line for every exon identified, its position in the genome, along with a number of attributes

1. gene_id: the internal locus id (XLOC_*) as defined in the tracking file.
2. transcript_id: the internal transcript id (TCONS_*) as defined in the tracking file.
3. exon_number: the position of that exon in the transcript.
4. gene_name: if mapped to an annotated CDS, the name of the gene the transcript has been mapped to.
5. oId: the transcript id assigned to that transcript by CuffLinks.
6. nearest_ref: the name of the nearest gene in the reference annotation.
7. class_code: Relationship between the CuffLinks transcript and the reference transcript.
8. tss_id: an identifier for the inferred transcription start site for the transcript that contains that exon. Determines which primary transcript this processed transcript is believed to come from.
9. p_id: if mapped to an annotated CDS, an identifier for the sequence coded for by that gene.

Comparison between RNA-Seq experiments

Finally, we can use the combined transcripts dataset to compare the relative expression levels of each exon and each transcript in both of our original input datasets, using CuffDiff.

1. Open the NGS: RNA Analysis → Cuffdiff tool.
2. Select the combined transcripts dataset from CuffCompare.
3. For the first BAM file, choose the output from TopHat of accepted reads for the GM12878 input dataset, and for the second, the one for the h1-hESC dataset.
4. Click Execute.

CuffDiff

CuffDiff tests the significance of gene and transcript expression levels in more than one condition. It takes as input the GTF-formatted dataset of transcripts, along with (at least) two SAM/BAM datasets containing the accepted mappings for the samples under consideration and outputs changes in expression at the level of transcripts, primary transcripts, and genes as well as changes in the relative abundance of transcripts sharing a common transcription start site (tss_id), and in the relative abundances of the primary transcripts of each gene. The p_id which was generated by CuffCompare is used to denote a coding sequence (and is only used when the reference sequence includes CDS records). To calculate whether the relative expression of a transcript isoform is significantly different between two samples, CuffDiff uses a two-tailed t-test which incorporates information about the variability in the number of fragments generated by the transcript across replicates (if available), and also incorporates any uncertainty in the expression estimate itself, which is calculated based on how many other transcript expression levels a read could be contributing to.

Options in CuffDiff

False Discovery Rate. The allowed false discovery rate (used for multiple hypothesis correction). Default: 0.05.
Min Alignment Count. The minimum number of alignments needed to conduct significance testing on changes for a transcript. If a transcript does not have the minimum number of alignments, no testing is performed, and any expression changes are deemed not significant, and that transcript does not contribute to correction for multiple testing. Default: 10.
Perform quartile normalization. As for CuffLinks, this option allows the exclusion of the contribution of the top 25 percent most highly expressed genes from the number of mapped fragments used in the FPKM denominator, to improve robustness of differential expression calls for less abundant genes and transcripts.
Perform Bias Correction. Again, similar to the option in CuffLinks, designed to significantly improve accuracy of transcript abundance estimates. Requires a reference sequence file. Detects sequences which are overrepresented due to library preparation or sequencing bias and corrects for this.

CuffDiff outputs many files, including FPKM tracking datasets at different levels (i.e., individual transcripts, gene (combines all transcripts sharing a gene_id), coding sequence (combines all transcripts sharing a p_id), transcript start site (combines all transcripts sharing a tss_id)) and differential expression test datasets for each of these same levels, testing the significance of the relative expression levels of each group between samples.

In addition to the standard information about each object (id, position, annotation), each of the FPKM tracking datasets also includes information about the estimated read depth for each sample, the FPKM +/- 95% CI for each sample as well as the quantification status for that sample (options include: OK (deconvolution successful), LOWDATA (too complex or shallowly sequenced), HIDATA (too many fragments in locus), or FAIL, (when an ill-conditioned covariance matrix or other numerical exception prevents deconvolution)).

The differential expression test datasets summarize the t-test result that calculates the significance of the differential expression levels between samples. These datasets include information about the calculated p-value, the corrected p-value (called q-value) adjusted for FDR using Benjamini-Hochberg multiple hypothesis correction and whether the q-value is significant or not.

We can now inspect these datasets and retrieve transcripts that may be of interest. For example, we can search for novel isoforms.

1. Open the Filter and Sort → Filter tool.
2. Select the transcript FPKM tracking dataset.
3. Use c2 == 'j' as a filter, where c2 is the column containing the class code for the relationship between the assembled CuffLinks transcript and the reference annotation. This will retrieve all assembled transcripts that do match any annotated isoforms ('j' indicating a new isoform).
4. Click Execute.
5. Open the Filter and Sort → Filter GTF data by attribute values_list tool.
6. Select the Cuffcompare combined transcripts dataset.
7. Select the transcript_id field from the pulldown menu and make sure that the filtered dataset containing only the significantly differentially expressed transcripts is selected as the dataset to filter on. Note that if you want to select on any other field, the filtered dataset must be manipulated such that the field you want to select on is the first field in the dataset.
8. Click Execute.
9. The result will be a GTF formatted dataset containing only the novel transcript isoforms, one exon per line, that is now viewable in a genome browser.

We can also search for transcripts that are significantly differentially expressed between the two samples.

1. Open the Filter and Sort → Filter tool.
2. Choose the transcript differential expression testing dataset generated by CuffDiff.
3. Filter on c14=='yes' (where c14 is the column that denotes whether the corrected differential expression is significant).
4. Click Execute.

Visualization in UCSC Genome Browser

The best way to investigate the data is to visualize it. The output of TopHat includes a BED and a BAM file which are both formats that the UCSC Genome Browser knows how to deal with. To view the data for the h1-hESC experiment, we can simply click the display at UCSC main link in the history preview window for the h1-hESC Tophat accepted hits dataset. Using the UCSC Genome Browser is useful because it already contains all the features that we may want to compare our results against (i.e., gene annotations). A new browser tab will be opened which now contains the TopHat accepted hits as a custom track. The default appearance of that custom track will be set to 'dense', but that can be changed to 'pack' which will enable us to see each read mapped to the genome. Clicking on any read will take you to a page with detailed information derived from the SAM file about the read, including the alignment of the sequence and the mapping quality.

Going back to the Galaxy browser window, we can now click the display at UCSC main link in the history preview window for the splice junctions file for the h1-hESC experiment and this will be added to the custom tracks and can be viewed simultaneously with the accepted hits file. We can now see the reads spanning exons, and the junctions that define those exon boundaries.

Finally, we can add the CuffLinks assembled transcripts dataset for the h1-hESC dataset to the visualization which shows the complete transcripts alongside the mapped reads, junctions, and reference genes.

Scroll around the genome browser to find examples of correctly and incorrectly called junctions or missed annotations (e.g., at positions chr19:274,000-297,000, chr19:1,010,000-1,020,000 or chr19:75,353-86,513). Of note is that this particular experiment fails to join many of the longer exons as there are not enough reads spanning their whole length, and annotates them as two (or more) different transcripts (e.g., KLF16).

Visualization in Galaxy

Galaxy also has a framework for visualizing results. Unlike the UCSC Genome Browser, though, we need to import all the annotations that we would like to compare our results to.

To create a new visualization in Galaxy:

1. Select New Visualization from the Visualization tab in the main Galaxy menu at the top of the page.
2. Name your visualization.
3. Assign the hg19 build as the reference genome.
4. Add datasets to your visualization by clicking on the Add Datasets to Visualization button. Choose both the accepted_hits and splice junctions files from your current history, as well as the uploaded chromosome 19 RefSeq annotations.
5. Choose Insert.
6. To view the data, choose chr19 from the Select Chrom/Contig pulldown menu. The controls are similar to those at the UCSC Genome Browser. To zoom in to the first section of chromosome 19 where all the hits are located, just drag a box over the base positions you want to see in greater detail.
7. To save your visualization, click on the small disk icon in the upper right hand corner.

ChIP-Seq analyses with MACS

ChIP-Seq

ChIP-Seq experiments are designed to map DNA-protein interactions by identifying all the genomic elements that are bound by a protein or transcription factor, coupling chromatin immunoprecipitation with NGS sequencing techniques. Although ChIP-Seq has many advantages over other ChIP-type experiments, the NGS side of it also lends it some idiosyncratic disadvantages. The first of these is that the reads represent only the ends of the ChIP fragments and the user has to extrapolate from these partials where the fragment lies on the genome. Secondly, there are regional biases along the genome, due to sequencing and mapping biases, chromatin structure and genome copy number variations.

MACS

For this exercise, we will be using the MACS (Model-based Analysis of ChIP-Seq) program to call peaks. MACS attempts to address both of these issues. The first issue, of reads representing the ends of the ChIP fragments, is dealt with by searching for patterns of bimodally mapped reads (since the expectation is that fragments are sequenced, on average, at the same rate from either end), with plus-strand reads enriched in one peak, and minus-strands in the other. It calculates the distance between the forward and reverse strand peaks, and then shifts each fragment inward (towards the center of the peak) by half that distance, resulting in the region between the two peaks now being narrowed somewhat and “filled in” as one peak. The second issue of dealing with biases along the genome is dealt with by using a control dataset (which is not available with all datasets). A control dataset will also show many of the same biases that are present in the experimental dataset, so the implication is that any time the experimental dataset shows peaks that are not mirrored by the control dataset, that these peaks are not accounted for by biases and are therefore real.

Options in MACS tool

Note that the version of MACS in Galaxy is not the latest version!
Paired end Sequencing. If this is selected, the tool will expect two input files, and if using a control, will expect two control files. Additionally, it will include a new option Best distance between Pair-End Tags which MACS will use to decide the best mapped locations for 3’ and 5’ pairs of reads (optimized over distance and incurred mismatches.)
Effective genome size. This is the mappable genome size and should be changed according to the genome you are working with. In general, the effective size will be smaller than the actual size of the genome, because of repetitive sequences, etc. The effective genome sizes for some of the more common organisms are: human: 2.7e+9, mouse: 1.87e+9, C. elegans: 9e+7, fruitfly: 1.2e+8. Default: 2.7e+9.
Tag size. This is the length of your tags, which should be specified, otherwise MACS will take the length of your first 10 sequences as being representative of your dataset.
Band width. This is used in the first step of finding bimodal peaks and is expected to approximate the sonication fragment size.

MFOLD. Only regions above the selected range of high-confidence enrichment ratio of background to build model are returned. Using two numbers here, delimited by a comma, will return only those regions that have ratios within those limits.
Wiggle. Saves all the shifted tag file locations as a wiggle file. If you choose to save this information, you also need to tell MACS how far out to extend each fragment (default is the distance calculated between the forward and reverse strand peaks) and at what resolution to save the wiggle information (the default is 10 bases.) Note that wiggle tracks can be converted to BigWig format using the Convert Formats → Wig-to-bigWig tool, and visualized in your preferred genome browser.
Background lambda. The lambda is used to describe the Poisson distribution of tags along the genome. If this is checked, then the background lambda is used as the local lambda and MACS will not consider the local bias at peak candidate regions.
3 levels of regions. This determines the different levels of regions that are used to calculate the local lambda. A large region like 10000bps will capture the bias from long range effects like open chromatin domains.
Build shifting model. If this is not selected, then instead of looking for bimodal pairs of peaks, calculating the distance between them and moving each tag inwards by d/2, you can tell MACS by exactly how much you want the tags moved in a 3' direction. For example, if you knew that your protein footprint was 200bp, you could choose not to build the shifting model and instead set the Arbitrary shift size to 100 (this parameter becomes available once this option is selected).

Download the input file

1. Open the Get Data → Upload File tool.
2. Download the input file at
http://chagall.med.cornell.edu/galaxy/chipseq/G1E-estradiol.fastqsanger
3. This is a reduced dataset (chr19) looking for CTCF binding sites, derived from the G1E line, a GATA1 null-derived line used as a model for erythropoiesis. In this case, the cell line has been induced to differentiate by estradiol treatment.
4. Choose the fastqsanger option from the File Format menu.
5. Associate this input file with the mm9 genome build.
6. Click Execute.
7. Open the NGS: QC and Manipulation → FASTQ Summary Statistics tool and run a quality control check on the imported dataset.
8. Visualize the results with Graph/Display Data → Boxplot.
9. From the boxplot, it can be seen that the sequences are 36 bases long, and all quality scores are above 30, so we do not have to trim or discard any data.

Map the reads against the mouse genome

1. Open the NGS: Mapping → Map with Bowtie for Illumina tool.
2. Select the fastqsanger formatted reads dataset.
3. Select the mm9 Full genome.
4. Click Execute.

Find peaks using MACS

Once the reads have been mapped to the mouse genome, we can run the MACS program to determine where the peaks are.

1. Open the NGS: Peak Calling → MACS tool.
2. Change the experiment name so that the track will have a unique name when we visualize it.
3. Select the dataset of reads mapped by Bowtie.
4. Change the Tag Size to 36 (the length of the reads as determined in the QC step).
5. Change the genome size to "1.87e+9" (effective size for mouse genome).
6. Click Execute.

Once the MACS program finishes, there are two results files, one with the positions of the peaks and the other a more general HTML report file. The html report has links to various statistics files, as well as the log of the MACS run. This includes, at the end of the output, the number of peaks that were called. In this case, we found 750 peaks.

Find peaks, using control data

Since some of the peaks that are predicted may be due to experimental set up and sequencing biases, it is often useful to use a control dataset (i.e., one where no transcription factor is bound to the DNA) to detect such biases and eliminate them from our predictions.

1. Open the Get Data → Upload File tool.
2. Download the control input file at

http://chagall.med.cornell.edu/galaxy/chipseq/G1E-estradiolControl.fastqsanger

3. This is a control dataset, derived from the same G1E line as that experiment, that has also been induced to differentiate by estradiol treatment, but has no CTCF bound to the DNA.
4. Choose the fastqsanger option from the File Format menu.
5. Associate this input file with the mm9 genome build.
6. Click Execute.
7. Open the NGS: QC and Manipulation → FASTQ Summary Statistics tool and run a quality control check on the imported dataset.
8. Visualize the results with Graph/Display Data → Boxplot tool.
9. From the boxplot, it can be seen that the sequences are 36 bases long, and all the reads have a median quality score of at least 26, so we do not have to trim or discard any data.
10. Open the NGS: Mapping → Map with Bowtie for Illumina tool.
11. Select the fastqsanger formatted control reads dataset.
12. Select the mm9 Full genome as the reference genome.
13. Click Execute.

Now we can compare the mapped reads from both the experiment and control.

14. Open the NGS: Peak Calling → MACS tool.
15. Select the dataset of experimental mapped reads as the ChIP-Seq Tag File.
16. Select the dataset of control mapped reads as the ChIP-Seq Control File.
17. Change the Tag Size to 36 (the length of the reads as determined in the QC step).

18. Change the genome size to “1.87e+9” (effective size for mouse genome.)
19. Click Execute.

This time, we get 852 peaks. This indicates that corrections due to sequencing bias along the genome go both ways, both removing and adding peaks. Since our output is a BED file, we can visualize the peaks at the UCSC Genome Browser by choosing the Display at UCSC main link.

Comparison between conditions

We can also compare predicted peaks between different conditions. Here, we will identify sites that have differential binding across the differentiated and undifferentiated states.

First, create a workflow from the above analysis, such that it takes as input two fastqsanger datasets, one a control and the other the experiment, maps both of them with Bowtie against the mm9 mouse genome build, and then analyzes the reads with MACS to come up with a set of predicted peaks.

1. Import two more datasets using the Get Data → Upload File tool. These datasets are from the same cell line, but this time without estradiol treatment (i.e., it remains undifferentiated). The experimental dataset is:

http://chagall.med.cornell.edu/galaxy/chipseq/G1E-undifferentiated.fastqsanger

2. and the control dataset is:

http://chagall.med.cornell.edu/galaxy/chipseq/G1E-undifferentiatedControl.fastqsanger

3. Run a quality control check on both, using the NGS: QC and Manipulation → FASTQ Summary Statistics and Graph/Display Data → Boxplot tool.
4. Use these files as input into the workflow you just created.

Note. If Bowtie is taking too long to run, copy over the four datasets of Bowtie-mapped reads from the accessible history and run MACS on them.
https://main.g2.bx.psu.edu/u/luce/h/mappingresults

Once we have access to all the peaks called by MACS, we can compare the peaks from the two samples to identify CTCF sites that are a) found in both conditions (exposed and not exposed to estradiol); b) found in the differentiated state (+ER4) but not in the undifferentiated state (-ER4); c) found in the undifferentiated state but not in the differentiated state.
To obtain a dataset of all peaks found in both conditions:

1. Open the Operate on Genomic Intervals → Intersect tool.
2. Choose the dataset of CTCF peaks identified with estradiol treatment as the first dataset.
3. Choose the dataset of CTCF peaks identified without estradiol treatment as the second dataset.
4. Click Execute.

To get a dataset of all CTCF peaks identified in the cell line when treated with estradiol, but not in the undifferentiated line:

5. Open the Operate on Genomic Intervals → Subtract tool.
6. Choose the dataset of CTCF peaks identified without estradiol treatment as the first dataset.
7. Choose the dataset of CTCF peaks identified with estradiol treatment as the second
8. Click Execute.

And finally, to get a dataset of all CTCF peaks identified in the undifferentiated cell line that disappear when treated with estradiol:

9. Open the Operate on Genomic Intervals → Subtract tool.
10. Choose the dataset of CTCF peaks identified with estradiol treatment as the first dataset.
11. Choose the dataset of CTCF peaks identified without estradiol treatment as the second
12. Click Execute.

We can create a custom track with information on all three datasets above for visualization in the UCSC Genome Browser.

1. Open the Graph / Display Data → Build Custom Track tool.
2. Add a new track for each of the three datasets above.
3. Give each track a different name and color, and specify pack as the visibility option.
4. Once the custom track has been created, choose the link to display it at UCSC in the history preview window. A region where examples of peaks from all three comparisons can be seen is chr19:3,156,415-3,556,414.

Finally, we might like to identify those sites predicted to bind CTCF in the estradiol-treated cell line that lie near promoters.

1. Open the Get Data → UCSC Main tool.
2. Select mouse as the organism and mm9 as the genome build.
3. Select the RefSeq genes track.
4. Make sure the BED output format is selected and the Send to Galaxy checkbox is checked.
5. Click Get Output.
6. Select the "promoters" by selecting 1000 bases upstream (or you can get the gene data now and then transform it with the Operate on Genomic Intervals → Get flanks tool.)
7. Click the Send query to Galaxy button.
8. Open the Operate on Genomic Intervals → Join tool.
9. Select the peaks identified for the cell line with estradiol treatment as the first dataset and the promoter dataset as the second.
10. Click Execute.

Running Galaxy in the Cloud

The Galaxy wiki page http://wiki.g2.bx.psu.edu/Admin/Cloud for setting up an instance of Galaxy in the Amazon cloud is very comprehensive and gives detailed step-by-step instructions.

Before starting to set up a Galaxy instance, you will have to set up and verify an Amazon Web Services account. This account gives you access to the Amazon cloud service. The payment scheme is such that you only pay for the resources you use.

To start a Galaxy CloudMan cluster, we need to start a master instance which will be used to control all of the needed services as well as worker instances which run the analysis jobs.

1. Log in to your Amazon Web Services account at http://aws.amazon.com/
2. From the My Account / Console tab, choose AWS Management Console and sign in.
3. Go to the Security Credentials options on the same tab and make a note of your Access Key ID and your Secret Access Key.
4. Go to the EC2 tab.
5. Make sure your AWS Region is set to US East (Virginia).
6. Click Launch Instance.
7. Choose the Classic Wizard in the pop-up window.
8. Click on the Community AMIs tab and select ami-46d4792f as your AMI.
9. Set Number of Instances to 1. This is the head node of the cluster.
10. For the instance type, select at least a **large** node.
11. Choose any availability zone. It does not matter which zone you choose the first time, but once selected, you must select this same zone every time you instantiate the given cluster.
12. Click Continue.
13. Enter your user data in the format below, which specifies the desired name of the cloud cluster and provides Galaxy CloudMan with user account information. Note that there must be a space between the colon and the value of the field

```
cluster_name: <DESIRED CLUSTER NAME>
password: <DESIRED Galaxy CloudMan WEB UI PASSWORD>
access_key: <YOUR AWS ACCESS KEY>
secret_key: <YOUR AWS SECRET KEY>
```

14. The next popup allows you to Set Metadata Tags for this instance. Set the Name tag for this instance, as that will appear in the instance list of the AWS EC2 Management Console.
15. Choose the key pair you created during the initial setup.
16. Select the security group in the initial setup, and the default group and continue.
17. Check your entries one more time, and then Launch the instance and wait (about 5 minutes on average) for the instance and CloudMan to boot.

18. Go to the AWS management console, and click Instances, then select the instance you just launched. You need to wait until the instance state is Running, and Status checks says “2/2 checks passed”.
19. Copy the URL that appears at the top of the instance details panels into a web browser and hit enter. You should see a “Welcome to Galaxy on the cloud” page.
20. Click on the “please use the cloud console” link.
21. Login to the instance by entering the password you specified in User Data when starting the master instance.
22. The first time you login to this instance's Galaxy CloudMan interface, an “Initial Cluster Configuration” popup will appear, asking you how much disk space you want to allocate for your data. This can be increased later.
23. Click on Start Cluster.
24. It will take a few minutes for the master node to come up. The Access Galaxy button will go from grayed out to active. Disk status will show a disk with a green plus on it. Service status for both Applications and Data will be green (instead of yellow).
25. Once the Access Galaxy button is no longer grayed out, you can add nodes to the cluster by either clicking the Add Nodes button.
26. Once the worker nodes are up, click the Access Galaxy button. This opens up a new window with Galaxy on the Cloud. You are now running an elastic and fully loaded and populated version of Galaxy on the cloud.
27. Register yourself as a new user on the cloud instance and continue as you normally would.

TRAINER'S MANUAL

Introduction to Next Generation Sequencing Hands-on Workshop

Bioplatforms Australia (BPA)
The Commonwealth Scientific and Industrial Research Organisation (CSIRO)

TRAINER'S MANUAL

Licensing

Contents

Workshop Information

The Trainers

Dr. Zhiliang Chen
Postdoctoral Research Associate
The University of New South Wales (UNSW), NSW
zhiliang@unsw.edu.au

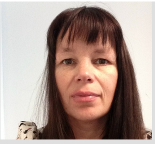
Dr. Susan Corley
Postdoctoral Research Associate
The University of New South Wales (UNSW), NSW
s.corley@unsw.edu.au

Dr. Nandan Deshpande
Postdoctoral Research Associate
The University of New South Wales (UNSW), NSW
n.deshpande@unsw.edu.au

Dr. Konsta Duesing
Research Team Leader - Statistics & Bioinformatics
CSIRO Animal, Food and Health Science, NSW
konsta.duesing@csiro.au

Dr. Matthew Field
Computational Biologist
The John Curtin School of Medical Research ANU College of Medicine, Biology & Environmen
ACT
matt.field@anu.edu.au

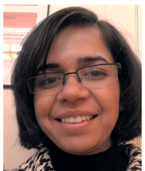
Dr. Xi (Sean) Li
Bioinformatics Analyst
Bioinformatics Core, CSIRO Mathematics, Informatics and Statistics, ACT
sean.li@csiro.au



Dr. Annette McGrath
Bioinformatics Core Leader at CSIRO
Bioinformatics Core, CSIRO Mathematics, Informatics and Statistics, ACT
Annette.Mcgrath@csiro.au

Mr. Sean McWilliam
Bioinformatics Analyst
CSIRO Animal, Food and Health Sciences, QLD
sean.mcwilliam@csiro.au

Dr. Paula Moolhuijzen
Senior Bioinformatics Officer
Centre for Comparative Genomics, Murdoch University, WA
pmoolhuijzen@ccg.murdoch.edu.au



Dr. Sonika Tyagi
Bioinformatics Supervisor
Australian Genome Research Facility Ltd, The Walter and Eliza Hall Institute, VIC
sonika.tyagi@agrf.org.au

Providing Feedback

While we endeavour to deliver a workshop with quality content and documentation in a venue conducive to an exciting, well run hands-on workshop with a bunch of knowledgeable and likable trainers, we know there are things we could do better.

Whilst we want to know what didn't quite hit the mark for you, what would be most helpful and least depressing, would be for you to provide ways to improve the workshop. i.e. constructive feedback. After all, if we knew something wasn't going to work, we wouldn't have done it or put it into the workshop in the first place! Remember, we're experts in the field of bioinformatics not experts in the field of biology!

Clearly, we also want to know what we did well! This gives us that "feel good" factor which will see us through those long days and nights in the lead up to such hands-on workshops!

With that in mind, we'll provide three really high tech mechanism through which you can provide anonymous feedback during the workshop:

1. A sheet of paper, from a flip-chart, sporting a "happy" face and a "not so happy" face. Armed with a stack of colourful post-it notes, your mission is to see how many comments you can stick on the "happy" side!
2. Some empty ruled pages at the back of this handout. Use them for your own personal notes or for write specific comments/feedback about the workshop as it progresses.
3. An online post-workshop evaluation survey. We'll ask you to complete this before you leave. If you've used the blank pages at the back of this handout to make feedback notes, you'll be able to provide more specific and helpful feedback with the least amount of brain-drain!

Document Structure

We have provided you with an electronic copy of the workshop's hands-on tutorial documents. We have done this for two reasons: 1) you will have something to take away with you at the end of the workshop, and 2) you can save time (mis)typing commands on the command line by using copy-and-paste.

We advise you to use Acrobat Reader to view the PDF. This is because it properly supports some features we have implemented to ensure that copy-and-paste of commands works as expected. This includes the appropriate copy-and-paste of special characters like tilde and hyphens as well as skipping line numbers for easy copy-and-past of whole code blocks.

> While you could fly through the hands-on sessions doing copy-and-paste you will learn more if you take the time, saved from not having to type all those commands, to understand what each command is doing!

The commands to enter at a terminal look something like this:

```
tophat --solexa-quals -g 2 --library-type fr-unstranded -j \
    annotation/Danio_rerio.Zv9.66.spliceSites -o tophat/ZV9_2cells \
    genome/ZV9 data/2cells_1.fastq data/2cells_2.fastq
```

The following styled code is not to be entered at a terminal, it is simply to show you the syntax of the command. You must use your own judgement to substitute in the correct arguments, options, filenames etc

```
tophat [options]* <index_base> <reads_1> <reads_2>
```

The following is an example how of R commands are styled:

```
R --no-save
library(plotrix)
data <- read.table("run_25/stats.txt", header=TRUE)
weighted.hist(data$short1_cov+data$short2_cov, data$lgth, breaks=0:70)
q()
```

The following icons are used in the margin, throughout the documentation to help you navigate around the document more easily:

- i Important
- ≣ For reference
- 👣 Follow these steps
- Q Questions to answer
- ! Warning - STOP and read
- ✸ Bonus exercise for fast learners
- ✸ Advanced exercise for super-fast learners

Resources Used

We have provided you with an environment which contains all the tools and data you need for the duration of this workshop. However, we also provide details about the tools and data used by each module at the start of the respective module documentation.

Module: Data Quality

Primary Author(s):
Sonika Tyagi sonika.tyagi@agrf.org.au

Contributor(s):
Nathan S. Watson-Haigh nathan.watson-haigh@awri.com.au

Key Learning Outcomes

After completing this practical the trainee should be able to:

- Assess the overall quality of NGS sequence reads
- Visualise the quality, and other associated matrices, of reads to decide on filters and cutoffs for cleaning up data ready for downstream analysis
- Clean up and pre-process the sequences data for further analysis

Resources You'll be Using

Tools Used

FastQC
http://www.bioinformatics.babraham.ac.uk/projects/fastqc/

FASTX-Toolkit
http://hannonlab.cshl.edu/fastx_toolkit/

Picard
http://picard.sourceforge.net/

Useful Links

FASTQ Encoding
http://en.wikipedia.org/wiki/FASTQ_format#Encoding

Introduction

Going on a blind date with your read set? For a better understanding of the consequences please check the data quality!

For the purpose of this tutorial we are focusing only on Illumina sequencing which uses 'sequence by synthesis' technology in a highly parallel fashion. Although Illumina high throughput sequencing provides highly accurate sequence data, several sequence artifacts, including base calling errors and small insertions/deletions, poor quality reads and primer/adapter contamination are quite common in the high throughput sequencing data. The primary errors are substitution errors. The error rates can vary from 0.5-2.0% with errors mainly rising in frequency at the 3' ends of reads.

One way to investigate sequence data quality is to visualize the quality scores and other metrics in a compact manner to get an idea about the quality of a read data set. Read data sets can be improved by post processing in different ways like trimming off low quality bases, cleaning up any sequencing adapters and removing PCR duplicates. We can also look at other statistics such as, sequence length distribution, base composition, sequence complexity, presence of ambiguous bases etc. to assess the overall quality of the data set.

Highly redundant coverage (>15X) of the genome can be used to correct sequencing errors in the reads before assembly and errors. Various k-mer based error correction methods exist but are beyond the scope of this tutorial.

Quality Value Encoding Schema

In order to use a single character to encode Phred qualities, ASCII characters are used (`http://shop.alterlinks.com/ascii-table/ascii-table-us.php`). All ASCII characters have a decimal number associated with them but the first 32 characters are non-printable (e.g. backspace, shift, return, escape). Therefore, the first printable ASCII character is number 33, the exclamation mark (!). In Phred+33 encoded quality values the exclamation mark takes the Phred quality score of zero.

Early Solexa (now Illumina) sequencing needed to encode negative quality values. Because ASCII characters < 33 are non-printable, using the Phred+33 encoding was not possible. Therefore, they simply moved the offset from 33 to 64 thus inventing the Phred+64 encoded quality values. In this encoding a Phred quality of zero is denoted by the ASCII number 64 (the @ character). Since Illumina 1.8, quality values are now encoded using Phred+33.

FASTQ does not provide a way to describe what quality encoding is used for the quality values. Therefore, you should find this out from your sequencing provider. Alternatively, you may be able to figure this out by determining what ASCII characters are present in the FASTQ file. E.g the presence of numbers in the quality strings, can only mean the quality values are Phred+33 encoded. However, due to the overlapping nature of the Phred+33 and Phred+64 encoding schema it is not always possible to identify what encoding is in use. For example, if the only characters seen in the quality string are (`@ABCDEFGHI`), then it is impossible to know if you have really good Phred+33 encoded qualities or really bad Phred+64 encoded qualities.

For a grapical representation of the different ASCII characters used in the two encoding schema see: http://en.wikipedia.org/wiki/FASTQ_format#Encoding.

Prepare the Environment

To investigate sequence data quality we will demonstrate tools called FastQC and FASTX-Toolkit. FastQC will process and present the reports in a visual manner. Based on the results, the sequence data can be processed using the FASTX-Toolkit. We will use one data set in this practical, which can be found in the QC directory on your desktop.

Open the Terminal and go to the directory where the data are stored:

```
cd ~/QC/
pwd
```

At any time, help can be displayed for FastQC using the following command:

```
fastqc -h
```

Quality Visualisation

We have a file for a good quality and bad quality statistics. FastQC generates results in the form of a zipped and unzipped directory for each input file.

Execute the following command on the two files:

```
fastqc -f fastq bad_example.fastq
fastqc -f fastq good_example.fastq
```

View the FastQC report file of the bad data using a web browser such as firefox.

```
firefox bad_example_fastqc.html &
```

The report file will have a Basic Statistics table and various graphs and tables for different quality statistics. E.g.:

Table 2: FastQC Basic Statistics table

Filename	bad_example.fastq
File type	Conventional base calls
Encoding	Sanger / Illumina 1.9
Total Sequences	40000
Filtered Sequences	0
Sequence length	100
%GC	48

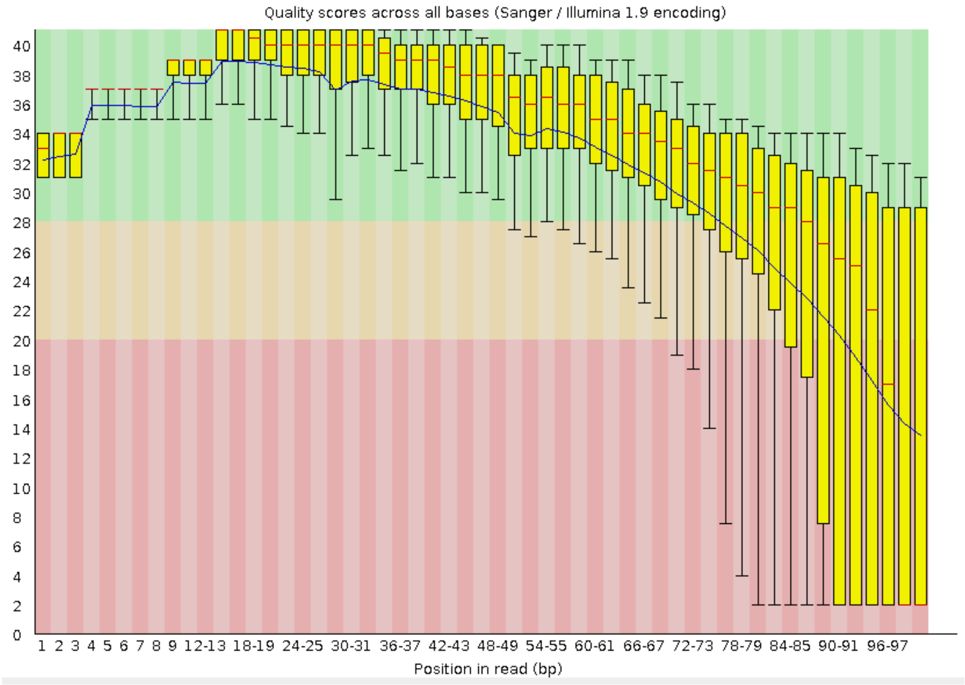


Figure 1: Per base sequence quality plot for `bad_example.fastq`.

A Phred quality score (or Q-score) expresses an error probability. In particular, it serves as a convenient and compact way to communicate very small error probabilities. The probability that base A is wrong ($P(\sim A)$) is expressed by a quality score, $Q(A)$, according to the relationship:

$$Q(A) = -10log10(P(\sim A))$$

The relationship between the quality score and error probability is demonstrated with the following table:

Table 3: Error probabilities associated with various quality (Q) values

Quality score, Q(A)	Error probability, P(∼A)	Accuracy of the base call
10	0.1	90%
20	0.01	99%
30	0.001	99.9%
40	0.0001	99.99%
50	0.00001	99.999%

How many sequences were there in your file? What is the read length? 40,000. read length=100bp

Does the quality score values vary throughout the read length? (hint: look at the 'per base sequence quality plot') Yes. Quality scores are dropping towards the end of the reads.

What is the quality score range you see? 2-40

At around which position do the scores start falling below Q20? Around 80 bp position

How can we trim the reads to filter out the low quality data? By trimming off the bases after a fixed position of the read or by trimming off bases based on the quality score.

Good Quality Data

View the FastQC report files `fastqc_report.html` to see examples of a good quality data and compare the quality plot with that of the `bad_example_fastqc`.

```
firefox good_example_fastqc.html &
```

Sequencing errors can complicate the downstream analysis, which normally requires that reads be aligned to each other (for genome assembly) or to a reference genome (for detection of mutations). Sequence reads containing errors may lead to ambiguous paths in the assembly or improper gaps. In variant analysis projects sequence reads are aligned against the reference genome. The errors in the reads may lead to more mismatches than expected from mutations alone. But if these errors can be removed or corrected, the read alignments and hence the variant detection will improve. The assemblies will also improve after pre-processing the reads with errors.

Read Trimming

Read trimming can be done in a variety of different ways. Choose a method which best suits your data. Here we are giving examples of fixed-length trimming and quality-based trimming.

Fixed Length Trimming

Low quality read ends can be trimmed using a fixed-length trimming. We will use the `fastx_trimmer` from the FASTX-Toolkit. Usage message to find out various options you can use with this tool. Type `fastx_trimmer -h` at anytime to display help.
We will now do fixed-length trimming of the `bad_example.fastq` file using the following command.

```
cd ~/QC
fastx_trimmer -h
fastx_trimmer -Q 33 -f 1 -l 80 -i bad_example.fastq -o \
    bad_example_trimmed01.fastq
```

We used the following options in the command above:

-Q 33 Indicates the input quality scores are Phred+33 encoded

-f First base to be retained in the output

-l Last base to be retained in the output

-i Input FASTQ file name

-o Output file name

Run FastQC on the trimmed file and visualise the quality scores of the trimmed file.

```
fastqc -f fastq bad_example_trimmed01.fastq
firefox bad_example_trimmed01_fastqc.html &
```

The output should look like:

Table 4: FastQC Basic Statistics table

Filename	bad_example_trimmed01.fastq
File type	Conventional base calls
Encoding	Sanger / Illumina 1.9
Total Sequences	40000
Filtered Sequences	0
Sequence length	80
%GC	48

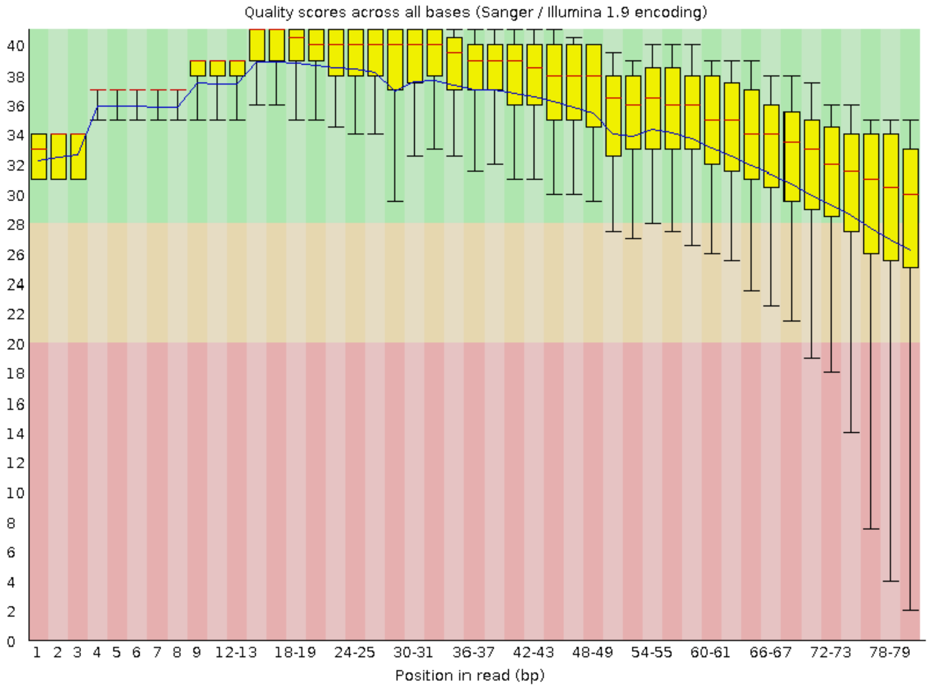


Figure 2: Per base sequence quality plot for the fixed-length trimmed `bad_example.fastq` reads.

What values would you use for `-f` if you wanted to trim off 10 bases at the 5' end of the reads? `-f 11`

Quality Based Trimming

Base call quality scores can also be used to dynamically determine the trim points for each read. A quality score threshold and minimum read length following trimming can be used to remove low quality data.

Run the following command to quality trim your data:

```
cd ~/QC
fastq_quality_trimmer -h
fastq_quality_trimmer -Q 33 -t 20 -l 50 -i bad_example.fastq -o \
    bad_example_quality_trimmed.fastq
```

-Q 33 Indicates the input quality scores are Phred+33 encoded

-t quality score cut-off

-l minimum length of reads to output

-i Input FASTQ file name

-o Output file name

Run FastQC on the quality trimmed file and visualise the quality scores.

```
fastqc -f fastq bad_example_quality_trimmed.fastq
firefox bad_example_quality_trimmed_fastqc.html &
```

The output should look like:

Table 5: FastQC Basic Statistics table

Filename	bad_example_quality_trimmed.fastq
File type	Conventional base calls
Encoding	Sanger / Illumina 1.9
Total Sequences	38976
Filtered Sequences	0
Sequence length	50-100
%GC	48

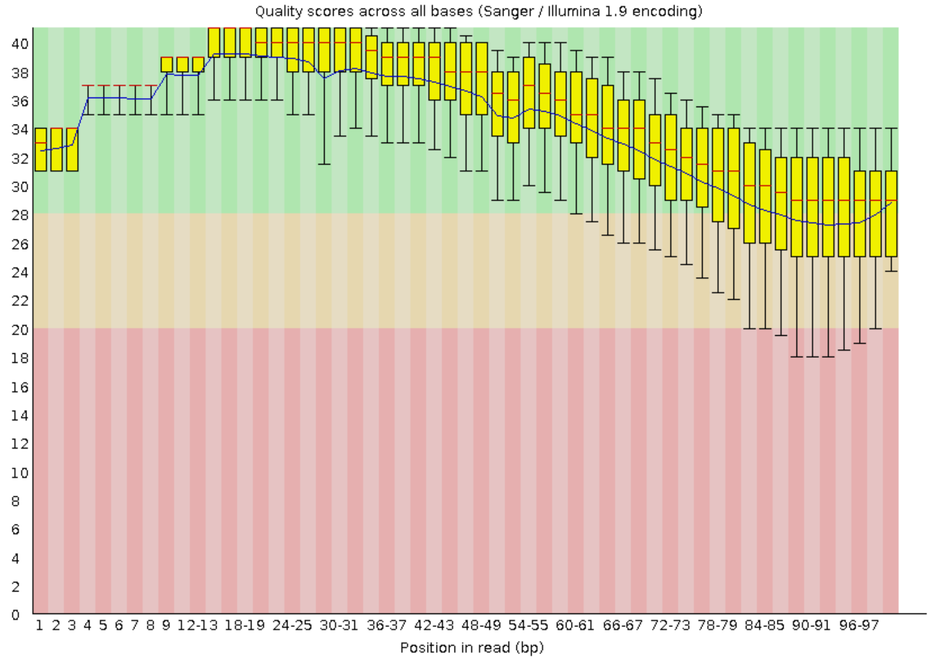


Figure 3: Per base sequence quality plot for the quality-trimmed `bad_example.fastq` reads.

How did the quality score range change with two types of trimming? Some poor quality bases (Q <20) are still present at the 3' end of the fixed-length trimmed reads. It also removes bases that are good quality.

Quality-based trimming retains the 3' ends of reads which have good quality scores.

Did the number of total reads change after two types of trimming? Quality trimming discarded >1000 reads. However, We retain a lot of maximal length reads which have good quality all the way to the ends.

What reads lengths were obtained after quality based trimming? 50-100

Reads <50 bp, following quality trimming, were discarded.

Did you observe adapter sequences in the data? No. (Hint: look at the overrepresented sequences.

How can you use -a option with fastqc ? (Hint: try fastqc -h). Adaptors can be supplied in a file for screening.

Adapter Clipping

Sometimes sequence reads may end up getting the leftover of adapters and primers used in the sequencing process. It's good practice to screen your data for these possible contamination for more sensitive alignment and assembly based analysis.
This is particularly important when read lengths can be longer than the molecules being sequenced. For example when sequencing miRNAs.

Various QC tools are available to screen and/or clip these adapter/primer sequences from your data. (e.g. FastQC, FASTX-Toolkit, cutadapt).
Here we are demonstrating `fastx_clipper` to trim a given adapter sequence.

```
cd ~/QC
fastx_clipper -h
fastx_clipper -v -Q 33 -l 20 -M 15 -a \
    GATCGGAAGAGCGGTTCAGCAGGAATGCCGAG -i bad_example.fastq -o \
    bad_example_clipped.fastq
```

An alternative tool, not installed on this system, for adapter clipping is `fastq-mcf`. A list of adapters is provided in a text file. For more information, see FastqMcf at http://code.google.com/p/ea-utils/wiki/FastqMcf.

Removing Duplicates

Duplicate reads are the ones having the same start and end coordinates. This may be the result of technical duplication (too many PCR cycles), or over-sequencing (very high fold coverage). It is very important to put the duplication level in context of your experiment. For example, duplication level in targeted or re-sequencing projects may mean something different in RNA-seq experiments. In RNA-seq experiments oversequencing is usually necessary when detecting low abundance transcripts.
The duplication level computed by FastQC is based on sequence identity at the end of reads. Another tool, Picard, determines duplicates based on identical start and end positions in SAM/BAM alignment files.

We will not cover Picard but provide the following for your information.

Picard is a suite of tools for performing many common tasks with SAM/BAM format files. For more information see the Picard website and information about the various command-line tools available:

http://picard.sourceforge.net/command-line-overview.shtml

Picard is installed on this system in `/tools/Picard/picard-default`

One of the Picard tools (MarkDuplicates) can be used to analyse and remove duplicates from the raw sequence data. The input for Picard is a sorted alignment file in BAM format. Short read aligners such as, bowtie, BWA and tophat can be used to align FASTQ files against a reference genome to generate SAM/BAM alignment format.

Interested users can use the following general command to run the MarkDuplicates tool at their leisure. You only need to provide a BAM file for the INPUT argument (not provided):

```
cd ~/QC
java -jar /tools/Picard/picard-default/MarkDuplicates.jar \
    INPUT=<alignment_file.bam> VALIDATION_STRINGENCY=LENIENT \
    OUTPUT=alignment_file.dup METRICS_FILE=alignment_file.matric \
    ASSUME_SORTED=true REMOVE_DUPLICATES=true
```

Module: Read Alignment

Primary Author(s):
Myrto Kostadima kostadim@ebi.ac.uk

Contributor(s):
Xi Li sean.li@csiro.au

Key Learning Outcomes

After completing this practical the trainee should be able to:

- Perform the simple NGS data alignment task against one interested reference data
- Interpret and manipulate the mapping output using SAMtools
- Visualise the alignment via a standard genome browser, e.g. IGV browser

Resources You'll be Using

Tools Used

Bowtie
`http://bowtie-bio.sourceforge.net/index.shtml`

Bowtie 2
`http://bowtie-bio.sourceforge.net/bowtie2/index.shtml`

Samtools
`http://picard.sourceforge.net/`

BEDTools
`http://code.google.com/p/bedtools/`

UCSC tools
`http://hgdownload.cse.ucsc.edu/admin/exe/`

IGV genome browser
`http://www.broadinstitute.org/igv/`

Useful Links

SAM Specification
`http://samtools.sourceforge.net/SAM1.pdf`

Explain SAM Flags
`http://picard.sourceforge.net/explain-flags.html`

Sources of Data

http://www.ebi.ac.uk/arrayexpress/experiments/E-GEOD-11431

Introduction

The goal of this hands-on session is to perform an unspliced alignment for a small subset of raw reads. We will align raw sequencing data to the mouse genome using Bowtie and then we will manipulate the SAM output in order to visualize the alignment on the IGV browser.

Prepare the Environment

We will use one data set in this practical, which can be found in the `ChIP-seq` directory on your desktop.

Open the Terminal.

First, go to the right folder, where the data are stored.

```
cd ~/ChIP-seq
```

The `.fastq` file that we will align is called `Oct4.fastq`. This file is based on Oct4 ChIP-seq data published by Chen *et al.* (2008). For the sake of time, we will align these reads to a single mouse chromosome.

Alignment

You already know that there are a number of competing tools for short read alignment, each with its own set of strengths, weaknesses, and caveats. Here we will try Bowtie, a widely used ultrafast, memory efficient short read aligner.

Bowtie has a number of parameters in order to perform the alignment. To view them all type

```
bowtie --help
```

Bowtie uses indexed genome for the alignment in order to keep its memory footprint small. Because of time constraints we will build the index only for one chromosome of the mouse genome. For this we need the chromosome sequence in FASTA format. This is stored in a file named `mm10`, under the subdirectory `bowtie_index`.

The indexed chromosome is generated using the command:

```
bowtie-build bowtie_index/mm10.fa bowtie_index/mm10
```

This command will output 6 files that constitute the index. These files that have the prefix `mm10` are stored in the `bowtie_index` subdirectory. To view if they files have been successfully created type:

```
ls -l bowtie_index
```

Now that the genome is indexed we can move on to the actual alignment. The first argument for `bowtie` is the basename of the index for the genome to be searched; in our case this is `mm10`. We also want to make sure that the output is in SAM format using the `-S` parameter. The last argument is the name of the FASTQ file.

Align the Oct4 reads using Bowtie:

```
bowtie bowtie_index/mm10 -S Oct4.fastq > Oct4.sam
```

The above command outputs the alignment in SAM format and stores them in the file `Oct4.sam`.

In general before you run Bowtie, you have to know what quality encoding your FASTQ files are in. The available FASTQ encodings for bowtie are:

`--phred33-quals` Input qualities are Phred+33 (default).

`--phred64-quals` Input qualities are Phred+64 (same as `--solexa1.3-quals`).

`--solexa-quals` Input qualities are from GA Pipeline ver. < 1.3.

`--solexa1.3-quals` Input qualities are from GA Pipeline ver. ≥ 1.3.

`--integer-quals` Qualities are given as space-separated integers (not ASCII).

The FASTQ files we are working with are Sanger encoded (Phred+33), which is the default for Bowtie.

Bowtie will take 2-3 minutes to align the file. This is fast compared to other aligners which sacrifice some speed to obtain higher sensitivity.

Look at the top 10 lines of the SAM file by typing:

```
head -n 10 Oct4.sam
```

Can you distinguish between the header of the SAM format and the actual alignments? The header line starts with the letter ‘@’, i.e.:

@HD	VN:1.0	SO:unsorted	
@SQ	SN:chr1	LN:197195432	
@PG	ID:Bowtie	VN:0.12.8	CL:“bowtie bowtie_index/mm10 -S Oct4.fastq”

While, the actual alignments start with read id, i.e.:

SRR002012.45	0	chr1	etc
SRR002012.48	16	chr1	etc

What kind of information does the header provide?

- @HD: Header line; VN: Format version; SO: the sort order of alignments.
- @SQ: Reference sequence information; SN: reference sequence name; LN: reference sequence length.
- @PG: Read group information; ID: Read group identifier; VN: Program version; CL: the command line that produces the alignment.

To which chromosome are the reads mapped? Chromosome 1.

Manipulate SAM output

SAM files are rather big and when dealing with a high volume of NGS data, storage space can become an issue. As we have already seen, we can convert SAM to BAM files (their binary equivalent that are not human readable) that occupy much less space.

Convert SAM to BAM using `samtools view` and store the output in the file `Oct4.bam`. You have to instruct `samtools view` that the input is in SAM format (`-S`), the output should be in BAM format (`-b`) and that you want the output to be stored in the file specified by the `-o` option:

```
samtools view -bSo Oct4.bam Oct4.sam
```

Compute summary stats for the Flag values associated with the alignments using:

```
samtools flagstat Oct4.bam
```

Visualize alignments in IGV

IGV is a stand-alone genome browser. Please check their website (http://www.broadinstitute.org/igv/) for all the formats that IGV can display. For our visualization purposes we will use the BAM and bigWig formats.

When uploading a BAM file into the genome browser, the browser will look for the index of the BAM file in the same folder where the BAM files is. The index file should have the same name as the BAM file and the suffix `.bai`. Finally, to create the index of a BAM file you need to make sure that the file is sorted according to chromosomal coordinates.

Sort alignments according to chromosomal position and store the result in the file with the prefix `Oct4.sorted`:

```
samtools sort Oct4.bam Oct4.sorted
```

Index the sorted file.

```
samtools index Oct4.sorted.bam
```

The indexing will create a file called `Oct4.sorted.bam.bai`. Note that you don't have to specify the name of the index file when running `samtools index`, it simply appends a `.bai` suffix to the input BAM file.

Another way to visualize the alignments is to convert the BAM file into a bigWig file. The bigWig format is for display of dense, continuous data and the data will be displayed as a graph. The resulting bigWig files are in an indexed binary format.

The BAM to bigWig conversion takes place in two steps. Firstly, we convert the BAM file into a bedgraph, called `Oct4.bedgraph`, using the tool `genomeCoverageBed` from BEDTools. Then we convert the bedgraph into a bigWig binary file called `Oct4.bw`, using `bedGraphToBigWig` from the UCSC tools:

```
genomeCoverageBed -bg -ibam Oct4.sorted.bam -g \
    bowtie_index/mouse.mm10.genome > Oct4.bedgraph
bedGraphToBigWig Oct4.bedgraph bowtie_index/mouse.mm10.genome Oct4.bw
```

Both of the commands above take as input a file called `mouse.mm10.genome` that is stored under the subdirectory `bowtie_index`. These genome files are tab-delimited and describe the size of the chromosomes for the organism of interest. When using the UCSC Genome Browser, Ensembl, or Galaxy, you typically indicate which species/genome build you are working with. The way you do this for BEDTools is to create a "genome" file, which simply lists the names of the chromosomes (or scaffolds, etc.) and their size (in basepairs).

BEDTools includes pre-defined genome files for human and mouse in the `genomes` subdirectory included in the BEDTools distribution.

Now we will load the data into the IGV browser for visualization. In order to launch IGV double click on the `IGV 2.3` icon on your Desktop. Ignore any warnings and when it opens you have to load the genome of interest.

On the top left of your screen choose from the drop down menu `Mus musculus (mm10)`. Then in order to load the desire files go to:

```
File > Load from File
```

On the pop up window navigate to Desktop > ChIP-seq folder and select the file `Oct4.sorted.bam`.

Repeat these steps in order to load `Oct4.bw` as well.

Select `chr1` from the drop down menu on the top left. Right click on the name of `Oct4.bw` and choose Maximum under the Windowing Function. Right click again and select Autoscale.

In order to see the aligned reads of the BAM file, you need to zoom in to a specific region. For example, look for gene `Lemd1` in the search box.

What is the main difference between the visualization of BAM and bigWig files? The actual alignment of reads that stack to a particular region can be displayed using the information stored in a BAM format. The bigWig format is for display of dense, continuous data that will be displayed in the Genome Browser as a graph.

Using the `+` button on the top right, zoom in to see more of the details of the alignments.

What do you think the different colors mean? The different color represents four nucleotides, e.g. blue is Cytidine (C), red is Thymidine (T).

Practice Makes Perfect!

In the ChIP-seq folder you will find the file `gfp.fastq`. Follow the above described analysis, from the bowtie alignment step, for this dataset as well. You will need these files for the ChIP-Seq module.

Module: ChIP-Seq

Primary Author(s):
Remco Loos, EMBL-EBI remco@ebi.ac.uk
Myrto Kostadima kostadim@ebi.ac.uk

Contributor(s):
Xi Li sean.li@csiro.au

Key Learning Outcomes

After completing this practical the trainee should be able to:

- Perform simple ChIP-Seq analysis, e.g. the detection of immuno-enriched areas using the chosen peak caller program MACS
- Visualize the peak regions through a genome browser, e.g. Ensembl, and identify the real peak regions
- Perform functional annotation and detect potential binding sites (motif) in the predicted binding regions using motif discovery tool, e.g. MEME.

Resources You'll be Using

Tools Used

MACS
`http://liulab.dfci.harvard.edu/MACS/index.html`

Ensembl
`http://www.ensembl.org`

PeakAnalyzer
`http://www.ebi.ac.uk/bertone/software`

MEME
`http://meme.ebi.edu.au/meme/tools/meme`

TOMTOM
`http://meme.ebi.edu.au/meme/tools/tomtom`

DAVID
`http://david.abcc.ncifcrf.gov`

GOstat
`http://gostat.wehi.edu.au`

Sources of Data

http://www.ebi.ac.uk/arrayexpress/experiments/E-GEOD-11431

Introduction

The goal of this hands-on session is to perform some basic tasks in the analysis of ChIP-seq data. In fact, you already performed the first step, alignment of the reads to the genome, in the previous session. We start from the aligned reads and we will find immuno-enriched areas using the peak caller MACS. We will visualize the identified regions in a genome browser and perform functional annotation and motif analysis on the predicted binding regions.

Prepare the Environment

The material for this practical can be found in the `ChIP-seq` directory on your desktop. This directory also contains an electronic version of this document, which can be useful to copy and paste commands. Please make sure that this directory also contains the SAM/BAM files you produced during the alignment practical.

If you didn't have time to align the control file called `gfp.fastq` during the alignment practical, please do it now. Follow the same steps, from the bowtie alignment step, as for the `Oct4.fastq` file.

In ChIP-seq analysis (unlike in other applications such as RNA-seq) it can be useful to exclude all reads that map to more than one location in the genome. When using Bowtie, this can be done using the `-m 1` option, which tells it to report only unique matches (See `bowtie --help` for more details).

Open the Terminal and go to the `ChIP-seq` directory:

```
cd ~/ChIP-seq
```

Finding enriched areas using MACS

MACS stands for Model based analysis of ChIP-seq. It was designed for identifying transcription factor binding sites. MACS captures the influence of genome complexity to evaluate the significance of enriched ChIP regions, and improves the spatial resolution of binding sites through combining the information of both sequencing tag position and orientation. MACS can be easily used for ChIP-Seq data alone, or with a control sample to increase specificity.

Consult the MACS help file to see the options and parameters:

```
macs --help
```

The input for MACS can be in ELAND, BED, SAM, BAM or BOWTIE formats (you just have to set the `--format` option).

Options that you will have to use include:

`-t` To indicate the input ChIP file.

`-c` To indicate the name of the control file.

`--format` To change the file format. The default format is bed.

`--name` To set the name of the output files.

`--gsize` This is the mappable genome size. With the read length we have, 70% of the genome is a fair estimation. Since in this analysis we include only reads from chromosome 1 (197Mbases), we will use a `--gsize` of 138Mbases (70% of 197Mbases).

`--tsize` To set the read length (look at the FASTQ files to check the length).

`--wig` To generate signal wig files for viewing in a genome browser. Since this process is time consuming, it is recommended to run MACS first with this flag off, and once you decide on the values of the parameters, run MACS again with this flag on.

`--diag` To generate a saturation table, which gives an indication whether the sequenced reads give a reliable representation of the possible peaks.

Now run macs using the following command:

```
macs -t <Oct4_aligned_bam_file> -c <gfp_aligned_bam_file> --format=BAM \
    --name=Oct4 --gsize=138000000 --tsize=26 --diag --wig
```

Look at the output saturation table (`Oct4_diag.xls`). To open this file file, right-click on it and choose “Open with” and select LibreOffice. Do you think that more sequencing is necessary?

Open the Excel peak file and view the peak details. Note that the number of tags (column 6) refers to the number of reads in the whole peak region and not the peak height.

Viewing results with the Ensembl genome browser

It is often instructive to look at your data in a genome browser. Before, we used IGV, a stand-alone browser, which has the advantage of being installed locally and providing fast access. Web-based genome browsers, like Ensembl or the UCSC browser, are slower, but provide more functionality. They do not only allow for more polished and flexible visualisation, but also provide easy access to a wealth of annotations and external data sources. This makes it straightforward to relate your data with information about repeat regions, known genes, epigenetic features or areas of cross-species conservation, to name just a few. As such, they are useful tools for exploratory analysis.

They will allow you to get a 'feel' for the data, as well as detecting abnormalities and problems. Also, exploring the data in such a way may give you ideas for further analyses.

Launch a web browser and go to the Ensembl website at `http://www.ensembl.org/index.html`

Choose the genome of interest (in this case, mouse) on the left side of the page, browse to any location in the genome or click one of the demo links provided on the web page.

Click on the **Manage your data** link on the left, then choose **Add your data** in the **Personal Data** tab.

Wig files are large so are inconvenient for uploading directly to the Ensemble Genome browser. Instead, we will convert it to an indexed binary format and put this into a web accessible place such as on a HTTP, HTTPS, or FTP server. This makes all the browsing process much faster. Detailed instructions for generating a bigWig from a wig type file can be found at:

`http://genome.ucsc.edu/goldenPath/help/bigWig.html`.

We have generated bigWig files in advance for you to upload to the Ensembl browser. They are at the following URL: `http://www.ebi.ac.uk/~remco/ChIP-Seq_course/Oct4.bw`

To visualise the data:

- Paste the location above in the field File URL.
- Choose data format bigWig.
- Choose some informative name and in the next window choose the colour of your preference.
- Click **Save** and close the window to return to the genome browser.

Repeat the process for the gfp control sample, located at:

`http://www.ebi.ac.uk/~remco/ChIP-Seq_course/gfp.bw`.

After uploading, to make sure your data is visible:

- Switch to the **Configure Region Image** tab
- Click **Your data** in the left panel
- Choose each of the uploaded *.bw files to confirm the **Wiggle plot** in **Change track style** pop up menu has been choosen.
- Closing the window will save these changes.

Go to a region on chromosome 1 (e.g. `1:34823162-35323161`), and zoom in and out to view the signal and peak regions. Be aware that the y-axis of each track is auto-scaled independently of each other, so bigger-looking peaks may not actually be bigger! Always look at the values on the left hand side axis.

What can you say about the profile of Oct4 peaks in this region? There are no significant Oct4 peaks over the selected region.

Compare it with H3K4me3 histone modification wig file we have generated at `http://www.ebi.ac.uk/~remco/ChIP-Seq_course/H3K4me3.bw`. H3K4me3 has a region that contains relatively high peaks than Oct4.

Jump to `1:36066594-36079728` for a sample peak. Do you think H3K4me3 peaks regions contain one or more modification sites? What about Oct4? Yes. There are roughly three peaks, which indicate the possibility of having more than one modification sites in this region.

For Oct4, no peak can be observed.

MACS generates its peak files in a file format called bed file. This is a simple text format containing genomic locations, specified by chromosome, begin and end positions, and some more optional information.

See `http://genome.ucsc.edu/FAQ/FAQformat.html#format1` for details.

Bed files can also be uploaded to the Ensembl browser.

Try uploading the peak file generated by MACS to Ensembl. Find the first peak in the file (use the `head` command to view the beginning of the bed file), and see if the peak looks convincing to you.

Annotation: From peaks to biological interpretation

In order to biologically interpret the results of ChIP-seq experiments, it is usually recommended to look at the genes and other annotated elements that are located in proximity to the identified enriched regions. This can be easily done using PeakAnalyzer.

Go to the PeakAnalyzer tool directory:

```
cd /tools/PeakAnalyzer/peakanalyzer-default
```

Launch the PeakAnalyzer program by typing:

```
java -jar PeakAnalyzer.jar &
```

The first window allows you to choose between the split application (which we will try next) and peak annotation. Choose the peak annotation option and click **Next**.

We would like to find the closest downstream genes to each peak, and the genes that overlap with the peak region. For that purpose you should choose the **NDG** option and click **Next**.

Fill in the location of the peak file `Oct4_peaks.bed`, and choose the mouse GTF as the annotation file. You don't have to define a symbol file since gene symbols are included in the GTF file.

Choose the output directory and run the program.

When the program has finished running, you will have the option to generate plots, by pressing the **Generate plots** button. This is only possible if R is installed on your computer, as it is on this system. A PDF file with the plots will be generated in the output folder. You could generate similar plots with Excel using the output files that were generated by PeakAnalyzer.

This list of closest downstream genes (contained in the file `Oct4_peaks.ndg.bed`) can be the basis of further analysis. For instance, you could look at the Gene Ontology terms associated with these genes to get an idea of the biological processes that may be affected. Web-based tools like DAVID (http://david.abcc.ncifcrf.gov) or GOstat (http://gostat.wehi.edu.au) take a list of genes and return the enriched GO categories.

We can pull out Ensemble Transcript IDs from the `Oct4_peaks.ndg.bed` file and write them to another file ready for use with DAVID or GOstat:

```
cut -f 5 Oct4_peaks.ndg.bed | sed '1 d' > Oct4_peaks.ndg.tid
```

Motif analysis

It is often interesting to find out whether we can associate identified the binding sites with a sequence pattern or motif. We will use MEME for motif analysis. The input for MEME should be a file in FASTA format containing the sequences of interest. In our case, these are the sequences of the identified peaks that probably contain Oct4 binding sites.

Since many peak-finding tools merge overlapping areas of enrichment, the resulting peaks tend to be much wider than the actual binding sites. Sub-dividing the enriched areas by accurately partitioning enriched loci into a finer-resolution set of individual binding sites, and fetching sequences from the summit region where binding motifs are most likely to appear enhances the quality of the motif analysis. Sub-peak summit sequences can be retrieved directly from the Ensembl database using PeakAnalyzer.

If you have closed the PeakAnalyzer running window, open it again. If it is still open, just go back to the first window.

Choose the split peaks utility and click **Next**. The input consists of files generated by most peak-finding tools: a file containing the chromosome, start and end locations of the enriched regions, and a `.wig` signal file describing the size and shape of each peak. Fill in the location of both files `Oct4_peaks.bed` and the wig file generated by MACS, which is under the `Oct4_MACS_wiggle/treat/` directory, check the option to **Fetch subpeak sequences** and click **Next**.

In the next window you have to set some parameters for splitting the peaks.

`Separation float` Keep the default value. This value determines when a peak will be separated into sub-peaks. This is the ratio between a valley and its neighbouring summit (the lower summit of the two). For example, if you set this height to be `0.5`, two sub-peaks will be separated only if the height of the lower summit is twice the height of the valley.

`Minimum height` Set this to be `5`. Only sub-peaks with at least this number of tags in their summit region will be separated. Change the organism name from the default human to mouse and run the program.

Since the program has to read large wig files, it will take a few minutes to run. Once the run is finished, two output files will be produced. The first describes the location of the sub-peaks, and the second is a FASTA file containing 300 sequences of length 61 bases, taken from the summit regions of the highest sub-peaks.

Open a web bowser and go to the MEME website at `http://meme.ebi.edu.au/meme/tools/meme`, and fill in the necessary details, such as:

- Your e-mail address

- The sub-peaks FASTA file `Oct4_peaks.bestSubPeaks.fa` (will need uploading), or just paste in the sequences.
- The number of motifs we expect to find (1 per sequence)
- The width of the desired motif (between 6 to 20)
- The maximum number of motifs to find (3 by default). For Oct4 one classical motif is known.

You will receive the results by e-mail. This usually doesn't take more than a few minutes.

Open the e-mail and click on the link that leads to the HTML results page.

Scroll down until you see the first motif logo. We would like to know if this motif is similar to any other known motif. We will use TOMTOM for this. Scroll down until you see the option **Submit this motif to**. Click the TOMTOM button to compare to known motifs in motif databases, and on the new page choose to compare your motif to those in the JASPAR and UniPROBE database.

Which motif was found to be the most similar to your motif? Sox2

Reference

Chen, X et al.: Integration of external signaling pathways with the core transcriptional network in embryonic stem cells. Cell 133:6, 1106-17 (2008).

Module: RNA-Seq

Primary Author(s):
Myrto Kostadima, EMBL-EBI kostadmi@ebi.ac.uk
Remco Loos, EMBL-EBI remco@ebi.ac.uk
Sonika Tyagi, AGRF sonika.tyagi@agrf.org.au

Contributor(s):
Nathan S. Watson-Haigh nathan.watson-haigh@awri.com.au
Susan M Corley s.corley@unsw.edu.au

Key Learning Outcomes

After completing this practical the trainee should be able to:

- Understand and perform a simple RNA-Seq analysis workflow.
- Perform gapped alignments to an indexed reference genome using TopHat.
- Perform transcript assembly using Cufflinks.
- Visualize transcript alignments and annotation in a genome browser such as IGV.
- Be able to identify differential gene expression between two experimental conditions.
- Be familiar with R environment and be able to run R based RNA-seq packages.

Resources You'll be Using

Tools Used

Tophat
`http://tophat.cbcb.umd.edu/`

Cufflinks
`http://cufflinks.cbcb.umd.edu/`

Samtools
`http://samtools.sourceforge.net/`

BEDTools
`http://code.google.com/p/bedtools/`

UCSC tools
`http://hgdownload.cse.ucsc.edu/admin/exe/`

IGV
`http://www.broadinstitute.org/igv/`

DAVID Functional Analysis
`http://david.abcc.ncifcrf.gov/`

edgeR pakcage
`http://http://www.bioconductor.org/packages/release/bioc/html/edgeR.html/`

CummeRbund manual
`http://www.bioconductor.org/packages/release/bioc/vignettes/cummeRbund/inst/doc/cummeRbund-manual.pdf`

Sources of Data

http://www.ebi.ac.uk/ena/data/view/ERR022484
http://www.ebi.ac.uk/ena/data/view/ERR022485
http://www.pnas.org/content/suppl/2008/12/16/0807121105.DCSupplemental

Introduction

The goal of this hands-on session is to perform some basic tasks in the downstream analysis of RNA-seq data. We will start from RNA-seq data aligned to the zebrafish genome using Tophat.

We will perform transcriptome reconstruction using Cufflinks and we will compare the gene expression between two different conditions in order to identify differentially expressed genes.

In the second part of the tutorial we will also be demonstrating usage of R-based packages to perform differential expression analysis. We will be using edgeR for the demonstration. The gene/tag counts generated from the alignment are used as input for edgeR.

Prepare the Environment

We will use a dataset derived from sequencing of mRNA from *Danio rerio* embryos in two different developmental stages. Sequencing was performed on the Illumina platform and generated 76bp paired-end sequence data using polyA selected RNA. Due to the time constraints of the practical we will only use a subset of the reads.

The data files are contained in the subdirectory called `data` and are the following:

`2cells_1.fastq` and `2cells_2.fastq`
These files are based on RNA-seq data of a 2-cell zebrafish embryo

`6h_1.fastq` and `6h_2.fastq`
These files are based on RNA-seq data of zebrafish embryos 6h post fertilization

Open the Terminal and go to the `RNA-seq` working directory:

```
cd ~/RNA-seq/
```

All commands entered into the terminal for this tutorial should be from within the ∼**`/RNA-seq`** directory.

Check that the `data` directory contains the above-mentioned files by typing:

```
ls data
```

Alignment

There are numerous tools for performing short read alignment and the choice of aligner should be carefully made according to the analysis goals/requirements. Here we will use Tophat, a widely used ultrafast aligner that performs spliced alignments.

Tophat is based on the Bowtie aligner and uses an indexed genome for the alignment to speed up the alignment and keep its memory footprint small. The the index for the *Danio rerio* genome has been created for you.

The command to create an index is as follows. You DO NOT need to run this command yourself - we have done this for you.

```
bowtie-build genome/Danio_rerio.Zv9.66.dna.fa genome/ZV9
```

Tophat has a number of parameters in order to perform the alignment. To view them all type:

```
tophat --help
```

The general format of the tophat command is:

```
tophat [options]* <index_base> <reads_1> <reads_2>
```

Where the last two arguments are the `.fastq` files of the paired end reads, and the argument before is the basename of the indexed genome.

The quality values in the FASTQ files used in this hands-on session are Phred+33 encoded. We explicitly tell tophat of this fact by using the command line argument `--solexa-quals`.

You can look at the first few reads in the file `data/2cells_1.fastq` with:

```
head -n 20 data/2cells_1.fastq
```

Some other parameters that we are going to use to run Tophat are listed below:

-g Maximum number of multihits allowed. Short reads are likely to map to more than one location in the genome even though these reads can have originated from only one of these regions. In RNA-seq we allow for a limited number of multihits, and in this case we ask Tophat to report only reads that map at most onto 2 different loci.

--library-type Before performing any type of RNA-seq analysis you need to know a few things about the library preparation. Was it done using a strand-specific protocol or not? If yes, which strand? In our data the protocol was NOT strand specific.

`-j` Improve spliced alignment by providing Tophat with annotated splice junctions. Pre-existing genome annotation is an advantage when analysing RNA-seq data. This file contains the coordinates of annotated splice junctions from Ensembl. These are stored under the sub-directory `annotation` in a file called `ZV9.spliceSites`.

`-o` This specifies in which subdirectory Tophat should save the output files. Given that for every run the name of the output files is the same, we specify different directories for each run.

It takes some time (approx. 20 min) to perform tophat spliced alignments, even for this subset of reads. Therefore, we have pre-aligned the `2cells` data for you using the following command:

You DO NOT need to run this command yourself - we have done this for you.

```
tophat --solexa-quals -g 2 --library-type fr-unstranded -j \
    annotation/Danio_rerio.Zv9.66.spliceSites -o tophat/ZV9_2cells \
    genome/ZV9 data/2cells_1.fastq data/2cells_2.fastq
```

Align the `6h` data yourself using the following command:

```
# Takes approx. 20mins
tophat --solexa-quals -g 2 --library-type fr-unstranded -j \
    annotation/Danio_rerio.Zv9.66.spliceSites -o tophat/ZV9_6h \
    genome/ZV9 data/6h_1.fastq data/6h_2.fastq
```

The `6h` read alignment will take approx. 20 min to complete. Therefore, we'll take a look at some of the files, generated by tophat, for the pre-computed `2cells` data.

Alignment Visualisation in IGV

The Integrative Genomics Viewer (IGV) is able to provide a visualisation of read alignments given a reference sequence and a BAM file. We'll visualise the information contained in the `accepted_hits.bam` and `junctions.bed` files for the pre-computed `2cells` data. The former, contains the tophat sliced alignments of the reads to the reference while the latter stores the coordinates of the splice junctions present in the data set.

Open the `RNA-seq` directory on your Desktop and double-click the `tophat` subdirectory and then the `ZV9_2cells` directory.

1. Launch IGV by double-clicking the "IGV 2.3.*" icon on the Desktop (ignore any warnings that you may get as it opens). *NOTE: IGV may take several minutes to load for the first time, please be patient.*

2. Choose "Zebrafish (Zv9)" from the drop-down box in the top left of the IGV window. Else you can also load the genome fasta file.

3. Load the `accepted_hits.sorted.bam` file by clicking the "File" menu, selecting "Load from File" and navigating to the `Desktop/RNA-seq/tophat/ZV9_2cells` directory.

4. Rename the track by right-clicking on its name and choosing "Rename Track". Give it a meaningful name like "2cells BAM".

5. Load the `junctions.bed` from the same directory and rename the track "2cells Junctions BED".

6. Load the Ensembl annotations file `Danio_rerio.Zv9.66.gtf` stored in the `RNA-seq/annotation` directory.

7. Navigate to a region on chromosome 12 by typing `chr12:20,270,921-20,300,943` into the search box at the top of the IGV window.

Keep zooming to view the bam file alignments

Some useful IGV manuals can be found below

http://www.broadinstitute.org/software/igv/interpreting_insert_size
http://www.broadinstitute.org/software/igv/alignmentdata

Can you identify the splice junctions from the BAM file? Slice junctions can be identified in the alignment BAM files. These are the aligned RNA-Seq reads that have skipped-bases from the reference genome (most likely introns).

Are the junctions annotated for `CBY1` consistent with the annotation? Read alignment supports an extended length in exon 5 to the gene model (cby1-001)

Are all annotated genes, from both RefSeq and Ensembl, expressed? No BX000473.1-201 is not expressed

Once tophat finishes aligning the 6h data you will need to sort the alignments found in the BAM file and then index the sorted BAM file.

```
samtools sort tophat/ZV9_6h/accepted_hits.bam \
    tophat/ZV9_6h/accepted_hits.sorted
samtools index tophat/ZV9_6h/accepted_hits.sorted.bam
```

Load the sorted BAM file into IGV, as described previously, and rename the track appropriately.

Isoform Expression and Transcriptome Assembly

There are a number of tools that perform reconstruction of the transcriptome and for this workshop we are going to use Cufflinks. Cufflinks can do transcriptome assembly either *ab initio* or using a reference annotation. It also quantifies the isoform expression in Fragments Per Kilobase of exon per Million fragments mapped (FPKM).

Cufflinks has a number of parameters in order to perform transcriptome assembly and quantification. To view them all type:

```
cufflinks --help
```

We aim to reconstruct the transcriptome for both samples by using the Ensembl annotation both strictly and as a guide. In the first case Cufflinks will only report isoforms that are included in the annotation, while in the latter case it will report novel isoforms as well.

The Ensembl annotation for *Danio rerio* is available in `annotation/Danio_rerio.Zv9.66.gtf`.

The general format of the `cufflinks` command is:

```
cufflinks [options]* <aligned_reads.(sam|bam)>
```

Where the input is the aligned reads (either in SAM or BAM format).

Some of the available parameters for Cufflinks that we are going to use to run Cufflinks are listed below:

`-o` Output directory.

`-G` Tells Cufflinks to use the supplied GTF annotations strictly in order to estimate isoform annotation.

`-b` Instructs Cufflinks to run a bias detection and correction algorithm which can significantly improve accuracy of transcript abundance estimates. To do this Cufflinks requires a multi-fasta file with the genomic sequences against which we have aligned the reads.

`-u` Tells Cufflinks to do an initial estimation procedure to more accurately weight reads mapping to multiple locations in the genome (multi-hits).

`--library-type` Before performing any type of RNA-seq analysis you need to know a few things about the library preparation. Was it done using a strand-specific protocol or not? If yes, which strand? In our data the protocol was NOT strand specific.

Perform transcriptome assembly, strictly using the supplied GTF annotations, for the `2cells` and `6h` data using cufflinks:

```
# 2cells data (takes approx. 5mins):
cufflinks -o cufflinks/ZV9_2cells_gtf -G \
    annotation/Danio_rerio.Zv9.66.gtf -b \
    genome/Danio_rerio.Zv9.66.dna.fa -u --library-type fr-unstranded \
    tophat/ZV9_2cells/accepted_hits.bam
# 6h data (takes approx. 5mins):
cufflinks -o cufflinks/ZV9_6h_gtf -G annotation/Danio_rerio.Zv9.66.gtf \
    -b genome/Danio_rerio.Zv9.66.dna.fa -u --library-type fr-unstranded \
    tophat/ZV9_6h/accepted_hits.bam
```

Cufflinks generates several files in the specified output directory. Here's a short description of these files:

`genes.fpkm_tracking` Contains the estimated gene-level expression values.

`isoforms.fpkm_tracking` Contains the estimated isoform-level expression values.

`skipped.gtf` Contains loci skipped as a result of exceeding the maximum number of fragments.

`transcripts.gtf` This GTF file contains Cufflinks' assembled isoforms.

The complete documentation can be found at: http://cufflinks.cbcb.umd.edu/manual.html#cufflinks_output

So far we have forced cufflinks, by using the `-G` option, to strictly use the GTF annotations provided and thus novel transcripts will not be reported. We can get cufflinks to perform a GTF-guided transcriptome assembly by using the `-g` option instead. Thus, novel transcripts will be reported.

GTF-guided transcriptome assembly is more computationally intensive than strictly using the GTF annotations. Therefore, we have pre-computed these GTF-guided assemblies for you and have placed the results under subdirectories:

`cufflinks/ZV9_2cells_gtf_guided` and `cufflinks/ZV9_6h_gft_guided`.

You DO NOT need to run these commands. We provide them so you know how we generated the the GTF-guided transcriptome assemblies:

```
# 2cells guided transcriptome assembly (takes approx. 30mins):
cufflinks -o cufflinks/ZV9_2cells_gtf_guided -g \
    annotation/Danio_rerio.Zv9.66.gtf -b \
    genome/Danio_rerio.Zv9.66.dna.fa -u --library-type fr-unstranded \
    tophat/ZV9_2cells/accepted_hits.bam
# 6h guided transcriptome assembly (takes approx. 30mins):
cufflinks -o cufflinks/ZV9_6h_gtf_guided -g \
    annotation/Danio_rerio.Zv9.66.gtf -b \
    genome/Danio_rerio.Zv9.66.dna.fa -u --library-type fr-unstranded \
    tophat/ZV9_6h/accepted_hits.bam
```

1. Go back to IGV and load the pre-computed, GTF-guided transcriptome assembly for the `2cells` data (`cufflinks/ZV9_2cells_gtf_guided/transcripts.gtf`).
2. Rename the track as "2cells GTF-Guided Transcripts".
3. In the search box type `ENSDART00000082297` in order for the browser to zoom in to the gene of interest.

Do you observe any difference between the Ensembl GTF annotations and the GTF-guided transcripts assembled by cufflinks (the "2cells GTF-Guided Transcripts" track)? Yes. It appears that the Ensembl annotations may have truncated the last exon. However, our data also doesn't contain reads that span between the last two exons.

Differential Expression

One of the stand-alone tools that perform differential expression analysis is Cuffdiff. We use this tool to compare between two conditions; for example different conditions could be control and disease, or wild-type and mutant, or various developmental stages.

In our case we want to identify genes that are differentially expressed between two developmental stages; a `2cells` embryo and `6h` post fertilization.

The general format of the cuffdiff command is:

```
cuffdiff [options]* <transcripts.gtf> \
    <sample1_replicate1.sam[,...,sample1_replicateM]> \
    <sample2_replicate1.sam[,...,sample2_replicateM.sam]>
```

Where the input includes a `transcripts.gtf` file, which is an annotation file of the genome of interest or the cufflinks assembled transcripts, and the aligned reads (either in SAM or BAM format) for the conditions. Some of the Cufflinks options that we will use to run the program are:

`-o` Output directory.

`-L` Labels for the different conditions

`-T` Tells Cuffdiff that the reads are from a time series experiment.

`-b` Instructs Cufflinks to run a bias detection and correction algorithm which can significantly improve accuracy of transcript abundance estimates. To do this Cufflinks requires a multi-fasta file with the genomic sequences against which we have aligned the reads.

`-u` Tells Cufflinks to do an initial estimation procedure to more accurately weight reads mapping to multiple locations in the genome (multi-hits).

`--library-type` Before performing any type of RNA-seq analysis you need to know a few things about the library preparation. Was it done using a strand-specific protocol or not? If yes, which strand? In our data the protocol was NOT strand specific.

`-C` Biological replicates and multiple group contrast can be defined here

Run cuffdiff on the tophat generated BAM files for the 2cells vs. 6h data sets:

```
cuffdiff -o cuffdiff/ -L ZV9_2cells,ZV9_6h -T -b \
    genome/Danio_rerio.Zv9.66.dna.fa -u --library-type fr-unstranded \
    annotation/Danio_rerio.Zv9.66.gtf \
    tophat/ZV9_2cells/accepted_hits.bam tophat/ZV9_6h/accepted_hits.bam
```

We are interested in the differential expression at the gene level. The results are reported by Cuffdiff in the file `cuffdiff/gene_exp.diff`. Look at the first few lines of the file using the following command:

```
head -n 20 cuffdiff/gene_exp.diff
```

We would like to see which are the most significantly differentially expressed genes. Therefore we will sort the above file according to the q value (corrected p value for multiple testing). The result will be stored in a different file called `gene_exp_qval.sorted.diff`.

```
sort -t$'\t' -g -k 13 cuffdiff/gene_exp.diff > \
    cuffdiff/gene_exp_qval.sorted.diff
```

Look again at the first few lines of the sorted file by typing:

```
head -n 20 cuffdiff/gene_exp_qval.sorted.diff
```

Copy an Ensembl transcript identifier from the first two columns for one of these genes (e.g. `ENSDARG00000077178`). Now go back to the IGV browser and paste it in the search box.

What are the various outputs generated by cuffdiff? Hint: Please refer to the `Cuffdiff output` section of the cufflinks manual online.

Do you see any difference in the read coverage between the `2cells` and `6h` conditions that might have given rise to this transcript being called as differentially expressed?

The coverage on the Ensembl browser is based on raw reads and no normalisation has taken place contrary to the FPKM values.

The read coverage of this transcript (`ENSDARG00000077178`) in the 2cells data set is much higher than in the 6h data set.

`Cuffquant` utility from the cufflinks suite can be used to generate the count files to be used with count based differential analysis methods such as, `edgeR` and `Deseq`.

Visualising the CuffDiff expression analysis

We will use an R-Bioconductor package called `cummeRbund` to visualise, manipulate and explore Cufflinks RNA-seq output. We will load an R environment and look at few quick tips to generate simple graphical output of the cufflinks analysis we have just run.

`CummeRbund` takes the cuffdiff output and populates a SQLite database with various type of output generated by cuffdiff e.g, genes, transcripts, transcription start site, isoforms and CDS regions. The data from this database can be accessed and processed easily. This package comes with a number of in-built plotting functions that are commonly used for visualising the expression data. We strongly recommend reading through the bioconductor manual and user guide of CummeRbund to learn about functionality of the tool. The reference is provided in the resource section.

Prepare the environment. Go to the `cuffdiff` output folder and copy the transcripts file there.

```
cd ~/RNA-seq/cuffdiff
cp ~/RNA-seq/annotation/Danio_rerio.Zv9.66.gtf ~/RNA-seq/cuffdiff
ls -l
```

Load the R environment

```
R (press enter)
```

Load the require R package.

```
library(cummeRbund)
```

Read in the cuffdiff output

```
cuff<-readCufflinks(dir="/home/trainee/Desktop/RNA-seq/cuffdiff", \
gtfFile='Danio_rerio.Zv9.66.gtf',genome="Zv9", rebuild=T)
```

Assess the distribution of FPKM scores across samples

```
pdf(file = "SCV.pdf", height = 6, width = 6)
dens<-csDensity(genes(cuff))
dens
dev.off()
```

Box plots of the FPKM values for each samples

```
pdf(file = "BoxP.pdf", height = 6, width = 6)
b<-csBoxplot(genes(cuff))
b
dev.off()
```

Accessing the data

```
sigGeneIds<-getSig(cuff,alpha=0.05,level="genes")
head(sigGeneIds)
sigGenes<-getGenes(cuff,sigGeneIds)
sigGenes
head(fpkm(sigGenes))
head(fpkm(isoforms(sigGenes)))
```

Plotting a heatmap of the differentially expressed genes

```
pdf(file = "heatmap.pdf", height = 6, width = 6)
h<-csHeatmap(sigGenes,cluster="both")
h
dev.off()
```

What options would you use to draw a density or boxplot for different replicates if available ? (Hint: look at the manual at Bioconductor website)

```
densRep<-csDensity(genes(cuff),replicates=T)
brep<-csBoxplot(genes(cuff),replicates=T)
```

How many differentially expressed genes did you observe? type 'summary(sigGenes)' on the R prompt to see.

Functional Annotation of Differentially Expressed Genes

After you have performed the differential expression analysis you are interested in identifying if there is any functionality enrichment for your differentially expressed genes. On your Desktop click:

```
Applications >> Internet >> Firefox Web Browser
```

And go to the following URL: http://david.abcc.ncifcrf.gov/ On the left side click on Functional Annotation. Then click on the Upload tab. Under the section Choose from File, click Choose File and navigate to the `cuffdiff` directory. Select the file called `globalDiffExprs_Genes_qval.01_top100.tab`. Under Step 2 select ENSEMBL_GENE_ID from the drop-down menu. Finally select Gene list and then press Submit List. Click on Gene Ontology and then click on the CHART button of the GOTERM_BP_ALL item.

Do these categories make sense given the samples we're studying? Developmental Biology

Browse around DAVID website and check what other information are available. Cellular component, Molecular function, Biological Processes, Tissue expression, Pathways, Literature, Protein domains

Differential Gene Expression Analysis using edgeR

The example we are working through today follows a case Study set out in the edgeR Users Guide (4.3 Androgen-treated prostate cancer cells (RNA-Seq, two groups) which is based on an experiment conducted by Li et al. (2008, Proc Natl Acad Sci USA, 105, 20179-84).

The researches used a prostate cancer cell line (LNCaP cells). These cells are sensitive to stimulation by male hormones (androgens). Three replicate RNA samples were collected from LNCaP cells treated with an androgen hormone (DHT). Four replicates were collected from cells treated with an inactive compound. Each of the seven samples was run on a lane (7 lanes) of an Illumina flow cell to produce 35 bp reads. The experimental design was therefore:

Table 6: Experimental design

Lane	**Treatment**	**Label**
1	Control	Con1
2	Control	Con2
3	Control	Con3
4	Control	Con4
5	DHT	DHT1
6	DHT	DHT2
7	DHT	DHT3

Prepare the environment and load R:

```
cd ~/RNA-seq/edgeR
R (press enter)
```

Once on the R prompt. Load libraries:

```
library(edgeR)
library(biomaRt)
library(gplots)
```

Read in count table and experimental design:

```
data <- read.delim("pnas_expression.txt", row.names=1, header=T)
targets <- read.delim("Targets.txt", header=T)
```

Create DGEList object:

```
y <- DGEList(counts=data[,1:7], group=targets$Treatment)
```

Change the column names of the object to align with treatment:

```
colnames(y) <- targets$Label
```

Check the dimensions of the object:

```
dim(y)
```

We see we have 37435 rows (i.e. genes) and 7 columns (samples).

Now we will filter out genes with low counts by only keeping those rows where the count per million (cpm) is at least 1 in at least three samples:

```
keep <-rowSums( cpm(y)>1) >=3
y <- y[keep, ]
```

How many rows (genes) are retained now dim(y) would give you 16494

How many genes were filtered out? do 37435-16494.

We will now perform normalization to take account of different library size:

```
y<-calcNormFactors(y)
```

We will check the calculated normalization factors:

```
y$samples
```

Lets have a look at whether the samples cluster by condition. (You should produce a plot as shown in Figure 4):

```
plotMDS(y)
```

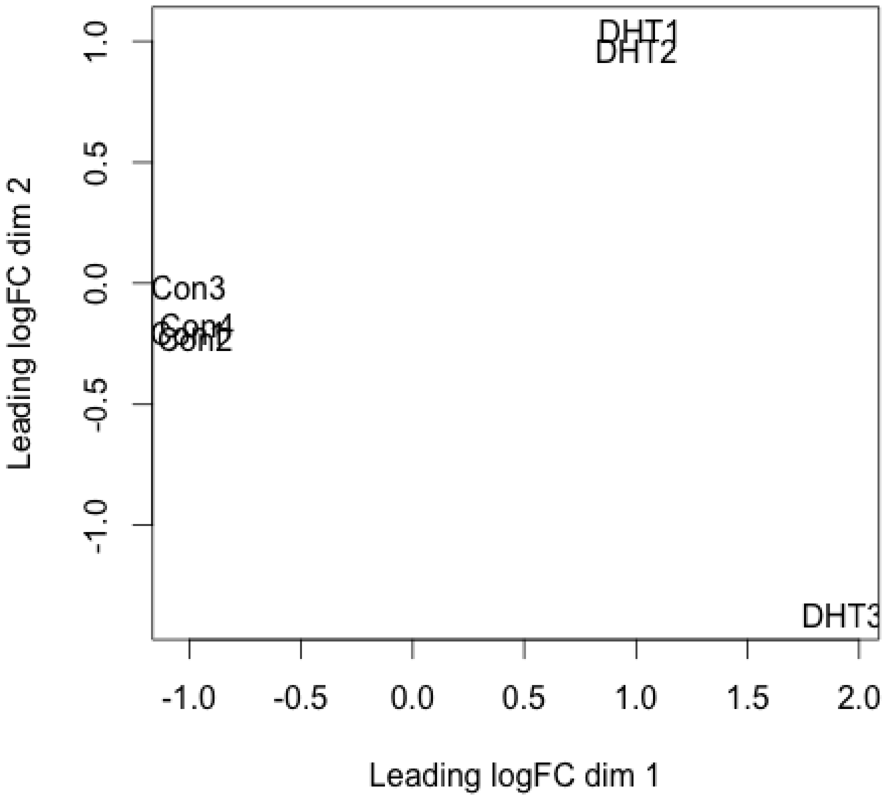


Figure 4: Visualization of sample clustering

We now estimate common and gene-specific dispersion:

```
y <- estimateCommonDisp(y)
y <- estimateTagwiseDisp(y)
```

We will plot the tagwise dispersion and the common dispersion (You should obtain a plot as shown in the Figure 5):

```
plotBCV(y)
```

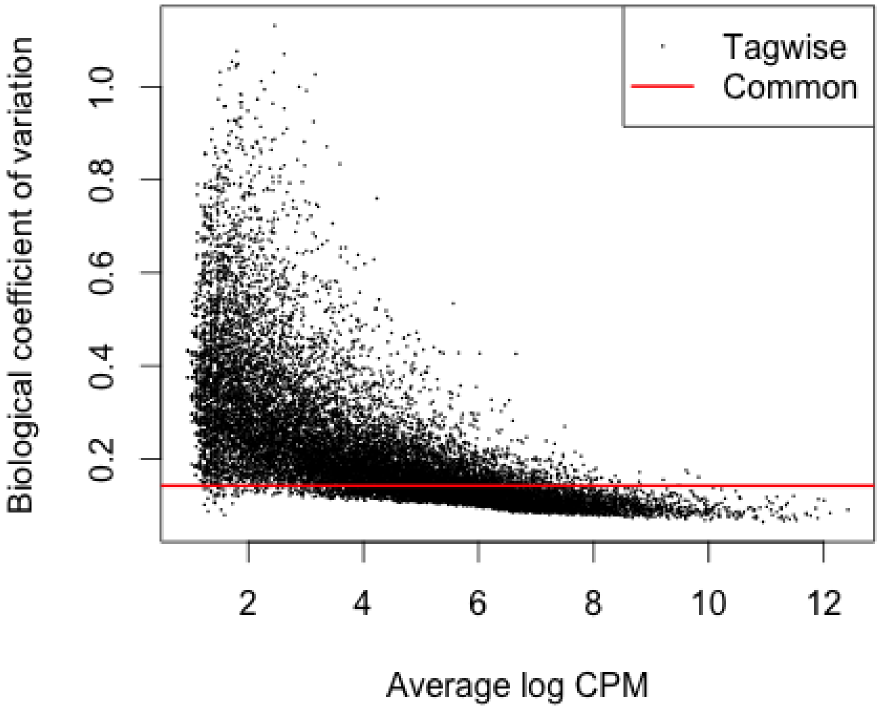


Figure 5: Visualization of sample clustering

We see here that the common dispersion estimates the overall Biological Coefficient of Variation (BCV) of the dataset averaged over all genes. The common dispersion is `0.02` and the BCV is the square root of the common dispersion (sqrt[0.02] = 0.14). A BCV of 14% is typical for cell line experiment.

We now test for differentially expressed BCV genes:

```
et <- exactTest(y)
```

Now we will use the topTags function to adjust for multiple testing. We will use the Benjimini Hochberg (”BH”) method and we will produce a table of results:

```
res <- topTags(et, n=nrow(y$counts), adjust.method="BH")$table
```

Let’s have a look at the first rows of the table:

```
head(res)
```

You can see we have the ensemble gene identifier in the first column, the log fold change in the second column, the the logCPM, the P-Value and the adjusted P-Value. The ensemble gene identifier is not as helpful as the gene symbol so let’s add in a column with the gene symbol. We will use the BiomaRt package to do this.

item We start by using the useMart function of BiomaRt to access the human data base of ensemble gene ids. Then we create a vector of our ensemble gene ids:

```
ensembl_names<-rownames(res)
ensembl<-useMart("ensembl", dataset="hsapiens_gene_ensembl")
```

We then use the function getBM to get the gene symbol data we want
Tthis can take about a minute or so to complete.

```
genemap <-getBM( attributes= c("ensembl_gene_id", "entrezgene", \
    "hgnc_symbol"), filters="ensembl_gene_id", values=ensembl_names, \
    mart=ensembl)
```

Have a look at the start of the genemap dataframe:

```
head(genemap)
```

We see that we have 3 columns, the ensemble id, the entrez gene id and the hgnc symbol We use the match function to match up our data with the data we have just retrieved from the database.

```
idx <- match(ensembl_names, genemap$ensembl_gene_id )
res$entrez <-genemap$entrezgene [ idx ]
res$hgnc_symbol <- genemap$hgnc_symbol [ idx ]
```

Next we have a look at the head of our res dataframe:

```
head(res)
```

As you see we have now added the hgnc symbol and the entrez id to our results.

Let's now make a subset of the most significant upregulated and downregulated genes:

```
de<-res[res$FDR<0.05, ]
de_upreg <-res[res$FDR<0.05 & res$logFC >0,]
de_downreg <-res[res$FDR<0.05 & res$logFC <0,]
```

How many differentially expressed genes are there? (Hint: Try str(de) 4429

How many upregulated genes and downregulated genes do we have? str(de_upreg) = 2345 str(de_downreg) = 2084

Lets write out these results:

```
write.csv( as.data.frame(de), file="DEGs.csv")
```

You can try running the list through DAVID for functional annotation. We will select top

100 genes from the differential expressed list and write those to a separate list.

```
de_top_3000 <-de[1:3000,]
de_top_gene_symbols <-de_top_3000$hgnc_symbol
write(de_top_gene_symbols, "DE_gene_symbols.txt", sep="\t")
```

You can now quit the R prompt

```
q()
```

Please note that the output files you are creating are saved in your present working directory. If you are not sure where you are in the file system try typing `pwd` on your command prompt to find out.

References

1. Trapnell, C., Pachter, L. & Salzberg, S. L. TopHat: discovering splice junctions with RNA-Seq. Bioinformatics 25, 1105-1111 (2009).

2. Trapnell, C. et al. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. Nat. Biotechnol. 28, 511-515 (2010).

3. Langmead, B., Trapnell, C., Pop, M. & Salzberg, S. L. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. 10, R25 (2009).

4. Roberts, A., Pimentel, H., Trapnell, C. & Pachter, L. Identification of novel transcripts in annotated genomes using RNA-Seq. Bioinformatics 27, 2325-2329 (2011).

5. Roberts, A., Trapnell, C., Donaghey, J., Rinn, J. L. & Pachter, L. Improving RNA-Seq expression estimates by correcting for fragment bias. Genome Biol. 12, R22 (2011).

6. Robinson MD, McCarthy DJ and Smyth GK. edgeR: a Bioconductor package for differential expression analysis of digital gene expression data. Bioinformatics, 26 (2010).

7. Robinson MD and Smyth GK Moderated statistical tests for assessing differences in tag abundance. Bioinformatics, 23, pp. -6.

8. Robinson MD and Smyth GK (2008). Small-sample estimation of negative binomial dispersion, with applications to SAGE data.âĂİ Biostatistics, 9.

9. McCarthy, J. D, Chen, Yunshun, Smyth and K. G (2012). Differential expression analysis of multifactor RNA-Seq experiments with respect to biological variation. Nucleic Acids Research, 40(10), pp. -9.

Module: *de novo* Genome Assembly

Primary Author(s):
Matthias Haimel mhaimel@ebi.ac.uk
Nathan S. Watson-Haigh nathan.watson-haigh@awri.com.au

Contributor(s):

Key Learning Outcomes

After completing this practical the trainee should be able to:

- Compile velvet with appropriate compile-time parameters set for a specific analysis
- Be able to choose appropriate assembly parameters
- Assemble a set of single-ended reads
- Assemble a set of paired-end reads from a single insert-size library
- Be able to visualise an assembly in AMOS Hawkeye
- Understand the importance of using paired-end libraries in *de novo* genome assembly

Resources You'll be Using

Although we have provided you with an environment which contains all the tools and data you will be using in this module, you may like to know where we have sourced those tools and data from.

Tools Used

Velvet
    http://www.ebi.ac.uk/~zerbino/velvet/

AMOS Hawkeye
    http://apps.sourceforge.net/mediawiki/amos/index.php?title=Hawkeye

gnx-tools
    https://github.com/mh11/gnx-tools

FastQC
    http://www.bioinformatics.bbsrc.ac.uk/projects/fastqc/

R
    http://www.r-project.org/

Sources of Data

- ftp://ftp.ensemblgenomes.org/pub/release-8/bacteria/fasta/Staphylococcus/s_aureus_mrsa252/dna/s_aureus_mrsa252.EB1_s_aureus_mrsa252.dna.chromosome.Chromosome.fa.gz
- http://www.ebi.ac.uk/ena/data/view/SRS004748
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR022/SRR022825/SRR022825.fastq.gz
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR022/SRR022823/SRR022823.fastq.gz
- http://www.ebi.ac.uk/ena/data/view/SRX008042
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR022/SRR022852/SRR022852_1.fastq.gz
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR022/SRR022852/SRR022852_2.fastq.gz
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR023/SRR023408/SRR023408_1.fastq.gz
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR023/SRR023408/SRR023408_2.fastq.gz
- http://www.ebi.ac.uk/ena/data/view/SRX000181
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR000/SRR000892/SRR000892.fastq.gz
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR000/SRR000893/SRR000893.fastq.gz
- http://www.ebi.ac.uk/ena/data/view/SRX007709
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR022/SRR022863/SRR022863_1.fastq.gz
- ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR022/SRR022863/SRR022863_2.fastq.gz

Introduction

The aim of this module is to become familiar with performing *de novo* genome assembly using Velvet, a de Bruijn graph based assembler, on a variety of sequence data.

Prepare the Environment

The first exercise should get you a little more comfortable with the computer environment and the command line.

First make sure that you are in your home directory by typing:

```
cd
```

and making absolutely sure you're there by typing:

```
pwd
```

Now create sub-directories for this and the two other velvet practicals. All these directories will be made as sub-directories of a directory for the whole course called NGS. For this you can use the following commands:

```
mkdir -p NGS/velvet/{part1,part2,part3}
```

The `-p` tells `mkdir` (make directory) to make any parent directories if they don't already exist. You could have created the above directories one-at-a-time by doing this instead:

```
mkdir NGS
mkdir NGS/velvet
mkdir NGS/velvet/part1
mkdir NGS/velvet/part2
mkdir NGS/velvet/part3
```

After creating the directories, examine the structure and move into the directory ready for the first velvet exercise by typing:

```
ls -R NGS
cd NGS/velvet/part1
pwd
```

Downloading and Compiling Velvet

For the duration of this workshop, all the software you require has been set up for you already. This might not be the case when you return to "real life". Many of the programs you will need, including velvet, are quite easy to set up, it might be instructive to try a couple.

Although you will be using the preinstalled version of velvet, it is useful to know how to compile velvet as some of the parameters you might like to control can only be set at compile time. You can find the latest version of velvet at:

http://www.ebi.ac.uk/~zerbino/velvet/

You could go to this URL and download the latest velvet version, or equivalently, you could type the following, which will download, unpack, inspect, compile and execute your locally compiled version of velvet:

```
cd ~/NGS/velvet/part1
pwd
tar xzf ~/NGS/Data/velvet_1.2.10.tgz
ls -R
cd velvet_1.2.10
make
./velveth
```

The standout displayed to screen when 'make' runs may contain an error message but it is ignored

Take a look at the executables you have created. They will be displayed as green by the command:

```
ls --color=always
```

The switch `--color`, instructs that files be coloured according to their type. This is often the default but we are just being explicit. By specifying the value `always`, we ensure that colouring is always applied, even from a script.

Have a look of the output the command produces and you will see that `MAXKMERLENGTH=31` and `CATEGORIES=2` parameters were passed into the compiler.

This indicates that the default compilation was set for de Bruijn graph k-mers of maximum size 31 and to allow a maximum of just 2 read categories. You can override these, and other, default configuration choices using command line parameters. Assume, you want to run velvet with a k-mer length of 41 using 3 categories, velvet needs to be recompiled to enable this functionality by typing:

```
make clean
make MAXKMERLENGTH=41 CATEGORIES=3
./velveth
```

Discuss with the persons next to you the following questions:
What are the consequences of the parameters you have given make for velvet? MAXKMERLENGTH: increase the max k-mer length from 31 to 41

CATEGORIES: paired-end data require to be put into separate categories. By increasing this parameter from 2 to 3 allows you to process 3 paired / mate-pair libraries and unpaired data.

Why does Velvet use k-mer 31 and 2 categories as default? Possibly a number of reason:
- odd number to avoid palindromes
- The first reads were very short (20-40 bp) and there were hardly any paired-end data
around so there was no need to allow for longer k-mer lengths / more categories.
- For programmers: 31 bp get stored in 64 bits (using 2bit encoding)

Should you get better results by using a longer k-mer length? If you can achieve a good k-mer coverage - yes.

velvet can also be used to process SOLID colour space data. To do this you need a further make parameter. With the following command clean away your last compilation and try the following parameters:

```
make clean
make MAXKMERLENGTH=41 CATEGORIES=3 color
./velveth_de
```

What effect would the following compile-time parameters have on velvet:
`OPENMP=Y` Turn on multithreading

`LONGSEQUENCES=Y` Assembling reads / contigs longer than 32kb long

`BIGASSEMBLY=Y` Using more than 2.2 billion reads

`VBIGASSEMBLY=Y` Not documented yet

`SINGLE_COV_CAT=Y` Merge all coverage statistics into a single variable - save memory

For a further description of velvet compile and runtime parameters please see the velvet Manual: `https://github.com/dzerbino/velvet/wiki/Manual`

Assembling Single-end Reads

The following exercise focuses on velvet using single-end reads, how the available parameters effect an assembly and how to measure and compare the changes.

Even though you have carefully compiled velvet in your own workspace, we will be use the pre-installed version.

The data you will use is from Staphylococcus aureus USA300 which has a genome of around 3MBases. The reads are unpaired Illumina, also known as single-end library.

The data for this section was obtained from the Sequence Read Archive (SRA), using `SRR022825` and `SRR022823` run data from SRA Sample `SRS004748`. The SRA experiment can be viewed at:

http://www.ebi.ac.uk/ena/data/view/SRS004748

To begin with, first move back to the directory you prepared for this exercise, create a new folder with a suitable name for this part and move into it. There is no need to download the read files, as they are already stored locally. Instead we will create symlinks to the files. Continue by copying (or typing):

```
cd ~/NGS/velvet/part1
mkdir SRS004748
cd SRS004748
pwd
ln -s ~/NGS/Data/SRR022825.fastq.gz ./
ln -s ~/NGS/Data/SRR022823.fastq.gz ./
ls -l
```

You are ready to process your data with Velvet. There are two main components to Velvet:

`velveth` Used to construct, from raw read data, a dataset organised in the fashion expected by the second component, `velvetg`.

`velvetg` The core of velvet where the de Bruijn graph assembly is built and manipulated.

You can always get further information about the usage of both velvet programs by typing `velvetg` or `velveth` in your terminal.

Now run `velveth` for the reads in `SRR022825.fastq.gz` and `SRR022823.fastq.gz` using the following options:

- A de Bruijn graph k-mer of 25

- An output directory called run_25

```
velveth run_25 25 -fastq.gz -short SRR022825.fastq.gz SRR022823.fastq.gz
```

`velveth` Once `velveth` finishes, move into the output directory `run_25` and have a look at what `velveth` has generated so far. The command `less` allows you to look at output files (press `q` to quit and return to the command prompt). Here are some other options for looking at file contents:

```
cd run_25
ls -l
head Sequences
cat Log
```

What did you find in the folder `run_25`? Sequences, Roadmaps, Log

Describe the content of the two `velveth` output files? Sequences: FASTA file version of provided reads
Roadmaps: Internal file of velvet - basic information about number of reads, k-mer size

What does the `Log` file store for you? Time stamp, Executed commands; velvet version + compiler parameters, results

Now move one directory level up and run `velvetg` on your output directory, with the commands:

```
cd ../
time velvetg run_25
```

Move back into your results directory to examine the effects of `velvetg`:

```
cd run_25
ls -l
```

What extra files do you see in the folder `run_25`? PreGraph, Graph, stats.txt, contigs.fa, LastGraph

What do you suppose they might represent? PreGraph, Graph, LastGraph: Velvet internal graph representation at different stages (see manual for more details about the file format)

stats.txt: tab-delimited description of the nodes of the graph incl. coverage information

contigs.fa: assembly output file

In the Log file in `run_25`, what is the N50? 4409 bp

Hopefully, we will have discussed what the N50 statistic is by this point. Broadly, it is the median (not average) of a sorted data set using the length of a set of sequences. Usually it is the length of the contig whose length, when added to the length of all longer contigs, makes a total greater that half the sum of the lengths of all contigs. Easy, but messy - a more formal definition can be found here:

http://www.broadinstitute.org/crd/wiki/index.php/N50

Backup the `contigs.fa` file and calculate the N50 (and the N25,N75) value with the command:

```
cp contigs.fa contigs.fa.0
gnx -min 100 -nx 25,50,75 contigs.fa
```

Does the value of N50 agree with the value stored in the Log file? No

If not, why do you think this might be? K-mer N50 vs bp N50; contig length cut-off value, estimated genome length

In order to improve our results, take a closer look at the standard options of `velvetg` by typing `velvetg` without parameters. For the moment focus on the two options `-cov_cutoff` and `-exp_cov`. Clearly `-cov_cutoff` will allow you to exclude contigs for which the k-mer coverage is low, implying unacceptably poor quality. The `-exp_cov` switch is used to give `velvetg` an idea of the coverage to expect.

If the expected coverage of any contig is substantially in excess of the suggested expected value, maybe this would indicate a repeat. For further details of how to choose the parameters, go to "Choice of a coverage cutoff":

http://wiki.github.com/dzerbino/velvet/

Briefly, the k-mer coverage (and much more information) for each contig is stored in the file `stats.txt` and can be used with R to visualize the k-mer coverage distribution. Take a look at the `stats.txt` file, start R, load and visualize the data using the following commands:

```
R --no-save --no-restore
install.packages('plotrix')
library(plotrix)
data <- read.table("stats.txt", header=TRUE)
weighted.hist(data$short1_cov, data$lgth, breaks=0:50)
```

A weighted histogram is a better way of visualizing the coverage information, because of noise (lots of very short contigs). You can see an example output below:

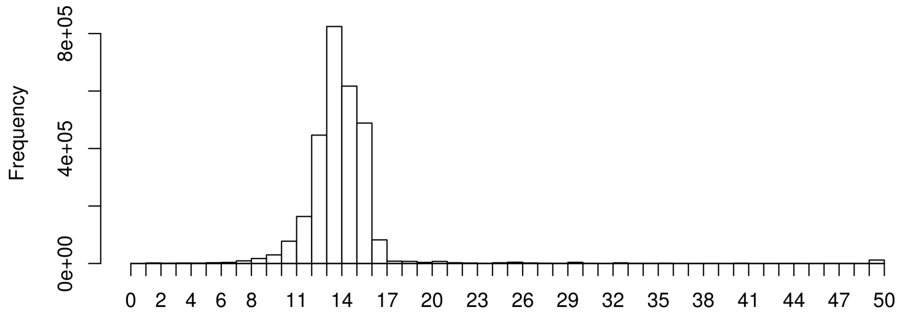


Figure 6: A weighted k-mer coverage histogram of the single-end reads.

After choosing the expected coverage and the coverage cut-off, you can exit R by typing:

```
q()
```

The weighted histogram suggests to me that the expected coverage is around 14 and that everything below 6 is likely to be noise. Some coverage is also represented at around 20, 30 and greater 50, which might be contamination or repeats (depending on the dataset), but at the moment this should not worry you. To see the improvements, rerun `velvetg` first with `-cov_cutoff` 6 and after checking the N50 use only / add `-exp_cov` 14 to the command line option. Also keep a copy of the contigs file for comparison:

```
cd ~/NGS/velvet/part1/SRS004748
time velvetg run_25 -cov_cutoff 6

# Make a copy of the run
cp run_25/contigs.fa run_25/contigs.fa.1

time velvetg run_25 -exp_cov 14
cp run_25/contigs.fa run_25/contigs.fa.2

time velvetg run_25 -cov_cutoff 6 -exp_cov 14
cp run_25/contigs.fa run_25/contigs.fa.3
```

What is the N50 with no parameter: 4,447 bp

What is the N50 with -cov_cutoff 6: 5,168 bp

What is the N50 with -exp_cov 14: 4,903 bp

What is the N50 with -cov_cutoff 6 -exp_cov 14: 5,417 bp

Did you notice a variation in the time velvetg took to run? If so, can you explain why that might be? Velvet reuses already calculated results (from PreGraph,Graph)

You were running velvetg with the -exp_cov and -cov_cutoff parameters. Now try to experiment using different cut-offs, expected parameters and also explore other settings (e.g. -max_coverage, -max_branch_length, -unused_reads, -amos_file, -read_trkg or see velvetg help menu).

Make some notes about the parameters you've played with and the results you obtained. -max_coverage: cut-off value for the upper range (like cov_cutoff for the lower range)
-max_branch_length: length of branch to look for bubble
-unused_reads: write unused reads into file
-amos_file: write AMOS message file
-read_trkg: tracking read (more memory usage) - automatically on for certain operations

AMOS Hawkeye

The -amos_file argument tells velvetg to output the assembly as an AMOS message file (*.afg) which can then be used by tools like Hawkeye from the AMOS suite of tools. Lets create the AMOS message file by running velvetg with some appropriate parameters:

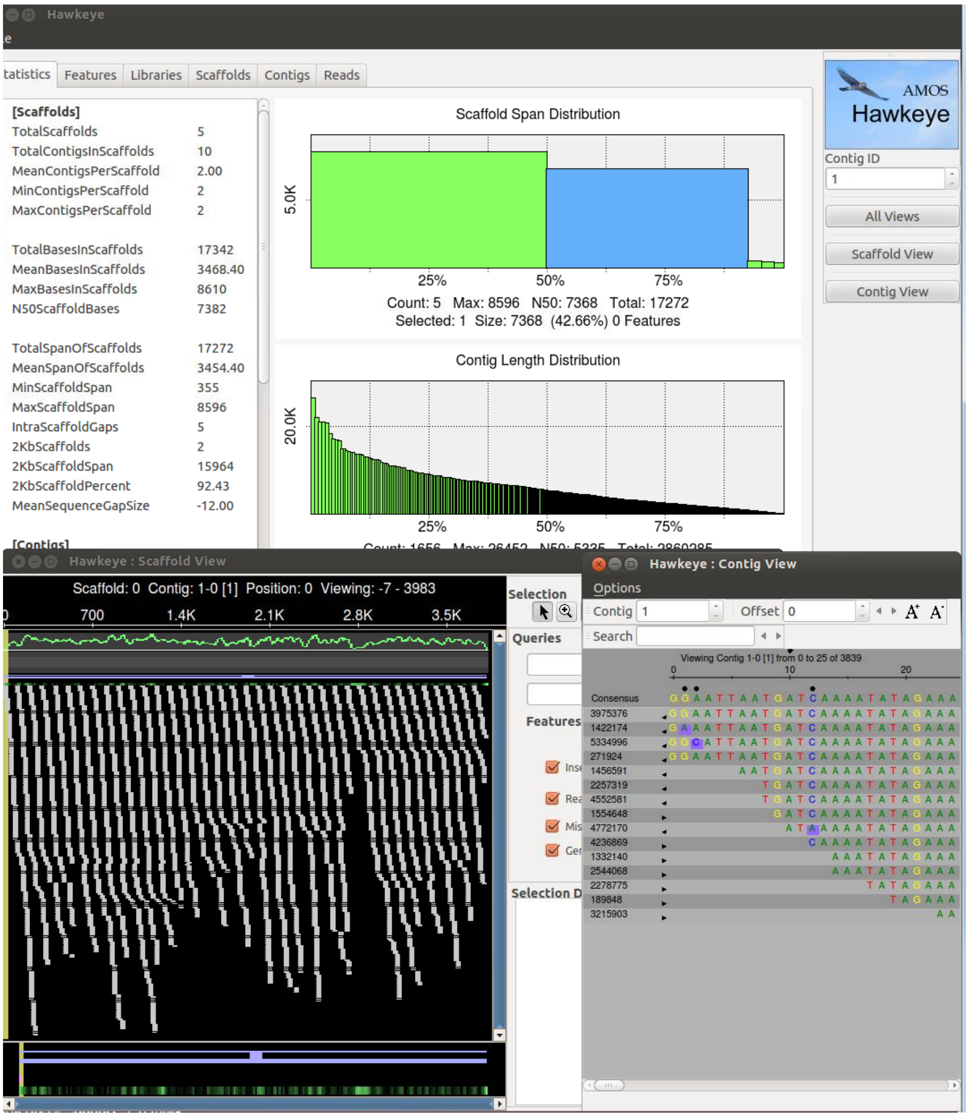
```
velvetg run_25 -cov_cutoff 6 -exp_cov 14 -amos_file yes
```

The -exp_cov argument to enable read-tracking -read_trkg yes in Velvet. Without read tracking enabled, very little read-level information can be output to the AMOS message file. This results in a pretty useless visualisation in Hawkeye! However, since reads are being tracked, the analysis takes longer and uses more memory.

Now convert the AMOS message file velvet_asm.afg into an AMOS bank using bank-transact and view the assembly with AMOS Hawkeye.

```
bank-transact -c -b run_25/velvet_asm.bnk -m run_25/velvet_asm.afg
hawkeye run_25/velvet_asm.bnk
```

Have a look around the interface, in particular try to look at the "Scaffold View" and "Contig View" of the larges scaffold. You should see something like this:



[Scaffolds]

TotalScaffolds	5
TotalContigsInScaffolds	10
MeanContigsPerScaffold	2.00
MinContigsPerScaffold	2
MaxContigsPerScaffold	2
TotalBasesInScaffolds	17342
MeanBasesInScaffolds	3468.40
MaxBasesInScaffolds	8610
N50ScaffoldBases	7382
TotalSpanOfScaffolds	17272
MeanSpanOfScaffolds	3454.40
MinScaffoldSpan	355
MaxScaffoldSpan	8596
IntraScaffoldGaps	5
2KbScaffolds	2
2KbScaffoldSpan	15964
2KbScaffoldPercent	92.43
MeanSequenceGapSize	-12.00

[Contigs]

Scaffold Span Distribution

Count: 5 Max: 8596 N50: 7368 Total: 17272
Selected: 1 Size: 7368 (42.66%) 0 Features

Contig Length Distribution

Scaffold: 0 Contig: 1-0 [1] Position: 0 Viewing: -7 - 3983

Contig ID 1

Contig 1 Offset 0

Viewing Contig 1-0 [1] from 0 to 25 of 3839

Figure 7:

If you have time, try running the `velvetg` command without the `-exp_cov` argument, create the AMOS bank and see how the assemblies look different in Hawkeye. Here's a hint:

```
velvetg run_25 -cov_cutoff 6 -amos_file yes
bank-transact -c -b run_25/velvet_asm.bnk -m run_25/velvet_asm.afg
hawkeye run_25/velvet_asm.bnk
```

Simple Assembly Simulation

The data for this section is from Staphylococcus aureus MRSA252, a genome closely related to the genome that provided the short read data in the earlier sections of this exercise. The sequence data this time is the fully assembled genome. The genome size is therefore known exactly and is 2,902,619 bp.

In this exercise you will process the single whole genome sequence with `velveth` and `velvetg`, look at the output only and go no further. The main intent of processing this whole genome is to compute its N50 value. This must clearly be very close to the ideal N50 value for the short reads assembly and so aid evaluation of that assembly.

To begin, move back to the main directory for this exercise, make a sub-directory for the processing of this data and move into it. All in one go, this would be:

```
cd ~/NGS/velvet/part1/
mkdir MRSA252
cd MRSA252
```

Next, you need to download the genome sequence from http://www.ensemblgenomes.org/ which holds five new sites, for bacteria, protists, fungi, plants and invertebrate metazoa. You could browse for the data you require or use the file which we have downloaded for you. For the easier of these options, make and check a symlink to the local file and with the commands:

```
ln -s ~\
    /NGS/Data/s_aureus_mrsa252.EB1_s_aureus_mrsa252.dna.chromosome.Chromosome.fa.gz \
    ./
ls -l
```

Usually Velvet expects relatively short sequence entries and for this reason has a read limit of 32,767 bp per sequence entry. As the genome size is 2,902,619 bp - longer as the allowed limit and does not fit with the standard settings into velvet. But like the maximum k-mer size option, you can tell Velvet during compile time, using `LONGSEQUENCES=Y`, to expect longer input sequences than usual. I already prepared the executable which you can use by typing `velveth_long` and `velvetg_long`.

Now, run `velveth_long`, using the file you either just downloaded or created a symlink to as the input:

```
velveth_long run_25 25 -fasta.gz -long \
    s_aureus_mrsa252.EB1_s_aureus_mrsa252.dna.chromosome.Chromosome.fa.gz
velvetg_long run_25
```

What is the N50? 24,142 bp

How does the N50 compare to the previous single end run (SRS004748)? Big difference

Does the total length differ from the input sequence length? 2,817,181 (stats) vs 2,902,619 (input)

What happens when you rerun velvet with a different k-mer length? K-mer 31: N50: 30,669 bp, total 2,822,878

Assembling Paired-end Reads

The use of paired-end data in *de novo* genome assembly results in better quality assemblies, particularly for larger, more complex genomes. In addition, paired-end constraint violation (expected distance and orientation of paired reads) can be used to identify misassemblies.

If you are doing *de novo* assembly, pay the extra and get paired-ends: they're worth it!

The data you will examine in this exercise is again from Staphylococcus aureus which has a genome of around 3MBases. The reads are Illumina paired end with an insert size of 350 bp.

The required data can be downloaded from the SRA. Specifically, the run data (SRR022852) from the SRA Sample SRS004748.

http://www.ebi.ac.uk/ena/data/view/SRS004748

The following exercise focuses on preparing the paired-end FASTQ files ready for Velvet, using Velvet in paired-end mode and comparing results with Velvet's 'auto' option.

First move to the directory you made for this exercise and make a suitable named directory for the exercise:

```
cd ~/NGS/velvet/part2
mkdir SRS004748
```

```
cd SRS004748
```

There is no need to download the read files, as they are already stored locally. You will simply create a symlink to this pre-downloaded data using the following commands:

```
ln -s ~/NGS/Data/SRR022852_?.fastq.gz ./
```

It is interesting to monitor the computer's resource utilisation, particularly memory. A simple way to do this is to open a second terminal and in it type:

```
top
```

`top` is a program that continually monitors all the processes running on your computer, showing the resources used by each. Leave this running and refer to it periodically throughout your Velvet analyses. Particularly if they are taking a long time or whenever your curiosity gets the better of you. You should find that as this practical progresses, memory usage will increase significantly.

Now, back to the first terminal, you are ready to run `velveth` and `velvetg`. The reads are `-shortPaired` and for the first run you should not use any parameters for `velvetg`.

From this point on, where it will be informative to time your runs. This is very easy to do, just prefix the command to run the program with the command `time`. This will cause UNIX to report how long the program took to complete its task.

Set the two stages of velvet running, whilst you watch the memory usage as reported by `top`. Time the `velvetg` stage:

```
velveth run_25 25 -fmtAuto -create_binary -shortPaired -separate \
    SRR022852_1.fastq.gz SRR022852_2.fastq.gz
time velvetg run_25
```

What does `-fmtAuto` and `-create_binary` do? (see help menu) `-fmtAuto` tries to detect the correct format of the input files e.g. FASTA, FASTQ and whether they are compressed or not.

`-create_binary` outputs sequences as a binary file. That means that `velvetg` can read the sequences from the binary file more quickly that from the original sequence files.

Comment on the use of memory and CPU for `velveth` and `velvetg`? `velveth` uses only one CPU while `velvetg` uses all possible CPUs for some parts of the calculation.

How long did `velvetg` take? My own measurements are:
`real 1m8.877s; user 4m15.324s; sys 0m4.716s`

TRAINER'S MANUAL

Next, after saving your `contigs.fa` file from being overwritten, set the cut-off parameters that you investigated in the previous exercise and rerun `velvetg`. time and monitor the use of resources as previously. Start with `-cov_cutoff 16` thus:

```
mv run_25/contigs.fa run_25/contigs.fa.0
time velvetg run_25 -cov_cutoff 16
```

Up until now, `velvetg` has ignored the paired-end information. Now try running `velvetg` with both `-cov_cutoff 16` and `-exp_cov 26`, but first save your `contigs.fa` file. By using `-cov_cutoff` and `-exp_cov`, `velvetg` tries to estimate the insert length, which you will see in the `velvetg` output. The command is, of course:

```
mv run_25/contigs.fa run_25/contigs.fa.1
time velvetg run_25 -cov_cutoff 16 -exp_cov 26
```

Comment on the time required, use of memory and CPU for `velvetg`? Runtime is lower when velvet can reuse previously calculated data. By using `-exp_cov`, the memory usage increases.

Which insert length does Velvet estimate? Paired-end library 1 has length: 228, sample standard deviation: 26

Next try running `velvetg` in ‘paired-end mode‘. This entails running `velvetg` specifying the insert length with the parameter `-ins_length` set to 350. Even though velvet estimates the insert length it is always advisable to check / provide the insert length manually as velvet can get the statistics wrong due to noise. Just in case, save your last version of `contigs.fa`. The commands are:

```
mv run_25/contigs.fa run_25/contigs.fa.2
time velvetg run_25 -cov_cutoff 16 -exp_cov 26 -ins_length 350
mv run_25/contigs.fa run_25/contigs.fa.3
```

How fast was this run? My own measurements are:
`real 0m29.792s; user 1m4.372s; sys 0m3.880s`

Take a look into the Log file.

What is the N50 value for the `velvetg` runs using the switches:
Base run: 19,510 bp `-cov_cutoff 16` 24,739 bp

`-cov_cutoff 16 -exp_cov 26` 61,793 bp

`-cov_cutoff 16 -exp_cov 26 -ins_length 350` n50 of 62,740 bp; max 194,649 bp; total 2,871,093 bp

Try giving the `-cov_cutoff` and/or `-exp_cov` parameters the value `auto`. See the `velvetg` help to show you how. The information Velvet prints during running includes information about the values used (coverage cut-off or insert length) when using the `auto` option.

What coverage values does Velvet choose (hint: look at the output that Velvet produces while running)? Median coverage depth = 26.021837
Removing contigs with coverage < 13.010918 ...

How does the N50 value change? n50 of 68,843 bp; max 194,645 bp; total 2,872,678 bp

Run `gnx` on all the `contig.fa` files you have generated in the course of this exercise. The command will be:

```
gnx -min 100 -nx 25,50,75 run_25/contigs.fa*
```

For which runs are there Ns in the `contigs.fa` file and why? contigs.fa.2, contigs.fa.3, contigs.fa
Velvet tries to use the provided (or infers) the insert length and fills ambiguous regions with Ns.

Comment on the number of contigs and total length generated for each run.

Filename	No. contigs	Total length	No. Ns
Contigs.fa.0	631	2,830,659	0
Contigs.fa.1	580	2,832,670	0
Contigs.fa.2	166	2,849,919	4,847
Contigs.fa.3	166	2,856,795	11,713
Contigs.fa	163	2,857,439	11,526

Table 7:

AMOS Hawkeye

We will now output the assembly in the AMOS massage format and visualise the assembly using AMOS Hawkeye.
Run `velvetg` with appropriate arguments and output the AMOS message file, then convert it to an AMOS bank and open it in Hawkeye:

```
time velvetg run_25 -cov_cutoff 16 -exp_cov 26 -ins_length 350 \
    -amos_file yes -read_trkg yes
time bank-transact -c -b run_25/velvet_asm.bnk -m run_25/velvet_asm.afg
```

```
hawkeye run_25/velvet_asm.bnk
```

Looking at the scaffold view of a contig, comment on the proportion of “happy mates” to “compressed mates” and “stretched mates”. Nearly all mates are compressed with no stretched mates and very few happy mates.

What is the mean and standard deviation of the insert size reported under the Libraries tab? Mean: 350 bp SD: 35 bp

Look at the actual distribution of insert sizes for this library. Can you explain where there is a difference between the mean and SD reported in those two places? We specified `-ins_length 350` to the `velvetg` command. Velvet uses this value, in the AMOS message file that it outputs, rather than its own estimate.

You can get AMOS to re-estimate the mean and SD of insert sizes using intra-contig pairs. First, close Hawkeye and then run the following commands before reopening the AMOS bank to see what has changed.

```
asmQC -b run_25/velvet_asm.bnk -scaff -recompute -update -numsd 2
hawkeye run_25/velvet_asm.bnk
```

Looking at the scaffold view of a contig, comment on the proportion of “happy mates” to “compressed mates” and “stretched mates”. There are only a few compressed and stretched mates compared to happy mates. There are similar numbers of stretched and compressed mates.

What is the mean and standard deviation of the insert size reported under the Libraries tab? TODO Mean: 226 bp SD: 25 bp

Look at the actual distribution of insert sizes for this library. Does the mean and SD reported in both places now match? Yes

Can you find a region with an unusually high proportion of stretched, compressed, incorrectly orientated or linking mates? What might this situation indicate? This would indicate a possible misassembly and worthy of further investigation.

Look at the largest scaffold, there are stacks of stretched pairs which span contig boundaries. This indicates that the gap size has been underestimated during the scaffolding phase.

Velvet and Data Quality

So far we have used the raw read data without performing any quality control or read trimming prior to doing our velvet assemblies.

Velvet does not use quality information present in FASTQ files.

For this reason, it is vitally important to perform read QC and quality trimming. In doing so, we remove errors/noise from the dataset which in turn means velvet will run faster, will use less memory and will produce a better assembly. Assuming we haven't compromised too much on coverage.

To investigate the effect of data quality, we will use the run data (SRR023408) from the SRA experiment SRX008042. The reads are Illumina paired end with an insert size of 92 bp.

Go back to the main directory for this exercise and create and enter a new directory dedicated to this phase of the exercise. The commands are:

```
cd ~/NGS/velvet/part2
mkdir SRX008042
cd SRX008042
```

Create symlinks to the read data files that we downloaded for you from the SRA:

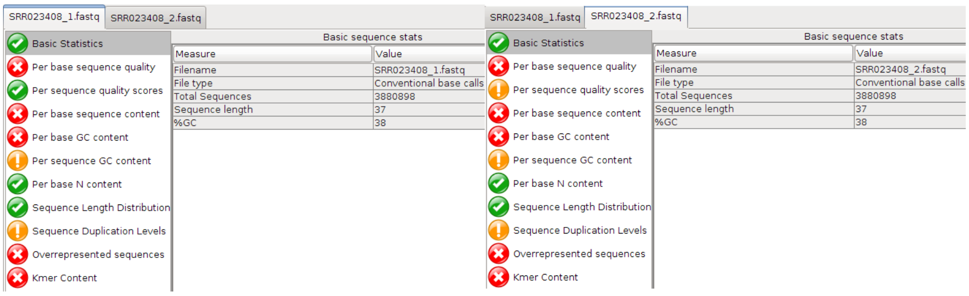
```
ln -s ~/NGS/Data/SRR023408_?.fastq.gz ./
```

We will use FastQC, a tool you should be familiar with, to visualise the quality of our data. We will use FastQC in the Graphical User Interface (GUI) mode.

Start FastQC and set the process running in the background, by using a trailing `&`, so we get control of our terminal back for entering more commands:

```
fastqc &
```

Open the two compressed FASTQ files (File – > Open) by selecting them both and clicking OK). Look at tabs for both files:



SRR023408_1.fastq

- ✓ Basic Statistics
- ✗ Per base sequence quality
- ✓ Per sequence quality scores
- ✗ Per base sequence content
- ✗ Per base GC content
- ! Per sequence GC content
- ✓ Per base N content
- ✓ Sequence Length Distribution
- ! Sequence Duplication Levels
- ✗ Overrepresented sequences
- ✗ Kmer Content

Basic sequence stats

Measure	Value
Filename	SRR023408_1.fastq
File type	Conventional base calls
Total Sequences	3880898
Sequence length	37
%GC	38

SRR023408_2.fastq

- ✓ Basic Statistics
- ✗ Per base sequence quality
- ! Per sequence quality scores
- ✗ Per base sequence content
- ✗ Per base GC content
- ! Per sequence GC content
- ✓ Per base N content
- ✓ Sequence Length Distribution
- ! Sequence Duplication Levels
- ✗ Overrepresented sequences
- ✗ Kmer Content

Basic sequence stats

Measure	Value
Filename	SRR023408_2.fastq
File type	Conventional base calls
Total Sequences	3880898
Sequence length	37
%GC	38

Figure 8:

Are the quality scores the same for both files? Overall yes

Which value varies? Per sequence quality scores

Take a look at the Per base sequence quality for both files. Did you note that it is not good for either file? The quality score of both files drop very fast. Qualities of the REV strand drop faster than the FWD strand. This is because the template has been sat around while the FWD strand was sequenced.

At which positions would you cut the reads if we did "fixed length trimming"? Looking at the "Per base quality" and "Per base sequence content", I would choose around 27

Why does the quality deteriorate towards the end of the read? Errors more likely for later cycles

Does it make sense to trim the 5' start of reads? Looking at the "Per base sequence content", yes - there is a clear signal at the beginning.

Have a look at the other options that FastQC offers.

Which other statistics could you use to support your trimming strategy? "Per base sequence content", "Per base GC content", "Kmer content", "Per base sequence quality"

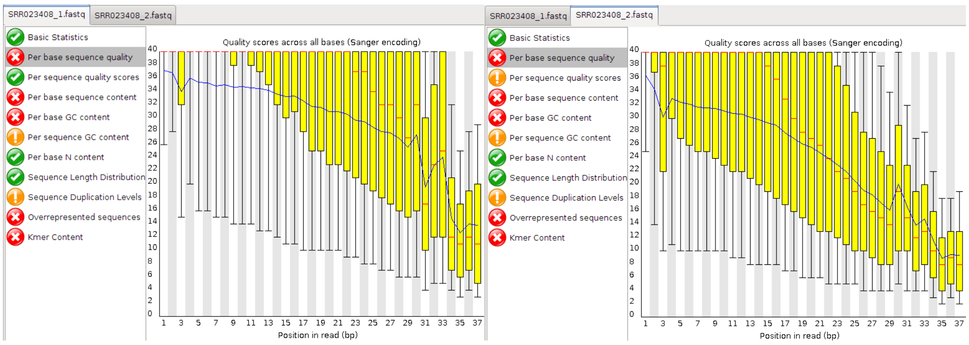

SRR023408_1.fastq
SRR023408_2.fastq
✓ Basic Statistics
✗ Per base sequence quality
✓ Per sequence quality scores
✗ Per base sequence content
✗ Per base GC content
! Per sequence GC content
✓ Per base N content
✓ Sequence Length Distribution
! Sequence Duplication Levels
✗ Overrepresented sequences
✗ Kmer Content
Quality scores across all bases (Sanger encoding)
40 38 36 34 32 30 28 26 24 22 20 18 16 14 12 10 8 6 4 2 0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37
Position in read (bp)
SRR023408_1.fastq
SRR023408_2.fastq
✓ Basic Statistics
✗ Per base sequence quality
! Per sequence quality scores
✗ Per base sequence content
✗ Per base GC content
! Per sequence GC content
✓ Per base N content
✓ Sequence Length Distribution
! Sequence Duplication Levels
✗ Overrepresented sequences
✗ Kmer Content
Quality scores across all bases (Sanger encoding)
40 38 36 34 32 30 28 26 24 22 20 18 16 14 12 10 8 6 4 2 0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37
Position in read (bp)

Figure 9:

Once you have decided what your trim points will be, close FastQC. We will use `fastx_trimmer` from the FASTX-Toolkit to perform fixed-length trimming. For usage information see the help:

```
fastx_trimmer -h
```

`fastx_trimmer` is not able to read compressed FASTQ files, so we first need to decompress the files ready for input.

The suggestion (hopefully not far from your own thoughts?) is that you trim your reads as follows:

```
gunzip < SRR023408_1.fastq.gz > SRR023408_1.fastq
gunzip < SRR023408_2.fastq.gz > SRR023408_2.fastq
fastx_trimmer -Q 33 -f 1 -l 32 -i SRR023408_1.fastq -o \
    SRR023408_trim1.fastq
fastx_trimmer -Q 33 -f 1 -l 27 -i SRR023408_2.fastq -o \
    SRR023408_trim2.fastq
```

Many NGS read files are large. This means that simply reading and writing files can become the bottleneck, also known as I/O bound. Therefore, it is often good practice to avoid unnecessary disk read/write.

We could do what is called pipelining to send a stream of data from one command to another, using the pipe (|) character, without the need for intermediary files. The following command would achieve this:

```
gunzip --to-stdout < SRR023408_1.fastq.gz | fastx_trimmer -Q 33 -f 4 \
    -l 32 -o SRR023408_trim1.fastq
gunzip --to-stdout < SRR023408_2.fastq.gz | fastx_trimmer -Q 33 -f 3 \
    -l 29 -o SRR023408_trim2.fastq
```

Now run `velveth` with a k-mer value of 21 for both the untrimmed and trimmed read files in `-shortPaired` mode. Separate the output of the two executions of `velveth` into suitably named directories, followed by `velvetg`:

```
# untrimmed reads
velveth run_21 21 -fmtAuto -create_binary -shortPaired -separate \
    SRR023408_1.fastq SRR023408_2.fastq
time velvetg run_21

# trimmed reads
velveth run_21trim 21 -fmtAuto -create_binary -shortPaired -separate \
    SRR023408_trim1.fastq SRR023408_trim2.fastq
time velvetg run_21trim
```

How long did the two `velvetg` runs take? run_25: `real 3m16.132s; user 8m18.261s; sys 0m7.317s`
run_25trim: `real 1m18.611s; user 3m53.140s; sys 0m4.962s`

What N50 scores did you achieve? Untrimmed: 11
Trimmed: 15

What were the overall effects of trimming? Time saving, increased N50, reduced coverage

The evidence is that trimming improved the assembly. The thing to do surely, is to run `velvetg` with the `-cov_cutoff` and `-exp_cov`. In order to use `-cov_cutoff` and `-exp_cov` sensibly, you need to investigate with R, as you did in the previous exercise, what parameter values to use. Start up R and produce the weighted histograms:

```
R --no-save
library(plotrix)
data <- read.table("run_21/stats.txt", header=TRUE)
data2 <- read.table("run_21trim/stats.txt", header=TRUE)
par(mfrow=c(1,2))
weighted.hist(data$short1_cov, data$lgth, breaks=0:50)
weighted.hist(data2$short1_cov, data2$lgth, breaks=0:50)
```

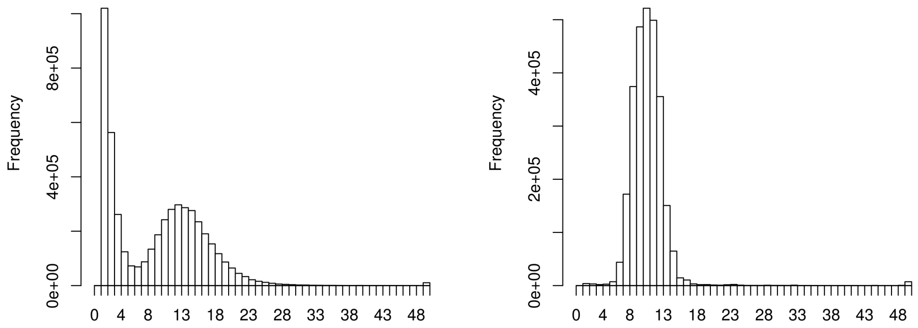


Figure 10: Weighted k-mer coverage histograms of the paired-end reads pre-trimmed (left) and post-trimmed (right).

For the untrimmed read histogram (left) there is an expected coverage of around 13 with a coverage cut-off of around 7. For the trimmed read histogram (right) there is an expected coverage of around 9 with a coverage cut-off of around 5.

If you disagree, feel free to try different settings, but first quit R before running `velvetg`:

```
q()
```

```
time velvetg run_21 -cov_cutoff 7 -exp_cov 13 -ins_length 92
time velvetg run_21trim -cov_cutoff 5 -exp_cov 9 -ins_length 92
```

How good does it look now?
Still not great Comment on:
Runtime Reduced runtime

Memory Lower memory usage

k-mer choice (Can you use k-mer 31 for a read of length 30 bp?) K-mer has to be lower than the read length and the K-mer coverage should be sufficient to produce results.

Does less data mean "worse" results? Not necessarily. If you have lots of data you can safely remove poor data without too much impact on overall coverage.

How would a smaller/larger k-mer size behave?

Compare the results, produced during the last exercises, with each other:

Metric	SRR022852	SRR023408	SRR023408.trimmed
Overall Quality (1-5)			
bp Coverage			
k-mer Coverage			
N50 (k-mer used)			

Table 8:

Metric	SRR022852	SRR023408	SRR023408.trimmed
Overall Quality (1-5)	2	5	4
bp Coverage	136 x (36 bp;11,374,488)	95x (37bp; 7761796)	82x (32bp; 7761796)
k-mer Coverage	45x	43x (21); 33x (25)	30x (21); 20.5x (25)
N50 (k-mer used)	68,843 (25)	2,803 (21)	2,914 (21)

Table 9:

What would you consider as the "best" assembly? SRR022852

If you found a candidate, why do you consider it as "best" assembly? Overall data quality and coverage

How else might you assess the the quality of an assembly? Hint: Hawkeye. By trying to identify paired-end constraint violations using AMOS Hawkeye.

Hybrid Assembly

Like the previous examples, the data you will examine in this exercise is again from Staphylococcus aureus which has a genome of around 3MB. The reads are 454 single end and Illumina paired end with an insert size of 170 bp. You already downloaded the required reads from the SRA in previous exercises. Specifically, the run data (SRR022863, SRR000892, SRR000893) from the SRA experiments SRX007709 and SRX000181.

The following exercise focuses on handing 454 long reads and paired-end reads with velvet and the differences in setting parameters.

First move to the directory you made for this exercise, make a suitable named directory for the exercise and check if all the three files are in place:

```
cd ~/NGS/velvet/part3
mkdir SRR000892-SRR022863
cd SRR000892-SRR022863
# 454 single end data
ln -s ~/NGS/Data/SRR00089[2-3].fastq.gz ./
# illumina paired end data
ln -s ~/NGS/Data/SRR022863_?.fastq.gz ./
```

The following command will run for a LONG time. This indicated the amount of calculations being preformed by Velvet to reach a conclusion. To wait for velvet to finish would exceed the time available in this workshop, but it is up to you to either let it run over night or kill the process by using the key combination CTRL+c.

```
velveth run_25 25 -fmtAuto -create_binary -long \
    SRR00089?.fastq.gz -shortPaired -separate \
    SRR022863_1.fastq.gz SRR022863_2.fastq.gz
time velvetg run_25
```

If you have decided to continue, we already inspected the weighted histograms for the short and long read library separately, you can reuse this for the cut-off values:

```
time velvetg run_25 -cov_cutoff 7 -long_cov_cutoff 9
```

What are your conclusions using Velvet in an hybrid assembly? 17 min: time velvetg run_25

Post-Workshop Information

Access to Computational Resources

By the end of the workshop we hope you're thinking one or more of the following:

- I'm interested in dabbling some more during my day-job!
- How do I access a Linux box like the one I've been using in the workshop - I *really* don't want the hassle of setting this all up myself!
- I'm hooked! I *really* want to get down and dirty with NGS data! What computational resources do I need, what do I have access to and how do I access them?

We're ecstatic you're thinking this way and want to help guide you! However, lets take this one step at a time.

The quickest way to dabble is to use a clone of the operating system (OS) you've been using during this workshop. That means you'll have hassle-free access to a plethora of pre-installed, pre-configured bioinformatics tools. You could even set it up to contain a copy of all the workshop data and handouts etc to go through the hands-on practicals in your own time!

We have created an image file (approx. 10 GBytes in size) of the NGS Training OS for you to use as you wish:

https://swift.rc.nectar.org.au:8888/v1/AUTH_33065ff5c34a4652aa2fefb292b3195a/VMs/NGSTrainingV1.2.1.vdi

We would advise one of the following two approaches for making use of it:

- Import it into VirtualBox to setup a virtual machine (VM) on your own computer.
- Instantiate a VM on the NeCTAR Research Cloud.

Setting up a VM using VirtualBox

This approach requires the least amount of mind-bending to get up and running. However, you will need to install some software. If you do not have administrator access or your system administrator is slow or unwilling to install the software, you may find using the NeCTAR Research Cloud to be viable alternative.

This approach will use, at most, the computational resources available on your own computer. If you are analysing non-microbial organisms or performing *de novo* assemblies, you may find these resources are insufficient. If this is the case, you really should speak to someone from IT support at your institution or get in touch with a bioinformatician for advise.

The software you need is VirtualBox, a freely available, Open Source virtualisation product from Oracle (https://www.virtualbox.org/). This software essentially allows you to run an operating system (the guest OS) within another (the host OS). VirtualBox is available for several different host OSes including MS Windows, OS X, Linux and Solaris (https://www.virtualbox.org/wiki/Downloads). Once VirtualBox is installed on your host OS, you can then install a guest OS inside VirtualBox. VirtualBox supports a lot of different OSes (https://www.virtualbox.org/wiki/Guest_OSes).

Here are the steps to setting up a VM in VirtualBox with our image file:

1. Download and install VirtualBox for your OS: https://www.virtualbox.org/wiki/Downloads
2. Start VirtualBox and click New to start the Create New Virtual Machine wizard
3. Give the VM a useful name like "NGS Training" and choose Linux and either Ubuntu or Ubuntu (64-bit) as the OS Type
4. Give the VM access to a reasonable amount of the host Oses memory. i.e. somewhere near the top of the green. If this value is < 2000 MB, you are likely to have insufficient memory for your NGS data analysis needs.
5. For the virtual hard disk, select "Use existing hard disk" and browse to and select the `NGSTrainingV1.2.1.vdi` file you downloaded.
6. Confirm remaining settings
7. Select the "NGS Training" VM and click Start to boot he machine.
8. Once booted, log into the VM as either `ubuntu` (a sudoer user; i.e. has admin rights) or as `ngstrainee` (a regular unprivileged user). See table below for passwords.

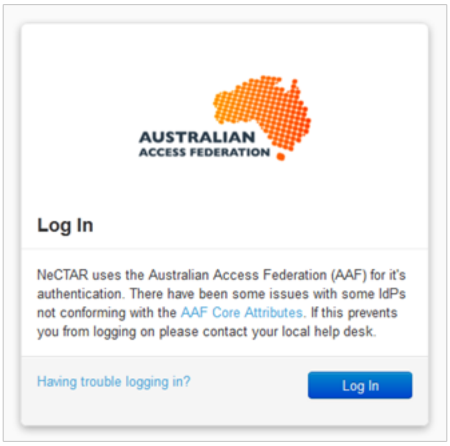
Setting up a VM using the NeCTAR Research Cloud

All Australian researchers, who are members of an institution which subscribes to the Australian Access Federation (AAF; http://www.aaf.edu.au/), have access to a small amount of computing resources (2 CPU's and 8 GBytes RAM) on the NeCTAR Research Cloud (http://nectar.org.au/research-cloud).

Login to the NeCTAR Research Cloud Dashboard

The online dashboard is a graphical interface for administering (creating, deleting, rebooting etc) your virtual machines (VMs) on the NeCTAR research cloud.

1. Go to the dashboard: http://dashboard.rc.nectar.org.au
2. When you see the following page, click the "Log In" button:

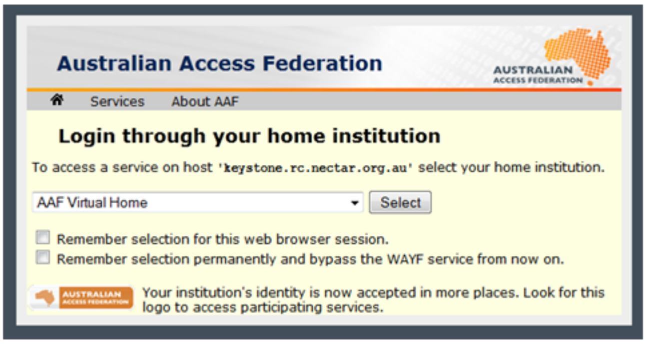


Log In

NeCTAR uses the Australian Access Federation (AAF) for it's authentication. There have been some issues with some IdPs not conforming with the AAF Core Attributes. If this prevents you from logging on please contact your local help desk.

Figure 11:

3. At the following screen, simply choose your institution from the dropdown box and click "Select". Now follow the on screen prompts and enter your regular institutional login details.



Login through your home institution

To access a service on host 'keystone.rc.nectar.org.au' select your home institution.

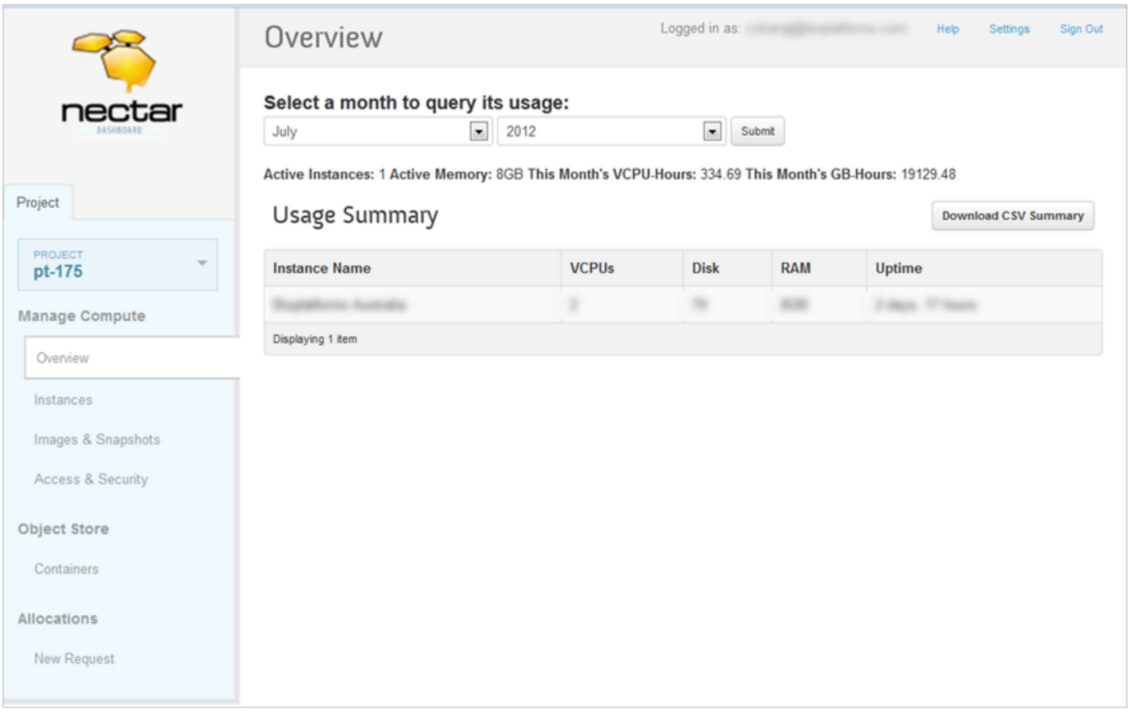
AAF Virtual Home ▾ Select

□ Remember selection for this web browser session.
□ Remember selection permanently and bypass the WAYF service from now on.

Your institution's identity is now accepted in more places. Look for this logo to access participating services.

Figure 12:

4. If you see the following screen, congratulations, you have successfully logged into the NeCTAR Research Cloud dashboard!



Overview

Logged in as: [redacted]

Select a month to query its usage:

July | 2012

Active Instances: 1 **Active Memory:** 8GB **This Month's VCPU-Hours:** 334.69 **This Month's GB-Hours:** 19129.48

Usage Summary

Instance Name	VCPUs	Disk	RAM	Uptime
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]

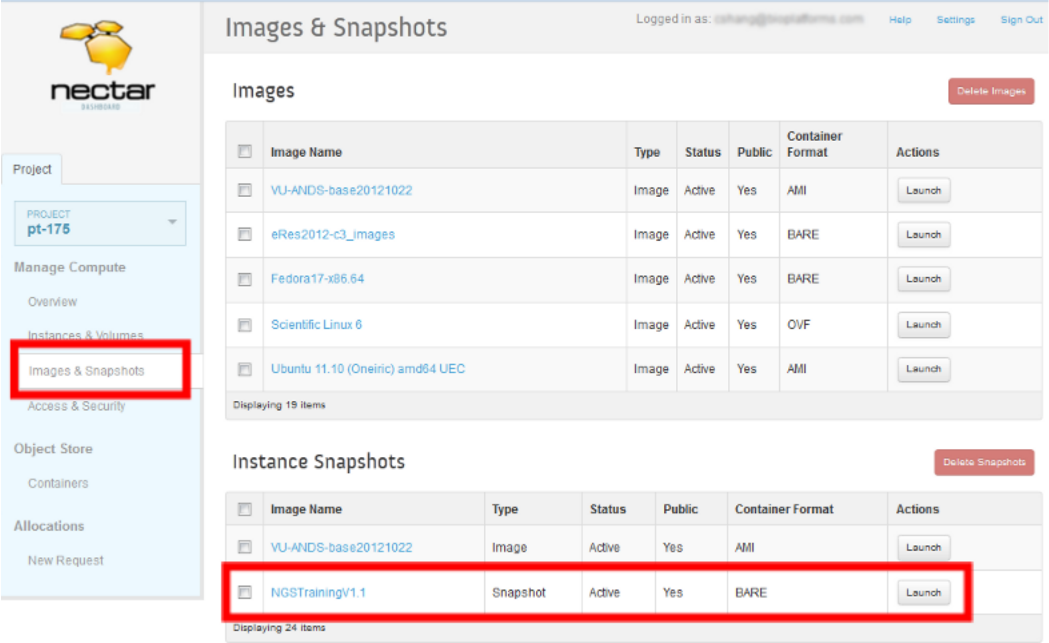
Displaying 1 item

Figure 13:

Instantiating Your Own VM

We will now show you how to instantiate the “NGS Training” image using your own personal cloud allocation.

1. In the NeCTAR Research Cloud dashboard, click “Images & Snapshots” to list all the publicly available images from which you can instantiate a VM. Under “Snapshots” Click the “Launch” button for the latest version of the “NGSTraining” snapshot:



Images & Snapshots

Images

□	Image Name	Type	Status	Public	Container Format	Actions
□	VU-ANDS-base20121022	Image	Active	Yes	AMI	Launch
□	eRes2012-c3_images	Image	Active	Yes	BARE	Launch
□	Fedora17-x86.64	Image	Active	Yes	BARE	Launch
□	Scientific Linux 6	Image	Active	Yes	OVF	Launch
□	Ubuntu 11.10 (Oneiric) amd64 UEC	Image	Active	Yes	AMI	Launch

Displaying 19 items

Instance Snapshots

□	Image Name	Type	Status	Public	Container Format	Actions
□	VU-ANDS-base20121022	Image	Active	Yes	AMI	Launch
□	NGSTrainingV1.1	Snapshot	Active	Yes	BARE	Launch

Displaying 24 items

Figure 14:

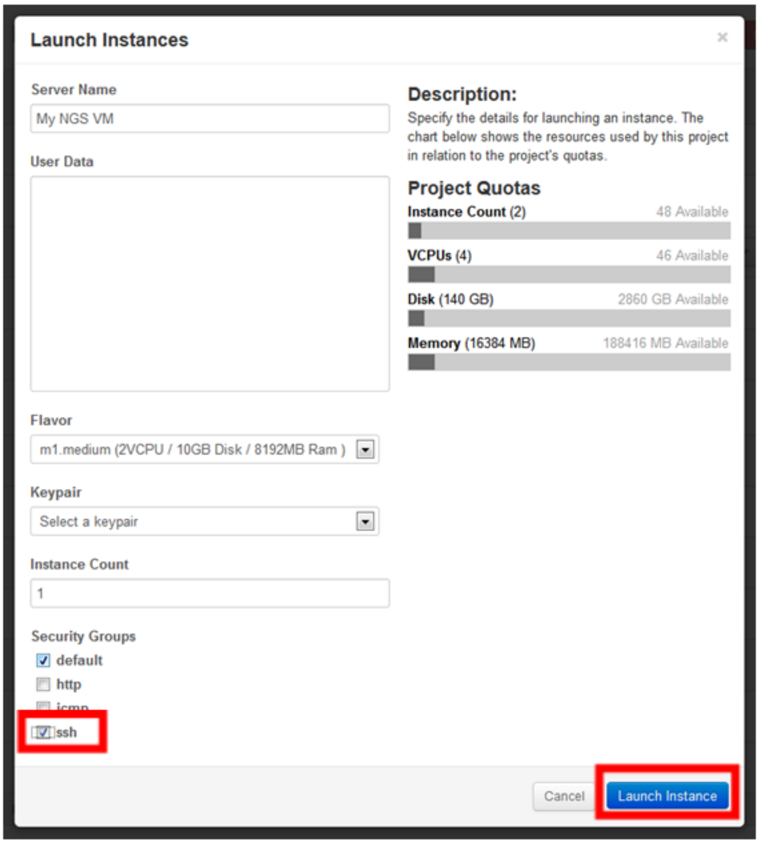
2. You will now see a "Launch Instances" window where you are required to enter some details about how you want the VM to be setup before clicking "Launch Instance". In the "Launch Instances" pop-up frame choose the following settings:

   Server Name A human readable name for your convenience. e.g. "My NGS VM"

   Flavor The resources you want to allocate to this VM. I suggest a Medium sized VM (2 CPUs and 8 GBytes RAM). This will use all your personal allocation, but anything less will probably be insufficient. You could request a new allocation of resources if you want to instantiate a larger VM with more memory.

   Security Groups Select SSH.

3. Click the "Launch Instance" button



Launch Instances

Server Name: My NGS VM

User Data

Flavor: m1.medium (2VCPU / 10GB Disk / 8192MB Ram)

Keypair: Select a keypair

Instance Count: 1

Security Groups

- ☑ default
- □ http
- □ icmp
- ☑ ssh

Description:

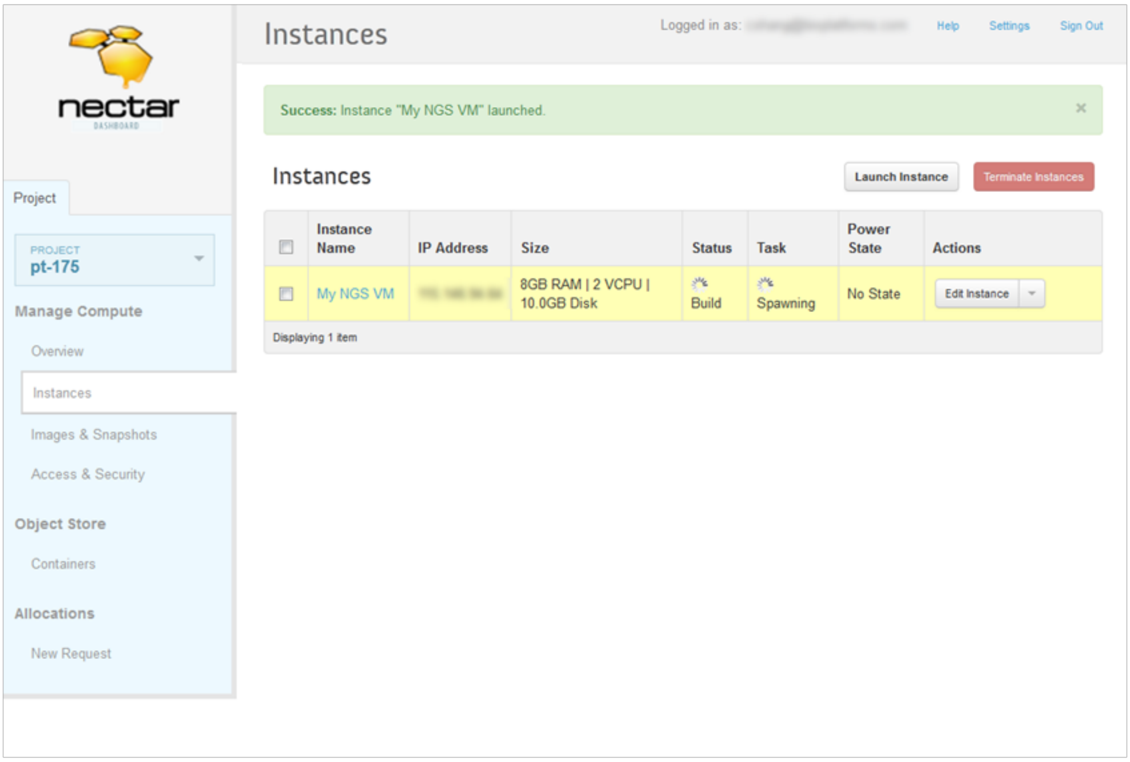
Specify the details for launching an instance. The chart below shows the resources used by this project in relation to the project's quotas.

Project Quotas

Instance Count (2)	48 Available
VCPUs (4)	46 Available
Disk (140 GB)	2860 GB Available
Memory (16384 MB)	188416 MB Available

Figure 15:

4. You will be taken to the “Instances” page and you will see the “Status” and “Task” column for your new VM is “Building” and “Spawning”. Once the “IP Address” cell is populated, take a note of it as you will need it for configuring the NX Client later on.



Instances

Success: Instance "My NGS VM" launched.

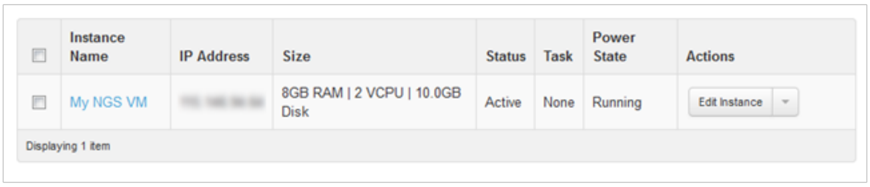
Instances

☐	Instance Name	IP Address	Size	Status	Task	Power State	Actions
☐	My NGS VM	[redacted]	8GB RAM | 2 VCPU | 10.0GB Disk	Build	Spawning	No State	Edit Instance

Displaying 1 item

Figure 16:

5. Once the Status and Task for the VM change to “Active and “None” respectively, your VM is powered up and is configuring itself. Congratulations, you have now instantiated a Virtual Machine! If you try to connect to the VM too quickly, you might not be successful. The OS may still be configuring itself, so give it a few minutes to finish before continuing.



☐	Instance Name	IP Address	Size	Status	Task	Power State	Actions
☐	My NGS VM	[redacted]	8GB RAM | 2 VCPU | 10.0GB Disk	Active	None	Running	Edit Instance

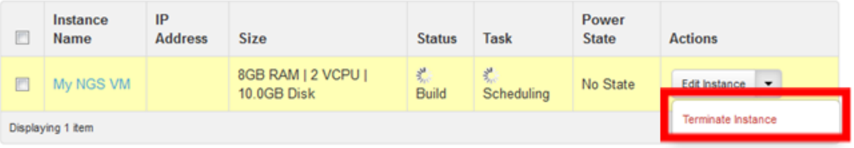
Displaying 1 item

Figure 17:

VM Stuck Building and Spawning

Sometimes, the cloud experiences a "hiccup" and a newly instantiated VM will get stuck in the "Build" and "Spawning" state (step 3) for more than a few minutes. This can be rectified by terminating the instance and creating a new VM from scratch:

1. Selecting "Terminate Instance" under the "Edit Instance" dropdown box:



□	Instance Name	IP Address	Size	Status	Task	Power State	Actions
□	My NGS VM		8GB RAM | 2 VCPU | 10.0GB Disk	Build	Scheduling	No State	Edit Instance ▾ Terminate Instance

Displaying 1 item

Figure 18:

2. Go back to step 1 of "Instantiating Your Own VM" and create the VM from scratch:

Remote Desktop with the NoMachine NX Client

During the workshop you were using the free NX client from NoMachine (`http://www.nomachine.com/`) to provide a remote desktop-like connection to VMs running on the NeCTAR Research Cloud. Therefore, we provide information on how to setup your local computer to connect to the VM you just instantiated in the steps above.

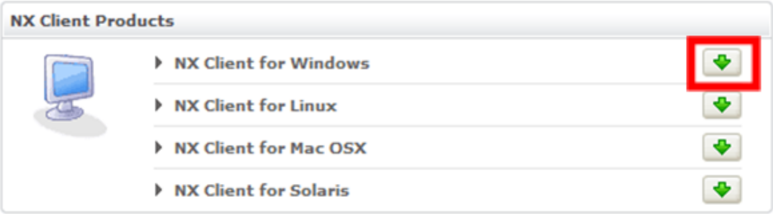
We assume that:

- You have administrator rights on your local computer for installing software.

NoMachine NX Client Installing

We show you instructions below for the MS Windows version of the NX Client, but procedures for other supported OSes (Linux, Mac OSX and Solaris) should be very similar.

1. Go to the NoMachine download page: `http://www.nomachine.com/download.php`
2. Click the download icon next to the NX Client for Windows:

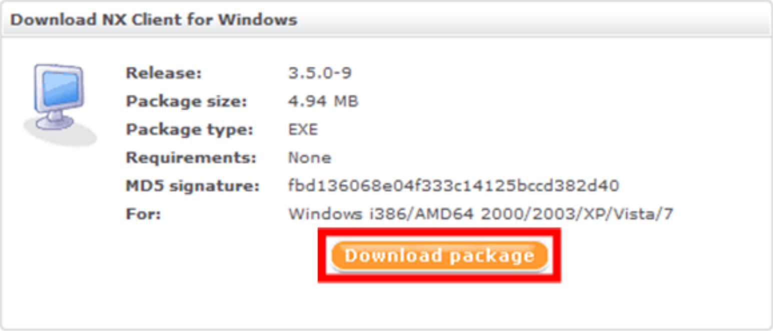


NX Client Products

- NX Client for Windows
- NX Client for Linux
- NX Client for Mac OSX
- NX Client for Solaris

Figure 19:

3. On the ”NX Client for Windows” page, click the ”Download package” button:



Download NX Client for Windows

Release:	3.5.0-9
Package size:	4.94 MB
Package type:	EXE
Requirements:	None
MD5 signature:	fbd136068e04f333c14125bccd382d40
For:	Windows i386/AMD64 2000/2003/XP/Vista/7

Figure 20:

4. Run the file you just downloaded (accepting defaults is fine)
5. Congratulations, you just installed the NoMachine NX Client!

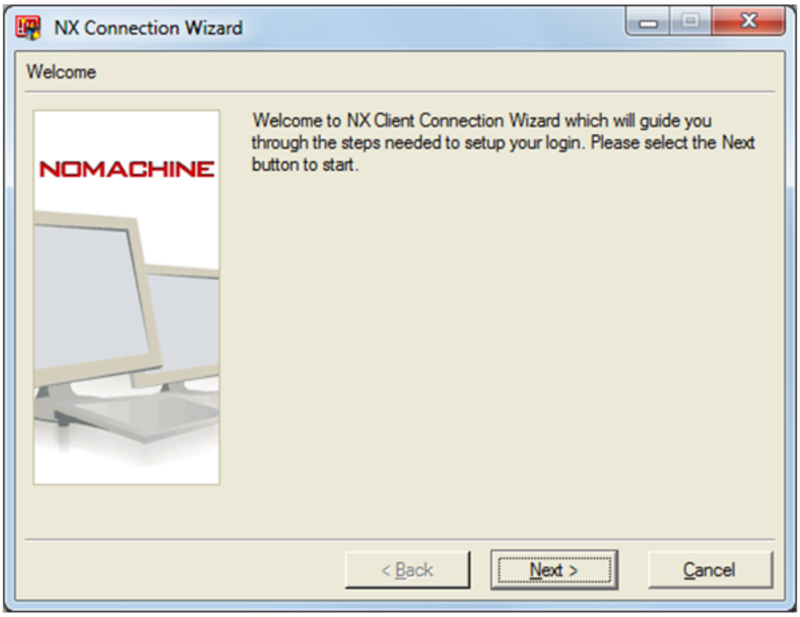
NoMachine NX Client Configuration

Now we have the NoMachine NX Client installed, we need to configure a new NX ”session” which will point to the VM we instantiated in the NeCTAR Research Cloud.

We assume that:

- You know the IP address of the VM you want to remote desktop into.

1. Start the NX Connection Wizard and click ”Next” to advance to the ”Session” settings page.


NX Connection Wizard

Welcome

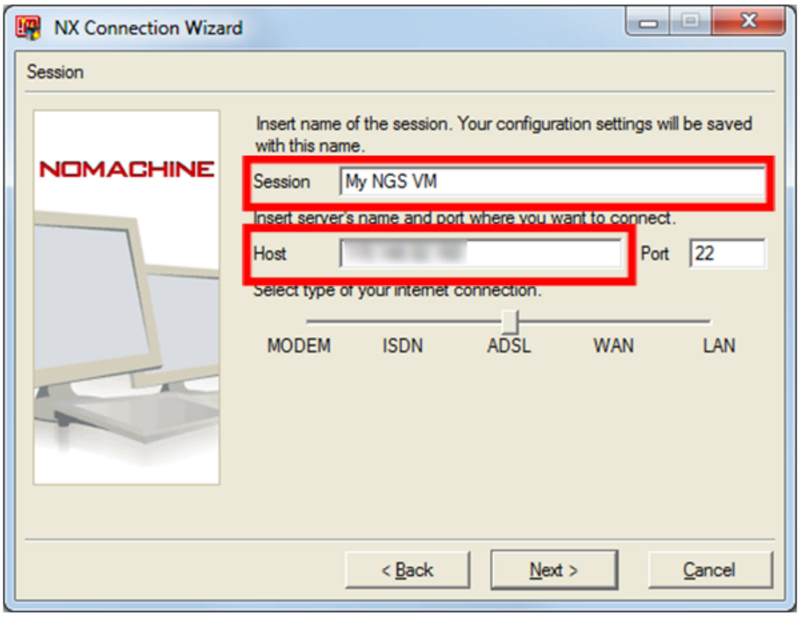
Welcome to NX Client Connection Wizard which will guide you through the steps needed to setup your login. Please select the Next button to start.

Figure 21:

2. On the ”Sessions” settings page enter the following details:

   Session A memorable name so you know which VM this session is pointing at. You could use the same name you chose for the VM you instantiated earlier e.g. ”NGS Training”.

   Host Enter the IP address of the VM you instantiated on the NeCTAR Research Cloud.


NX Connection Wizard

Session

Insert name of the session. Your configuration settings will be saved with this name.

Session: My NGS VM

Insert server's name and port where you want to connect.

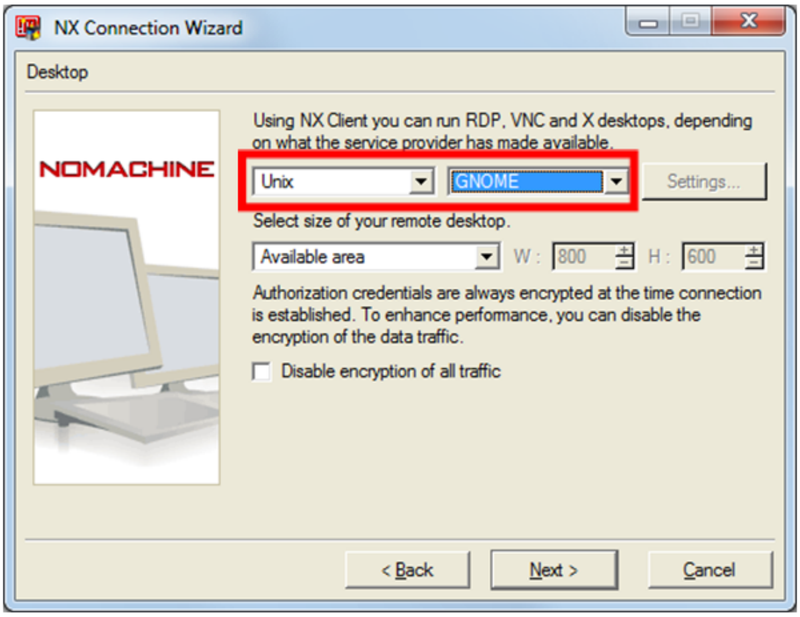
Host: [redacted] Port: 22

Select type of your internet connection.

MODEM ISDN ADSL WAN LAN

Figure 22:

3. Click ”Next” to advance to the ”Desktop” settings page. You should use the ”Unix GNOME” setting.



Desktop

Using NX Client you can run RDP, VNC and X desktops, depending on what the service provider has made available.

Unix GNOME

Select size of your remote desktop.

Available area W: 800 H: 600

Authorization credentials are always encrypted at the time connection is established. To enhance performance, you can disable the encryption of the data traffic.

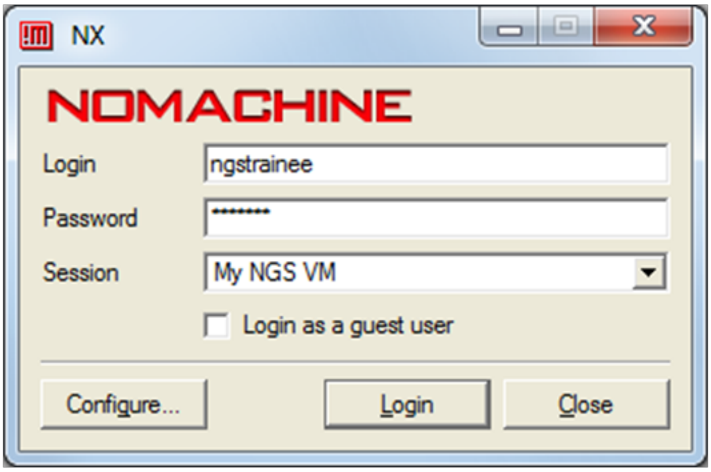
□ Disable encryption of all traffic

Figure 23:

4. Click ”Next” and ”Finish” to complete the wizard.

Connecting to a VM

If you just completed the NX Connection Wizard described above, the wizard should have opened the NX Client window. If not, run the ”NX Client”. You will be presented with a window like this:



Login ngstrainee

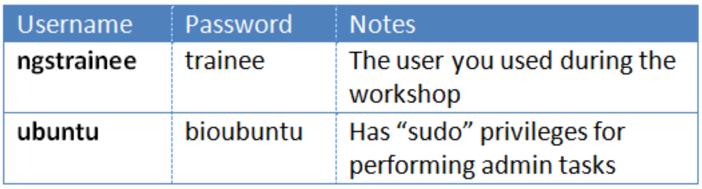
Password ********

Session My NGS VM

□ Login as a guest user

Figure 24:

The ”Login” and ”Password” boxes in the NX Client are for user accounts setup on the VM. By default our image, from which you instantiated your VM, has two preconfigured users:



Username	Password	Notes
ngstrainee	trainee	The user you used during the workshop
ubuntu	bioubuntu	Has “sudo” privileges for performing admin tasks

Figure 25:

Unless you know what you are doing, we suggest you use the `ngstrainee` user account details to initiate an NX connection to your VM. In less than a minute, you should see an NX Window showing the desktop of your VM:

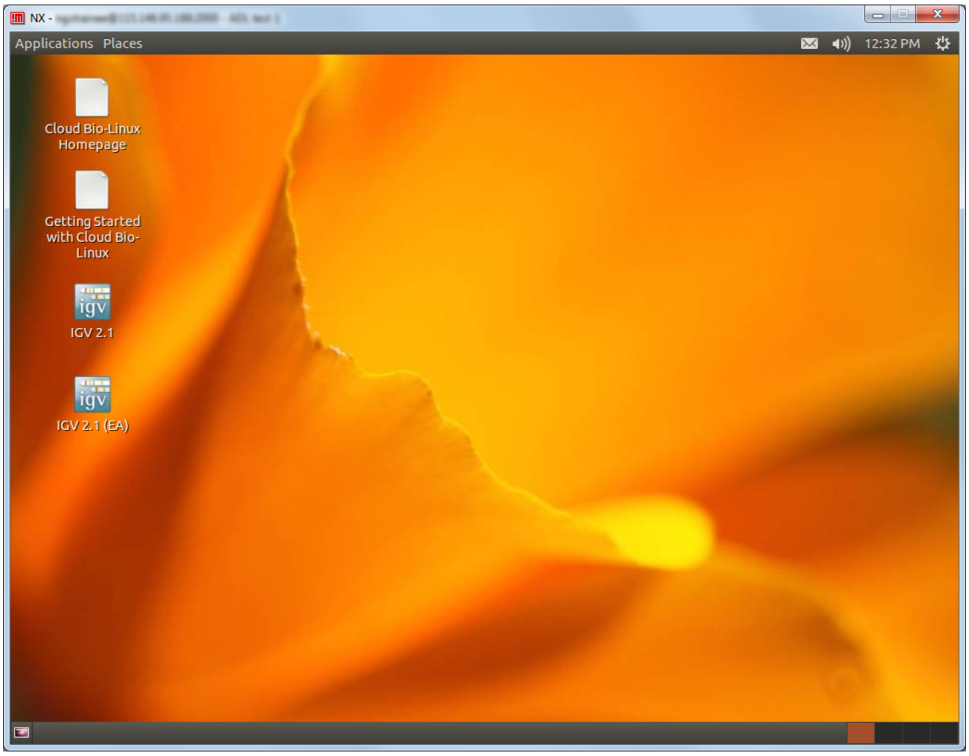


Figure 26:

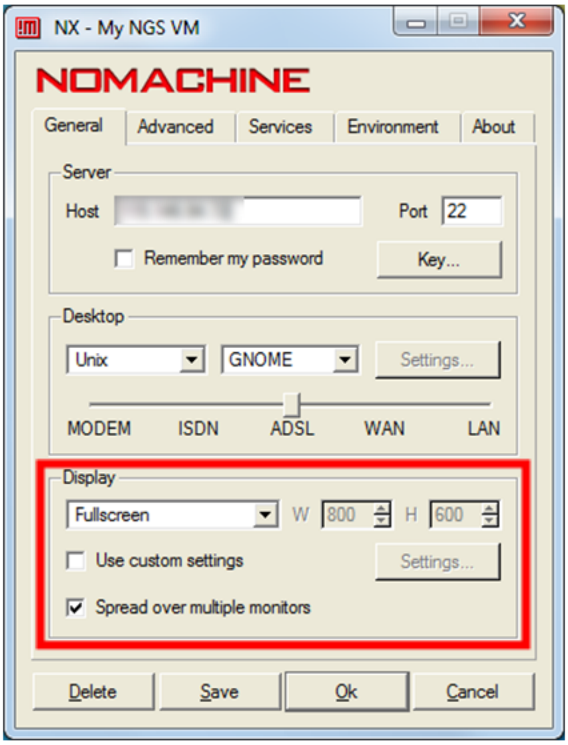
NX Connection Failure

In the event that you don't get the NX Window with your VM's desktop displaying inside it. The most common errors are:

- You failed to select the "ssh" security group when instantiating the VM. You'll need to terminate the instance and create a new VM from scratch
- You failed to select "Unix GNOME" when you configured the NX Client session. You'll need to reconfigure the session using the NX Client
- Your institutions firewall blocks TCP port 22. You may need to request this port to be opened by your local network team or configure the NX client to use a proxy server.

Advanced Configuration

In the session configuration, you can configure the size of the NX Window in which the desktop of the VM is drawn:



NX - My NGS VM

Server

Host [redacted] Port 22

□ Remember my password

Desktop

Unix GNOME

Display

Fullscreen W 800 H 600

□ Use custom settings

☑ Spread over multiple monitors

Figure 27:

This can be useful if you want to:

- Have the NX Window occupy the entire screen, without window decorations. This is often desirable if you wish to ”hide” the host OS from the person sitting at the computer running the NX Client.
- Have the NX Window spread over multiple monitors.

Access to Workshop Documents

This document has been written in LaTeX and deposited in a public github repository (https://github.com/nathanhaigh/ngs_workshop). The documentation has been released under a Creative Commons Attribution 3.0 Unported License (see the Licence page at the beginning of this handout).

For convienience, you can access up-to-date PDF versions of the LaTeX documents at:

Trainee Handout
https://github.com/nathanhaigh/ngs_workshop/raw/master/trainee_handout.pdf

Trainer Handout
https://github.com/nathanhaigh/ngs_workshop/raw/master/trainer_handout.pdf

Access to Workshop Data

Once you have created a VM from our image file, either locally using VirtualBox or on the NeCTAR Research Cloud, you can configure the system with the workshop documents and data. This way you can revisit and work through this workshop in your own time.

In order to do this, we have provided you with access to a shell script which should be executed on your NGS Training VM by the ubuntu user. This pulls approx. 3.3 GBytes of data from the NeCTAR Cloud storage and configures the system for running this workshop:

```
# As the ubuntu user run the following commands:
cd /tmp
wget https://github.com/nathanhaigh/ngs_workshop/raw/master/\
workshop_deployment/NGS_workshop_deployment.sh
bash NGS_workshop_deployment.sh
```

While you're at it, you may also like to change the timezone of your VM to match that of your own. To do this simply run the following commands as the ubuntu user:

```
TZ="Australia/Adelaide"
echo "$TZ" | sudo tee /etc/timezone
sudo dpkg-reconfigure --frontend noninteractive tzdata
```

For further information about what this script does and possible command line arguments, see the script's help:

```
bash NGS_workshop_deployment.sh -h
```

For further information about setting up the VM for the workshop, please see:

https://github.com/nathanhaigh/ngs_workshop/blob/master/workshop_deployment/README.md

Space for Personal Notes or Feedback

Space for Personal Notes or Feedback

Space for Personal Notes or Feedback

An introduction to Linux for bioinformatics

Paul Stothard

January 17, 2016

Contents

1 Introduction

Linux is a free operating system for computers that is similar in many ways to proprietary Unix operating systems. The field of bioinformatics relies heavily on Linux-based computers and software. Although most bioinformatics programs can be compiled to run on Mac OS X and Windows systems, it is often more convenient to install and use the software on a Linux system, as pre-compiled binaries are usually available, and much of the program documentation is often targeted to the Linux user. For most users, the simplest way to access a Linux system is by connecting from their primary Mac or Windows machine. This type of arrangement allows several users to run software on a single Linux system, which can be maintained by an experienced systems administrator. Although there are other ways for inexperienced users to become familiar with Linux (installing Linux on a PC, using a Live CD to run Linux, running a Linux virtual machine), this document focuses on accessing a remote Linux machine using a text-based terminal. Many powerful statistics and bioinformatics programs can be run in this manner.

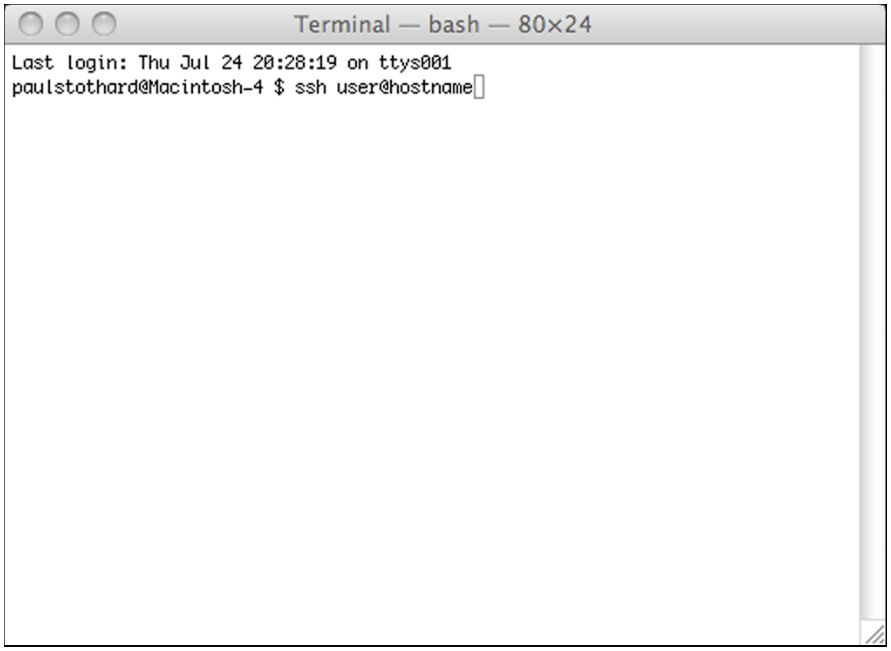
2 Getting started

2.1 Obtaining a Linux user account

To gain access to a Linux-based machine you first need to speak a system administrator (sysadmin) to obtain a user name, hostname (or IP address), and password. Once you have this information you can access your account. Alternatively, you can run a Linux virtual machine on top of your present operating system. If you are using a Linux virtual machine you can skip ahead to the section called "Your home directory" after launching the machine and opening a Bash terminal).

2.2 How to access your account from Mac OS X

Mac OS X includes a Terminal application (located in the **Applications** >> **Utilities folder**), which can be used to connect to other systems (Figure 2.2). Launch Terminal and at the command prompt, enter **ssh user@hostname**, replacing "user" and "hostname" with the user name and machine name you have been assigned. Press enter and you should be prompted for a password. The first time you try to connect to your account, a warning message may appear. Ignore the message and allow the connection to be established.



```
Last login: Thu Jul 24 20:28:19 on ttys001
paulstothard@Macintosh-4 $ ssh user@hostname
```

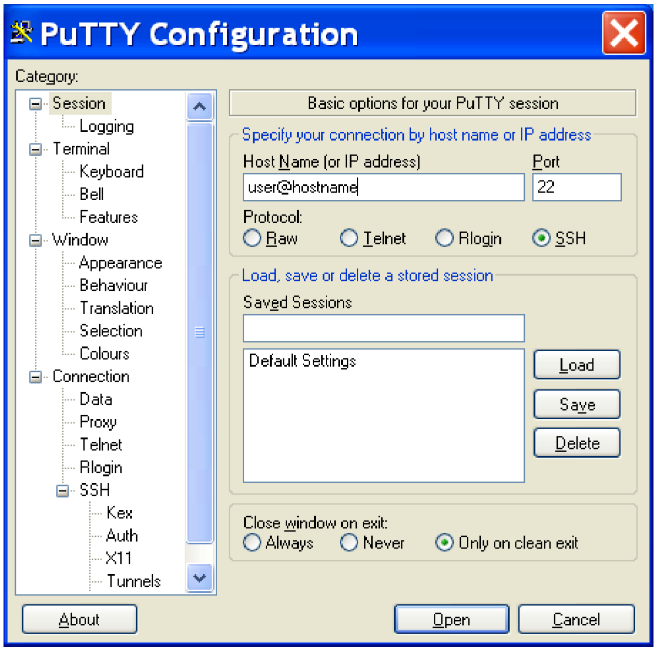
Figure 1: The Mac OS X Terminal application.

2.3 How to access your account from Windows

2.3.1 Using PuTTY

On Windows systems you can use a variety of programs to connect to a Linux system. PuTTY[1] is a free program already available on many Windows machines, including most of the student-accessible computers at the University of Alberta. If PuTTY is not installed you can download an executable from the PuTTY website. Launching PuTTY will open a configuration window resembling the one shown in Figure 2.3.1. Click **Session** in the left pane and then in the **Host Name (or IP address)** text box enter **user@hostname**, replacing “user” and “hostname” with the user name and machine name you have been assigned. Click **Open** to establish a connection with the remote system. The first time you try to connect to your account, a warning message may appear. Ignore the message and allow the connection to be established. A new terminal window will open, and you will be prompted for a password.

[1] http://www.chiark.greenend.org.uk/~sgtatham/putty/



Host Name (or IP address): user@hostname

Port: 22

Protocol: ○ Raw ○ Telnet ○ Rlogin ● SSH

Saved Sessions: Default Settings

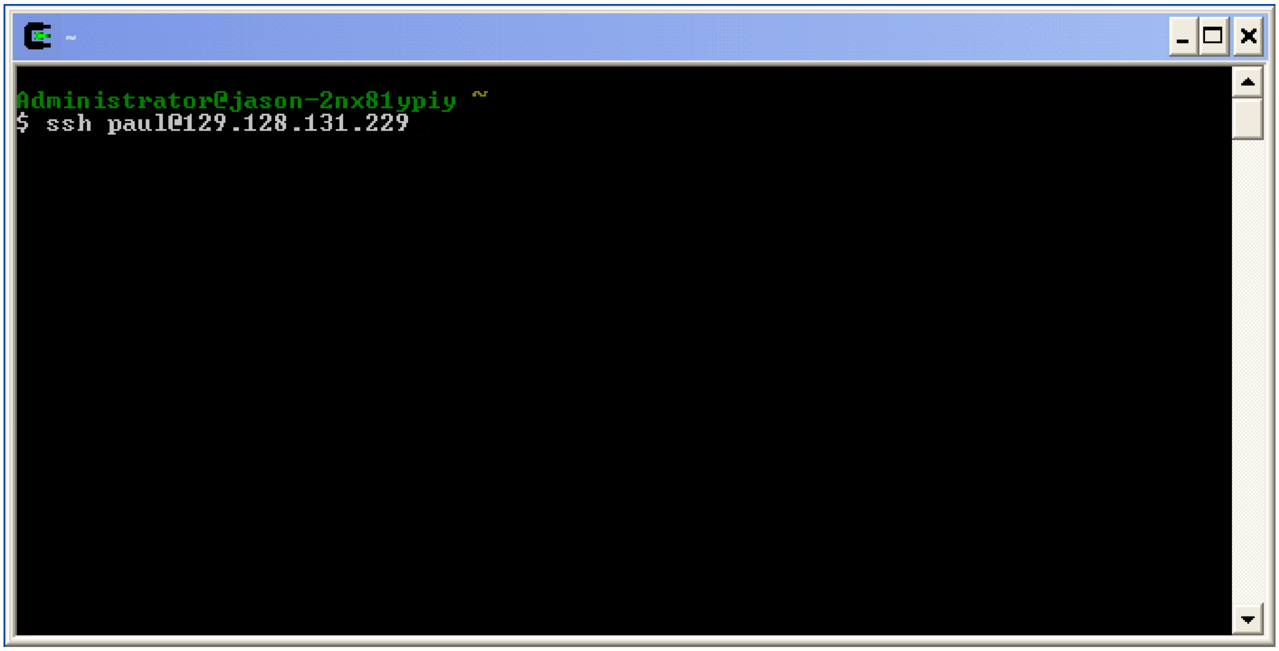
Close window on exit: ○ Always ○ Never ● Only on clean exit

Figure 2: The PuTTY Telnet/SSH client running on Windows.

2.3.2 Using Cygwin

Cygwin[2] is a program that can be installed on Windows to provide a Linux-like environment. An advantage of using Cygwin is that you gain access to a lot of the standard Linux utilities, without having to connect to another computer. For example, after launching Cygwin, you can use many of the Linux commands described elsewhere in this guide, such as **pwd** and **mkdir**. You can also edit and run Bash scripts and Perl programs, and you can manage software repositories using subversion (these topics, apart from Bash scripts, are not covered in the tutorial). If you choose to install Cygwin, you can use it to access your remote Linux account by entering **ssh user@hostname** (replace "user" and "hostname" with the user name and machine name you have been assigned) on the command line (Figure 2.3.2).

[2] `http://www.cygwin.com/`



```
Administrator@jason-2nx81ypiy ~
$ ssh paul@129.128.131.229
```

Figure 3: Cygwin, a Linux-like environment for Windows.

2.4 Your home directory

Once you have successfully signed in to your account, you can start exploring the directory structure of the remote computer. In your terminal you should see a command prompt, which usually consists of your user name and the name of the computer on which your account resides.

The examples in this guide will include **$** as the command prompt to illustrate that the commands should be entered into the terminal, and to differentiate the entered commands from the output they return.

When you sign in you will be located in your home directory. To see where this directory is located in the file system, use the **pwd** command:

```
pwd
```

In this case the output indicates that the current working directory is **paul** and that the **paul** directory is located inside of the **home** directory. When you enter **pwd** after signing in it should show that you are in a directory with the same name as your user name. The

home directory is located inside the **/** directory, which is also called the “root” directory. If at any time you want to return to your home directory, use the **cd ~** command. As you will see, your home directory is where you can create and delete your own files and directories.

2.5 Some basic commands

As in the previous example, you can see which directory is the current directory by using the **pwd** command:

```
pwd
```

To change to a different directory, use the **cd** command (**cd** means change directory):

```
cd /home
pwd
```

Now you should be in the **home** directory. To see what is inside of this directory, use the **ls** command (**ls** stands for list):

```
ls
```

The folders you see will likely differ from these. Now switch back to your home directory:

```
cd ~
```

In addition to real directory names, you can supply certain alias terms to the **cd** command. One of these is the ~ character, which represents your home directory. Another is **..**, which represents the directory above the current directory. Try the following:

```
cd ~
cd ..
ls
pwd
cd ~
pwd
```

As you can see, **cd**, **ls**, and **pwd** can be used to explore the Linux file system. Don’t forget that you can always use **cd ~** to move back to your home directory.

2.6 More commands and command-line options

The **pwd**, **ls**, and **cd** commands point to programs on the Linux system that perform specific tasks and return output. When you enter the commands, the system runs the corresponding program for you. There are many other useful commands available.

To see which user you are signed in as, use the **whoami** command:

```
whoami
```

To see who else is signed in to the same system, use the **who** command:

```
who
```

To see the current time and date, using the **date** command:

```
date
```

To create your own directories use the **mkdir** (make directory) command:

```
cd ~
mkdir seqs
cd seqs
mkdir proteins
cd proteins
pwd
```

To create a new file, use the **touch** command:

```
cd ~/seqs/proteins
touch my_sequence.txt
ls -l
```

As you will see later, there are other ways to create new files (output redirection for example). In the last command above, the **-l** (a lowercase “L”, not a “1”) option was used with the **ls** command. The **-l** indicates that you want the directory contents shown in the “long listing” format. Most commands accept a variety of options. To see which options are available for a certain command, you can try typing **man** followed by the command name (**man ls** for example to see what options are available for the **ls** command), or the command name followed by **--help** (**ls --help** for example). One of these two methods usually provides information. To see what options can be used with **ls**, enter **man ls**. To get through the list of options that appears, keep pressing Space until the page stops scrolling, then enter “q” to return to the command prompt:

```
man ls
```

To delete a file, use the **rm** (remove) command:

```
cd ~/seqs/proteins
rm my_sequence.txt
ls
```

To remove a directory, use the **rmdir** (remove directory) command:

```
cd ~/seqs
rmdir proteins
ls
```

To copy a file, use the **cp** (copy) command:

```
cd ~/seqs/
touch testfile1
cp testfile1 testfile2
ls
```

To rename a file, or to move it to another directory, use the **mv** (move) command:

```
cd ~
touch testfile3
mv testfile3 junk
mkdir testdir
mv junk testdir
cd testdir
ls
```

The commands covered so far represent a small but useful subset of the many commands available on a typical Linux system [1].

3 Transferring files to and from your Linux account

3.1 Transferring files between Mac OS X and Linux

3.1.1 Using the Terminal application

Recall that Mac OS X includes a Terminal application (located in the **Applications** >> **Utilities** folder), which can be used to connect to other systems. This terminal can also be used to transfer files, thanks to the **scp** command.

Try transferring a file from your Mac to your Linux account using the Terminal application:

1. Launch the Terminal program.

2. Switch to your home directory on the Mac using the command **cd ~**.

3. Create a text file containing your home directory listing using **ls -l > myfiles.txt** (you will learn more about the meaning of **>** later).

4. Now use the **scp** command on your Mac to transfer the file you created to your Linux account. This command requires two values: the file you want to transfer and the destination. Be sure to replace "user" with your user name, and replace hostname with the real hostname or IP address of the Linux system you want to connect to:

   ```
   scp myfiles.txt user@hostname:~/
   ```

You should be prompted for your user account password. Remember, in the above example you are running the **scp** command on your Mac, not from your Linux account.

Now, delete the **myfiles.txt** file on your Mac, and see if you can use **scp** to retrieve the file from your Linux account:

1. In the terminal on your Mac, switch to your home directory using **cd~**.

2. Delete the **myfiles.txt** file using **rm myfiles.txt**.

3. Use the **scp** command to copy **myfiles.txt** from your Linux account back to your Mac. Remember to replace "hostname" and "user" with the appropriate values when you enter the command:

```
scp user@hostname:~/myfiles.txt ./
```

The above command will prompt you for your Linux account password. Remember that **./** means “current directory”. This informs the **scp** program that you would like the file **myfiles.txt** in your home directory on the remote system to be copied to the current directory on your computer.

3.1.2 Using Fugu

For users who prefer to use a graphical interface when transferring files between Mac and Linux, there is the freely available Fugu program.[3] To use Fugu, launch the program and enter the hostname of the computer you wish to connect to in the **Connect to** text area, and enter your Linux account name in the **Username** text area (Figure 3.1.2). Click **Connect** to connect to the remote system. You will be prompted for your Linux account password. Once you are connected to your Linux account you should be able to copy files between systems by dragging files and folders.

3.2 Transferring files between Windows and Linux

The simplest way to transfer files between Linux and Windows is to use the freely available WinSCP program.[4] WinSCP is already installed on many Windows systems, including those provided for student use at the University of Alberta. To use WinSCP, launch the program and enter the appropriate information into the **Host name**, **User name**, and **Password** text areas (Figure 3.2). Click **Login** to connect to the remote system. Once you are connected you should be able to transfer files and directories between systems using the simple graphical interface.

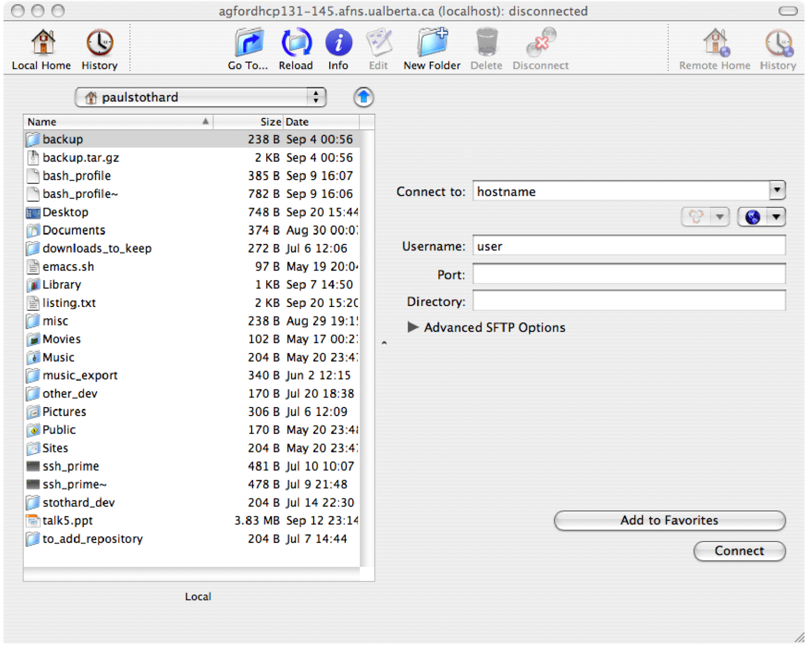
3.3 File transfer exercise

To test your ability to transfer files to your Linux account, download the following file to your Mac or Windows system using a web browser:

`http://www.ualberta.ca/~stothard/downloads/sample_sequences.zip`

[3] `http://rsug.itd.umich.edu/software/fugu/`

[4] `http://sourceforge.net/projects/winscp/`



paulstothard

Name	Size	Date
backup	238 B	Sep 4 00:56
backup.tar.gz	2 KB	Sep 4 00:56
bash_profile	385 B	Sep 9 16:07
bash_profile~	782 B	Sep 9 16:06
Desktop	748 B	Sep 20 15:44
Documents	374 B	Aug 30 00:0
downloads_to_keep	272 B	Jul 6 12:06
emacs.sh	97 B	May 19 20:0
Library	1 KB	Sep 7 14:50
listing.txt	2 KB	Sep 20 15:2
misc	238 B	Aug 29 19:1
Movies	102 B	May 17 00:2
Music	204 B	May 20 23:4
music_export	340 B	Jun 2 12:15
other_dev	170 B	Jul 20 18:38
Pictures	306 B	Jul 6 12:09
Public	170 B	May 20 23:4
Sites	204 B	May 20 23:4
ssh_prime	481 B	Jul 10 10:07
ssh_prime~	478 B	Jul 9 21:48
stothard_dev	204 B	Jul 14 22:30
talk5.ppt	3.83 MB	Sep 12 23:14
to_add_repository	204 B	Jul 7 14:44

Connect to: hostname

Username: user

Port:

Directory:

▶ Advanced SFTP Options

Figure 4: Fugu, a graphical SSH and SCP tool for Mac OS X.

Once the file has been downloaded to your system, use the file transfer methods outlined above to transfer the file to your Linux account. Be sure to perform this exercise, as the sample_sequences.zip file will be used later on in this tutorial.

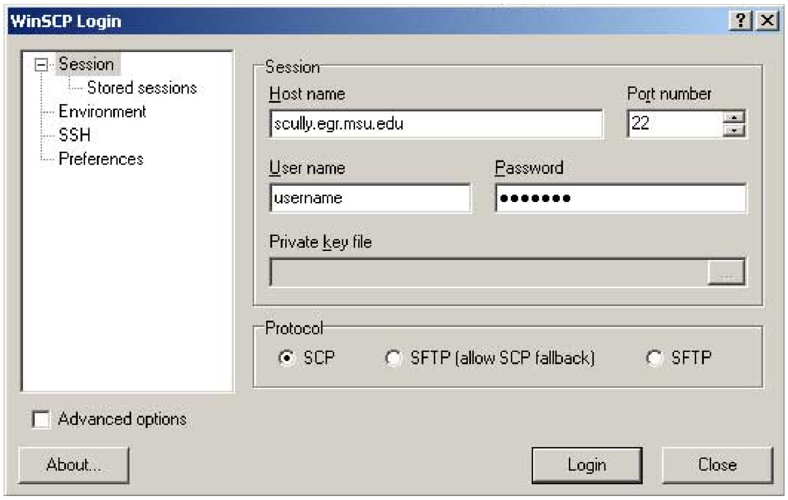
4 Understanding Linux

4.1 Paths

Many commands require that you supply a directory or file name. For example, if you enter the command **touch** without specifying a file name, an error message is returned:

```
touch
```

Directory and file names like “testdir” and “my_sequences.txt” are called relative paths, since they specify the location of the file or directory in relation to the current working directory. For example, if you are located in your home directory, the command



Host name: scully.egr.msu.edu
Port number: 22
User name: username
Password: ●●●●●●●
Private key file:
Protocol: ● SCP ○ SFTP (allow SCP fallback) ○ SFTP
□ Advanced options

Figure 5: WinSCP.

mkdir some_dir will create a directory called "some_dir" in your home directory.

In the following example two directories are created, and **cd** is used to switch to "dir2" so that a new file can be created there using **touch**:

```
cd ~
mkdir dir1
cd dir1
mkdir dir2
cd dir2
touch somefile
```

Alternatively, by specifying the names of the directories in the paths, the same directory structure and file can be created without using **cd** to switch to the new directories (in this example the **rm -rf dir1** is used to remove the existing "dir1" and all of its contents):

```
cd ~
rm -rf dir1
mkdir dir1
mkdir dir1/dir2
touch dir1/dir2/somefile
```

Relative paths can use **..** to refer to the parent directory (i.e. the directory above the current directory). In the following example, **..** is used twice in the path passed to **touch**, to create a file called “somefile2” in the directory two levels up from the current directory (which in this case is your home directory):

```
cd ~
cd dir1/dir2
```

In contrast to relative paths, absolute paths specify the name of a file or directory in relation to the root (top) directory. Absolute paths always begin with a forward slash (the forward slash at the beginning of a path represents the root directory). In the following example, an absolute path is used to instruct **ls** to list the contents of the “etc” directory, which is used by Linux to store configuration files (i.e. the “etc” directory located in the root directory):

```
ls /etc
```

Absolute paths are useful because their interpretation doesn’t depend on which directory is the current working directory.

4.2 Typing shortcuts

There are some useful tricks to save typing on the command line. One is to use Tab to complete a command name or a file name. When you press Tab, the system will try to complete the text you have partially entered, based on which characters are found in known command names and file names. If there are multiple possible matches, the system will not guess the matching text, however, if you press Tab twice a list of the possible matches will be given. This method of using Tab on the command line is called “Tab completion”.

Try the following:

```
cd ~
mkdir a_new_directory
```

Now begin by typing “cd a” and press Tab instead of entering the full directory name. The full name should appear automatically.

To see what happens when there are multiple possibilities for a command or filename, begin by typing “mk” and press Tab twice. You should see a list of commands starting with the letters “mk”.

Another useful command-line shortcut is to use the up and down arrow keys to scroll through commands you have recently used (**ls** is sometimes not stored in this list since it is easy to type). If you scroll to a command you want to use again, press Enter to execute the command.

4.3 File permissions

Linux uses file permissions to prevent accidental file deletion and to protect data from being manipulated by others. Each file and directory is associated with three types of file permissions: “user”, “group”, and “other” permissions. The meanings of these terms are discussed below. To see the permission information for a directory or file, use the **ls** command with the **-l** option. Try the following set of commands:

```
cd ~
mkdir somedir
cd somedir
touch somefile
mkdir anotherdir
ls -l
```

The permission information for the directory **anotherdir** and the file **somefile** is given in the file listing. The first column is the file type and the file permissions (**drwxr-xr-x** for example). The third column is the owner of the file or directory (probably you), and the column after that is the group that owns the file. A group is simply a collection of users (groups are created by the sysadmin). A group can be used, for example, to allow different users to collaborate on a particular set of files, while protecting the files from editing by users not in the designated group.

As mentioned above, the first column contains the file type and permission information. The **anotherdir** entry begins with **d**, indicating it is a directory. The **somefile** entry begins with a **-**, which indicates that it is a file. The first three letters after the file type letter are the permissions for the user who owns the file or the directory. The next three letters are the permissions for the group that owns the file or directory. The final three letters define the access permissions for other users. The meanings of the letters are the following:

r (read permission) indicates that the file can be read. In the case of a directory this means that the contents of the directory can be listed.

w (write permission) indicates that the file can be modified. In the case of a directory this means that the contents of the directory can be changed (i.e. create new files, delete existing files, or rename files).

x (execute permission) indicates that the file can be executed as a program. In the case of a directory, the execute attribute means you have permission to enter a directory (i.e. make it the current working directory).

Returning to the example above, the permissions for **somefile** are **rw-r--r--**. This series of characters means that the owner of the file has the read and write permissions (**rw-**). Other users in the group users can read the file but not write or execute it (**r--**). Similarly, all other users can only read the file (**r--**).

To change file permissions, use the **chmod** command. For example, the following changes the permissions associated with **somefile** so that only the owner of the file (**paul** in this case) can read it (along with the **root** user, who will be discussed later):

```
chmod go-r somefile
ls -l
```

The **go-r** portion of the above **chmod** command means "from the group (**g**) and other (**o**) permission sets take away the read permission (**r**)."

To allow everyone to read the file **somefile** you could modify the permissions using the following:

```
chmod a+r somefile
```

The **a** in the above command means "all users". To refer to different types of users separately, use **u** (user who owns the file), **g** (group that owns the file), and **o** (other users).

To see how permissions protect files and directories, try to delete the **/etc** directory, which contains important system files:

```
rmdir /etc
```

Although the full rationale behind permissions may not be apparent to you at this time, it is important to remember that they do exist and that they control who can do what to specific files and directories. These permissions also automatically apply to any program you run on a Linux system. For example, if you run a program that attempts to copy a file for which you do not have the read permission, the program will be denied access to the file and it will not be able to make the copy.

4.4 Redirecting output

Many commands return textual output (i.e. the **ls** command) that is written to the terminal window. You can redirect the output to a file instead, by providing a filename preceded by the '>' character. Use the following to create a file called **my_listing.txt** containing the output of the **ls -l** command:

```
cd ~
ls -l > my_listing.txt
```

To examine the contents of the **my_listing.txt** file, use the **more** command:

```
more my_listing.txt
```

Note that **more** is useful for viewing the contents of a file, one page at a time. To advance a page press Space. To return to the command prompt, enter "q".

Output redirection is useful for commands that return a lot of output. It is often used so that the output can be used or processed at a later time.

4.5 Piping output

With pipes, which are represented by the | character, it is possible to send the output of one program to another program as input. Consider the following command:

```
cat /etc/services | sort | tail -n 10
```

The above example uses the **cat** command to extract the text from the file **/etc/services**. The text is then piped to the **sort** program, which sorts the lines alphanumerically. Finally, the sorted text is piped to the **tail** program, which displays the last 10 lines of the text. Piping provides a convenient way to perform a series of data manipulations.

4.6 Using locate and find

Occasionally you may want to search a Linux system for a particular file. A simple way to do this is to use the **locate** command:

```
locate blastall
```

In this example the **locate** program was used to search for files or directories matching the name "blastall" (**blastall** is a program that can be used to search DNA and protein sequence databases). The **locate** program does not actually search the Linux file system. Instead it uses a database that is usually updated daily. By using an optimized database, **locate** is able to find items quickly, however you may not obtain results for files recently added to the system.

Another tool for searching for files of interest is **find**. This command accepts several options for specifying the types of files you want. For example, you can search for files based on name, size, owner, modification date, and permissions.

To find files in the **/etc** directory that end with ".conf" and that are more than 10 kilobytes in size you could use this command:

```
find /etc -name "*.conf" -size +10k
```

4.7 Working with tar and zip files

Sometimes you may want to compress a file or a group of files into a zip file or a tar file. Alternatively, you may have a tar or zip file you wish to extract. Note that tar files, which are similar to zip files, are frequently used on Linux-based systems.

To explore the **zip** and **tar** commands, first, create a text file using the following:

```
cd ~
wget www.google.ca -O google.html
```

The **wget** command can be used to download web-based files. In this example it is used to write the google homepage to a file called google.html.

To create a zip file of **google.html** use the **zip** command:

```
zip -r google.zip google.html
rm google.html
```

The **-r** (for recursive) is not necessary in the above example. However, it is needed if you want to if you want to zip the contents of a directory.

To extract this zip file use the **unzip** command:

```
unzip google.zip
```

To create a tar file use the **tar** command:

```
tar -cvf google.tar google.html
```

The **-c** option tells **tar** that you would like to create an archive (as opposed to extract one), and **-v** indicates that you want the **tar** program to be verbose (i.e. print comments and progress messages). **-f** is used to specify the name of the archive you would like to create (**google.tar**). This command is generally used on directory containing multiple files, rather than on a single file as in the previous example.

To extract this tar file use the **tar** command again, this time with the **-x** (extract) option:

```
tar -xvf google.tar
```

To create a tar file that is also compressed (like a zip file), use the **-z** option:

```
tar -cvzf google.tar.zip google.html
```

Note that a directory name can be specified instead of the name of a file. To extract tar.gz or tar.zip files use **tar** with the **-z** option:

```
tar -xvzf google.tar.zip
```

4.8 Wildcard characters

Sign in to your Linux account and locate the **sample_sequences.zip** file you transferred to your home directory (see the **File transfer exercise** section). Extract the file and then use **ls -l** to examine the files that are produced:

```
unzip sample_sequences.zip
ls -l
```

As you can see, several ".fasta" files were extracted from the **sample_sequences.zip** archive. Suppose you want to organize your home directory by placing these new sequence files into a single directory. You can do this easily using the * wildcard character. Try the following:

```
cd ~
mkdir sequences
mv *.fasta sequences
```

The * represents any text. The ***.fasta** instructs the **mv** command to move any file that ends with ".fasta" from the current directory to the **sequences** directory. The * can be used will other commands as a simple way to refer to multiple files with similar names.

4.9 The grep command

A useful command for searching the contents of files is **grep**. **grep** can be used to look for specific text in one or more files. In this example you will use **grep** to examine whether the fasta files you downloaded contain a properly formatted title line. The title line should start with the **>** character, and there should not be any additional **>** characters in the file. Try the following command:

```
grep -r ">" sequences
```

The **-r** option stands for “recursive” and tells **grep** to examine all the files inside the specified directory. The ‘>’ is the text you want to search for, and **sequences** is the directory you want to search. For each match encountered, **grep** returns the name of the file and the contents of the line containing the match. As you will see when you run the above command, each file contains a single title line as expected.

4.10 The root user

You may have noticed that when you are signed in to your user account you are unable to access many of the files and directories on the Linux system. One way to gain access to these files is to sign in as user **root**. However, you are unlikely to be given the password for the root user, since it is usually reserved for sysadmins, so that they install new programs, create new user accounts, etc. Even sysadmins do not usually sign in as the **root** user, since a small mistake when typing a command can have drastic consequences. Instead, they switch to the **root** user only when they need to perform a specific task that they are unable to perform as a regular user.

4.11 The Linux file system

So far you have worked inside your home directory, which is located in **/home**. You may wonder what the other directories found on the typical Linux system are used for. Here is a short description of what is typically stored in the directories:

/bin contains several useful programs that can be used by the **root** user and standard users. For example, the **ls** program is located in **/bin**.

/boot contains files used during startup.

/dev contains files that represent hardware components of the system. When data is written to these files it is redirected to the corresponding hardware device.

/etc contains system configuration files.

/home the user home directories.

/initrd information used for booting.

/lib software components used by many different programs.

/lost+found files saved during system failures are stored here.

/misc for miscellaneous purposes.

/mnt a directory that can be used to access external file systems, such as CD-ROMs and digital cameras.

/opt usually contains third-party software.

/proc a virtual file system containing information about system resources.

/root the root user's home directory.

/sbin essential programs used by the system and by the root user.

/tmp temporary space that can be used by the system and by users.

/usr programs, libraries, and documentation for all user-related programs.

/var contains log files and files created during processes such as printing and downloading.

To see which of these directories is present on the Linux system you are using, perform the following:

```
cd /
ls
```

4.12 Editing a text file using vi

Sometimes you may want to make changes to a text file while signed in to your Linux account. There are several programs available for this purpose, one of which is called **vi**. **vi** is somewhat difficult to operate, since you have to use keyboard shortcuts for all the commands you typically access using menus in other text editing applications. However, by learning a few key commands you can comfortably edit text files using **vi**. In the example below, you will use **vi** to edit a text file containing multiple DNA sequences.

Many bioinformatics programs, such as **clustalw**, read in multiple sequences from a single file. Each sequence in the file usually needs to be in fasta format, as in the following example:

```
>seq 1
gatattta
>seq2
attatcc
>seq3
etc
```

To combine the p53 sequences in your **sequences** directory into a single file, use the following:

```
cd ~/sequences
cat *p53.fasta > all_p53_seqs.fasta
```

Now examine the **all_p53_seqs.fasta** file using the **more** command. In a fasta file containing multiple sequences, each sequence should have a separate title (titles normally begin with a **>** character). In the current **all_p53_seqs.fasta** file the first sequence record is missing the **>**. To edit this sequence's title, begin by opening the file in **vi**:

```
vi all_p53_seqs.fasta
```

Next, press **i** to enter insert mode. Use the arrows on the keyboard to move the cursor to top left if it isn't already there, and then type ">". Press Esc to leave the insert mode. To save the changes and quit, type **:wq** and press Enter. If you had problems editing the file and wish to quit without saving, press Esc and then type **:q!** and press Enter.

Use **vi** to correct the sequence title in the **bos_taurus_p53.fasta** file too, as we will be using this file in the future.

Note that the goal of this exercise was to introduce you to **vi**. Usually you will not need to edit your sequence files in this manner. However, you may find **vi** useful for making changes to your **.bashrc** file and for creating and modifying Bash scripts (both of these are described below).

5 Bioinformatics tools

5.1 EMBOSS

Now that you have been exposed to several of the built-in Linux commands and the Linux file system, you are ready to use some third party bioinformatics applications. One of these applications is called EMBOSS (The European Molecular Biology Open Software Suite).[5] EMBOSS contains several powerful bioinformatics programs for performing tasks such as sequence alignment, PCR primer design, and protein property prediction [2]. To see whether EMBOSS is installed on the Linux system you are using, try the following:

```
which showalign
```

showalign is one of the programs included with the EMBOSS package. In the above command, **which** is used to look for the **showalign** program on your PATH (the meaning of "PATH" is explained in more detail below). If this command returns something like "/usr/local/bin/showalign", then EMBOSS is likely installed. If instead it returns "no showalign in ..", then talk to your sysadmin.

EMBOSS includes numerous applications. In the following examples you will explore just a few of them. First, switch to your **sequences** directory, which should contain several sequences in fasta format.

```
cd ~/sequences
```

Now, use the EMBOSS **transeq** program to translate the *Bos taurus* p53 nucleotide sequence into a protein sequence (note that the **** below is used to split the command across multiple lines–when typing the command press Enter after the **** or omit the **** and type the entire command on one line):

```
transeq -sequence bos_taurus_p53.fasta -outseq bovine_p53_protein
```

To see the resulting protein sequence use:

```
cat bovine_p53_protein
```

Next, perform a global sequence alignment of two of the p53 sequences using **needle**. Note that when you run this command you will be prompted for some additional information. For this example you can press Enter each time you are prompted for information,

[5] http://emboss.sourceforge.net/

to indicate that you would like to use the default program settings:

```
needle macaca_mulatta_p53.fasta xenopus_laevis_p53.fasta -outfile \
    alignment
```

To examine the alignment that is generated use:

```
more alignment
```

Finally, use the **pepstats** program to obtain protein statistics for the protein sequence you created using **transeq**:

```
pepstats bovine_p53_protein -outfile stats
```

To examine the output use:

```
more stats
```

5.2 Using ClustalW

clustalw[6] is a powerful sequence alignment program that can be used to generate large multiple alignments [3]. To see whether **clustalw** is installed on the Linux system you are using, use the **which** command again:

```
which clustalw
```

This command should return the full path to the **clustalw** program. If it returns “no clustalw in ..”, talk to your sysadmin.

The **clustalw** program offers several command-line options for controlling the sequence alignment process. To see these options, enter **clustalw -options**. In the following example **clustalw** is used to align the sequences in the **all_p53_seqs.fasta** file:

```
cd ~
clustalw -infile=sequences/all_p53_seqs.fasta -outfile=alignment -align
```

To view the completed alignment, use **more**:

```
more alignment
```

[6] http://www.ebi.ac.uk/Tools/clustalw2/index.html

5.3 Performing a BLAST search

BLAST[7] is a powerful program for comparing a sequence of interest to large databases of existing sequences [4]. By identifying related sequences you can gain insight into the function and evolution of the genes and proteins you are interested. The BLAST program can be installed on Windows, Mac, and Linux machines. However, to run BLAST on your own computer you also need to download the sequence databases you wish to search. These databases can be very large, and they become outdated quickly, since new sequences are continually added. For these reasons, many users prefer to submit sequences using the web interfaces provided by NCBI. The main drawback of using the web interface is that you can only submit one sequence at a time. If you have a large collection of sequences you wish to analyze, this approach can be very time consuming.

To avoid these issues you can use the **remote_blast_client.pl** program.[8] To download **remote_blast_client.pl** to your Linux account, use the following command:

```
wget http://www.ualberta.ca/~stothard/downloads/remote_blast_client.\
    zip --user-agent=IE
```

Now unzip the file you downloaded (don't forget about Tab completion–you can type "unzip re" and then press Tab to get the full file name):

```
unzip remote_blast_client.zip
```

Change the permissions on the **remote_blast_client.pl** file so that you can execute it:

```
chmod u+x remote_blast_client/remote_blast_client.pl
```

Now use the **remote_blast_client.pl** program to perform a BLAST search for each of the sequences in the **all_p53_seqs.fasta** file you created in your **sequences** directory:

```
cd ~
./remote_blast_client/remote_blast_client.pl -i \
    sequences/all_p53_seqs.fasta -o blast_results.txt -b blastn -d nr
```

The **-i** option in the previous command is used to specify which file contains the sequences you wish to submit and the **-o** is used to specify where you want the results saved. The **-b** and **-d** options are used to specify which BLAST program and database you want to use. The BLAST search may take a few minutes to complete. As the script

[7] `http://blast.ncbi.nlm.nih.gov/Blast.cgi`

[8] Note that NCBI provides a similar tool, called **netblast**, which is available at `ftp://ftp.ncbi.nih.gov/blast/executables/LATEST/`

runs it will give you information about what it is doing. If you wish to cancel the search, use Ctrl-C. Note that Ctrl-C can be used to return to the command prompt for many other programs too.

Once the program has stopped running you can examine the results using **more**. Note that this script returns results in a compact tabular format that does not include alignments.

To perform a BLAST search without relying on NCBI's servers you can use the **blastall** program. First you need to format a sequence database using the **formatdb** program. The following command formats the genomic sequence of E. coli (which was included in the **sample_sequences.zip** file) so that it can be used as a BLAST database:

```
cd ~
formatdb -i sequences/e_coli.fasta -p F
```

To see what the **-i** and **-p** options are used to indicate, try the following:

```
formatdb --help
```

You are now ready to search the E. coli database using any fasta or multi-fasta sequence as the query. The following command compares the two 16S rRNA sequences in **16S_rRNA.fasta** to the E. coli genome:

```
cd ~
blastall -i sequences/16S_rRNA.fasta -d sequences/e_coli.fasta -p \
    blastn -o local_blast_results.txt
```

To examine the results, use **more**:

```
more local_blast_results.txt
```

5.4 Performing a BLAT search

BLAT[9] is a powerful tool for searching for sequences of interest in a completed genome or proteome. It is faster than BLAST, and is much better at aligning cDNA sequences to genomic sequence, because it looks for splice site consensus sequences [5]. BLAT is less sensitive than BLAST, and is thus most useful for comparisons involving sequences from the same species (e.g. a human cDNA vs. the human genome) or closely related species (e.g. a human cDNA vs. the chimp genome).

[9] http://www.kentinformatics.com/

The following command uses the **blat** program to compare a bovine insulin cDNA to bovine chromosome 29:

```
cd ~
blat sequences/bos_taurus_chromosome_29.fasta \
    sequences/bos_taurus_insulin_cDNA.fasta blat_chr_29_output.txt
```

Note that **blat** interprets the first file to be the database, the second to be the query, and the third to be the output. The output returned by **blat** contains the coordinates of similar regions but not a sequence alignment. To generate a sequence alignment from the coordinates, use the **pslPretty** program, which is included with BLAT:

```
cd ~
pslPretty blat_chr_29_output.txt \
    sequences/bos_taurus_chromosome_29.fasta \
    sequences/bos_taurus_insulin_cDNA.fasta blat_chr_29_alignment.txt
```

To view the alignment, use **more**:

```
more blat_chr_29_alignment.txt
```

The **blat** program is usually used to compare sequences to a full genome rather than a single chromosome. The following commands download a complete bovine genome sequence:

```
cd ~
mkdir bovine_genome
cd bovine_genome
wget -c -A "Chr*" -R "ChrY*" ftp://ftp.cbcb.umd.edu/pub/data/assembly/\
    Bos_taurus/Bos_taurus_UMD_3.1/*
gunzip *.fa.gz
```

If you plan on performing several **blat** searches against a genome, you may want to convert the chromosome sequence text files to “2bit” files. The 2bit format is more compact and can lead to faster searches (the **faToTwoBit** program used below is included with **blat**):

```
faToTwoBit Chr1.fa Chr1.2bit
```

To convert all the chromosome text files to 2bit files you can use **find** followed by **xargs**. In the command below, **find** is used to obtain a list of the chromosome sequence files. The file list is passed to **xargs**, which builds a **faToTwoBit** command for each file, replacing all instances of “{}” with the name of file:

```
cd ~
find bovine_genome -name "*.fa" | xargs -I{} faToTwoBit {} {}.2bit
```

Another way to process multiple files is to use the **-exec** option of **find**. The command placed after **-exec** is run for each file found by **find**. All instances of “{}” become the name of the file when the command is executed:

```
cd ~
find bovine_genome -name "*.fa" -exec faToTwoBit {} {}.2bit \;
```

The **-exec** approach is generally preferred because it works when filenames contain spaces. The resulting 2bit files (one per chromosome) can now be used as the target sequences in **blat** searches. The following uses **find**, **xargs**, and **blat** to compare each bovine chromosome to a single query sequence (**bos_taurus_insulin_cDNA.fasta**):

```
cd ~
find bovine_genome -name "*.2bit" | xargs -I{} blat {} \
    sequences/bos_taurus_insulin_cDNA.fasta {}.insulin.out
```

To quickly examine all the output files produced by **blat** you can use the following:

```
cat bovine_genome/*insulin.out | more
```

To perform searches for multiple queries, first create a text file containing a list of the query sequence files, one filename per line. Pass this list file to **blat** in place of the query. This approach is faster than manually running a separate search for each query, in part because each chromosome sequence needs to be loaded into memory just once. The following creates a file called **query_list.txt** containing the names of all the files in the **sequences** directory that have “bos_taurus” in their title and are less than 1 MB in size. The list file is then used as the query in **blat** searches against each bovine chromosome:

```
cd ~
find ./sequences -name "*bos_taurus*" -size -1000k > query_list.txt
find bovine_genome -name "*.2bit" | xargs -I{} blat {} query_list.txt \
    {}.list.out
```

To convert the **blat** results to alignments, first create a file containing a list of the chromosome sequence files that were searched. This “targets” file can then be passed to **pslPretty** along with the **query_list.txt** file you created earlier. These filename lists are used by **pslPretty** when it obtains the sequences located between the match coordinates given in the “list.out” files:

```
cd ~
find bovine_genome -name "*.fa" > target_list.txt
find bovine_genome -name "*list.out" | xargs -I{} pslPretty {} \
   target_list.txt query_list.txt {}.pretty
```

The resulting alignments can be viewed using the following:

```
cat bovine_genome/*pretty | more
```

6 Streamlining data analysis

6.1 The .bashrc file

When you sign in to your Linux account, a file in your home directory called **.bashrc** is run by the system. This file contains commands that are used to control the behavior of the **bash** program (**bash** is the program that passes the commands you type to the actual programs that do the work). You may not have noticed this file in your home directory, because by default the **ls** command does not show files that start with a “.” character. To see all the files in your home directory, use **ls** with the **-a** option.

```
cd ~
ls -a
```

In the following exercise you will make a few minor changes to your **.bashrc** file using using **vi**. Start by copying your **.bashrc** file so that you can go back to the existing version if the changes you make create problems:

```
cp .bashrc bashrc_backup
```

Now open your **.bashrc** file in **vi**:

```
vi .bashrc
```

Remember to press **i** to enter insert mode. Add the following text below the existing contents:

```
alias rm="rm -i"
alias la="ls -al"
```

Now press Esc to leave insert mode, and then type **:wq** to save your changes and exit **vi**. The first line you added to your **.bashrc** file will tell **bash** (the program that handles the commands you type) to always pass the **-i** option to the **rm** program when you enter the **rm** command. The **-i** option tells **rm** that you want to be warned before any files are actually deleted, and that you want to have the option of canceling the delete process. The second line tells **bash** that you want to use the command **la** to call the **ls** program with the **-a** and **-l** options (show all files and use the long listing format). Defining the **la** command in your **.bashrc** simply saves you the trouble of remembering and typing the full command for listing all files.

Try the new **la** command:

```
la
```

Notice that the **bash** program is saying that it doesn't know what is meant by **la**, even though you defined it in the **.bashrc** file. Remember that the **.bashrc** file is only read when you log in. To tell **bash** to read your **.bashrc** file again, use the **source** command:

```
source .bashrc
```

The **la** command should now be recognized by **bash**.

6.2 Modifying $PATH and other environment variables

Try running the **remote_blast_client.pl** program from your home directory by typing the name of the program:

```
remote_blast_client.pl
```

The **bash** program, which interprets the commands you enter, doesn't know anything about the **remote_blast_client.pl** program. This is the reason you had to enter the exact location of the **remote_blast_client.pl** script when you ran it in the previous example (**./remote_blast_client/remote_blast_client.pl**). Remember that the **./** means the current directory.

Whenever you type a command, **bash** searches for a program with the same name as the command you enter, and for alias commands you specified in your **.bashrc** file. The **bash** program does not however, search the entire file system for a matching program, as this would be very time consuming. Instead, it searches a specified set of directories. The names of these directories are stored in an environment variable called **$PATH**. To see what is currently stored in your path, use the following:

```
echo $PATH
```

Notice that the directory containing **remote_blast_client.pl** is not stored in the **$PATH** variable. You can temporarily add it using the following

```
export PATH=$PATH:~/remote_blast_client
```

To see that it was added, enter **echo $PATH** again. Note that this change to **$PATH** only lasts while you are signed in. To make the change permanent, you could add the above **export** command to the end of your **.bashrc** file using **vi**.

Now that the **$PATH** variable contains information about where to find **remote_blast_client.pl**, try entering the following in your home directory:

```
remote_blast_client.pl
```

The **remote_blast_client.pl** program should start. Press Ctrl-C to return to the command line.

Although the benefits of editing the **$PATH** variable are minor in this case (it isn't difficult to enter the full path to the **remote_blast_client.pl**), understanding environment variables and how to modify them is very important. Indeed, many programs require that you add new environment variables to your **.bashrc** file.

6.3 Writing a simple Bash script

Sometimes you may find it hard to remember what command-line options a program like **remote_blast_client.pl** or **clustalw** requires. Furthermore, you may always use the same options, making all the typing seem quite repetitive. Suppose you want to be able to sign in to your user account and quickly perform an alignment of whatever sequences you have stored in a file called **dna.fasta**. You can do this by writing a simple Bash script that contains the command you want to use.

First create the file that will contain your script:

```
touch align_dna.sh
```

Now edit the file in **vi**:

```
vi align_dna.sh
```

Using **vi**, add the following text (remember to press i to enter insert mode):

```
#!/bin/bash
clustalw -infile=dna.fasta -outfile=dna.alignment -align -type=dna
```

Save your changes and exit **vi**.

Use **chmod** to make your script executable:

```
chmod u+x align_dna.sh
```

To test your script, first create a file called **dna.fasta** in your home directory by copying the fasta file you created previously in your **sequences** directory:

```
cd ~
cp sequences/all_p53_seqs.fasta ./dna.fasta
```

Now execute your script:

```
./align_dna.sh
```

You should see output from **clustalw** appear, and a file called **dna.alignment** should be created. Bash scripts are useful because they help to automate analysis steps, since you do not need to enter a lot of text, and you can be sure the same parameters are used each time. It is possible to build complex scripts consisting of many commands.

7 Summary

This tutorial has provided a brief introduction to the Linux operating system, and in particular the use of the command line. Although the transition to the command line can seem difficult at first, it is well worth the effort if you plan on working with large data sets, such as those arising from high-throughput sequencing projects. If you would like to become even more proficient at analyzing data you should learn a programming language. Linux is well-suited to such an endeavor, because of the wealth of programming tools available. Once you can program you can perform almost any analysis you can imagine.

References

[1] http://ss64.com/bash/

[2] Rice P, Longden I, Bleasby A (2000) EMBOSS: The European Molecular Biology Open Software Suite (2000) Trends Genet 16:276-277.

[3] Chenna R, Sugawara H, Koike T, Lopez R, Gibson TJ, Higgins DG, Thompson JD (2003) Multiple sequence alignment with the Clustal series of programs. Nucleic Acids Res 31:3497-500.

[4] Altschul SF, Madden TL, Schäffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res 25:3389-3402.

[5] Kent WJ (2002) BLAT–the BLAST-like alignment tool. Genome Res 12:656-664.

R for Beginners

Emmanuel Paradis

Institut des Sciences de l'Évolution
Université Montpellier II
F-34095 Montpellier cédex 05
France

E-mail: *paradis@isem.univ-montp2.fr*

I thank Julien Claude, Christophe Declercq, Élodie Gazave, Friedrich Leisch, Louis Luangkesron, François Pinard, and Mathieu Ros for their comments and suggestions on earlier versions of this document. I am also grateful to all the members of the R Development Core Team for their considerable efforts in developing R and animating the discussion list 'rhelp'. Thanks also to the R users whose questions or comments helped me to write "R for Beginners". Special thanks to Jorge Ahumada for the Spanish translation.

Contents

1 Preamble

The goal of the present document is to give a starting point for people newly interested in R. I chose to emphasize on the understanding of how R works, with the aim of a beginner, rather than expert, use. Given that the possibilities offered by R are vast, it is useful to a beginner to get some notions and concepts in order to progress easily. I tried to simplify the explanations as much as I could to make them understandable by all, while giving useful details, sometimes with tables.

R is a system for statistical analyses and graphics created by Ross Ihaka and Robert Gentleman[1]. R is both a software and a language considered as a dialect of the S language created by the AT&T Bell Laboratories. S is available as the software S-PLUS commercialized by Insightful[2]. There are important differences in the designs of R and of S: those who want to know more on this point can read the paper by Ihaka & Gentleman (1996) or the R-FAQ[3], a copy of which is also distributed with R.

R is freely distributed under the terms of the *GNU General Public Licence*[4]; its development and distribution are carried out by several statisticians known as the *R Development Core Team*.

R is available in several forms: the sources (written mainly in C and some routines in Fortran), essentially for Unix and Linux machines, or some pre-compiled binaries for Windows, Linux, and Macintosh. The files needed to install R, either from the sources or from the pre-compiled binaries, are distributed from the internet site of the *Comprehensive R Archive Network* (CRAN)[5] where the instructions for the installation are also available. Regarding the distributions of Linux (Debian, . . .), the binaries are generally available for the most recent versions; look at the CRAN site if necessary.

R has many functions for statistical analyses and graphics; the latter are visualized immediately in their own window and can be saved in various formats (jpg, png, bmp, ps, pdf, emf, pictex, xfig; the available formats may depend on the operating system). The results from a statistical analysis are displayed on the screen, some intermediate results (*P*-values, regression coefficients, residuals, . . .) can be saved, written in a file, or used in subsequent analyses.

The R language allows the user, for instance, to program loops to successively analyse several data sets. It is also possible to combine in a single program different statistical functions to perform more complex analyses. The

[1] Ihaka R. & Gentleman R. 1996. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5: 299–314.

[2] See http://www.insightful.com/products/splus/default.asp for more information

[3] http://cran.r-project.org/doc/FAQ/R-FAQ.html

[4] For more information: http://www.gnu.org/

[5] http://cran.r-project.org/

R users may benefit from a large number of programs written for S and available on the internet[6], most of these programs can be used directly with R.

At first, R could seem too complex for a non-specialist. This may not be true actually. In fact, a prominent feature of R is its flexibility. Whereas a classical software displays immediately the results of an analysis, R stores these results in an "object", so that an analysis can be done with no result displayed. The user may be surprised by this, but such a feature is very useful. Indeed, the user can extract only the part of the results which is of interest. For example, if one runs a series of 20 regressions and wants to compare the different regression coefficients, R can display only the estimated coefficients: thus the results may take a single line, whereas a classical software could well open 20 results windows. We will see other examples illustrating the flexibility of a system such as R compared to traditional softwares.

[6] For example: http://stat.cmu.edu/S/

2 A few concepts before starting

Once R is installed on your computer, the software is executed by launching the corresponding executable. The prompt, by default '`>`', indicates that R is waiting for your commands. Under Windows using the program Rgui.exe, some commands (accessing the on-line help, opening files, ...) can be executed via the pull-down menus. At this stage, a new user is likely to wonder "What do I do now?" It is indeed very useful to have a few ideas on how R works when it is used for the first time, and this is what we will see now.

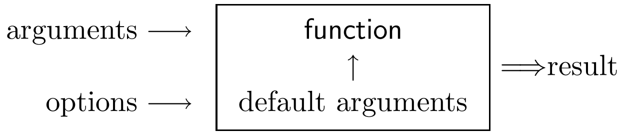
We shall see first briefly how R works. Then, I will describe the "assign" operator which allows creating objects, how to manage objects in memory, and finally how to use the on-line help which is very useful when running R.

2.1 How R works

The fact that R is a language may deter some users who think "I can't program". This should not be the case for two reasons. First, R is an interpreted language, not a compiled one, meaning that all commands typed on the keyboard are directly executed without requiring to build a complete program like in most computer languages (C, Fortran, Pascal, ...).

Second, R's syntax is very simple and intuitive. For instance, a linear regression can be done with the command `lm(y ~ x)` which means "fitting a linear model with y as response and x as predictor". In R, in order to be executed, a function *always* needs to be written with parentheses, even if there is nothing within them (e.g., `ls()`). If one just types the name of a function without parentheses, R will display the content of the function. In this document, the names of the functions are generally written with parentheses in order to distinguish them from other objects, unless the text indicates clearly so.

When R is running, variables, data, functions, results, etc, are stored in the active memory of the computer in the form of *objects* which have a *name*. The user can do actions on these objects with *operators* (arithmetic, logical, comparison, ...) and *functions* (which are themselves objects). The use of operators is relatively intuitive, we will see the details later (p. 25). An R function may be sketched as follows:



The arguments can be objects ("data", formulae, expressions, ...), some of which could be defined by default in the function; these default values may be modified by the user by specifying options. An R function may require no argument: either all arguments are defined by default (and their values can be modified with the options), or no argument has been defined in the function. We will see later in more details how to use and build functions (p. 67). The present description is sufficient for the moment to understand how R works.

All the actions of R are done on objects stored in the active memory of the computer: no temporary files are used (Fig. 1). The readings and writings of files are used for input and output of data and results (graphics, . . .). The user executes the functions via some commands. The results are displayed directly on the screen, stored in an object, or written on the disk (particularly for graphics). Since the results are themselves objects, they can be considered as data and analysed as such. Data files can be read from the local disk or from a remote server through internet.

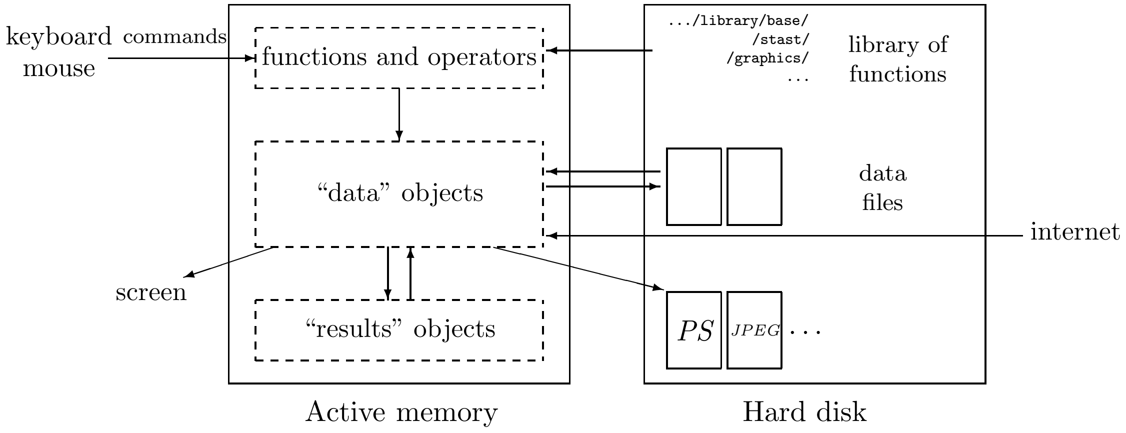


Figure 1: A schematic view of how R works.

The functions available to the user are stored in a library localised on the disk in a directory called R_HOME/library (R_HOME is the directory where R is installed). This directory contains *packages* of functions, which are themselves structured in directories. The package named base is in a way the core of R and contains the basic functions of the language, particularly, for reading and manipulating data. Each package has a directory called R with a file named like the package (for instance, for the package base, this is the file R_HOME/library/base/R/base). This file contains all the functions of the package.

One of the simplest commands is to type the name of an object to display its content. For instance, if an object n contents the value 10:

```
> n
[1] 10
```

The digit 1 within brackets indicates that the display starts at the first element of `n`. This command is an implicit use of the function `print` and the above example is similar to `print(n)` (in some situations, the function `print` must be used explicitly, such as within a function or a loop).

The name of an object must start with a letter (A–Z and a–z) and can include letters, digits (0–9), dots (.), and underscores (_). R discriminates between uppercase letters and lowercase ones in the names of the objects, so that `x` and `X` can name two distinct objects (even under Windows).

2.2 Creating, listing and deleting the objects in memory

An object can be created with the "assign" operator which is written as an arrow with a minus sign and a bracket; this symbol can be oriented left-to-right or the reverse:

```
> n <- 15
> n
[1] 15
> 5 -> n
> n
[1] 5
> x <- 1
> X <- 10
> x
[1] 1
> X
[1] 10
```

If the object already exists, its previous value is erased (the modification affects only the objects in the active memory, not the data on the disk). The value assigned this way may be the result of an operation and/or a function:

```
> n <- 10 + 2
> n
[1] 12
> n <- 3 + rnorm(1)
> n
[1] 2.208807
```

The function `rnorm(1)` generates a normal random variate with mean zero and variance unity (p. 17). Note that you can simply type an expression without assigning its value to an object, the result is thus displayed on the screen but is not stored in memory:

```
> (10 + 2) * 5
[1] 60
```

The assignment will be omitted in the examples if not necessary for understanding.

The function `ls` lists simply the objects in memory: only the names of the objects are displayed.

```
> name <- "Carmen"; n1 <- 10; n2 <- 100; m <- 0.5
> ls()
[1] "m"    "n1"   "n2"   "name"
```

Note the use of the semi-colon to separate distinct commands on the same line. If we want to list only the objects which contain a given character in their name, the option `pattern` (which can be abbreviated with `pat`) can be used:

```
> ls(pat = "m")
[1] "m"    "name"
```

To restrict the list of objects whose names start with this character:

```
> ls(pat = "^m")
[1] "m"
```

The function `ls.str` displays some details on the objects in memory:

```
> ls.str()
m :  num 0.5
n1 :  num 10
n2 :  num 100
name :  chr "Carmen"
```

The option `pattern` can be used in the same way as with `ls`. Another useful option of `ls.str` is `max.level` which specifies the level of detail for the display of composite objects. By default, `ls.str` displays the details of all objects in memory, included the columns of data frames, matrices and lists, which can result in a very long display. We can avoid to display all these details with the option `max.level = -1`:

```
> M <- data.frame(n1, n2, m)
> ls.str(pat = "M")
M : ‘data.frame’:       1 obs. of  3 variables:
 $ n1: num 10
 $ n2: num 100
 $ m : num 0.5
> ls.str(pat="M", max.level=-1)
M : ‘data.frame’:       1 obs. of  3 variables:
```

To delete objects in memory, we use the function `rm`: `rm(x)` deletes the object `x`, `rm(x,y)` deletes both the objects `x` et `y`, `rm(list=ls())` deletes all the objects in memory; the same options mentioned for the function `ls()` can then be used to delete selectively some objects: `rm(list=ls(pat="^m"))`.

2.3 The on-line help

The on-line help of R gives very useful information on how to use the functions. Help is available directly for a given function, for instance:

```
> ?lm
```

will display, within R, the help page for the function `lm()` (*linear model*). The commands `help(lm)` and `help("lm")` have the same effect. The last one must be used to access help with non-conventional characters:

```
> ?*
Error: syntax error
> help("*")
Arithmetic              package:base              R Documentation

Arithmetic Operators
...
```

Calling help opens a page (this depends on the operating system) with general information on the first line such as the name of the package where is (are) the documented function(s) or operators. Then comes a title followed by sections which give detailed information.

Description: brief description.

Usage: for a function, gives the name with all its arguments and the possible options (with the corresponding default values); for an operator gives the typical use.

Arguments: for a function, details each of its arguments.

Details: detailed description.

Value: if applicable, the type of object returned by the function or the operator.

See Also: other help pages close or similar to the present one.

Examples: some examples which can generally be executed without opening the help with the function `example`.

For beginners, it is good to look at the section **Examples**. Generally, it is useful to read carefully the section **Arguments**. Other sections may be encountered, such as **Note**, **References** or **Author(s)**.

By default, the function `help` only searches in the packages which are loaded in memory. The option `try.all.packages`, which default is `FALSE`, allows to search in all packages if its value is `TRUE`:

```
> help("bs")
No documentation for 'bs' in specified packages and libraries:
you could try 'help.search("bs")'
> help("bs", try.all.packages = TRUE)
Help for topic 'bs' is not in any loaded package but
can be found in the following packages:

  Package              Library
  splines              /usr/lib/R/library
```

Note that in this case the help page of the function `bs` is not displayed. The user can display help pages from a package not loaded in memory using the option `package`:

```
> help("bs", package = "splines")
bs                package:splines                R Documentation

B-Spline Basis for Polynomial Splines

Description:

     Generate the B-spline basis matrix for a polynomial spline.
...
```

The help in html format (read, e.g., with Netscape) is called by typing:

```
> help.start()
```

A search with keywords is possible with this html help. The section **See Also** has here hypertext links to other function help pages. The search with keywords is also possible in R with the function `help.search`. The latter looks for a specified topic, given as a character string, in the help pages of all installed packages. For instance, `help.search("tree")` will display a list of the functions which help pages mention "tree". Note that if some packages have been recently installed, it may be useful to refresh the database used by `help.search` using the option `rebuild` (e.g., `help.search("tree", rebuild = TRUE)`).

The fonction `apropos` finds all functions which name contains the character string given as argument; only the packages loaded in memory are searched:

```
> apropos(help)
[1] "help"         ".helpForCall" "help.search"
[4] "help.start"
```

3 Data with R

3.1 Objects

We have seen that R works with objects which are, of course, characterized by their names and their content, but also by *attributes* which specify the kind of data represented by an object. In order to understand the usefulness of these attributes, consider a variable that takes the value 1, 2, or 3: such a variable could be an integer variable (for instance, the number of eggs in a nest), or the coding of a categorical variable (for instance, sex in some populations of crustaceans: male, female, or hermaphrodite).

It is clear that the statistical analysis of this variable will not be the same in both cases: with R, the attributes of the object give the necessary information. More technically, and more generally, the action of a function on an object depends on the attributes of the latter.

All objects have two *intrinsic* attributes: *mode* and *length*. The mode is the basic type of the elements of the object; there are four main modes: numeric, character, complex[7], and logical (`FALSE` or `TRUE`). Other modes exist but they do not represent data, for instance function or expression. The length is the number of elements of the object. To display the mode and the length of an object, one can use the functions `mode` and `length`, respectively:

```
> x <- 1
> mode(x)
[1] "numeric"
> length(x)
[1] 1
> A <- "Gomphotherium"; compar <- TRUE; z <- 1i
> mode(A); mode(compar); mode(z)
[1] "character"
[1] "logical"
[1] "complex"
```

Whatever the mode, missing data are represented by `NA` (*not available*). A very large numeric value can be specified with an exponential notation:

```
> N <- 2.1e23
> N
[1] 2.1e+23
```

R correctly represents non-finite numeric values, such as $\pm\infty$ with `Inf` and `-Inf`, or values which are not numbers with `NaN` (*not a number*).

[7] The mode complex will not be discussed in this document.

```
> x <- 5/0
> x
[1] Inf
> exp(x)
[1] Inf
> exp(-x)
[1] 0
> x - x
[1] NaN
```

A value of mode character is input with double quotes ". It is possible to include this latter character in the value if it follows a backslash \. The two charaters altogether \" will be treated in a specific way by some functions such as `cat` for display on screen, or `write.table` to write on the disk (p. 14, the option `qmethod` of this function).

```
> x <- "Double quotes \" delimitate R's strings."
> x
[1] "Double quotes \" delimitate R's strings."
> cat(x)
Double quotes " delimitate R's strings.
```

Alternatively, variables of mode character can be delimited with single quotes ('); in this case it is not necessary to escape double quotes with backslashes (but single quotes must be!):

```
> x <- 'Double quotes " delimitate R\'s strings.'
> x
[1] "Double quotes \" delimitate R's strings."
```

The following table gives an overview of the type of objects representing data.

object	**modes**	**several modes possible in the same object?**
vector	numeric, character, complex *or* logical	No
factor	numeric *or* character	No
array	numeric, character, complex *or* logical	No
matrix	numeric, character, complex *or* logical	No
data frame	numeric, character, complex *or* logical	Yes
ts	numeric, character, complex *or* logical	No
list	numeric, character, complex, logical, function, expression, . . .	Yes

A vector is a variable in the commonly admitted meaning. A factor is a categorical variable. An array is a table with k dimensions, a matrix being a particular case of array with $k = 2$. Note that the elements of an array or of a matrix are all of the same mode. A data frame is a table composed with one or several vectors and/or factors all of the same length but possibly of different modes. A 'ts' is a time series data set and so contains additional attributes such as frequency and dates. Finally, a list can contain any type of object, included lists!

For a vector, its mode and length are sufficient to describe the data. For other objects, other information is necessary and it is given by *non-intrinsic* attributes. Among these attributes, we can cite *dim* which corresponds to the dimensions of an object. For example, a matrix with 2 lines and 2 columns has for `dim` the pair of values [2, 2], but its length is 4.

3.2 Reading data in a file

For reading and writing in files, R uses the working directory. To find this directory, the command `getwd()` (*get working directory*) can be used, and the working directory can be changed with `setwd("C:/data")` or `setwd("/home/paradis/R")`. It is necessary to give the path to a file if it is not in the working directory.[8]

R can read data stored in text (ASCII) files with the following functions: `read.table` (which has several variants, see below), `scan` and `read.fwf`. R can also read files in other formats (Excel, SAS, SPSS, ...), and access SQL-type databases, but the functions needed for this are not in the package base. These functionalities are very useful for a more advanced use of R, but we will restrict here to reading files in ASCII format.

The function `read.table` has for effect to create a data frame, and so is the main way to read data in tabular form. For instance, if one has a file named data.dat, the command:

```
> mydata <- read.table("data.dat")
```

will create a data frame named `mydata`, and each variable will be named, by default, `V1`, `V2`, ... and can be accessed individually by `mydata$V1`, `mydata$V2`, ..., or by `mydata["V1"]`, `mydata["V2"]`, ..., or, still another solution, by `mydata[, 1]`, `mydata[,2 ]`, ...[9] There are several options whose default values (i.e. those used by R if they are omitted by the user) are detailed in the following table:

```
read.table(file, header = FALSE, sep = "", quote = "\"’", dec = ".",
```

[8] Under Windows, it is useful to create a short-cut of Rgui.exe then edit its properties and change the directory in the field "Start in:" under the tab "Short-cut": this directory will then be the working directory if R is started from this short-cut.

[9] There is a difference: `mydata$V1` and `mydata[, 1]` are vectors whereas `mydata["V1"]` is a data frame. We will see later (p. 18) some details on manipulating objects.

```
           row.names, col.names, as.is = FALSE, na.strings = "NA",
           colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#")
```

`file`	the name of the file (within `""` or a variable of mode character), possibly with its path (the symbol \ is not allowed and must be replaced by /, even under Windows), or a remote access to a file of type URL (http://...)
`header`	a logical (`FALSE` or `TRUE`) indicating if the file contains the names of the variables on its first line
`sep`	the field separator used in the file, for instance `sep="\t"` if it is a tabulation
`quote`	the characters used to cite the variables of mode character
`dec`	the character used for the decimal point
`row.names`	a vector with the names of the lines which can be either a vector of mode character, or the number (or the name) of a variable of the file (by default: `1`, `2`, `3`, . . .)
`col.names`	a vector with the names of the variables (by default: `V1`, `V2`, `V3`, . . .)
`as.is`	controls the conversion of character variables as factors (if `FALSE`) or keeps them as characters (`TRUE`); `as.is` can be a logical, numeric or character vector specifying the variables to be kept as character
`na.strings`	the value given to missing data (converted as `NA`)
`colClasses`	a vector of mode character giving the classes to attribute to the columns
`nrows`	the maximum number of lines to read (negative values are ignored)
`skip`	the number of lines to be skipped before reading the data
`check.names`	if `TRUE`, checks that the variable names are valid for R
`fill`	if `TRUE` and all lines do not have the same number of variables, "blanks" are added
`strip.white`	(conditional to `sep`) if `TRUE`, deletes extra spaces before and after the character variables
`blank.lines.skip`	if `TRUE`, ignores "blank" lines
`comment.char`	a character defining comments in the data file, the rest of the line after this character is ignored (to disable this argument, use `comment.char = ""`)

The variants of `read.table` are useful since they have different default values:

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, ...)
read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
          fill = TRUE, ...)
read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
           fill = TRUE, ...)
read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
            fill = TRUE, ...)
```

The function scan is more flexible than read.table. A difference is that it is possible to specify the mode of the variables, for example:

```
> mydata <- scan("data.dat", what = list("", 0, 0))
```

reads in the file data.dat three variables, the first is of mode character and the next two are of mode numeric. Another important distinction is that scan() can be used to create different objects, vectors, matrices, data frames, lists, ... In the above example, mydata is a list of three vectors. By default, that is if what is omitted, scan() creates a numeric vector. If the data read do not correspond to the mode(s) expected (either by default, or specified by what), an error message is returned. The options are the followings.

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
     quote = if (sep=="\n") "" else "'\"", dec = ".",
     skip = 0, nlines = 0, na.strings = "NA",
     flush = FALSE, fill = FALSE, strip.white = FALSE, quiet = FALSE,
     blank.lines.skip = TRUE, multi.line = TRUE, comment.char = "",
     allowEscapes = TRUE)
```

file	the name of the file (within ""), possibly with its path (the symbol \ is not allowed and must be replaced by /, even under Windows), or a remote access to a file of type URL (http://...); if file="", the data are entered with the keyboard (the entree is terminated by a blank line)
what	specifies the mode(s) of the data (numeric by default)
nmax	the number of data to read, or, if what is a list, the number of lines to read (by default, scan reads the data up to the end of file)
n	the number of data to read (by default, no limit)
sep	the field separator used in the file
quote	the characters used to cite the variables of mode character
dec	the character used for the decimal point
skip	the number of lines to be skipped before reading the data
nlines	the number of lines to read
na.string	the value given to missing data (converted as NA)
flush	a logical, if TRUE, scan goes to the next line once the number of columns has been reached (allows the user to add comments in the data file)
fill	if TRUE and all lines do not have the same number of variables, "blanks" are added
strip.white	(conditional to sep) if TRUE, deletes extra spaces before and after the character variables
quiet	a logical, if FALSE, scan displays a line showing which fields have been read
blank.lines.skip	if TRUE, ignores blank lines
multi.line	if what is a list, specifies if the variables of the same individual are on a single line in the file (FALSE)
comment.char	a character defining comments in the data file, the rest of the line after this character is ignored (the default is to have this disabled)
allowEscapes	specifies whether C-style escapes (e.g., '\t') be processed (the default) or read as verbatim

The function `read.fwf` can be used to read in a file some data in *fixed width format*:

```
read.fwf(file, widths, header = FALSE, sep = "\t",
         as.is = FALSE, skip = 0, row.names, col.names,
         n = -1, buffersize = 2000, ...)
```

The options are the same than for `read.table()` except `widths` which specifies the width of the fields (`buffersize` is the maximum number of lines read simultaneously). For example, if a file named data.txt has the data indicated on the right, one can read the data with the following command:

```
A1.501.2
A1.551.3
B1.601.4
B1.651.5
C1.701.6
C1.751.7
```

```
> mydata <- read.fwf("data.txt", widths=c(1, 4, 3))
> mydata
  V1   V2  V3
1  A 1.50 1.2
2  A 1.55 1.3
3  B 1.60 1.4
4  B 1.65 1.5
5  C 1.70 1.6
6  C 1.75 1.7
```

3.3 Saving data

The function `write.table` writes in a file an object, typically a data frame but this could well be another kind of object (vector, matrix, ...). The arguments and options are:

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"))
```

`x`	the name of the object to be written
`file`	the name of the file (by default the object is displayed on the screen)
`append`	if `TRUE` adds the data without erasing those possibly existing in the file
`quote`	a logical or a numeric vector: if `TRUE` the variables of mode character and the factors are written within `""`, otherwise the numeric vector indicates the numbers of the variables to write within `""` (in both cases the names of the variables are written within `""` but not if `quote = FALSE`)
`sep`	the field separator used in the file
`eol`	the character to be used at the end of each line (`"\n"` is a carriage-return)
`na`	the character to be used for missing data
`dec`	the character used for the decimal point
`row.names`	a logical indicating whether the names of the lines are written in the file
`col.names`	id. for the names of the columns
`qmethod`	specifies, if `quote=TRUE`, how double quotes `"` included in variables of mode character are treated: if `"escape"` (or `"e"`, the default) each `"` is replaced by `\"`, if `"d"` each `"` is replaced by `""`

To write in a simpler way an object in a file, the command `write(x, file="data.txt")` can be used, where `x` is the name of the object (which can be a vector, a matrix, or an array). There are two options: `nc` (or `ncol`) which defines the number of columns in the file (by default `nc=1` if `x` is of mode character, `nc=5` for the other modes), and `append` (a logical) to add the data without deleting those possibly already in the file (`TRUE`) or deleting them if the file already exists (`FALSE`, the default).

To record a group of objects of any type, we can use the command `save(x, y, z, file= "xyz.RData")`. To ease the transfert of data between different machines, the option `ascii = TRUE` can be used. The data (which are now called a *workspace* in R's jargon) can be loaded later in memory with `load("xyz.RData")`. The function `save.image()` is a short-cut for `save(list =ls(all=TRUE), file=".RData")`.

3.4 Generating data

3.4.1 Regular sequences

A regular sequence of integers, for example from 1 to 30, can be generated with:

```
> x <- 1:30
```

The resulting vector `x` has 30 elements. The operator '`:`' has priority on the arithmetic operators within an expression:

```
> 1:10-1
 [1] 0 1 2 3 4 5 6 7 8 9
> 1:(10-1)
[1] 1 2 3 4 5 6 7 8 9
```

The function `seq` can generate sequences of real numbers as follows:

```
> seq(1, 5, 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

where the first number indicates the beginning of the sequence, the second one the end, and the third one the increment to be used to generate the sequence. One can use also:

```
> seq(length=9, from=1, to=5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

One can also type directly the values using the function `c`:

```
> c(1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

It is also possible, if one wants to enter some data on the keyboard, to use the function `scan` with simply the default options:

```
> z <- scan()
1: 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
10:
Read 9 items
> z
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

The function `rep` creates a vector with all its elements identical:

```
> rep(1, 30)
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

The function `sequence` creates a series of sequences of integers each ending by the numbers given as arguments:

```
> sequence(4:5)
[1] 1 2 3 4 1 2 3 4 5
> sequence(c(10,5))
 [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5
```

The function `gl` (*generate levels*) is very useful because it generates regular series of factors. The usage of this fonction is `gl(k, n)` where `k` is the number of levels (or classes), and `n` is the number of replications in each level. Two options may be used: `length` to specify the number of data produced, and `labels` to specify the names of the levels of the factor. Examples:

```
> gl(3, 5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels:  1 2 3
> gl(3, 5, length=30)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels:  1 2 3
> gl(2, 6, label=c("Male", "Female"))
 [1] Male   Male   Male   Male   Male   Male
 [7] Female Female Female Female Female Female
Levels:  Male Female
> gl(2, 10)
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
Levels:  1 2
> gl(2, 1, length=20)
 [1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
Levels:  1 2
> gl(2, 2, length=20)
 [1] 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2
Levels:  1 2
```

Finally, `expand.grid()` creates a data frame with all combinations of vectors or factors given as arguments:

```
> expand.grid(h=c(60,80), w=c(100, 300), sex=c("Male", "Female"))
   h   w    sex
1 60 100   Male
2 80 100   Male
3 60 300   Male
4 80 300   Male
5 60 100 Female
6 80 100 Female
7 60 300 Female
8 80 300 Female
```

3.4.2 Random sequences

law	**function**
Gaussian (normal)	`rnorm(n, mean=0, sd=1)`
exponential	`rexp(n, rate=1)`
gamma	`rgamma(n, shape, scale=1)`
Poisson	`rpois(n, lambda)`
Weibull	`rweibull(n, shape, scale=1)`
Cauchy	`rcauchy(n, location=0, scale=1)`
beta	`rbeta(n, shape1, shape2)`
'Student' (t)	`rt(n, df)`
Fisher–Snedecor (F)	`rf(n, df1, df2)`
Pearson (χ^2)	`rchisq(n, df)`
binomial	`rbinom(n, size, prob)`
multinomial	`rmultinom(n, size, prob)`
geometric	`rgeom(n, prob)`
hypergeometric	`rhyper(nn, m, n, k)`
logistic	`rlogis(n, location=0, scale=1)`
lognormal	`rlnorm(n, meanlog=0, sdlog=1)`
negative binomial	`rnbinom(n, size, prob)`
uniform	`runif(n, min=0, max=1)`
Wilcoxon's statistics	`rwilcox(nn, m, n)`, `rsignrank(nn, n)`

It is useful in statistics to be able to generate random data, and R can do it for a large number of probability density functions. These functions are of the form `r`*`func`*`(n, p1, p2, ...)`, where *`func`* indicates the probability distribution, `n` the number of data generated, and `p1`, `p2`, . . . are the values of the parameters of the distribution. The above table gives the details for each distribution, and the possible default values (if none default value is indicated, this means that the parameter must be specified by the user).

Most of these functions have counterparts obtained by replacing the letter `r` with `d`, `p` or `q` to get, respectively, the probability density (`d`*`func`*`(x, ...)`), the cumulative probability density (`p`*`func`*`(x, ...)`), and the value of quantile (`q`*`func`*`(p, ...)`, with $0 < \texttt{p} < 1$). The last two series of functions can be used to find critical values or P-values of statistical tests. For instance, the critical values for a two-tailed test following a normal distribution at the 5% threshold are:

```
> qnorm(0.025)
[1] -1.959964
> qnorm(0.975)
[1] 1.959964
```

For the one-tailed version of the same test, either `qnorm(0.05)` or `1 - qnorm(0.95)` will be used depending on the form of the alternative hypothesis.

The P-value of a test, say $\chi^2 = 3.84$ with $df = 1$, is:

```
> 1 - pchisq(3.84, 1)
[1] 0.05004352
```

3.5 Manipulating objects

3.5.1 Creating objects

We have seen previously different ways to create objects using the assign operator; the mode and the type of objects so created are generally determined implicitly. It is possible to create an object and specifying its mode, length, type, etc. This approach is interesting in the perspective of manipulating objects. One can, for instance, create an 'empty' object and then modify its elements successively which is more efficient than putting all its elements together with `c()`. The indexing system could be used here, as we will see later (p. 26).

It can also be very convenient to create objects from others. For example, if one wants to fit a series of models, it is simple to put the formulae in a list, and then to extract the elements successively to insert them in the function `lm`.

At this stage of our learning of R, the interest in learning the following functionalities is not only practical but also didactic. The explicit construction of objects gives a better understanding of their structure, and allows us to go further in some notions previously mentioned.

Vector. The function `vector`, which has two arguments `mode` and `length`, creates a vector which elements have a value depending on the mode specified as argument: 0 if numeric, `FALSE` if logical, or `""` if character. The following functions have exactly the same effect and have for single argument the length of the vector: `numeric()`, `logical()`, and `character()`.

Factor. A factor includes not only the values of the corresponding categorical variable, but also the different possible levels of that variable (even if they are not present in the data). The function factor creates a factor with the following options:

```
factor(x, levels = sort(unique(x), na.last = TRUE),
       labels = levels, exclude = NA, ordered = is.ordered(x))
```

levels specifies the possible levels of the factor (by default the unique values of the vector x), labels defines the names of the levels, exclude the values of x to exclude from the levels, and ordered is a logical argument specifying whether the levels of the factor are ordered. Recall that x is of mode numeric or character. Some examples follow.

```
> factor(1:3)
[1] 1 2 3
Levels:  1 2 3
> factor(1:3, levels=1:5)
[1] 1 2 3
Levels:  1 2 3 4 5
> factor(1:3, labels=c("A", "B", "C"))
[1] A B C
Levels:  A B C
> factor(1:5, exclude=4)
[1] 1  2  3  NA 5
Levels:  1 2 3 5
```

The function levels extracts the possible levels of a factor:

```
> ff <- factor(c(2, 4), levels=2:5)
> ff
[1] 2 4
Levels:  2 3 4 5
> levels(ff)
[1] "2" "3" "4" "5"
```

Matrix. A matrix is actually a vector with an additional attribute (dim) which is itself a numeric vector with length 2, and defines the numbers of rows and columns of the matrix. A matrix can be created with the function matrix:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
       dimnames = NULL)
```

The option byrow indicates whether the values given by data must fill successively the columns (the default) or the rows (if TRUE). The option dimnames allows to give names to the rows and columns.

```
> matrix(data=5, nr=2, nc=2)
     [,1] [,2]
[1,]    5    5
[2,]    5    5
> matrix(1:6, 2, 3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(1:6, 2, 3, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Another way to create a matrix is to give the appropriate values to the dim attribute (which is initially NULL):

```
> x <- 1:15
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> dim(x)
NULL
> dim(x) <- c(5, 3)
> x
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
```

Data frame. We have seen that a data frame is created implicitly by the function read.table; it is also possible to create a data frame with the function data.frame. The vectors so included in the data frame must be of the same length, or if one of the them is shorter, it is “recycled” a whole number of times:

```
> x <- 1:4; n <- 10; M <- c(10, 35); y <- 2:4
> data.frame(x, n)
  x  n
1 1 10
2 2 10
```

```
3 3 10
4 4 10
> data.frame(x, M)
  x  M
1 1 10
2 2 35
3 3 10
4 4 35
> data.frame(x, y)
Error in data.frame(x, y) :
    arguments imply differing number of rows: 4, 3
```

If a factor is included in a data frame, it must be of the same length than the vector(s). It is possible to change the names of the columns with, for instance, `data.frame(A1=x, A2=n)`. One can also give names to the rows with the option `row.names` which must be, of course, a vector of mode character and of length equal to the number of lines of the data frame. Finally, note that data frames have an attribute dim similarly to matrices.

List. A list is created in a way similar to data frames with the function `list`. There is no constraint on the objects that can be included. In contrast to `data.frame()`, the names of the objects are not taken by default; taking the vectors `x` and `y` of the previous example:

```
> L1 <- list(x, y); L2 <- list(A=x, B=y)
> L1
[[1]]
[1] 1 2 3 4

[[2]]
[1] 2 3 4

> L2
$A
[1] 1 2 3 4

$B
[1] 2 3 4

> names(L1)
NULL
> names(L2)
[1] "A" "B"
```

Time-series. The function `ts` creates an object of class `"ts"` from a vector (single time-series) or a matrix (multivariate time-series), and some options which characterize the series. The options, with the default values, are:

```
ts(data = NA, start = 1, end = numeric(0), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"), class, names)
```

`data` a vector or a matrix

`start` the time of the first observation, either a number, or a vector of two integers (see the examples below)

`end` the time of the last observation specified in the same way than `start`

`frequency` the number of observations per time unit

`deltat` the fraction of the sampling period between successive observations (ex. `1/12` for monthly data); only one of `frequency` or `deltat` must be given

`ts.eps` tolerance for the comparison of series. The frequencies are considered equal if their difference is less than `ts.eps`

`class` class to give to the object; the default is `"ts"` for a single series, and `c("mts", "ts")` for a multivariate series

`names` a vector of mode character with the names of the individual series in the case of a multivariate series; by default the names of the columns of `data`, or `Series 1`, `Series 2`, ...

A few examples of time-series created with `ts`:

```
> ts(1:10, start = 1959)
Time Series:
Start = 1959
End = 1968
Frequency = 1
 [1]  1  2  3  4  5  6  7  8  9 10
> ts(1:47, frequency = 12, start = c(1959, 2))
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1959       1   2   3   4   5   6   7   8   9  10  11
1960  12  13  14  15  16  17  18  19  20  21  22  23
1961  24  25  26  27  28  29  30  31  32  33  34  35
1962  36  37  38  39  40  41  42  43  44  45  46  47
> ts(1:10, frequency = 4, start = c(1959, 2))
     Qtr1 Qtr2 Qtr3 Qtr4
1959         1    2    3
1960    4    5    6    7
1961    8    9   10
> ts(matrix(rpois(36, 5), 12, 3), start=c(1961, 1), frequency=12)
         Series 1 Series 2 Series 3
```

```
Jan 1961        8        5        4
Feb 1961        6        6        9
Mar 1961        2        3        3
Apr 1961        8        5        4
May 1961        4        9        3
Jun 1961        4        6       13
Jul 1961        4        2        6
Aug 1961       11        6        4
Sep 1961        6        5        7
Oct 1961        6        5        7
Nov 1961        5        5        7
Dec 1961        8        5        2
```

Expression. The objects of mode expression have a fundamental role in R. An expression is a series of characters which makes sense for R. All valid commands are expressions. When a command is typed directly on the keyboard, it is then *evaluated* by R and executed if it is valid. In many circumstances, it is useful to construct an expression without evaluating it: this is what the function `expression` is made for. It is, of course, possible to evaluate the expression subsequently with `eval()`.

```
> x <- 3; y <- 2.5; z <- 1
> exp1 <- expression(x / (y + exp(z)))
> exp1
expression(x/(y + exp(z)))
> eval(exp1)
[1] 0.5749019
```

Expressions can be used, among other things, to include equations in graphs (p. 42). An expression can be created from a variable of mode character. Some functions take expressions as arguments, for example `D` which returns partial derivatives:

```
> D(exp1, "x")
1/(y + exp(z))
> D(exp1, "y")
-x/(y + exp(z))^2
> D(exp1, "z")
-x * exp(z)/(y + exp(z))^2
```

3.5.2 Converting objects

The reader has surely realized that the differences between some types of objects are small; it is thus logical that it is possible to convert an object from a type to another by changing some of its attributes. Such a conversion will be done with a function of the type `as.`*`something`*. R (version 2.1.0) has, in the packages base and utils, 98 of such functions, so we will not go in the deepest details here.

The result of a conversion depends obviously of the attributes of the converted object. Genrally, conversion follows intuitive rules. For the conversion of modes, the following table summarizes the situation.

Conversion to	Function	Rules
numeric	`as.numeric`	`FALSE` → 0
		`TRUE` → 1
		`"1"`, `"2"`, ... → 1, 2, ...
		`"A"`, ... → `NA`
logical	`as.logical`	0 → `FALSE`
		other numbers → `TRUE`
		`"FALSE"`, `"F"` → `FALSE`
		`"TRUE"`, `"T"` → `TRUE`
		other characters → `NA`
character	`as.character`	1, 2, ... → `"1"`, `"2"`, ...
		`FALSE` → `"FALSE"`
		`TRUE` → `"TRUE"`

There are functions to convert the types of objects (`as.matrix`, `as.ts`, `as.data.frame`, `as.expression`, ...). These functions will affect attributes other than the modes during the conversion. The results are, again, generally intuitive. A situation frequently encountered is the conversion of factors into numeric values. In this case, R does the conversion with the numeric coding of the levels of the factor:

```
> fac <- factor(c(1, 10))
> fac
[1] 1 10
Levels:  1 10
> as.numeric(fac)
[1] 1 2
```

This makes sense when considering a factor of mode character:

```
> fac2 <- factor(c("Male", "Female"))
> fac2
[1] Male   Female
Levels: Female Male
> as.numeric(fac2)
[1] 2 1
```

Note that the result is not `NA` as may have been expected from the table above.

To convert a factor of mode numeric into a numeric vector but keeping the levels as they are originally specified, one must first convert into character, then into numeric.

```
> as.numeric(as.character(fac))
[1] 1 10
```

This procedure is very useful if in a file a numeric variable has also non-numeric values. We have seen that read.table() in such a situation will, by default, read this column as a factor.

3.5.3 Operators

We have seen previously that there are three main types of operators in R[10]. Here is the list.

		Operators			
Arithmetic		Comparison		Logical	
+	addition	<	lesser than	! x	logical NOT
-	subtraction	>	greater than	x & y	logical AND
*	multiplication	<=	lesser than or equal to	x && y	id.
/	division	>=	greater than or equal to	x | y	logical OR
^	power	==	equal	x || y	id.
%%	modulo	!=	different	xor(x, y)	exclusive OR
%/%	integer division				

The arithmetic and comparison operators act on two elements (x + y, a < b). The arithmetic operators act not only on variables of mode numeric or complex, but also on logical variables; in this latter case, the logical values are coerced into numeric. The comparison operators may be applied to any mode: they return one or several logical values.

The logical operators are applied to one (!) or two objects of mode logical, and return one (or several) logical values. The operators "AND" and "OR" exist in two forms: the single one operates on each elements of the objects and returns as many logical values as comparisons done; the double one operates on the first element of the objects.

It is necessary to use the operator "AND" to specify an inequality of the type $0 < x < 1$ which will be coded with: 0 < x & x < 1. The expression 0 < x < 1 is valid, but will not return the expected result: since both operators are the same, they are executed successively from left to right. The comparison 0 < x is first done and returns a logical value which is then compared to 1 (TRUE or FALSE < 1): in this situation, the logical value is implicitly coerced into numeric (1 or 0 < 1).

[10] The following characters are also operators for R: $, @, [, [[, :, ?, <-, <<-, =, ::. A table of operators describing precedence rules can be found with ?Syntax.

```
> x <- 0.5
> 0 < x < 1
[1] FALSE
```

The comparison operators operate on *each* element of the two objects being compared (recycling the values of the shortest one if necessary), and thus returns an object of the same size. To compare 'wholly' two objects, two functions are available: `identical` and `all.equal`.

```
> x <- 1:3; y <- 1:3
> x == y
[1] TRUE TRUE TRUE
> identical(x, y)
[1] TRUE
> all.equal(x, y)
[1] TRUE
```

`identical` compares the internal representation of the data and returns `TRUE` if the objects are strictly identical, and `FALSE` otherwise. `all.equal` compares the "near equality" of two objects, and returns `TRUE` or display a summary of the differences. The latter function takes the approximation of the computing process into account when comparing numeric values. The comparison of numeric values on a computer is sometimes surprising!

```
> 0.9 == (1 - 0.1)
[1] TRUE
> identical(0.9, 1 - 0.1)
[1] TRUE
> all.equal(0.9, 1 - 0.1)
[1] TRUE
> 0.9 == (1.1 - 0.2)
[1] FALSE
> identical(0.9, 1.1 - 0.2)
[1] FALSE
> all.equal(0.9, 1.1 - 0.2)
[1] TRUE
> all.equal(0.9, 1.1 - 0.2, tolerance = 1e-16)
[1] "Mean relative  difference: 1.233581e-16"
```

3.5.4 Accessing the values of an object: the indexing system

The indexing system is an efficient and flexible way to access selectively the elements of an object; it can be either *numeric* or *logical*. To access, for example, the third value of a vector `x`, we just type `x[3]` which can be used either to extract or to change this value:

```
> x <- 1:5
> x[3]
[1] 3
> x[3] <- 20
> x
[1]  1  2 20  4  5
```

The index itself can be a vector of mode numeric:

```
> i <- c(1, 3)
> x[i]
[1]  1 20
```

If `x` is a matrix or a data frame, the value of the ith line and jth column is accessed with `x[i, j]`. To access all values of a given row or column, one has simply to omit the appropriate index (without forgetting the comma!):

```
> x <- matrix(1:6, 2, 3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> x[, 3] <- 21:22
> x
     [,1] [,2] [,3]
[1,]    1    3   21
[2,]    2    4   22
> x[, 3]
[1] 21 22
```

You have certainly noticed that the last result is a vector and not a matrix. The default behaviour of R is to return an object of the lowest dimension possible. This can be altered with the option `drop` which default is `TRUE`:

```
> x[, 3, drop = FALSE]
     [,1]
[1,]   21
[2,]   22
```

This indexing system is easily generalized to arrays, with as many indices as the number of dimensions of the array (for example, a three dimensional array: `x[i, j, k]`, `x[, , 3]`, `x[, , 3, drop = FALSE]`, and so on). It may be useful to keep in mind that indexing is made with square brackets, while parentheses are used for the arguments of a function:

```
> x(1)
Error: couldn’t find function "x"
```

Indexing can also be used to suppress one or several rows or columns using negative values. For example, x[-1,] will suppress the first row, while x[-c(1, 15),] will do the same for the 1st and 15th rows. Using the matrix defined above:

```
> x[, -1]
     [,1] [,2]
[1,]    3   21
[2,]    4   22
> x[, -(1:2)]
[1] 21 22
> x[, -(1:2), drop = FALSE]
     [,1]
[1,]   21
[2,]   22
```

For vectors, matrices and arrays, it is possible to access the values of an element with a comparison expression as the index:

```
> x <- 1:10
> x[x >= 5] <- 20
> x
 [1]  1  2  3  4 20 20 20 20 20 20
> x[x == 1] <- 25
> x
 [1] 25  2  3  4 20 20 20 20 20 20
```

A practical use of the logical indexing is, for instance, the possibility to select the even elements of an integer variable:

```
> x <- rpois(40, lambda=5)
> x
 [1]  5  9  4  7  7  6  4  5 11  3  5  7  1  5  3  9  2  2  5  2
[21]  4  6  6  5  4  5  3  4  3  3  3  7  7  3  8  1  4  2  1  4
> x[x %% 2 == 0]
 [1] 4 6 4 2 2 2 4 6 6 4 4 8 4 2 4
```

Thus, this indexing system uses the logical values returned, in the above examples, by comparison operators. These logical values can be computed beforehand, they then will be recycled if necessary:

```
> x <- 1:40
> s <- c(FALSE, TRUE)
> x[s]
 [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
```

Logical indexing can also be used with data frames, but with caution since different columns of the data drame may be of different modes.

For lists, accessing the different elements (which can be any kind of object) is done either with single or with double square brackets: the difference is that with single brackets a list is returned, whereas double brackets *extract* the object from the list. The result can then be itself indexed as previously seen for vectors, matrices, etc. For instance, if the third object of a list is a vector, its ith value can be accessed using `my.list[[3]][i]`, if it is a three dimensional array using `my.list[[3]][i, j, k]`, and so on. Another difference is that `my.list[1:2]` will return a list with the first and second elements of the original list, whereas `my.list[[1:2]]` will no not give the expected result.

3.5.5 Accessing the values of an object with names

The *names* are labels of the elements of an object, and thus of mode character. They are generally optional attributes. There are several kinds of names (*names*, *colnames*, *rownames*, *dimnames*).

The *names* of a vector are stored in a vector of the same length of the object, and can be accessed with the function `names`.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("a", "b", "c")
> x
a b c
1 2 3
> names(x)
[1] "a" "b" "c"
> names(x) <- NULL
> x
[1] 1 2 3
```

For matrices and data frames, *colnames* and *rownames* are labels of the columns and rows, respectively. They can be accessed either with their respective functions, or with `dimnames` which returns a list with both vectors.

```
> X <- matrix(1:4, 2)
> rownames(X) <- c("a", "b")
> colnames(X) <- c("c", "d")
> X
  c d
a 1 3
b 2 4
> dimnames(X)
[[1]]
[1] "a" "b"
```

```
[[2]]
[1] "c" "d"
```

For arrays, the names of the dimensions can be accessed with `dimnames`:

```
> A <- array(1:8, dim = c(2, 2, 2))
> A
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

> dimnames(A) <- list(c("a", "b"), c("c", "d"), c("e", "f"))
> A
, , e

  c d
a 1 3
b 2 4

, , f

  c d
a 5 7
b 6 8
```

If the elements of an object have names, they can be extracted by using them as indices. Actually, this should be termed 'subsetting' rather than 'extraction' since the attributes of the original object are kept. For instance, if a data frame `DF` contains the variables `x`, `y`, and `z`, the command `DF["x"]` will return a data frame with just `x`; `DF[c("x", "y")]` will return a data frame with both variables. This works with lists as well if the elements in the list have names.

As the reader surely realizes, the index used here is a vector of mode character. Like the numeric or logical vectors seen above, this vector can be defined beforehand and then used for the extraction.

To extract a vector or a factor from a data frame, on can use the operator `$` (e.g., `DF$x`). This also works with lists.

3.5.6 The data editor

It is possible to use a graphical spreadsheet-like editor to edit a “data” object. For example, if `X` is a matrix, the command `data.entry(X)` will open a graphic editor and one will be able to modify some values by clicking on the appropriate cells, or to add new columns or rows.

The function `data.entry` modifies directly the object given as argument without needing to assign its result. On the other hand, the function `de` returns a list with the objects given as arguments and possibly modified. This result is displayed on the screen by default, but, as for most functions, can be assigned to an object.

The details of using the data editor depend on the operating system.

3.5.7 Arithmetics and simple functions

There are numerous functions in R to manipulate data. We have already seen the simplest one, `c` which concatenates the objects listed in parentheses. For example:

```
> c(1:5, seq(10, 11, 0.2))
 [1]  1.0  2.0  3.0  4.0  5.0 10.0 10.2 10.4 10.6 10.8 11.0
```

Vectors can be manipulated with classical arithmetic expressions:

```
> x <- 1:4
> y <- rep(1, 4)
> z <- x + y
> z
[1] 2 3 4 5
```

Vectors of different lengths can be added; in this case, the shortest vector is recycled. Examples:

```
> x <- 1:4
> y <- 1:2
> z <- x + y
> z
[1] 2 4 4 6
> x <- 1:3
> y <- 1:2
> z <- x + y
Warning message:
longer object length
 is not a multiple of shorter object length in: x + y
> z
[1] 2 4 4
```

Note that R returned a warning message and not an error message, thus the operation has been performed. If we want to add (or multiply) the same value to all the elements of a vector:

```
> x <- 1:4
> a <- 10
> z <- a * x
> z
[1] 10 20 30 40
```

The functions available in R for manipulating data are too many to be listed here. One can find all the basic mathematical functions (`log`, `exp`, `log10`, `log2`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `abs`, `sqrt`, ...), special functions (`gamma`, `digamma`, `beta`, `besselI`, ...), as well as diverse functions useful in statistics. Some of these functions are listed in the following table.

`sum(x)`	sum of the elements of `x`
`prod(x)`	product of the elements of `x`
`max(x)`	maximum of the elements of `x`
`min(x)`	minimum of the elements of `x`
`which.max(x)`	returns the index of the greatest element of `x`
`which.min(x)`	returns the index of the smallest element of `x`
`range(x)`	id. than `c(min(x), max(x))`
`length(x)`	number of elements in `x`
`mean(x)`	mean of the elements of `x`
`median(x)`	median of the elements of `x`
`var(x)` or `cov(x)`	variance of the elements of `x` (calculated on $n - 1$); if `x` is a matrix or a data frame, the variance-covariance matrix is calculated
`cor(x)`	correlation matrix of `x` if it is a matrix or a data frame (1 if `x` is a vector)
`var(x, y)` or `cov(x, y)`	covariance between `x` and `y`, or between the columns of `x` and those of `y` if they are matrices or data frames
`cor(x, y)`	linear correlation between `x` and `y`, or correlation matrix if they are matrices or data frames

These functions return a single value (thus a vector of length one), except `range` which returns a vector of length two, and `var`, `cov`, and `cor` which may return a matrix. The following functions return more complex results.

`round(x, n)`	rounds the elements of `x` to `n` decimals
`rev(x)`	reverses the elements of `x`
`sort(x)`	sorts the elements of `x` in increasing order; to sort in decreasing order: `rev(sort(x))`
`rank(x)`	ranks of the elements of `x`

`log(x, base)`	computes the logarithm of `x` with base `base`
`scale(x)`	if `x` is a matrix, centers and reduces the data; to center only use the option `center=FALSE`, to reduce only `scale=FALSE` (by default `center=TRUE, scale=TRUE`)
`pmin(x,y,...)`	a vector which ith element is the minimum of `x[i]`, `y[i]`, ...
`pmax(x,y,...)`	id. for the maximum
`cumsum(x)`	a vector which ith element is the sum from `x[1]` to `x[i]`
`cumprod(x)`	id. for the product
`cummin(x)`	id. for the minimum
`cummax(x)`	id. for the maximum
`match(x, y)`	returns a vector of the same length than `x` with the elements of `x` which are in `y` (`NA` otherwise)
`which(x == a)`	returns a vector of the indices of `x` if the comparison operation is true (`TRUE`), in this example the values of `i` for which `x[i] == a` (the argument of this function must be a variable of mode logical)
`choose(n, k)`	computes the combinations of k events among n repetitions $= n!/[(n-k)!k!]$
`na.omit(x)`	suppresses the observations with missing data (`NA`) (suppresses the corresponding line if `x` is a matrix or a data frame)
`na.fail(x)`	returns an error message if `x` contains at least one `NA`
`unique(x)`	if `x` is a vector or a data frame, returns a similar object but with the duplicate elements suppressed
`table(x)`	returns a table with the numbers of the differents values of `x` (typically for integers or factors)
`table(x, y)`	contingency table of `x` and `y`
`subset(x, ...)`	returns a selection of `x` with respect to criteria (`...`, typically comparisons: `x$V1 < 10`); if `x` is a data frame, the option `select` gives the variables to be kept (or dropped using a minus sign)
`sample(x, size)`	resample randomly and without replacement `size` elements in the vector `x`, the option `replace = TRUE` allows to resample with replacement

3.5.8 Matrix computation

R has facilities for matrix computation and manipulation. The functions `rbind` and `cbind` bind matrices with respect to the lines or the columns, respectively:

```
> m1 <- matrix(1, nr = 2, nc = 2)
> m2 <- matrix(2, nr = 2, nc = 2)
> rbind(m1, m2)
     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    2    2
[4,]    2    2
> cbind(m1, m2)
     [,1] [,2] [,3] [,4]
```

```
[1,]    1    1    2    2
[2,]    1    1    2    2
```

The operator for the product of two matrices is '%*%'. For example, considering the two matrices m1 and m2 above:

```
> rbind(m1, m2) %*% cbind(m1, m2)
     [,1] [,2] [,3] [,4]
[1,]    2    2    4    4
[2,]    2    2    4    4
[3,]    4    4    8    8
[4,]    4    4    8    8
> cbind(m1, m2) %*% rbind(m1, m2)
     [,1] [,2]
[1,]   10   10
[2,]   10   10
```

The transposition of a matrix is done with the function t; this function works also with a data frame.

The function diag can be used to extract or modify the diagonal of a matrix, or to build a diagonal matrix.

```
> diag(m1)
[1] 1 1
> diag(rbind(m1, m2) %*% cbind(m1, m2))
[1] 2 2 8 8
> diag(m1) <- 10
> m1
     [,1] [,2]
[1,]   10    1
[2,]    1   10
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
> v <- c(10, 20, 30)
> diag(v)
     [,1] [,2] [,3]
[1,]   10    0    0
[2,]    0   20    0
[3,]    0    0   30
> diag(2.1, nr = 3, nc = 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]  2.1  0.0  0.0    0    0
[2,]  0.0  2.1  0.0    0    0
[3,]  0.0  0.0  2.1    0    0
```

R has also some special functions for matrix computation. We can cite here `solve` for inverting a matrix, `qr` for decomposition, `eigen` for computing eigenvalues and eigenvectors, and `svd` for singular value decomposition.

4 Graphics with R

R offers a remarkable variety of graphics. To get an idea, one can type `demo(graphics)` or `demo(persp)`. It is not possible to detail here the possibilities of R in terms of graphics, particularly since each graphical function has a large number of options making the production of graphics very flexible.

The way graphical functions work deviates substantially from the scheme sketched at the beginning of this document. Particularly, the result of a graphical function cannot be assigned to an object[11] but is sent to a *graphical device*. A graphical device is a graphical window or a file.

There are two kinds of graphical functions: the *high-level plotting functions* which create a new graph, and the *low-level plotting functions* which add elements to an existing graph. The graphs are produced with respect to *graphical parameters* which are defined by default and can be modified with the function `par`.

We will see in a first time how to manage graphics and graphical devices; we will then somehow detail the graphical functions and parameters. We will see a practical example of the use of these functionalities in producing graphs. Finally, we will see the packages grid and lattice whose functioning is different from the one summarized above.

4.1 Managing graphics

4.1.1 Opening several graphical devices

When a graphical function is executed, if no graphical device is open, R opens a graphical window and displays the graph. A graphical device may be open with an appropriate function. The list of available graphical devices depends on the operating system. The graphical windows are called `X11` under Unix/Linux and `windows` under Windows. In all cases, one can open a graphical window with the command `x11()` which also works under Windows because of an alias towards the command `windows()`. A graphical device which is a file will be open with a function depending on the format: `postscript()`, `pdf()`, `png()`, . . . The list of available graphical devices can be found with `?device`.

The last open device becomes the active graphical device on which all subsequent graphs are displayed. The function `dev.list()` displays the list of open devices:

```
> x11(); x11(); pdf()
> dev.list()
```

[11] There are a few remarkable exceptions: `hist()` and `barplot()` produce also numeric results as lists or matrices.

```
X11 X11 pdf
  2   3   4
```

The figures displayed are the device numbers which must be used to change the active device. To know what is the active device:

```
> dev.cur()
pdf
  4
```

and to change the active device:

```
> dev.set(3)
X11
  3
```

The function `dev.off()` closes a device: by default the active device is closed, otherwise this is the one which number is given as argument to the function. R then displays the number of the new active device:

```
> dev.off(2)
X11
  3
> dev.off()
pdf
  4
```

Two specific features of the Windows version of R are worth mentioning: a Windows Metafile device can be open with the function `win.metafile`, and a menu "History" displayed when the graphical window is selected allowing recording of all graphs drawn during a session (by default, the recording system is off, the user switches it on by clicking on "Recording" in this menu).

4.1.2 Partitioning a graphic

The function `split.screen` partitions the active graphical device. For example:

```
> split.screen(c(1, 2))
```

divides the device into two parts which can be selected with `screen(1)` or `screen(2)`; `erase.screen()` deletes the last drawn graph. A part of the device can itself be divided with `split.screen()` leading to the possibility to make complex arrangements.

These functions are incompatible with others (such as `layout` or `coplot`) and must not be used with multiple graphical devices. Their use should be limited, for instance, to graphical exploration of data.

The function `layout` partitions the active graphic window in several parts where the graphs will be displayed successively. Its main argument is a matrix with integer numbers indicating the numbers of the "sub-windows". For example, to divide the device into four equal parts:

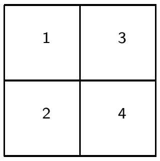
```
> layout(matrix(1:4, 2, 2))
```

It is of course possible to create this matrix previously allowing to better visualize how the device is divided:

```
> mat <- matrix(1:4, 2, 2)
> mat
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> layout(mat)
```

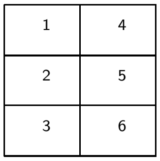
To actually visualize the partition created, one can use the function `layout.show` with the number of sub-windows as argument (here 4). In this example, we will have:

```
> layout.show(4)
```

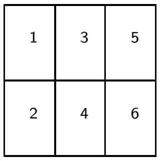


The following examples show some of the possibilities offered by `layout()`.

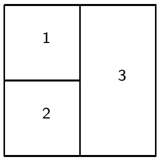
```
> layout(matrix(1:6, 3, 2))
> layout.show(6)
```



```
> layout(matrix(1:6, 2, 3))
> layout.show(6)
```



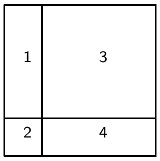
```
> m <- matrix(c(1:3, 3), 2, 2)
> layout(m)
> layout.show(3)
```



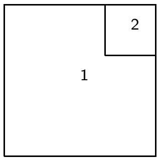
In all these examples, we have not used the option `byrow` of `matrix()`, the sub-windows are thus numbered column-wise; one can just specify `matrix(..., byrow=TRUE)` so that the sub-windows are numbered row-wise. The numbers in the matrix may also be given in any order, for example, `matrix(c(2, 1, 4, 3), 2, 2)`.

By default, `layout()` partitions the device with regular heights and widths: this can be modified with the options `widths` and `heights`. These dimensions are given relatively[12]. Examples:

```
> m <- matrix(1:4, 2, 2)
> layout(m, widths=c(1, 3),
           heights=c(3, 1))
> layout.show(4)
```

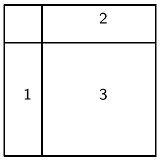


```
> m <- matrix(c(1,1,2,1),2,2)
> layout(m, widths=c(2, 1),
           heights=c(1, 2))
> layout.show(2)
```

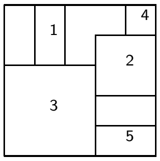


Finally, the numbers in the matrix can include zeros giving the possibility to make complex (or even esoterical) partitions.

```
> m <- matrix(0:3, 2, 2)
> layout(m, c(1, 3), c(1, 3))
> layout.show(3)
```



```
> m <- matrix(scan(), 5, 5)
1:  0 0 3 3 3 1 1 3 3 3
11:  0 0 3 3 3 0 2 2 0 5
21:  4 2 2 0 5
26:
Read 25 items
> layout(m)
> layout.show(5)
```



[12] They can be given in centimetres, see `?layout`.

4.2 Graphical functions

Here is an overview of the high-level graphical functions in R.

`plot(x)`	plot of the values of `x` (on the y-axis) ordered on the x-axis
`plot(x, y)`	bivariate plot of `x` (on the x-axis) and `y` (on the y-axis)
`sunflowerplot(x, y)`	id. but the points with similar coordinates are drawn as a flower which petal number represents the number of points
`pie(x)`	circular pie-chart
`boxplot(x)`	"box-and-whiskers" plot
`stripchart(x)`	plot of the values of `x` on a line (an alternative to `boxplot()` for small sample sizes)
`coplot(x~y | z)`	bivariate plot of `x` and `y` for each value (or interval of values) of `z`
`interaction.plot (f1, f2, y)`	if `f1` and `f2` are factors, plots the means of `y` (on the y-axis) with respect to the values of `f1` (on the x-axis) and of `f2` (different curves); the option `fun` allows to choose the summary statistic of `y` (by default `fun=mean`)
`matplot(x,y)`	bivariate plot of the first column of `x` *vs.* the first one of `y`, the second one of `x` *vs.* the second one of `y`, etc.
`dotchart(x)`	if `x` is a data frame, plots a Cleveland dot plot (stacked plots line-by-line and column-by-column)
`fourfoldplot(x)`	visualizes, with quarters of circles, the association between two dichotomous variables for different populations (`x` must be an array with `dim=c(2, 2, k)`, or a matrix with `dim=c(2, 2)` if $k = 1$)
`assocplot(x)`	Cohen–Friendly graph showing the deviations from independence of rows and columns in a two dimensional contingency table
`mosaicplot(x)`	'mosaic' graph of the residuals from a log-linear regression of a contingency table
`pairs(x)`	if `x` is a matrix or a data frame, draws all possible bivariate plots between the columns of `x`
`plot.ts(x)`	if `x` is an object of class `"ts"`, plot of `x` with respect to time, `x` may be multivariate but the series must have the same frequency and dates
`ts.plot(x)`	id. but if `x` is multivariate the series may have different dates and must have the same frequency
`hist(x)`	histogram of the frequencies of `x`
`barplot(x)`	histogram of the values of `x`
`qqnorm(x)`	quantiles of `x` with respect to the values expected under a normal law
`qqplot(x, y)`	quantiles of `y` with respect to the quantiles of `x`
`contour(x, y, z)`	contour plot (data are interpolated to draw the curves), `x` and `y` must be vectors and `z` must be a matrix so that `dim(z)=c(length(x), length(y))` (`x` and `y` may be omitted)
`filled.contour (x, y, z)`	id. but the areas between the contours are coloured, and a legend of the colours is drawn as well
`image(x, y, z)`	id. but the actual data are represented with colours
`persp(x, y, z)`	id. but in perspective
`stars(x)`	if `x` is a matrix or a data frame, draws a graph with segments or a star where each row of `x` is represented by a star and the columns are the lengths of the segments

`symbols(x, y, ...)`	draws, at the coordinates given by `x` and `y`, symbols (circles, squares, rectangles, stars, thermometres or "boxplots") which sizes, colours, etc, are specified by supplementary arguments
`termplot(mod.obj)`	plot of the (partial) effects of a regression model (`mod.obj`)

For each function, the options may be found with the on-line help in R. Some of these options are identical for several graphical functions; here are the main ones (with their possible default values):

`add=FALSE` if `TRUE` superposes the plot on the previous one (if it exists)

`axes=TRUE` if `FALSE` does not draw the axes and the box

`type="p"` specifies the type of plot, `"p"`: points, `"l"`: lines, `"b"`: points connected by lines, `"o"`: id. but the lines are over the points, `"h"`: vertical lines, `"s"`: steps, the data are represented by the top of the vertical lines, `"S"`: id. but the data are represented by the bottom of the vertical lines

`xlim=, ylim=` specifies the lower and upper limits of the axes, for example with `xlim=c(1, 10)` or `xlim=range(x)`

`xlab=, ylab=` annotates the axes, must be variables of mode character

`main=` main title, must be a variable of mode character

`sub=` sub-title (written in a smaller font)

4.3 Low-level plotting commands

R has a set of graphical functions which affect an already existing graph: they are called *low-level plotting commands*. Here are the main ones:

`points(x, y)`	adds points (the option `type=` can be used)
`lines(x, y)`	id. but with lines
`text(x, y, labels, ...)`	adds text given by `labels` at coordinates (x,y); a typical use is: `plot(x, y, type="n"); text(x, y, names)`
`mtext(text, side=3, line=0, ...)`	adds text given by `text` in the margin specified by `side` (see `axis()` below); `line` specifies the line from the plotting area
`segments(x0, y0, x1, y1)`	draws lines from points (x0,y0) to points (x1,y1)
`arrows(x0, y0, x1, y1, angle= 30, code=2)`	id. with arrows at points (x0,y0) if `code=2`, at points (x1,y1) if `code=1`, or both if `code=3`; `angle` controls the angle from the shaft of the arrow to the edge of the arrow head
`abline(a,b)`	draws a line of slope `b` and intercept `a`
`abline(h=y)`	draws a horizontal line at ordinate `y`
`abline(v=x)`	draws a vertical line at abcissa `x`
`abline(lm.obj)`	draws the regression line given by `lm.obj` (see section 5)

`rect(x1, y1, x2, y2)`	draws a rectangle which left, right, bottom, and top limits are `x1`, `x2`, `y1`, and `y2`, respectively
`polygon(x, y)`	draws a polygon linking the points with coordinates given by `x` and `y`
`legend(x, y, legend)`	adds the legend at the point (`x`,`y`) with the symbols given by `legend`
`title()`	adds a title and optionally a sub-title
`axis(side, vect)`	adds an axis at the bottom (`side=1`), on the left (`2`), at the top (`3`), or on the right (`4`); `vect` (optional) gives the abcissa (or ordinates) where tick-marks are drawn
`box()`	adds a box around the current plot
`rug(x)`	draws the data `x` on the x-axis as small vertical lines
`locator(n, type="n", ...)`	returns the coordinates (x, y) after the user has clicked `n` times on the plot with the mouse; also draws symbols (`type="p"`) or lines (`type="l"`) with respect to optional graphic parameters (`...`); by default nothing is drawn (`type="n"`)

Note the possibility to add mathematical expressions on a plot with `text(x, y, expression(...))`, where the function `expression` transforms its argument in a mathematical equation. For example,

```
> text(x, y, expression(p == over(1, 1+e^-(beta*x+alpha))))
```

will display, on the plot, the following equation at the point of coordinates (x, y):

$$\mathrm{p} = \frac{1}{1 + \mathrm{e}^{-(\beta\mathrm{x}+\alpha)}}$$

To include in an expression a variable we can use the functions `substitute` and `as.expression`; for example to include a value of R^2 (previously computed and stored in an object named `Rsquared`):

```
> text(x, y, as.expression(substitute(R^2==r, list(r=Rsquared))))
```

will display on the plot at the point of coordinates (x, y):

$$\mathrm{R}^2 = 0.9856298$$

To display only three decimals, we can modify the code as follows:

```
> text(x, y, as.expression(substitute(R^2==r,
+                           list(r=round(Rsquared, 3)))))
```

will display:

$$\mathrm{R}^2 = 0.986$$

Finally, to write the R in italics:

```
> text(x, y, as.expression(substitute(italic(R)^2==r,
+                           list(r=round(Rsquared, 3)))))
```

$$R^2 = 0.986$$

4.4 Graphical parameters

In addition to low-level plotting commands, the presentation of graphics can be improved with graphical parameters. They can be used either as options of graphic functions (but it does not work for all), or with the function `par` to change permanently the graphical parameters, i.e. the subsequent plots will be drawn with respect to the parameters specified by the user. For instance, the following command:

```
> par(bg="yellow")
```

will result in all subsequent plots drawn with a yellow background. There are 73 graphical parameters, some of them have very similar functions. The exhaustive list of these parameters can be read with `?par`; I will limit the following table to the most usual ones.

`adj`	controls text justification with respect to the left border of the text so that `0` is left-justified, `0.5` is centred, `1` is right-justified, values > 1 move the text further to the left, and negative values further to the right; if two values are given (e.g., `c(0, 0)`) the second one controls vertical justification with respect to the text baseline
`bg`	specifies the colour of the background (e.g., `bg="red"`, `bg="blue"`; the list of the 657 available colours is displayed with `colors()`)
`bty`	controls the type of box drawn around the plot, allowed values are: `"o"`, `"l"`, `"7"`, `"c"`, `"u"` ou `"]"` (the box looks like the corresponding character); if `bty="n"` the box is not drawn
`cex`	a value controlling the size of texts and symbols with respect to the default; the following parameters have the same control for numbers on the axes, `cex.axis`, the axis labels, `cex.lab`, the title, `cex.main`, and the sub-title, `cex.sub`
`col`	controls the colour of symbols; as for `cex` there are: `col.axis`, `col.lab`, `col.main`, `col.sub`
`font`	an integer which controls the style of text (`1`: normal, `2`: italics, `3`: bold, `4`: bold italics); as for `cex` there are: `font.axis`, `font.lab`, `font.main`, `font.sub`
`las`	an integer which controls the orientation of the axis labels (`0`: parallel to the axes, `1`: horizontal, `2`: perpendicular to the axes, `3`: vertical)
`lty`	controls the type of lines, can be an integer (`1`: solid, `2`: dashed, `3`: dotted, `4`: dotdash, `5`: longdash, `6`: twodash), or a string of up to eight characters (between `"0"` and `"9"`) which specifies alternatively the length, in points or pixels, of the drawn elements and the blanks, for example `lty="44"` will have the same effet than `lty=2`
`lwd`	a numeric which controls the width of lines
`mar`	a vector of 4 numeric values which control the space between the axes and the border of the graph of the form `c(bottom, left, top, right)`, the default values are `c(5.1, 4.1, 4.1, 2.1)`
`mfcol`	a vector of the form `c(nr,nc)` which partitions the graphic window as a matrix of `nr` lines and `nc` columns, the plots are then drawn in columns (see section 4.1.2)
`mfrow`	id. but the plots are then drawn in line (see section 4.1.2)
`pch`	controls the type of symbol, either an integer between 1 and 25, or any single character within `""` (Fig. 2)
`ps`	an integer which controls the size in points of texts and symbols

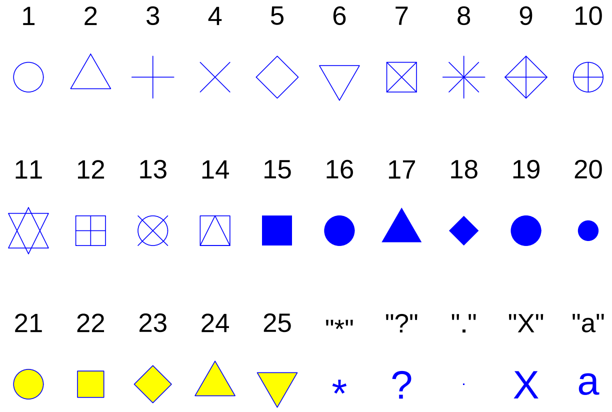


Figure 2: The plotting symbols in R (`pch=1:25`). The colours were obtained with the options `col="blue", bg="yellow"`, the second option has an effect only for the symbols 21–25. Any character can be used (`pch="*"`, `"?"`, `"."`, . . .).

`pty`	a character which specifies the type of the plotting region, `"s"`: square, `"m"`: maximal
`tck`	a value which specifies the length of tick-marks on the axes as a fraction of the smallest of the width or height of the plot; if `tck=1` a grid is drawn
`tcl`	id. but as a fraction of the height of a line of text (by default `tcl=-0.5`)
`xaxt`	if `xaxt="n"` the x-axis is set but not drawn (useful in conjunction with `axis(side=1, ...)`)
`yaxt`	if `yaxt="n"` the y-axis is set but not drawn (useful in conjunction with `axis(side=2, ...)`)

4.5 A practical example

In order to illustrate R's graphical functionalities, let us consider a simple example of a bivariate graph of 10 pairs of random variates. These values were generated with:

```
> x <- rnorm(10)
> y <- rnorm(10)
```

The wanted graph will be obtained with `plot()`; one will type the command:

```
> plot(x, y)
```

and the graph will be plotted on the active graphical device. The result is shown on Fig. 3. By default, R makes graphs in an "intelligent" way:

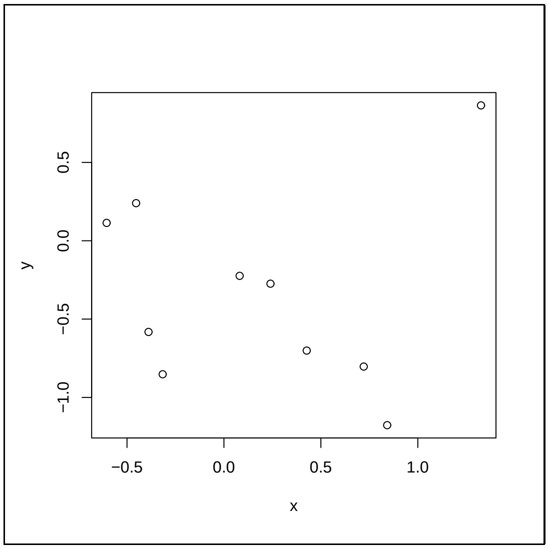


Figure 3: The function `plot` used without options.

the spaces between tick-marks on the axes, the placement of labels, etc, are calculated so that the resulting graph is as intelligible as possible.

The user may, nevertheless, change the way a graph is presented, for instance, to conform to a pre-defined editorial style, or to give it a personal touch for a talk. The simplest way to change the presentation of a graph is to add options which will modify the default arguments. In our example, we can modify significantly the figure in the following way:

```
plot(x, y, xlab="Ten random values", ylab="Ten other values",
     xlim=c(-2, 2), ylim=c(-2, 2), pch=22, col="red",
     bg="yellow", bty="l", tcl=0.4,
     main="How to customize a plot with R", las=1, cex=1.5)
```

The result is on Fig. 4. Let us detail each of the used options. First, `xlab` and `ylab` change the axis labels which, by default, were the names of the variables. Then, `xlim` and `ylim` allow us to define the limits on both axes[13]. The graphical parameter `pch` is used here as an option: `pch=22` specifies a square which contour and background colours may be different and are given by, respectively, `col` and `bg`. The table of graphical parameters gives the meaning of the modifications done by `bty`, `tcl`, `las` and `cex`. Finally, a title is added with the option `main`.

The graphical parameters and the low-level plotting functions allow us to go further in the presentation of a graph. As we have seen previously, some graphical parameters cannot be passed as arguments to a function like `plot`.

[13] By default, R adds 4% on each side of the axis limit. This behaviour may be altered by setting the graphical parameters `xaxs="i"` and `yaxs="i"` (they can be passed as options to `plot()`).

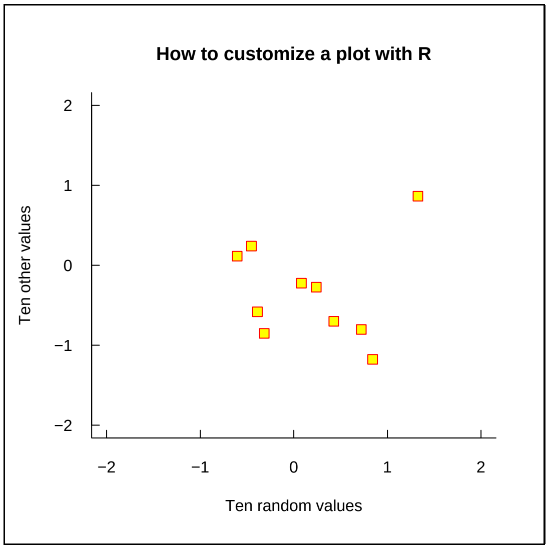


Figure 4: The function `plot` used with options.

We will now modify some of these parameters with `par()`, it is thus necessary to type several commands. When the graphical parameters are changed, it is useful to save their initial values beforehand to be able to restore them afterwards. Here are the commands used to obtain Fig. 5.

```
opar <- par()
par(bg="lightyellow", col.axis="blue", mar=c(4, 4, 2.5, 0.25))
plot(x, y, xlab="Ten random values", ylab="Ten other values",
     xlim=c(-2, 2), ylim=c(-2, 2), pch=22, col="red", bg="yellow",
     bty="l", tcl=-.25, las=1, cex=1.5)
title("How to customize a plot with R (bis)", font.main=3, adj=1)
par(opar)
```

Let us detail the actions resulting from these commands. First, the default graphical parameters are copied in a list called here `opar`. Three parameters will be then modified: `bg` for the colour of the background, `col.axis` for the colour of the numbers on the axes, and `mar` for the sizes of the margins around the plotting region. The graph is drawn in a nearly similar way to Fig. 4. The modifications of the margins allowed to use the space around the plotting area. The title here is added with the low-level plotting function `title` which allows to give some parameters as arguments without altering the rest of the graph. Finally, the initial graphical parameters are restored with the last command.

Now, total control! On Fig. 5, R still determines a few things such as the number of tick marks on the axes, or the space between the title and the plotting area. We will see now how to totally control the presentation of the graph. The approach used here is to plot a “blank” graph with `plot(..., type="n")`, then to add points, axes, labels, etc, with low-level plotting func-

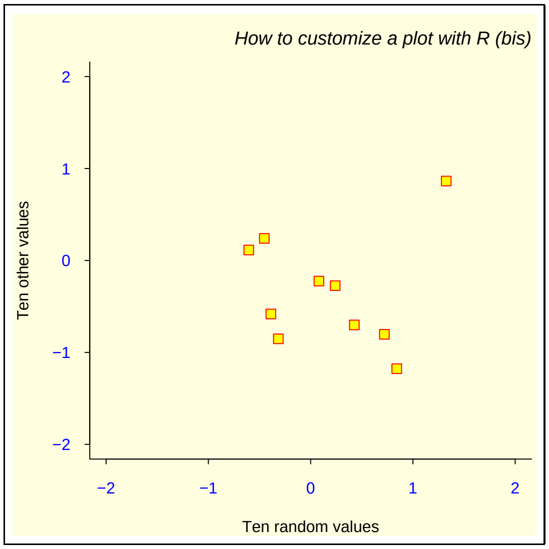


Figure 5: The functions `par`, `plot` and `title`.

tions. We will fancy a few arrangements such as changing the colour of the plotting area. The commands follow, and the resulting graph is on Fig. 6.

```
opar <- par()
par(bg="lightgray", mar=c(2.5, 1.5, 2.5, 0.25))
plot(x, y, type="n", xlab="", ylab="", xlim=c(-2, 2),
     ylim=c(-2, 2), xaxt="n", yaxt="n")
rect(-3, -3, 3, 3, col="cornsilk")
points(x, y, pch=10, col="red", cex=2)
axis(side=1, c(-2, 0, 2), tcl=-0.2, labels=FALSE)
axis(side=2, -1:1, tcl=-0.2, labels=FALSE)
title("How to customize a plot with R (ter)",
      font.main=4, adj=1, cex.main=1)
mtext("Ten random values", side=1, line=1, at=1, cex=0.9, font=3)
mtext("Ten other values", line=0.5, at=-1.8, cex=0.9, font=3)
mtext(c(-2, 0, 2), side=1, las=1, at=c(-2, 0, 2), line=0.3,
      col="blue", cex=0.9)
mtext(-1:1, side=2, las=1, at=-1:1, line=0.2, col="blue", cex=0.9)
par(opar)
```

Like before, the default graphical parameters are saved, and the colour of the background and the margins are modified. The graph is then drawn with `type="n"` to not plot the points, `xlab=""`, `ylab=""` to not write the axis labels, and `xaxt="n"`, `yaxt="n"` to not draw the axes. This results in drawing only the box around the plotting area, and defining the axes with respect to `xlim` et `ylim`. Note that we could have used the option `axes=FALSE` but in this case neither the axes, nor the box would have been drawn.

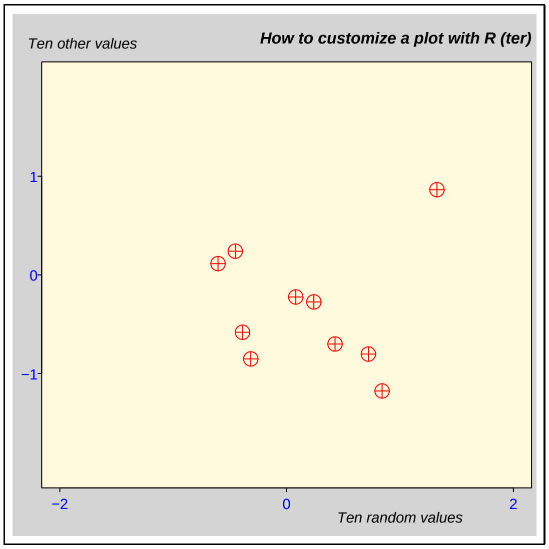


Figure 6: A “hand-made” graph.

The elements are then added in the plotting region so defined with some low-level plotting functions. Before adding the points, the colour inside the plotting area is changed with `rect()`: the size of the rectangle are chosen so that it is substantially larger than the plotting area.

The points are plotted with `points()`; a new symbol was used. The axes are added with `axis()`: the vector given as second argument specifies the coordinates of the tick-marks. The option `labels=FALSE` specifies that no annotation must be written with the tick-marks. This option also accepts a vector of mode character, for example `labels=c("A", "B", "C")`.

The title is added with `title()`, but the font is slightly changed. The annotations on the axes are written with `mtext()` (*marginal text*). The first argument of this function is a vector of mode character giving the text to be written. The option `line` indicates the distance from the plotting area (by default `line=0`), and `at` the coordinnate. The second call to `mtext()` uses the default value of `side` (3). The two other calls to `mtext()` pass a numeric vector as first argument: this will be converted into character.

4.6 The grid and lattice packages

The packages grid and lattice implement the grid and lattice systems. Grid is a new graphical mode with its own system of graphical parameters which are distinct from those seen above. The two main distinctions of grid compared to the base graphics are:

- a more flexible way to split graphical devices using *viewports* which could be overpalling (graphical objects may even be shared among distinct viewports, e.g., arrows);

- graphical objects (*grob*) may be modified or removed from a graph without requiring the re-draw all the graph (as must be done with base graphics).

Grid graphics cannot usually be combined or mixed with base graphics (the gridBase package must be used to do this). However, it is possible to use both graphical modes in the same session on the same graphical device.

Lattice is essentially the implementation in R of the Trellis graphics of S-PLUS. Trellis is an approach for visualizing multivariate data which is particularly appropriate for the exploration of relations or interactions among variables[14]. The main idea behind lattice (and Trellis as well) is that of conditional multiple graphs: a bivariate graph will be split in several graphs with respect to the values of a third variable. The function `coplot` uses a similar approach, but lattice offers much wider functionalities. Lattice uses the grid graphical mode.

Most functions in lattice take a formula as their main argument[15], for example `y ~ x`. The formula `y ~ x | z` means that the graph of `y` with respect to `x` will be plotted as several graphs with respect to the values of `z`.

The following table gives the main functions in lattice. The formula given as argument is the typical necessary formula, but all these functions accept a conditional formula (`y ~ x | z`) as main argument; in the latter case, a multiple graph, with respect to the values of `z`, is plotted as will be seen in the examples below.

`barchart(y ~ x)`	histogram of the values of `y` with respect to those of `x`
`bwplot(y ~ x)`	"box-and-whiskers" plot
`densityplot(~ x)`	density functions plot
`dotplot(y ~ x)`	Cleveland dot plot (stacked plots line-by-line and column-by-column)
`histogram(~ x)`	histogram of the frequencies of `x`
`qqmath(~ x)`	quantiles of `x` with respect to the values expected under a theoretical distribution
`stripplot(y ~ x)`	single dimension plot, `x` must be numeric, `y` may be a factor
`qq(y ~ x)`	quantiles to compare two distributions, `x` must be numeric, `y` may be numeric, character, or factor but must have two 'levels'
`xyplot(y ~ x)`	bivariate plots (with many functionalities)

`levelplot(z ~ x*y)`
`contourplot(z ~ x*y)`
	coloured plot of the values of `z` at the coordinates given by `x` and `y` (`x`, `y` and `z` are all of the same length)

`cloud(z ~ x*y)`	3-D perspective plot (points)
`wireframe(z ~ x*y)`	id. (surface)
`splom(~ x)`	matrix of bivariate plots
`parallel(~ x)`	parallel coordinates plot

Let us see now some examples in order to illustrate a few aspects of lattice. The package must be loaded in memory with the command `library(lattice)`

[14] http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/index.html

[15] `plot()` also accepts a formula as its main argument: if `x` and `y` are two vectors of the same length, `plot(y ~ x)` and `plot(x, y)` will give identical graphs.

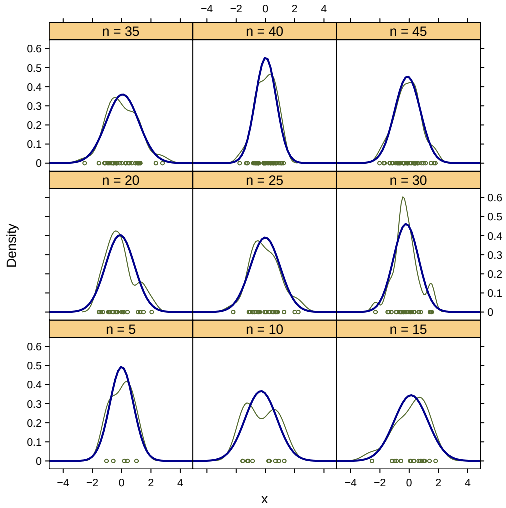


Figure 7: The function `densityplot`.

so that the functions can be accessed.

Let us start with the graphs of density functions. Such graphs can be done simply with `densityplot(~ x)` which will plot a curve of the empirical density function with the points corresponding to the observations on the x-axis (similarly to `rug()`). Our example will be slightly more complicated with the superposition, on each plot, of the curves of empirical density and those predicted from a normal law. It is necessary to use the argument `panel` which defines what is drawn on each plot. The commands are:

```
n <- seq(5, 45, 5)
x <- rnorm(sum(n))
y <- factor(rep(n, n), labels=paste("n =", n))
densityplot(~ x | y,
            panel = function(x, ...) {
                panel.densityplot(x, col="DarkOliveGreen", ...)
                panel.mathdensity(dmath=dnorm,
                                  args=list(mean=mean(x), sd=sd(x)),
                                  col="darkblue")
            })
```

The first three lines of command generate a random sample of independent normal variates which is split in sub-samples of size equal to 5, 10, 15, . . . , and 45. Then comes the call to `densityplot` producing a plot for each sub-sample. `panel` takes as argument a function. In our example, we have defined a function which calls two functions pre-defined in lattice: `panel.densityplot` to draw the empirical density function, and `panel.mathdensity` to draw the density function predicted from a normal law. The function `panel.densityplot` is called by default if no argument is given to `panel`: the command `densityplot`

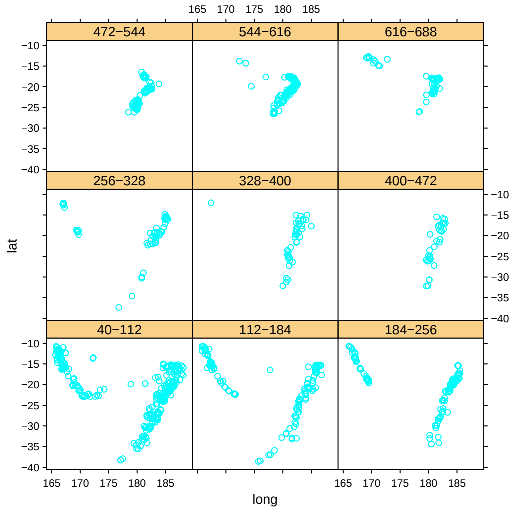


Figure 8: The function `xyplot` with the data “quakes”.

(`~ x | y`) would have resulted in the same graph than Fig. 7 but without the blue curves.

The next examples are taken, more or less modified, from the help pages of lattice, and use some data sets available in R: the locations of 1000 seismic events near the Fiji Islands, and some flower measurements made on three species of iris.

Fig. 8 shows the geographic locations of the seismic events with respect to depth. The commands necessary for this graph are:

```
data(quakes)
mini <- min(quakes$depth)
maxi <- max(quakes$depth)
int <- ceiling((maxi - mini)/9)
inf <- seq(mini, maxi, int)
quakes$depth.cat <- factor(floor(((quakes$depth - mini) / int)),
                           labels=paste(inf, inf + int, sep="-"))
xyplot(lat ~ long | depth.cat, data = quakes)
```

The first command loads the data `quakes` in memory. The five next commands create a factor by dividing the depth (variable `depth`) in nine equally-ranged intervals: the levels of this factor are labelled with the lower and upper bounds of these intervals. It then suffices to call the function `xyplot` with the appropriate formula and an argument `data` indicating where `xyplot` must look for the variables[16].

With the data `iris`, the overlap among the different species is sufficiently small so they can be plotted on the figure (Fig. 9). The commands are:

[16] `plot()` cannot take an argument `data`, the location of the variables must be given explicitly, for example `plot(quakes$long ~ quakes$lat)`.

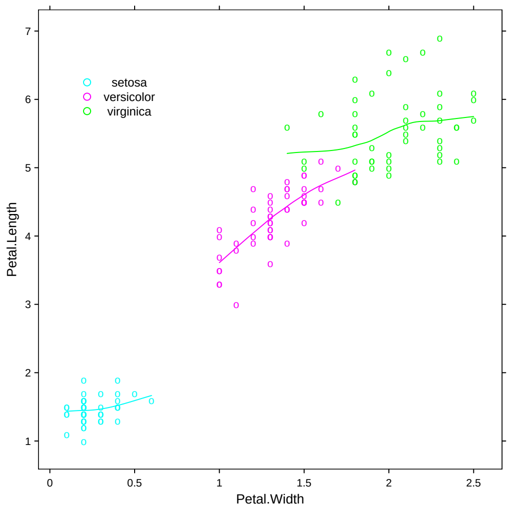


Figure 9: The function `xyplot` with the data “iris”.

```
data(iris)
xyplot(
  Petal.Length ~ Petal.Width, data = iris, groups=Species,
  panel = panel.superpose,
  type = c("p", "smooth"), span=.75,
  auto.key = list(x = 0.15, y = 0.85)
)
```

The call to the function `xyplot` is here a bit more complex than in the previous example and uses several options that we will detail. The option `groups`, as suggested by its name, defines groups that will be used by the other options. We have already seen the option `panel` which defines how the different groups will be represented on the graph: we use here a pre-defined function `panel.superpose` in order to superpose the groups on the same plot. No option is passed to `panel.superpose`, the default colours will be used to distinguish the groups. The option `type`, like in `plot()`, specifies how the data are represented, but here we can give several arguments as a vector: `"p"` to draw points and `"smooth"` to draw a smooth curve which degree of smoothness is specified by `span`. The option `auto.key` adds a legend to the graph: it is only necessary to give, as a list, the coordinates where the legend is to be plotted. Note that here these coordinates are relative to the size of the plot (i.e. in [0, 1]).

We will see now the function `splom` with the same data on iris. The following commands were used to produce Fig. 10:

```
splom(
  ~iris[1:4], groups = Species, data = iris, xlab = "",
  panel = panel.superpose,
```

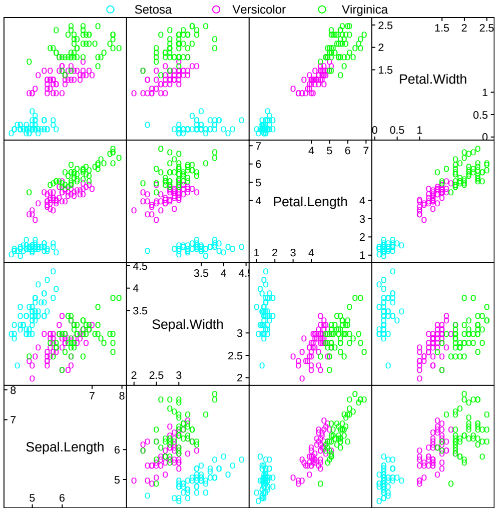


Figure 10: The function `splom` with the data “iris” (1).

```
  auto.key = list(columns = 3)
)
```

The main argument is this time a matrix (the four first columns of `iris`). The result is the set of possible bivariate plots among the columns of the matrix, like the standard function `pairs`. By default, `splom` adds the text “Scatter Plot Matrix” under the x-axis: to avoid this, the option `xlab=""` was used. The other options are similar to the previous example, except that `columns = 3` for `auto.key` was specified so the legend is displayed in three columns.

Fig. 10 could have been done with `pairs()`, but this latter function cannot make conditional graphs like on Fig. 11. The code used is relatively simple:

```
splom(~iris[1:3] | Species, data = iris, pscales = 0,
      varnames = c("Sepal\nLength", "Sepal\nWidth", "Petal\nLength"))
```

The sub-graphs being relatively small, we added two options to improve the legibility of the figure: `pscales = 0` removes the tick-marks on the axes (all sub-graphs are drawn on the same scales), and the names of the variables were re-defined to display them on two lines (`"\n"` codes for a line break in a character string).

The last example uses the method of parallel coordinates for the exploratory analysis of multivariate data. The variables are arranged on an axis (e.g., the y-axis), and the observed values are plotted on the other axis (the variables are scaled similarly, e.g., by standardizing them). The different values of the same individual are joined by a line. With the data `iris`, Fig. 12 is obtained with the following code:

```
parallel(~iris[, 1:4] | Species, data = iris, layout = c(3, 1))
```

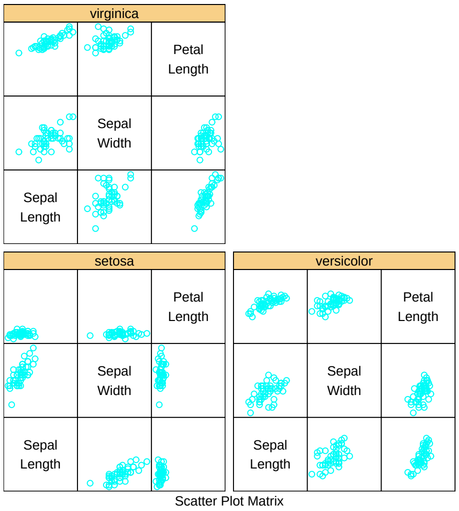


Figure 11: The function `splom` with the data “iris” (2).

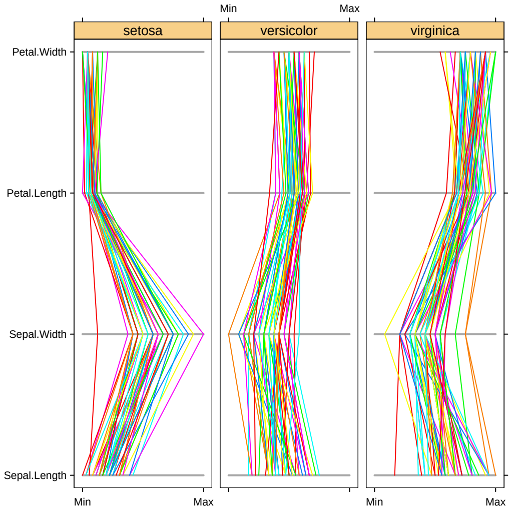


Figure 12: The function `parallel` with the data “iris”.

5 Statistical analyses with R

Even more than for graphics, it is impossible here to go in the details of the possibilities offered by R with respect to statistical analyses. My goal here is to give some landmarks with the aim to have an idea of the features of R to perform data analyses.

The package stats contains functions for a wide range of basic statistical analyses: classical tests, linear models (including least-squares regression, generalized linear models, and analysis of variance), distributions, summary statistics, hierarchical clustering, time-series analysis, nonlinear least squares, and multivariate analysis. Other statistical methods are available in a large number of packages. Some of them are distributed with a base installation of R and are labelled *recommanded*, and many other packages are *contributed* and must be installed by the user.

We will start with a simple example which requires no other package than stats in order to introduce the general approach to data analysis in R. Then, we will detail some notions, *formulae* and *generic functions*, which are useful whatever the type of analysis performed. We will conclude with an overview on packages.

5.1 A simple example of analysis of variance

The function for the analysis of variance in stats is aov. In order to try it, let us take a data set distributed with R: InsectSprays. Six insecticides were tested in field conditions, the observed response was the number of insects. Each insecticide was tested 12 times, thus there are 72 observations. We will not consider here the graphical exploration of the data, but will focus on a simple analysis of variance of the response with respect to the insecticide. After loading the data in memory with the function data, the analysis is performed after a square-root transformation of the response:

```
> data(InsectSprays)
> aov.spray <- aov(sqrt(count) ~ spray, data = InsectSprays)
```

The main (and mandatory) argument of aov is a formula which specifies the response on the left-hand side of the tilde symbol ~ and the predictor on the right-hand side. The option data = InsectSprays specifies that the variables must be found in the data frame InsectSprays. This syntax is equivalent to:

```
> aov.spray <- aov(sqrt(InsectSprays$count) ~ InsectSprays$spray)
```

or still (if we know the column numbers of the variables):

```
> aov.spray <- aov(sqrt(InsectSprays[, 1]) ~ InsectSprays[, 2])
```

The first syntax is to be preferred since it is clearer.

The results are not displayed since they are assigned to an object called `aov.spray`. We will then used some functions to extract the results, for example `print` to display a brief summary of the analysis (mostly the estimated parameters) and `summary` to display more details (included the statistical tests):

```
> aov.spray
Call:
   aov(formula = sqrt(count) ~ spray, data = InsectSprays)

Terms:
                   spray Residuals
Sum of Squares  88.43787  26.05798
Deg. of Freedom        5        66

Residual standard error: 0.6283453
Estimated effects may be unbalanced
> summary(aov.spray)
            Df Sum Sq Mean Sq F value    Pr(>F)
spray        5 88.438  17.688  44.799 < 2.2e-16 ***
Residuals   66 26.058   0.395
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

We may remind that typing the name of the object as a command is similar to the command `print(aov.spray)`. A graphical representation of the results can be done with `plot()` or `termplot()`. Before typing `plot(aov.spray)` we will divide the graphics into four parts so that the four diagnostics plots will be done on the same graph. The commands are:

```
> opar <- par()
> par(mfcol = c(2, 2))
> plot(aov.spray)
> par(opar)
> termplot(aov.spray, se=TRUE, partial.resid=TRUE, rug=TRUE)
```

and the resulting graphics are on Figs. 13 and 14.

5.2 Formulae

Formulae are a key element in statistical analyses with R: the notation used is the same for (almost) all functions. A formula is typically of the form `y ~` *`model`* where `y` is the analysed response and *`model`* is a set of terms for which some parameters are to be estimated. These terms are separated with arithmetic symbols but they have here a particular meaning.

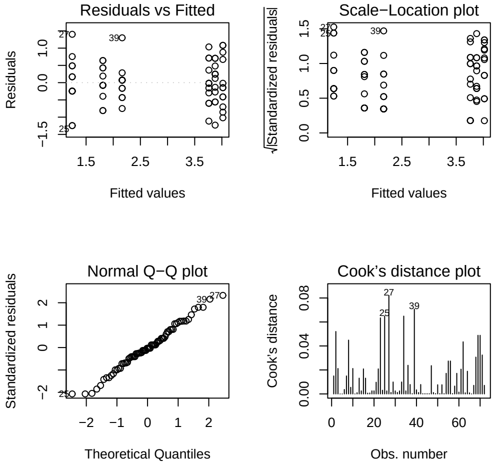


Figure 13: Graphical representation of the results from the function `aov` with `plot()`.

`a+b` additive effects of `a` and of `b`

`X` if `X` is a matrix, this specifies an additive effect of each of its columns, i.e. `X[,1]+X[,2]+...+X[,ncol(X)]`; some of the columns may be selected with numeric indices (e.g., `X[,2:4]`)

`a:b` interactive effect between `a` and `b`

`a*b` additive and interactive effects (identical to `a+b+a:b`)

`poly(a, n)` polynomials of `a` up to degree n

`^n` includes all interactions up to level n, i.e. `(a+b+c)^2` is identical to `a+b+c+a:b+a:c+b:c`

`b %in% a` the effects of `b` are nested in `a` (identical to `a+a:b`, or `a/b`)

`-b` removes the effect of `b`, for example: `(a+b+c)^2-a:b` is identical to `a+b+c+a:c+b:c`

`-1` `y~x-1` is a regression through the origin (id. for `y~x+0` or `0+y~x`)

`1` `y~1` fits a model with no effects (only the intercept)

`offset(...)` adds an effect to the model without estimating any parameter (e.g., `offset(3*x)`)

We see that the arithmetic operators of R have in a formula a different meaning than they have in expressions. For example, the formula `y~x1+x2` defines the model $y = \beta_1 x_1 + \beta_2 x_2 + \alpha$, and not (if the operator `+` would have its usual meaning) $y = \beta(x_1 + x_2) + \alpha$. To include arithmetic operations in a formula, we can use the function `I`: the formula `y~I(x1+x2)` defines the model $y = \beta(x_1 + x_2) + \alpha$. Similarly, to define the model $y = \beta_1 x + \beta_2 x^2 + \alpha$, we will use the formula `y ~ poly(x, 2)` (and not `y ~ x + x^2`). However, it is

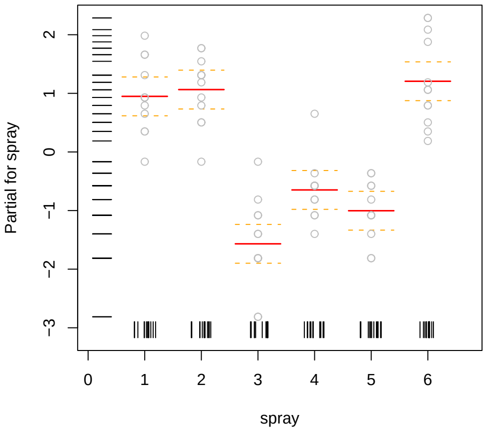


Figure 14: Graphical representation of the results from the function `aov` with `termplot()`.

possible to include a function in a formula in order to transform a variable as seen above with the insect sprays analysis of variance.

For analyses of variance, `aov()` accepts a particular syntax to define random effects. For instance, `y ~ a + Error(b)` means the additive effects of the fixed term `a` and the random one `b`.

5.3 Generic functions

We remember that R's functions act with respect to the attributes of the objects possibly passed as arguments. The *class* is an attribute deserving some attention here. It is very common that the R statistical functions return an object of class with the same name (e.g., `aov` returns an object of class `"aov"`, `lm` returns one of class `"lm"`). The functions that we can use subsequently to extract the results will act specifically with respect to the class of the object. These functions are called *generic*.

For instance, the function which is most often used to extract results from analyses is `summary` which displays detailed results. Whether the object given as argument is of class `"lm"` (linear model) or `"aov"` (analysis of variance), it sounds obvious that the information to display will not be the same. The advantage of generic functions is that the syntax is the same in all cases.

An object containing the results of an analysis is generally a list, and the way it is displayed is determined by its class. We have already seen this notion that the action of a function depends on the kind of object given as argument. It is a general feature of R[17]. The following table gives the main generic func-

[17] There are more than 100 generic functions in R.

tions which can be used to extract information from objects resulting from an analysis. The typical usage of these functions is:

```
> mod <- lm(y ~ x)
> df.residual(mod)
[1] 8
```

`print`	returns a brief summary
`summary`	returns a detailed summary
`df.residual`	returns the number of residual degrees of freedom
`coef`	returns the estimated coefficients (sometimes with their standard-errors)
`residuals`	returns the residuals
`deviance`	returns the deviance
`fitted`	returns the fitted values
`logLik`	computes the logarithm of the likelihood and the number of parameters
`AIC`	computes the Akaike information criterion or AIC (depends on `logLik()`)

A function like `aov` or `lm` returns a list with its different elements corresponding to the results of the analysis. If we take our example of an analysis of variance with the data `InsectSprays`, we can look at the structure of the object returned by `aov`:

```
> str(aov.spray, max.level = -1)
List of 13
 - attr(*, "class")= chr [1:2] "aov" "lm"
```

Another way to look at this structure is to display the names of the object:

```
> names(aov.spray)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "contrasts"
[10] "xlevels"       "call"          "terms"
[13] "model"
```

The elements can then be extracted as we have already seen:

```
> aov.spray$coefficients
(Intercept)      sprayB      sprayC      sprayD
  3.7606784   0.1159530  -2.5158217  -1.5963245
     sprayE      sprayF
 -1.9512174   0.2579388
```

`summary()` also creates a list which, in the case of `aov()`, is simply a table of tests:

```
> str(summary(aov.spray))
List of 1
 $ :Classes anova  and ‘data.frame’:    2 obs. of  5 variables:
  ..$ Df     : num [1:2] 5 66
  ..$ Sum Sq : num [1:2] 88.4 26.1
  ..$ Mean Sq: num [1:2] 17.688  0.395
  ..$ F value: num [1:2] 44.8   NA
  ..$ Pr(>F) : num [1:2] 0 NA
 - attr(*, "class")= chr [1:2] "summary.aov" "listof"
> names(summary(aov.spray))
NULL
```

Generic functions do not generally perform any action on objects: they call the appropriate function with respect to the class of the argument. A function called by a generic is a *method* in R's jargon. Schematically, a method is constructed as ***generic.cls***, where ***cls*** is the class of the object. For instance, in the case of `summary`, we can display the corresponding methods:

```
> apropos("^summary")
 [1] "summary"               "summary.aov"
 [3] "summary.aovlist"       "summary.connection"
 [5] "summary.data.frame"    "summary.default"
 [7] "summary.factor"        "summary.glm"
 [9] "summary.glm.null"      "summary.infl"
[11] "summary.lm"            "summary.lm.null"
[13] "summary.manova"        "summary.matrix"
[15] "summary.mlm"           "summary.packageStatus"
[17] "summary.POSIXct"       "summary.POSIXlt"
[19] "summary.table"
```

We can see the difference for this generic in the case of a linear regression, compared to an analysis of variance, with a small simulated example:

```
> x <- y <- rnorm(5)
> lm.spray <- lm(y ~ x)
> names(lm.spray)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
> names(summary(lm.spray))
 [1] "call"          "terms"         "residuals"
 [4] "coefficients"  "sigma"         "df"
 [7] "r.squared"     "adj.r.squared" "fstatistic"
[10] "cov.unscaled"
```

The following table shows some generic functions that do supplementary analyses from an object resulting from an analysis, the main argument being this latter object, but in some cases a further argument is necessary like for `predict` or `update`.

`add1`	tests successively all the terms that can be added to a model
`drop1`	tests successively all the terms that can be removed from a model
`step`	selects a model with AIC (calls `add1` and `drop1`)
`anova`	computes a table of analysis of variance or deviance for one or several models
`predict`	computes the predicted values for new data from a fitted model
`update`	re-fits a model with a new formula or new data

There are also various utilities functions that extract information from a model object or a formula, such as `alias` which finds the linearly dependent terms in a linear model specified by a formula.

Finally, there are, of course, graphical functions such as `plot` which displays various diagnostics, or `termplot` (see the above example), though this latter function is not generic but calls `predict`.

5.4 Packages

The following table lists the *standard* packages which are distributed with a base installation of R. Some of them are loaded in memory when R starts; this can be displayed with the function `search`:

```
> search()
[1] ".GlobalEnv"        "package:methods"
[3] "package:stats"     "package:graphics"
[5] "package:grDevices" "package:utils"
[7] "package:datasets"  "Autoloads"
[9] "package:base"
```

The other packages may be used after being loaded:

```
> library(grid)
```

The list of the functions in a package can be displayed with:

```
> library(help = grid)
```

or by browsing the help in html format. The information relative to each function can be accessed as previously seen (p. 7).

Package	Description
base	base R functions
datasets	base R datasets
grDevices	graphics devices for base and grid graphics
graphics	base graphics
grid	grid graphics
methods	definition of methods and classes for R objects and programming tools
splines	regression spline functions and classes
stats	statistical functions
stats4	statistical functions using S4 classes
tcltk	functions to interface R with Tcl/Tk graphical user interface elements
tools	tools for package development and administration
utils	R utility functions

Many *contributed* packages add to the list of statistical methods available in R. They are distributed separately, and must be installed and loaded in R. A complete list of the contributed packages, with descriptions, is on the CRAN Web site[18]. Several of these packages are *recommanded* since they cover statistical methods often used in data analysis. The recommended packages are often distributed with a base installation of R. They are briefly described in the following table.

Package	Description
boot	resampling and bootstraping methods
class	classification methods
cluster	clustering methods
foreign	functions for reading data stored in various formats (S3, Stata, SAS, Minitab, SPSS, Epi Info)
KernSmooth	methods for kernel smoothing and density estimation (including bivariate kernels)
lattice	Lattice (Trellis) graphics
MASS	contains many functions, tools and data sets from the libraries of “Modern Applied Statistics with S” by Venables & Ripley
mgcv	generalized additive models
nlme	linear and non-linear mixed-effects models
nnet	neural networks and multinomial log-linear models
rpart	recursive partitioning
spatial	spatial analyses (“kriging”, spatial covariance, . . .)
survival	survival analyses

[18] http://cran.r-project.org/src/contrib/PACKAGES.html

There are two other main repositories of R packages: the Omegahat Project for Statistical Computing[19] which focuses on web-based applications and interfaces between softwares and languages, and the Bioconductor Project[20] specialized in bioinformatic applications (particularly for the analysis of micro-array data).

The procedure to install a package depends on the operating system and whether R was installed from the sources or pre-compiled binaries. In the latter situation, it is recommended to use the pre-compiled packages available on CRAN's site. Under Windows, the binary Rgui.exe has a menu "Packages" allowing to install packages via internet from the CRAN Web site, or from zipped files on the local disk.

If R was compiled, a package can be installed from its sources which are distributed as a '.tar.gz' file. For instance, if we want to install the package gee, we will first download the file gee_4.13-6.tar.gz (the number 4.13-6 indicates the version of the package; generally only one version is available on CRAN). We will then type from the system (and not in R) the command:

```
R CMD INSTALL gee_4.13-6.tar.gz
```

There are several useful functions to manage packages such as `installed.packages`, `CRAN.packages`, or `download.packages`. It is also useful to type regularly the command:

```
> update.packages()
```

which checks the versions of the packages installed against those available on CRAN (this command can be called from the menu "Packages" under Windows). The user can then update the packages with more recent versions than those installed on the computer.

[19] http://www.omegahat.org/R/

[20] http://www.bioconductor.org/

6 Programming with R in pratice

Now that we have done an overview of R's functionalities, let us return to the language and programming. We will see a few simple ideas likely to be used in practice.

6.1 Loops and vectorization

An advantage of R compared to softwares with pull-down menus is the possibility to program simply a series of analyses which will be executed successively. This is common to any computer language, but R has some particular features which make programming easier for non-specialists.

Like other languages, R has some *control structures* which are not dissimilar to those of the C language. Suppose we have a vector x, and for each element of x with the value b, we want to give the value 0 to another variable y, otherwise 1. We first create a vector y of the same length than x:

```
y <- numeric(length(x))
for (i in 1:length(x)) if (x[i] == b) y[i] <- 0 else y[i] <- 1
```

Several instructions can be executed if they are placed within braces:

```
for (i in 1:length(x)) {
    y[i] <- 0
    ...
}

if (x[i] == b) {
    y[i] <- 0
    ...
}
```

Another possible situation is to execute an instruction as long as a condition is true:

```
while (myfun > minimum) {
    ...
}
```

However, loops and control structures can be avoided in most situations thanks to a feature of R: *vectorization.* Vectorization makes loops implicit in expression, and we have seen many cases. Let us consider the addition of two vectors:

```
> z <- x + y
```

This addition could be written with a loop, as this is done in most languages:

```
> z <- numeric(length(x))
> for (i in 1:length(z)) z[i] <- x[i] + y[i]
```

In this case, it is necessary to create the vector `z` beforehand because of the use of the indexing system. We realize that this explicit loop will work only if `x` and `y` are of the same length: it must be changed if this is not true, whereas the first expression will work in all situations.

The conditional executions (`if ... else`) can be avoided with the use of the logical indexing; coming back to the above example:

```
> y[x == b] <- 0
> y[x != b] <- 1
```

In addition to being simpler, vectorized expressions are computationally more efficient, particularly with large quantities of data.

There are also several functions of the type 'apply' which avoids writing loops. `apply` acts on the rows and/or columns of a matrix, its syntax is `apply(X, MARGIN, FUN, ...)`, where `X` is a matrix, `MARGIN` indicates whether to consider the rows (`1`), the columns (`2`), or both (`c(1, 2)`), `FUN` is a function (or an operator, but in this case it must be specified within brackets) to apply, and `...` are possible optional arguments for `FUN`. A simple example follows.

```
> x <- rnorm(10, -5, 0.1)
> y <- rnorm(10, 5, 2)
> X <- cbind(x, y)  # the columns of X keep the names "x" and "y"
> apply(X, 2, mean)
        x         y
-4.975132  4.932979
> apply(X, 2, sd)
        x         y
0.0755153 2.1388071
```

`lapply()` acts on a list: its syntax is similar to `apply` and it returns a list.

```
> forms <- list(y ~ x, y ~ poly(x, 2))
> lapply(forms, lm)
[[1]]

Call:
FUN(formula = X[[1]])

Coefficients:
```

```
(Intercept)            x
     31.683        5.377


[[2]]

Call:
FUN(formula = X[[2]])

Coefficients:
(Intercept)  poly(x, 2)1  poly(x, 2)2
     4.9330       1.2181      -0.6037
```

sapply() is a flexible variant of lapply() which can take a vector or a matrix as main argument, and returns its results in a more user-friendly form, generally as a table.

6.2 Writing a program in R

Typically, an R program is written in a file saved in ASCII format and named with the extension '.R'. A typical situation where a program is useful is when one wants to do the same tasks several times. In our first example, we want to do the same plot for three different species of birds, the data being in three distinct files. We will proceed step by step, and see different ways to program this very simple problem.

First, let us make our program in the most intuitive way by executing successively the needed commands, taking care to partition the graphical device beforehand.

```
layout(matrix(1:3, 3, 1))           # partition the graphics
data <- read.table("Swal.dat")      # read the data
plot(data$V1, data$V2, type="l")
title("swallow")                    # add a title
data <- read.table("Wren.dat")
plot(data$V1, data$V2, type="l")
title("wren")
data <- read.table("Dunn.dat")
plot(data$V1, data$V2, type="l")
title("dunnock")
```

The character '#' is used to add comments in a program: R then goes to the next line.

The problem of this first program is that it may become quite long if we want to add other species. Moreover, some commands are executed several times, thus they can be grouped together and executed after changing some arguments. The strategy used here is to put these arguments in vectors of mode character, and then use the indexing to access these different values.

```
layout(matrix(1:3, 3, 1))               # partition the graphics
species <- c("swallow", "wren", "dunnock")
file <- c("Swal.dat" , "Wren.dat", "Dunn.dat")
for(i in 1:length(species)) {
    data <- read.table(file[i])         # read the data
    plot(data$V1, data$V2, type="l")
    title(species[i])                   # add a title
}
```

Note that there are no double quotes around `file[i]` in `read.table()` since this argument is of mode character.

Our program is now more compact. It is easier to add other species since the vectors containing the species and file names are at the beginning of the program.

The above programs will work correctly if the data files '.dat' are located in the working directory of R, otherwise the user must either change the working directory, or specifiy the path in the program (for example: `file <- "/home/paradis/data/Swal.dat"`). If the program is written in the file Mybirds.R, it will be called by typing:

```
> source("Mybirds.R")
```

Like for any input from a file, it is necessary to give the path to access the file if it is not in the working directory.

6.3 Writing your own functions

We have seen that most of R's work is done with functions which arguments are given within parentheses. Users can write their own functions, and these will have exactly the same properties than other functions in R.

Writing your own functions allows an efficient, flexible, and rational use of R. Let us come back to our example of reading some data followed by plotting a graph. If we want to do this operation in different situations, it may be a good idea to write a function:

```
myfun <- function(S, F)
{
    data <- read.table(F)
    plot(data$V1, data$V2, type="l")
    title(S)
}
```

To be executed, this function must be loaded in memory, and this can be done in several ways. The lines of the function can be typed directly on the keyboard, like any other command, or copied and pasted from an editor. If the function has been saved in a text file, it can be loaded with `source()`

like another program. If the user wants some functions to be loaded each time when R starts, they can be saved in a workspace `.RData` which will be loaded in memory if it is in the working directory. Another possibility is to configure the file '.Rprofile' or 'Rprofile' (see `?Startup` for details). Finally, it is possible to create a package, but this will not be discussed here (see the manual "Writing R Extensions").

Once the function is loaded, we will be able with a single command to read the data and plot the graph, for instance with `myfun("swallow", "Swal.dat")`. Thus, we have now a third version of our program:

```
layout(matrix(1:3, 3, 1))
myfun("swallow", "Swal.dat")
myfun("wren", "Wrenn.dat")
myfun("dunnock", "Dunn.dat")
```

We may also use `sapply()` leading to a fourth version of our program:

```
layout(matrix(1:3, 3, 1))
species <- c("swallow", "wren", "dunnock")
file <- c("Swal.dat" , "Wren.dat", "Dunn.dat")
sapply(species, myfun, file)
```

In R, it is not necessary to declare the variables used within a function. When a function is executed, R uses a rule called *lexical scoping* to decide whether an object is local to the function, or global. To understand this mechanism, let us consider the very simple function below:

```
> foo <- function() print(x)
> x <- 1
> foo()
[1] 1
```

The name `x` is not used to create an object within `foo()`, so R will seek in the *enclosing* environment if there is an object called `x`, and will print its value (otherwise, a message error is displayed, and the execution is halted).

If `x` is used as the name of an object within our function, the value of `x` in the global environment is not used.

```
> x <- 1
> foo2 <- function() { x <- 2; print(x) }
> foo2()
[1] 2
> x
[1] 1
```

This time `print()` uses the object `x` that is defined within its environment, that is the environment of `foo2`.

The word "*enclosing*" above is important. In our two example functions, there are *two* environments: the global one, and the one of the function `foo` or `foo2`. If there are three or more nested environments, the search for the objects is made progressively from a given environment to the enclosing one, and so on, up to the global one.

There are two ways to specify arguments to a function: by their positions or by their names (also called *tagged arguments*). For example, let us consider a function with three arguments:

```
foo <- function(arg1, arg2, arg3) {...}
```

`foo()` can be executed without using the names `arg1`, ..., if the corresponding objects are placed in the correct position, for instance: `foo(x, y, z)`. However, the position has no importance if the names of the arguments are used, e.g. `foo(arg3 = z, arg2 = y, arg1 = x)`. Another feature of R's functions is the possibility to use default values in their definition. For instance:

```
foo <- function(arg1, arg2 = 5, arg3 = FALSE) {...}
```

The commands `foo(x)`, `foo(x, 5, FALSE)`, and `foo(x, arg3 = FALSE)` will have exactly the same result. The use of default values in a function definition is very useful, particularly when used with tagged arguments (i.e. to change only one default value such as `foo(x, arg3 = TRUE)`.

To conclude this section, let us see another example which is not purely statistical, but it illustrates the flexibility of R. Consider we wish to study the behaviour of a non-linear model: Ricker's model defined by:

$$N_{t+1} = N_t \exp\left[r\left(1 - \frac{N_t}{K}\right)\right]$$

This model is widely used in population dynamics, particularly of fish. We want, using a function, to simulate this model with respect to the growth rate r and the initial number in the population N_0 (the carrying capacity K is often taken equal to 1 and this value will be taken as default); the results will be displayed as a plot of numbers with respect to time. We will add an option to allow the user to display only the numbers in the last few time steps (by default all results will be plotted). The function below can do this numerical analysis of Ricker's model.

```
ricker <- function(nzero, r, K=1, time=100, from=0, to=time)
{
    N <- numeric(time+1)
    N[1] <- nzero
    for (i in 1:time) N[i+1] <- N[i]*exp(r*(1 - N[i]/K))
    Time <- 0:time
    plot(Time, N, type="l", xlim=c(from, to))
}
```

Try it yourself with:

```
> layout(matrix(1:3, 3, 1))
> ricker(0.1, 1); title("r = 1")
> ricker(0.1, 2); title("r = 2")
> ricker(0.1, 3); title("r = 3")
```

7 Literature on R

Manuals. Several manuals are distributed with R in R_HOME/doc/manual/:

- *An Introduction to R* [R-intro.pdf],
- *R Installation and Administration* [R-admin.pdf],
- *R Data Import/Export* [R-data.pdf],
- *Writing R Extensions* [R-exts.pdf],
- *R Language Definition* [R-lang.pdf].

The files may be in different formats (pdf, html, texi, ...) depending on the type of installation.

FAQ. R is also distributed with an FAQ (*Frequently Asked Questions*) localized in the directory R_HOME/doc/html/. A version of the R-FAQ is regularly updated on CRAN's Web site:

http://cran.r-project.org/doc/FAQ/R-FAQ.html

On-line resources. The CRAN Web site hosts several documents, bibliographic resources, and links to other sites. There are also a list of publications (books and articles) about R or statistical methods[21] and some documents and tutorials written by R's users[22].

Mailing lists. There are four discussion lists on R; to subscribe, send a message, or read the archives see: http://www.R-project.org/mail.html.

The general discussion list 'r-help' is an interesting source of information for the users of R (the three other lists are dedicated to annoucements of new versions, and for developers). Many users have sent to 'r-help' functions or programs which can be found in the archives. If a problem is encountered with R, it is thus important to proceed in the following order before sending a message to 'r-help':

1. read carefully the on-line help (possibly using the search engine);
2. read the R-FAQ;
3. search the archives of 'r-help' at the above address, or by using one of the search engines developed on some Web sites[23];
4. read the "posting guide"[24] before sending your question(s).

[21] http://www.R-project.org/doc/bib/R-publications.html

[22] http://cran.r-project.org/other-docs.html

[23] The addresses of these sites are listed at http://cran.r-project.org/search.html

[24] http://www.r-project.org/posting-guide.html

R News. The electronic journal *R News* aims to fill the gap between the electronic discussion lists and traditional scientific publications. The first issue was published on January 2001[25].

Citing R in a publication. Finally, if you mention R in a publication, you must cite the following reference:

> R Development Core Team (2005). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL: http://www.R-project.org.

[25] http://cran.r-project.org/doc/Rnews/

High-throughput sequence analysis with *R* and *Bioconductor*

Marc Carlson, Valerie Obenchain, Hervé Pagès, Paul Shannon, Daniel Tenenbaum, Martin Morgan[*]

June 2012

Contents

[*] mcarlson,vobencha,hpages,pshannon,dtenenba,mtmorgan@fhcrc.org

Table 1: Tentative schedule.

R / *Bioconductor* for Sequence Analysis

> *R* data types & functions; help; objects; essential packages, efficient programming (Section 2). Working with ranges, strings, reads and alignments. Quality assessment (Sections 3, 4).

RNA-Seq

> Differential representation, gene set enrichment, annotation, exon use (Sections 5, 7).

ChIP-Seq

> Peak calling (3rd party); collated experiments; motifs; annotation (Section 6, 7).

Variant Annotation

> Common work flows; variants in and around genes, amino acid and coding consequences (Sections 8).

1 Introduction

1.1 This workshop

This workshop introduces use of *R* and *Bioconductor* for analysis of high-throughput sequence data. The workshop is structured as a series of short remarks followed by group exercises. The exercises explore the diversity of tasks for which *R* / *Bioconductor* are appropriate, but are far from comprehensive.

The goals of the workshop are to: (1) develop familiarity with *R* / *Bioconductor* software for high-throughput analysis; (2) expose key statistical issues in the analysis of sequence data; and (3) provide inspiration and a framework for further independent exploration. An approximate schedule is shown in Table 1.

1.2 *Bioconductor*

Bioconductor is a collection of *R* packages for the analysis and comprehension of high-throughput genomic data. *Bioconductor* started more than 10 years ago. It gained credibility for its statistically rigorous approach to microarray pre-preprocessing and analysis of designed experiments, and integrative and reproducible approaches to bioinformatic tasks. There are now more than 500 *Bioconductor* packages for expression and other microarrays, sequence analysis, flow cytometry, imaging, and other domains. The *Bioconductor* web site provides installation, package repository, help, and other documentation.

The *Bioconductor* web site is at `bioconductor.org`. Features include:

- Introductory work flows.
- A manifest of *Bioconductor* packages arranged in BiocViews.
- Annotation (data bases of relevant genomic information, e.g., Entrez gene ids in model organisms, KEGG pathways) and experiment data (containing relatively comprehensive data sets and their analysis) packages.
- Mailing lists, including searchable archives, as the primary source of help.
- Course and conference information, including extensive reference material.
- General information about the project.

- Package developer resources, including guidelines for creating and submitting new packages.

Exercise 1
Scavenger hunt. Spend five minutes tracking down the following information.

a. *From the Bioconductor web site, instructions for installing or updating Bioconductor packages.*

b. *A list of all packages in the current release of Bioconductor.*

c. *The URL of the Bioconductor mailing list subscription page.*

Solution: Possible solutions from the *Bioconductor* web site are, e.g., `http://bioconductor.org/install/` (installation instructions), `http://bioconductor.org/packages/release/bioc/` (current software packages), and `http://bioconductor.org/help/mailing-list/` (mailing lists).

1.3 High-throughput sequence analysis

Recent technological developments introduce high-throughput sequencing approaches. A variety of experimental protocols and analysis work flows address gene expression, regulation, and encoding of genetic variants. Experimental protocols produce a large number (millions per sample) of short (e.g., 35-100, single or paired-end) nucleotide sequences. These are aligned to a reference or other genome. Analysis work flows use the alignments to infer levels of gene expression (RNA-seq), binding of regulatory elements to genomic locations (ChIP-seq), or prevalence of structural variants (e.g., SNPs, short indels, large-scale genomic rearrangements). Sample sizes range from minimal replication (e.g,. 2 samples per treatment group) to thousands of individuals.

1.4 Statistical programming

Many academic and commercial software products are available; why would one use *R* and *Bioconductor*? One answer is to ask about the demands high-throughput genomic data places on effective computational biology software.

Effective computational biology software High-throughput questions make use of large data sets. This applies both to the primary data (microarray expression values, sequenced reads, etc.) and also to the annotations on those data (coordinates of genes and features such as exons or regulatory regions; participation in biological pathways, etc.). Large data sets place demands on our tools that preclude some standard approaches, such as spread sheets. Likewise, intricate relationships between data and annotation, and the diversity of research questions, require flexibility typical of a programming language rather than a narrowly-enabled graphical user interface.

Analysis of high-throughput data is necessarily statistical. The volume of data requires that it be appropriately summarized before any sort of comprehension is possible. The data are produced by advanced technologies, and these introduce artifacts (e.g., probe-specific bias in microarrays; sequence or base calling bias in RNA-seq experiments) that need to be accommodated to avoid incorrect or inefficient inference. Data sets typically derive from designed experiments, requiring a statistical approach both to account for the design and to correctly address the large number of observed values (e.g., gene expression or sequence tag counts) and small number of samples accessible in typical experiments.

Research needs to be reproducible. Reproducibility is both an ideal of the scientific method, and a pragmatic requirement. The latter comes from the long-term and multi-participant nature of contemporary science. An analysis will be performed for the initial experiment, revisited again during manuscript preparation, and revisited during reviews or in determining next steps. Likewise, analyses typically involve a team of individuals with diverse domains of expertise. Effective collaborations result when it is easy to reproduce, perhaps with minor modifications, an existing result, and when sophisticated statistical or bioinformatic analyses can be effectively conveyed to other group members.

Science moves very quickly. This is driven by the novel questions that are the hallmark of discovery, and by technological innovation and accessibility. Rapidity of scientific development places significant burdens on software, which must also move quickly. Effective software cannot be too polished, because that requires that the correct analyses are 'known' and that significant resources of time and money have been invested in developing the software; this implies software that is tracking the trailing edge of innovation. On the other hand, leading-edge software cannot be too idiosyncratic; it must be usable by a wider audience than the creator of the software, and fit in with other software relevant to the analysis.

Effective software must be accessible. Affordability is one aspect of accessibility. Another is transparent implementation, where the novel software is sufficiently documented and source code accessible enough for the assumptions, approaches, practical implementation decisions, and inevitable coding errors to be assessed by other skilled practitioners. A final aspect of affordability is that the software is actually usable. This is achieved through adequate documentation, support forums, and training opportunities.

***Bioconductor* as effective computational biology software** What features of *R* and *Bioconductor* contribute to its effectiveness as a software tool?

Bioconductor is well suited to handle extensive data and annotation. *Bioconductor* 'classes' represent high-throughput data and their annotation in an integrated way. *Bioconductor* methods use advanced programming techniques or *R* resources (such as transparent data base or network access) to minimize memory requirements and integrate with diverse resources. Classes and methods coordinate complicated data sets with extensive annotation. Nonetheless, the basic model for object manipulation in *R* involves vectorized in-memory representations. For this reason, particular programming paradigms (e.g., block processing of data streams; explicit parallelism) or hardware resources (e.g., large-memory computers) are sometimes required when dealing with extensive data.

R is ideally suited to addressing the statistical challenges of high-throughput data. Three examples include the development of the 'RMA' and other normalization algorithm for microarray pre-processing, use of moderated t-statistics for assessing microarray differential expression, and development of negative binomial approaches to estimating dispersion read counts necessary for appropriate analysis of RNAseq designed experiments.

Many of the 'old school' aspects of *R* and *Bioconductor* facilitate reproducible research. An analysis is often represented as a text-based script. Reproducing the analysis involves re-running the script; adjusting how the analysis is performed involves simple text-editing tasks. Beyond this, *R* has the notion of a 'vignette', which represents an analysis as a LaTeX document with embedded *R* commands. The *R* commands are evaluated when the document is built, thus reproducing the analysis. The use of LaTeX means that the symbolic manipulations in the script are augmented with textual explanations and justifications for the approach taken; these include graphical and tabular summaries at appropriate places in the analysis. *R* includes facilities for reporting the exact version of *R* and associated packages used in an analysis so that, if needed, discrepancies between software versions can be tracked down and their importance evaluated. While users often think of *R* packages as providing new functionality, packages are also used to enhance reproducibility by encapsulating a single analysis. The package can contain data sets, vignette(s) describing the analysis, *R* functions that might have been written, scripts for key data processing stages, and documentation (via standard *R* help mechanisms) of what the functions, data, and packages are about.

The *Bioconductor* project adopts practices that facilitate reproducibility. Versions of *R* and *Bioconductor* are released twice each year. Each *Bioconductor* release is the result of development, in a separate branch, during the previous six months. The release is built daily against the corresponding version of *R* on Linux, Mac, and Windows platforms, with an extensive suite of tests performed. The `biocLite` function ensures that each release of *R* uses the corresponding *Bioconductor* packages. The user thus has access to stable and tested package versions. *R* and *Bioconductor* are effective tools for reproducible research.

R and *Bioconductor* exist on the leading portion of the software life cycle. Contributors are primarily from academic institutions, and are directly involved in novel research activities. New developments are made available in a familiar format, i.e., the *R* language, packaging, and build systems. The rich set of facilities in *R* (e.g., for advanced statistical analysis or visualization) and the extensive resources in *Bioconductor* (e.g., for annotation using third-party data such as Biomart or UCSC genome browser tracks) mean that innovations can be directly incorporated into existing work flows. The 'development' branches of *R* and *Bioconductor* provide an environment where contributors can explore new approaches without alienating their user base.

R and *Bioconductor* also fair well in terms of accessibility. The software is freely available. The source code is easily and fully accessible for critical evaluation. The *R* packaging and check system requires that all functions are documented. *Bioconductor* requires that each package contain vignettes to illustrate the use of the software. There are very active *R* and *Bioconductor* mailing lists for immediate support, and regular training and conference activities for professional development.

Table 2: Selected *Bioconductor* packages for high-throughput sequence analysis.

Concept	Packages
Data representation	*IRanges*, *GenomicRanges*, *GenomicFeatures*, *Biostrings*, *BSgenome*, *girafe*.
Input / output	*ShortRead* (fastq), *Rsamtools* (bam), *rtracklayer* (gff, wig, bed), *VariantAnnotation* (vcf), *R453Plus1Toolbox* (454).
Annotation	*GenomicFeatures*, *ChIPpeakAnno*, *VariantAnnotation*.
Alignment	*Rsubread*, *Biostrings*.
Visualization	*ggbio*, *Gviz*.
Quality assessment	*qrqc*, *seqbias*, *ReQON*, *htSeqTools*, *TEQC*, *Rolexa*, *ShortRead*.
RNA-seq	*BitSeq*, *cqn*, *cummeRbund*, *DESeq*, *DEXSeq*, *EDASeq*, *edgeR*, *gage*, *goseq*, *iASeq*, *tweeDEseq*.
ChIP-seq, etc.	*BayesPeak*, *baySeq*, *ChIPpeakAnno*, *chipseq*, *ChIPseqR*, *ChIPsim*, *CSAR*, *DiffBind*, *MEDIPS*, *mosaics*, *NarrowPeaks*, *nucleR*, *PICS*, *PING*, *REDseq*, *Repitools*, *TSSi*.
Motifs	*BCRANK*, *cosmo*, *cosmoGUI*, *MotIV*, *seqLogo*, *rGADEM*.
3C, etc.	*HiTC*, *r3Cseq*.
Copy number	*cn.mops*, *CNAnorm*, *exomeCopy*, *seqmentSeq*.
Microbiome	*phyloseq*, *DirichletMultinomial*, *clstutils*, *manta*, *mcaGUI*.
Work flows	*ArrayExpressHTS*, *Genominator*, *easyRNASeq*, *oneChannelGUI*, *rnaSeqMap*.
Database	*SRAdb*.

1.5 *Bioconductor* for high-throughput sequence analysis

Table 2 enumerates many of the packages available for sequence analysis. The table includes packages for representing sequence-related data (e.g., *GenomicRanges*, *Biostrings*), as well as domain-specific analysis such as RNA-seq (e.g., *edgeR*, *DEXSeq*), ChIP-seq (e.g,. *ChIPpeakAnno*, *DiffBind*), and SNPs and copy number variation (e.g., *genoset*, *ggtools*, *VariantAnnotation*).

1.6 Resources

Dalgaard [4] provides an introduction to statistical analysis with *R*. Kabaloff [9] provides a broad survey of *R*. Matloff [15] introduces *R* programming concepts. Chambers [3] provides more advanced insights into *R*. Gentleman [5] emphasizes use of *R* for bioinformatic programming tasks. The *R* web site enumerates additional publications from the user community.

The RStudio environment provides a nice, cross-platform environment for working in *R*.

2 *R*

R is an open-source statistical programming language. It is used to manipulate data, to perform statistical analyses, and to present graphical and other results. *R* consists of a core language, additional 'packages' distributed with the *R* language, and a very large number of packages contributed by the broader community. Packages add specific functionality to an *R* installation. *R* has become the primary language of academic statistical analyses, and is widely used in diverse areas of research, government, and industry.

R has several unique features. It has a surprisingly 'old school' interface: users type commands into a console; scripts in plain text represent work flows; tools other than *R* are used for editing and other tasks. *R* is a flexible programming language, so while one person might use functions provided by *R* to accomplish advanced analytic tasks, another might implement their own functions for novel data types. As a programming language, *R* adopts syntax and grammar that differ from many other languages: objects in *R* are 'vectors', and functions are 'vectorized' to operate on all elements of the object; *R* objects have 'copy on change' and 'pass by value' semantics, reducing unexpected consequences for users at the expense of less efficient memory use; common paradigms in other languages, such as the 'for' loop, are encountered much less commonly in *R*. Many authors contribute to *R*, so there can be a frustrating inconsistency of documentation and interface. *R* grew up in the academic community, so authors have not shied away from trying new approaches. Common statistical analyses are very well-developed.

2.1 *R* data types

Opening an *R* session results in a prompt. The user types instructions at the prompt. Here is an example:

```
> ## assign values 5, 4, 3, 2, 1 to variable 'x'
> x <- c(5, 4, 3, 2, 1)
> x
```

```
[1] 5 4 3 2 1
```

The first line starts with a `#` to represent a comment; the line is ignored by *R*. The next line creates a variable `x`. The variable is assigned (using `<-`, we could have used `=` almost interchangeably) a value. The value assigned is the result of a call to the `c` function. That it is a function call is indicated by the symbol named followed by parentheses, `c()`. The `c` function takes zero or more arguments, and returns a vector. The vector is the value assigned to `x`. *R* responds to this line with a new prompt, ready for the next input. The next line asks *R* to display the value of the variable `x`. *R* responds by printing `[1]` to indicate that the subsequent number is the first element of the vector. It then prints the value of `x`.

R has many features to aid common operations. Entering sequences is a very common operation, and expressions of the form `2:4` create a sequence from `2` to `4`. Subsetting one vector by another is enabled with `[`. Here we create an integer sequence from 2 to 4, and use the sequence as an index to select the second, third, and fourth elements of `x`

```
> x[2:4]
```

```
[1] 4 3 2
```

R functions operate on variables. Functions are usually vectorized, acting on all elements of their argument and obviating the need for explicit iteration. Functions can generate warnings when performing suspect operations, or errors if evaluation cannot proceed; try `log(-1)`.

```
> log(x)
```

```
[1] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000
```

Essential data types *R* has a number of standard data types, to represent `integer`, `numeric` (floating point), `complex`, `character`, `logical` (boolean), and `raw` (byte) data. It is possible to convert between data types, and to discover the type or mode of a variable.

```
> c(1.1, 1.2, 1.3)         # numeric
```

```
[1] 1.1 1.2 1.3
```

```
> c(FALSE, TRUE, FALSE)    # logical
```

```
[1] FALSE  TRUE FALSE
```

```
> c("foo", "bar", "baz")   # character, single or double quote ok
```

```
[1] "foo" "bar" "baz"
```

```
> as.character(x)          # convert 'x' to character
```

```
[1] "5" "4" "3" "2" "1"
```

```
> typeof(x)                # the number 5 is numeric, not integer
```

```
[1] "double"
```

```
> typeof(2L)               # append 'L' to force integer
```

```
[1] "integer"
```

```
> typeof(2:4)              # ':' produces a sequence of integers
```

```
[1] "integer"
```

R includes data types particularly useful for statistical analysis, including `factor` to represent categories and `NA` (used in any vector) to represent missing values.

```
> sex <- factor(c("Male", "Female", NA), levels=c("Female", "Male"))
> sex
```

```
[1] Male   Female <NA>
Levels: Female Male
```

Lists, data frames, and matrices All of the vectors mentioned so far are homogeneous, consisting of a single type of element. A `list` can contain a collection of different types of elements and, like all vectors, these elements can be named to create a key-value association.

```
> lst <- list(a=1:3, b=c("foo", "bar"), c=sex)
> lst

$a
[1] 1 2 3

$b
[1] "foo" "bar"

$c
[1] Male   Female <NA>
Levels: Female Male
```

Lists can be subset like other vectors to get another list, or subset with `[[` to retrieve the actual list element; as with other vectors, subsetting can use names

```
> lst[c(3, 1)]             # another list

$c
[1] Male   Female <NA>
Levels: Female Male

$a
[1] 1 2 3

> lst[["a"]]               # the element itself, selected by name

[1] 1 2 3
```

A `data.frame` is a list of equal-length vectors, representing a rectangular data structure not unlike a spread sheet. Each column of the data frame is a vector, so data types must be homogenous within a column. A `data.frame` can be subset by row or column, and columns can be accessed with `$` or `[[`.

```
> df <- data.frame(age=c(27L, 32L, 19L),
+                  sex=factor(c("Male", "Female", "Male")))
> df

  age    sex
1  27   Male
2  32 Female
3  19   Male

> df[c(1, 3),]

  age  sex
1  27 Male
3  19 Male
```

```
> df[df$age > 20,]

  age    sex
1  27   Male
2  32 Female
```

A `matrix` is also a rectangular data structure, but subject to the constraint that all elements are the same type. A matrix is created by taking a vector, and specifying the number of rows or columns the vector is to represent. On subsetting, *R* coerces a single column `data.frame` or single row or column `matrix` to a vector if possible; use `drop=FALSE` to stop this behavior.

```
> m <- matrix(1:12, nrow=3)
> m

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> m[c(1, 3), c(2, 4)]

     [,1] [,2]
[1,]    4   10
[2,]    6   12

> m[, 3]

[1] 7 8 9

> m[, 3, drop=FALSE]

     [,1]
[1,]    7
[2,]    8
[3,]    9
```

An `array` is a data structure for representing homogenous, rectangular data in higher dimensions.

S3 and S4 classes More complicated data structures are represented using the 'S3' or 'S4' object system. Objects are often created by functions (for example, `lm`, below), with parts of the object extracted or assigned using *accessor* functions. The following generates 1000 random normal deviates as `x`, and uses these to create another 1000 deviates `y` that are linearly related to `x` but with some error. We fit a linear regression using a 'formula' to describe the relationship between variables, summarize the results in a familiar ANOVA table, and access `fit` (an S3 object) for the residuals of the regression, using these as input first to the `var` (variance) and then `sqrt` (square-root) functions. Objects can be interrogated for their class.

```
> x <- rnorm(1000, sd=1)
> y <- x + rnorm(1000, sd=.5)
> fit <- lm(y ~ x)       # formula describes linear regression
> fit                    # an 'S3' object

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
    0.01043      1.00035

> anova(fit)

Analysis of Variance Table

Response: y
           Df Sum Sq Mean Sq F value    Pr(>F)
x           1 945.90  945.90  3756.4 < 2.2e-16 ***
Residuals 998 251.31    0.25
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> sqrt(var(resid(fit)))  # residuals accessor and subsequent transforms

[1] 0.5015571

> class(fit)

[1] "lm"
```

Many *Bioconductor* packages implement S4 objects to represent data. S3 and S4 systems are quite different from a programmer's perspective, but fairly similar from a user's perspective: both systems encapsulate complicated data structures, and allow for methods specialized to different data types; accessors are used to extract information from the objects.

Functions *R* functions accept arguments, and return values. Arguments can be required or optional. Some functions may take variable numbers of arguments, e.g., the columns in a `data.frame`

```
> y <- 5:1
> log(y)

[1] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000

> args(log)        # arguments 'x' and 'base'; see ?log

function (x, base = exp(1))
NULL

> log(y, base=2)   # 'base' is optional, with default value
```

```
[1] 2.321928 2.000000 1.584963 1.000000 0.000000

> try(log())       # 'x' required; 'try' continues even on error
> args(data.frame) # ... represents variable number of arguments

function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
    stringsAsFactors = default.stringsAsFactors())
NULL
```

Arguments can be matched by name or position. If an argument appears after ..., it must be named.

```
> log(base=2, y)   # match argument 'base' by name, 'x' by position

[1] 2.321928 2.000000 1.584963 1.000000 0.000000
```

A function such as `anova` is a *generic* that provides an overall signature but dispatches the actual work to the *method* corresponding to the class(es) of the arguments used to invoke the generic. A generic may have fewer arguments than a method, as with the S3 function `anova` and its method `anova.glm`.

```
> args(anova)

function (object, ...)
NULL

> args(anova.glm)

function (object, ..., dispersion = NULL, test = NULL)
NULL
```

The ... argument in the `anova` generic means that additional arguments are possible; the `anova` generic hands these arguments to the method it dispatches to.

2.2 Useful functions

R has a very large number of functions. The following is a brief list of those that might be commonly used and particularly useful.

`dir`, `read.table` (and friends), `scan` List files in a directory, read spreadsheet-like data into *R*, efficiently read homogenous data (e.g., a file of numeric values) to be represented as a matrix.

`c`, `factor`, `data.frame`, `matrix` Create a vector, factor, data frame or matrix.

`summary`, `table`, `xtabs` Summarize, create a table of the number of times elements occur in a vector, cross-tabulate two or more variables.

`t.test`, `aov`, `lm`, `anova` Basic comparison of two (`t.test`) groups, or several groups via analysis of variance / linear models (`aov` output is probably more familiar to biologists), or compare simpler with more complicated models (`anova`).

`dist`, `hclust` Cluster data.

`plot` Plot data.

`ls`, `str`, `library`, `search` List objects in the current (or specified) workspace, or peak at the structure of an object; add a library to or describe the search path of attached packages.

`lapply`, `sapply`, `mapply` Apply a function to each element of a list (`lapply`, `sapply`) or to elements of several lists (`mapply`).

`with` Conveniently access columns of a data frame or other element without having to repeat the name of the data frame.

`match`, `%in%` Report the index or existence of elements from one vector that match another.

`split`, `cut` Split one vector by an equal length factor, cut a single vector into intervals encoded as levels of a factor.

`strsplit`, `grep`, `sub` Operate on character vectors, splitting it into distinct fields, searching for the occurrence of a patterns using regular expressions (see `?regex`, or substituting a string for a regular expression.

`install.packages` Install a package from an on-line repository into your *R*.

`traceback`, `debug`, `browser` Report the sequence of functions under evaluatino at the time of the error; enter a debugger when a particular function or statement is invoked.

See the help pages (e.g., `?lm`) and examples (`example(match)`) for each of these functions

Exercise 2

This exercise uses data describing 128 microarray samples as a basis for exploring R functions. Covariates such as age, sex, type, stage of the disease, etc., are in a data file `pData.csv`.

The following command creates a variable `pdataFiles` that is the location of a comma-separated value ('csv') file to be used in the exercise. A csv file can be created using, e.g., 'Save as...' in spreadsheet software.

```
> pdataFile <- system.file(package="SequenceAnalysisData", "extdata",
+                          "pData.csv")
```

Input the csv file using `read.table`, assigning the input to a variable `pdata`. Use `dim` to find out the dimensions (number of rows, number of columns) in the object. Are there 128 rows? Use `names` or `colnames` to list the names of the columns of `pdata`. Use `summary` to summarize each column of the data. What are the data types of each column in the data frame?

A data frame is a list of equal length vectors. Select the 'sex' column of the data frame using `[[` or `$`. Pause to explain to your neighbor why this subsetting works. Since a data frame is a list, use `sapply` to ask about the class of each column in the data frame. Explain to your neighbor what this produces, and why.

Use `table` to summarize the number of males and females in the sample. Consult the help page `?table` to figure out additional arguments required to include `NA` values in the tabulation.

The `mol.biol` column summarizes molecular biological attributes of each sample. Use `table` to summarize the different molecular biology levels in the sample. Use `%in%` to create a logical vector of the samples that are either `BCR/ABL` or `NEG`. Subset the original phenotypic data to contain those samples that are `BCR/ABL` or `NEG`.

After subsetting, what are the levels of the `mol.biol` column? Set the levels to be `BCR/ABL` and `NEG`, i.e., the levels in the subset.

One would like covariates to be similar across groups of interest. Use `t.test` to assess whether `BCR/ABL` and `NEG` have individuals with similar age. To do this, use a `formula` that describes the response `age` in terms of the predictor `mol.biol`. If age is not independent of molecular biology, what complications might this introduce into subsequent analysis? Use

Solution: Here we input the data and explore basic properties.

```
> pdata <- read.table(pdataFile)
> dim(pdata)
```

```
[1] 128  21
```

```
> names(pdata)
```

```
 [1] "cod"            "diagnosis"      "sex"            "age"
 [5] "BT"             "remission"      "CR"             "date.cr"
 [9] "t.4.11."        "t.9.22."        "cyto.normal"    "citog"
[13] "mol.biol"       "fusion.protein" "mdr"            "kinet"
[17] "ccr"            "relapse"        "transplant"     "f.u"
[21] "date.last.seen"
```

```
> summary(pdata)
```

```
      cod           diagnosis     sex          age             BT
 10005  :  1   11/15/1997:  2   F   :42   Min.   : 5.00   B2     :36
 1003   :  1   1/15/1997 :  2   M   :83   1st Qu.:19.00   B3     :23
 remission                  CR           date.cr    t.4.11.
 CR  :99   CR                :96   11/11/1997: 3   Mode :logical
 REF :15   DEATH IN CR       : 3   10/18/1999: 2   FALSE:86
  t.9.22.        cyto.normal              citog        mol.biol
 Mode :logical   Mode :logical   normal       :24   ALL1/AF4:10
 FALSE:67        FALSE:69        simple alt.  :15   BCR/ABL :37
   fusion.protein   mdr          kinet        ccr          relapse
 p190     :17     NEG :101   dyploid:94   Mode :logical   Mode :logical
 p190/p210: 8     POS : 24   hyperd.:27   FALSE:74       FALSE:35
 transplant                     f.u        date.last.seen
 Mode :logical   REL              :61   12/15/1997: 2
 FALSE:91        CCR              :23   12/31/2002: 2
 [ reached getOption("max.print") -- omitted 5 rows ]
```

A data frame can be subset as if it were a matrix, or a list of column vectors.

```
> head(pdata[,"sex"], 3)
```

```
[1] M M F
Levels: F M
```

```
> head(pdata$sex, 3)
```

```
[1] M M F
Levels: F M
```

```
> head(pdata[["sex"]], 3)

[1] M M F
Levels: F M

> sapply(pdata, class)

           cod      diagnosis            sex            age             BT
      "factor"       "factor"       "factor"      "integer"       "factor"
     remission             CR        date.cr        t.4.11.        t.9.22.
      "factor"       "factor"       "factor"      "logical"      "logical"
   cyto.normal          citog       mol.biol fusion.protein            mdr
     "logical"       "factor"       "factor"       "factor"       "factor"
         kinet            ccr        relapse     transplant            f.u
      "factor"      "logical"      "logical"      "logical"       "factor"
date.last.seen
      "factor"
```

The number of males and females, including `NA`, is

```
> table(pdata$sex, useNA="ifany")

   F    M <NA>
  42   83    3
```

An alternative version of this uses the `with` function: `with(pdata, table(sex, useNA="ifany"))`.

The `mol.biol` column contains the following samples:

```
> with(pdata, table(mol.biol, useNA="ifany"))

mol.biol
ALL1/AF4  BCR/ABL E2A/PBX1      NEG   NUP-98  p15/p16
      10       37        5       74        1        1
```

A logical vector indicating that the corresponding row is either `BCR/ABL` or `NEG` is constructed as

```
> ridx <- pdata$mol.biol %in% c("BCR/ABL", "NEG")
```

We can get a sense of the number of rows selected via `table` or `sum` (discuss with your neighbor what `sum` does, and why the answer is the same as the number of `TRUE` values in the result of the `table` function).

```
> table(ridx)

ridx
FALSE  TRUE
   17   111

> sum(ridx)

[1] 111
```

The original data frame can be subset to contain only BCR/ABL or NEG samples using the logical vector ridx that we created.

```
> pdata1 <- pdata[ridx,]
```

The levels of each factor reflect the levels in the original object, rather than the levels in the subset object, e.g.,

```
> levels(pdata$mol.biol)
```

```
[1] "ALL1/AF4" "BCR/ABL"  "E2A/PBX1" "NEG"      "NUP-98"   "p15/p16"
```

These can be re-coded by updating the new data frame to contain a factor with the desired levels.

```
> pdata1$mol.biol <- factor(pdata1$mol.biol)
> table(pdata1$mol.biol)
```

```
BCR/ABL     NEG
     37      74
```

To ask whether age differs between molecular biology types, we use a formula age ~ mol.biol to describe the relationship ('age as a function of molecular biology') that we wish to test

```
> with(pdata1, t.test(age ~ mol.biol))
```

```
        Welch Two Sample t-test

data:  age by mol.biol
t = 4.8172, df = 68.529, p-value = 8.401e-06
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  7.13507 17.22408
sample estimates:
mean in group BCR/ABL     mean in group NEG
             40.25000              28.07042
```

This summary can be visualize with, e.g., the boxplot function

```
> ## not evaluated
> boxplot(age ~ mol.biol, pdata1)
```

Molecular biology seem to be strongly associated with age; individuals in the NEG group are considerably younger than those in the BCR/ABL group. We might wish to include age as a covariate in any subsequent analysis seeking to relate molecular biology to gene expression.

Table 3: Selected base and contributed packages.

Package	Description
base	Data input and essential manipulation; scripting and programming concepts.
stats	Essential statistical and plotting functions.
lattice, *ggplot2*	Approaches to advanced graphics.
methods	'S4' classes and methods.
parallel	Facilities for parallel evaluation.

2.3 Packages

Packages provide functionality beyond that available in base *R*. There are over 3000 packages in CRAN (comprehensive *R* archive network) and more than 500 *Bioconductor* packages. Packages are contributed by diverse members of the community; they vary in quality (many are excellent) and sometimes contain idiosyncratic aspects to their implementation. Table 3 outlines key base packages and selected contributed packages; see a local CRAN mirror (including the task views summarizing packages in different domains) and *Bioconductor* for additional contributed packages.

The *lattice* package illustrates the value packages add to base *R*. *lattice* is distributed with *R* but not loaded by default. It provides a very expressive way to visualize data. The following example plots yield for a number of barley varieties, conditioned on site and grouped by year. Figure 1 is read from the lower left corner. Note the common scales, efficient use of space, and not-too-pleasing default color palette. The Morris sample appears to be mis-labeled for 'year', an apparent error in the original data. Find out about the built-in data set used in this example with `?barley`.

```
> library(lattice)
> dotplot(variety ~ yield | site, data = barley, groups = year,
+         key = simpleKey(levels(barley$year), space = "right"),
+         xlab = "Barley Yield (bushels/acre)",
+         aspect=0.5, layout = c(2,3), ylab=NULL)
```

New packages can be added to an *R* installation using `install.packages`. A package is installed only once per *R* installation, but needs to be loaded (with `library`) in each session in which it is used. Loading a package also loads any package that it depends on. Packages loaded in the current session are displayed with `search`. The ordering of packages returned by `search` represents the order in which the global environment (where commands entered at the prompt are evaluated) and attached packages are searched for symbols; it is possible for a package earlier in the search path to mask symbols later in the search path; these can be disambiguated using `::`.

```
> length(search())

[1] 11

> search()
```

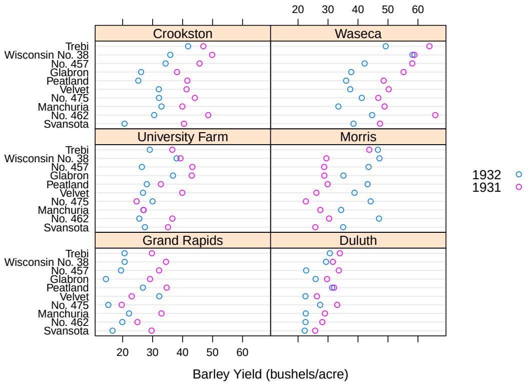


Figure 1: Variety yield conditional on site and grouped by year, for the `barley` data set.

```
 [1] ".GlobalEnv"             "package:lattice"        "package:BiocInstaller"
 [4] "package:stats"          "package:graphics"       "package:grDevices"
 [7] "package:utils"          "package:datasets"       "package:methods"
[10] "Autoloads"              "package:base"
```

```
> base::log(1:3)
```

```
[1] 0.0000000 0.6931472 1.0986123
```

Exercise 3
Use the `library` *function to load the SequenceAnalysis package. Use the* `sessionInfo` *function to verify that you are using R version 2.15.0 and current packages, similar to those reported here. What other packages were loaded along with SequenceAnalysis?*

Solution:

```
> library(SequenceAnalysis)
> sessionInfo()
```

2.4 Help

Find help using the *R* help system. Start a web browser with

```
> help.start()
```

The 'Search Engine and Keywords' link is helpful in day-to-day use.

Manual pages Use manual pages to find detailed descriptions of the arguments and return values of functions, and the structure and methods of classes. Find help within an *R* session as

```
> ?data.frame
> ?lm
> ?anova             # a generic function
> ?anova.lm          # an S3 method, specialized for 'lm' objects
```

S3 methods can be queried interactively. For S3,

```
> methods(anova)

[1] anova.glm     anova.glmlist anova.lm      anova.loess*  anova.mlm
[6] anova.nls*

   Non-visible functions are asterisked

> methods(class="glm")

 [1] add1.glm*           anova.glm           confint.glm*
 [4] cooks.distance.glm* deviance.glm*       drop1.glm*
 [7] effects.glm*        extractAIC.glm*     family.glm*
[10] formula.glm*        influence.glm*      logLik.glm*
[13] model.frame.glm     nobs.glm*           predict.glm
[16] print.glm           residuals.glm       rstandard.glm
[19] rstudent.glm        summary.glm         vcov.glm*
[22] weights.glm*

   Non-visible functions are asterisked
```

It is often useful to view a method definition, either by typing the method name at the command line or, for 'non-visible' methods, using `getAnywhere`:

```
> anova.lm
> getAnywhere("anova.loess")
```

For instance, the source code of a function is printed if the function is invoked without parentheses. Here we discover that the function `head` (which returns the first 6 elements of anything) defined in the *utils* package, is an S3 generic (indicated by `UseMethod`) and has several methods. We use `head` to look at the first six lines of the `head` method specialized for `matrix` objects.

```
> utils::head

function (x, ...)
UseMethod("head")
<environment: namespace:utils>

> methods(head)

[1] head.data.frame* head.default*    head.ftable*     head.function*
[5] head.matrix      head.table*

   Non-visible functions are asterisked
```

```
> head(head.matrix)
```

```
1 function (x, n = 6L, ...)
2 {
3     stopifnot(length(n) == 1L)
4     n <- if (n < 0L)
5         max(nrow(x) + n, 0L)
6     else min(n, nrow(x))
```

S4 classes and generics are queried in a similar way to S3 classes and generics, but with different syntax, as for the `complement` generic in the *Biostrings* package:

```
> library(Biostrings)
> showMethods(complement)
```

```
Function: complement (package Biostrings)
x="DNAString"
x="DNAStringSet"
x="MaskedDNAString"
x="MaskedRNAString"
x="RNAString"
x="RNAStringSet"
x="XStringViews"
```

Methods defined on the `DNAStringSet` class of *Biostrings* can be found with

```
> showMethods(class="DNAStringSet", where=getNamespace("Biostrings"))
```

Obtaining help on S4 classes and methods requires syntax such as

```
> class ? DNAStringSet
> method ? "complement,DNAStringSet"
```

The specification of method and class in the latter must not contain a space after the comma. The definition of a method can be retrieved as

```
> selectMethod(complement, "DNAStringSet")
```

Vignettes Vignettes, especially in *Bioconductor* packages, provide an extensive narrative describing overall package functionality. Use

```
> vignette(package="SequenceAnalysis")
```

to see, in your web browser, vignettes available in the *SequenceAnalysis* package. Vignettes usually consist of text with embedded *R* code, a form of literate programming. The vignette can be read as a PDF document, while the *R* source code is present as a script file ending with extension `.R`. The script file can be sourced or copied into an *R* session to evaluate exactly the commands used in the vignette.

Exercise 4
Scavenger hunt. Spend five minutes tracking down the following information.

a. *The package containing the* `library` *function.*

b. The author of the `alphabetFrequency` function, defined in the Biostrings package.

c. A description of the GappedAlignments class.

d. The number of vignettes in the GenomicRanges package.

Solution: Possible solutions are found with the following *R* commands

```
> ?library
> library(Biostrings)
> ?alphabetFrequency
> class?GappedAlignments
> vignette(package="GenomicRanges")
```

2.5 Efficient scripts

There are often many ways to accomplish a result in *R*, but these different ways often have very different speed or memory requirements. For small data sets these performance differences are not that important, but for large data sets (e.g., high-throughput sequencing; genome-wide association studies, GWAS) or complicated calculations (e.g., bootstrapping) performance can be important. There are several approaches to achieving efficient *R* programming.

Easy solutions Several common performance bottlenecks often have easy solutions; these are outlined here.

Text files often contain more information, for example 1000's of individuals at millions of SNPs, when only a subset of the data is required, e.g., during algorithm development. Reading in all the data can be demanding in terms of both memory and time. A solution is to use arguments such as `colClasses` to specify the columns and their data types that are required, and to use `nrow` to limit the number of rows input. For example, the following ignores the first and fourth column, reading in only the second and third (as type `integer` and `numeric`).

```
> ## not evaluated
> colClasses <-
+   c("NULL", "integer", "numeric", "NULL")
> df <- read.table("myfile", colClasses=colClasses)
```

R is vectorized, so traditional programming `for` loops are often not necessary. Rather than calculating 100000 random numbers one at a time, or squaring each element of a vector, or iterating over rows and columns in a matrix to calculate row sums, invoke the single function that performs each of these operations.

```
> x <- runif(100000); x2 <- x^2
> m <- matrix(x2, nrow=1000); y <- rowSums(m)
```

This often requires a change of thinking, turning the sequence of operations 'inside-out'. For instance, calculate the log of the square of each element of a vector by calculating the square of all elements, followed by the log of all elements `x2 <- x^2; x3 <- log(x2)`, or simply `logx2 <- log(x^2)`.

It may sometimes be natural to formulate a problem as a `for` loop, or the formulation of the problem may require that a `for` loop be used. In these circumstances the appropriate strategy is to pre-allocate the `result` object, and to fill the result in during loop iteration.

```
> ## not evaluated
> result <- numeric(nrow(df))
> for (i in seq_len(nrow(df)))
+   result[[i]] <- some_calc(df[i,])
```

Some *R* operations are helpful in general, but misleading or inefficient in particular circumstances. An example is the behavior of `unlist` when the list is named – *R* creates new names that have been made unique. This can be confusing (e.g., when Entrez gene identifiers are 'mangled' to unintentionally look like other identifiers) and expensive (when a large number of new names need to be created). Avoid creating unnecessary names, e.g.,

```
> unlist(list(a=1:2)) # name 'a' becomes 'a1', 'a2'

a1 a2
 1  2

> unlist(list(a=1:2), use.names=FALSE)   # no names

[1] 1 2
```

Names can be very useful for avoiding book-keeping errors, but are inefficient for repeated look-ups; use vectorized access or numeric indexing.

Moderate solutions Several solutions to inefficient code require greater knowledge to implement.

Using appropriate functions can greatly influence performance; it takes experience to know when an appropriate function exists. For instance, the `lm` function could be used to assess differential expression of each gene on a microarray, but the *limma* package implements this operation in a way that takes advantage of the experimental design that is common to each probe on the microarray, and does so in a very efficient manner.

```
> ## not evaluated
> library(limma) # microarray linear models
> fit <- lmFit(eSet, design)
```

Using appropriate algorithms can have significant performance benefits, especially as data becomes larger. This solution requires moderate skills, because one has to be able to think about the complexity (e.g., expected number of operations) of an algorithm, and to identify algorithms that accomplish the same goal in fewer steps. For example, a naive way of identifying which of 100 numbers are in a set of size 10 might look at all 100 × 10 combinations of numbers (i.e., polynomial time), but a faster way is to create a 'hash' table of one of the set of elements and probe that for each of the other elements (i.e., linear time). The latter strategy is illustrated with

```
> x <- 1:100; s <- sample(x, 10)
> inS <- x %in% s
```

R is an interpreted language, and for very challenging computational problems it may be appropriate to write critical stages of an analysis in a compiled language like C or Fortran, or to use an existing programming library (e.g., the BOOST graph library) that efficiently implements advanced algorithms. *R* has a well-developed interface to C or Fortran, so it is 'easy' to do this. This places a significant burden on the person implementing the solution, requiring knowledge of two or more computer languages and of the interface between them.

Measuring performance When trying to improve performance, one wants to ensure (a) that the new code is actually faster than the previous code, and (b) both solutions arrive at the same, correct, answer.

The `system.time` function is a straight-forward way to measure the length of time a portion of code takes to evaluate. Here we see that the use of `apply` to calculate row sums of a matrix is much less efficient than the specialized `rowSums` function.

```
> m <- matrix(runif(200000), 20000)
> replicate(5, system.time(apply(m, 1, sum))[[1]])

[1] 0.176 0.168 0.168 0.164 0.168

> replicate(5, system.time(rowSums(m))[[1]])

[1] 0.000 0.004 0.000 0.000 0.000
```

Usually it is appropriate to replicate timings to average over vagaries of system use, and to shuffle the order in which timings of alternative algorithms are calculated to avoid artifacts such as initial memory allocation.

Speed is an important metric, but equivalent results are also needed. The functions `identical` and `all.equal` provide different levels of assessing equivalence, with `all.equal` providing ability to ignore some differences, e.g., in the names of vector elements.

```
> res1 <- apply(m, 1, sum)
> res2 <- rowSums(m)
> identical(res1, res2)

[1] TRUE

> identical(c(1, -1), c(x=1, y=-1))

[1] FALSE

> all.equal(c(1, -1), c(x=1, y=-1),
+           check.attributes=FALSE)

[1] TRUE
```

Two additional functions for assessing performance are Rprof and tracemem; these are mentioned only briefly here. The Rprof function profiles *R* code, presenting a summary of the time spent in each part of several lines of *R* code. It is useful for gaining insight into the location of performance bottlenecks when these are not readily apparent from direct inspection. Memory management, especially copying large objects, can frequently contribute to poor performance. The tracemem function allows one to gain insight into how *R* manages memory; insights from this kind of analysis can sometimes be useful in restructuring code into a more efficient sequence.

2.6 Warnings, errors, and debugging

R signals unexpected results through warnings and errors. Warnings occur when the calculation produces an unusual result that nonetheless does not preclude further evaluation. For instance log(-1) results in a value NaN ('not a number') that allows computation to continue, but at the same time signals an warning

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

Errors result when the inputs or outputs of a function are such that no further action can be taken, e.g., trying to take the square root of a character vector

```
> sqrt("two")
Error in sqrt("two") : Non-numeric argument to mathematical function
```

Warnings and errors occurring at the command prompt are usually easy to diagnose. They can be more enigmatic when occurring in a function, and exacerbated by sometimes cryptic (when read out of context) error messages.

An initial step in coming to terms with errors is to simplify the problem as much as possible, aiming for a 'reproducible' error. The reproducible error might involve a very small (even trivial) data set that immediately provokes the error. Often the process of creating a reproducible example helps to clarify what the error is, and what possible solutions might be.

Invoking traceback() immediately after an error occurs provides a 'stack' of the function calls that were in effect when the error occurred. This can help understand the context in which the error occurred. Knowing the context, one might use debug to enter into a browser (see ?browser) that allows one to step through the function in which the error occurred.

It can sometimes be useful to use global options (see ?options) to influence what happens when an error occurs. Two common global options are error and warn. Setting error=recover combines the functionality of traceback and debug, allowing the user to enter the browser at any level of the call stack in effect at the time the error occurred. Default error behavior can be restored with options(error=NULL). Setting warn=2 causes warnings to be promoted to errors. For instance, initial investigation of an error might show that the error occurs when one of the arguments to a function has value NaN. The error might be accompanied by a warning message that the NaN has been introduced, but because warnings are by default not reported immediately it is not clear where the `NaN` comes from. `warn=2` means that the warning is treated as an error, and hence can be debugged using `traceback`, `debug`, and so on.

Additional useful debugging functions include `browser`, `trace`, and `setBreakpoint`.

Fixme: tryCatch

Table 4: Selected *Bioconductor* packages for representing and manipulating ranges, strings, and other data structures.

Package	Description
IRanges	Defines important classes (e.g., *IRanges*, *Rle*) and methods (e.g., `findOverlaps`, `countOverlaps`) for representing and manipulating ranges of consecutive values. Also introduces *DataFrame*, *SimpleList* and other classes tailored to representing very large data.
GenomicRanges	Range-based classes tailored to sequence representation (e.g., *GRanges*, *GRangesList*), with information about strand and sequence name.
GenomicFeatures	Foundation for manipulating data bases of genomic ranges, e.g., representing coordinates and organization of exons and transcripts of known genes.
Biostrings	Classes (e.g., *DNAStringSet*) and methods (e.g., `alphabetFrequency`, `pairwiseAlignment`) for representing and manipulating DNA and other biological sequences.
BSgenome	Representation and manipulation of large (e.g., whole-genome) sequences.

3 Ranges and Strings

Many *Bioconductor* packages are available for analysis of high-throughput sequence data. This section introduces two essential ways in which sequence data are manipulated. Ranges describe both aligned reads and features of interest on the genome. Sets of DNA strings represent the reads themselves and the nucleotide sequence of reference genomes. Key packages are summarized in Table 4.

3.1 Genomic ranges

Next-generation sequencing data consists of a large number of short reads. These are, typically, aligned to a reference genome. Basic operations are performed on the alignment, asking e.g., how many reads are aligned in a genomic range defined by nucleotide coordinates (e.g., in the exons of a gene), or how many nucleotides from all the aligned reads cover a set of genomic coordinates. How is this type of data, the aligned reads and the reference genome, to be represented in *R* in a way that allows for effective computation?

The *IRanges*, *GenomicRanges*, and *GenomicFeatures* *Bioconductor* packages provide the essential infrastructure for these operations; we start with the *GRanges* class, defined in *GenomicRanges*.

GRanges Instances of *GRanges* are used to specify genomic coordinates. Suppose we wish to represent two *D. melanogaster* genes. The first is located on the positive strand of chromosome 3R, from position 19967117 to 19973212. The second is on the minus strand of the X chromosome, with 'left-most' base at 18962306, and right-most base at 18962925. The coordinates are *1-based* (i.e., the first nucleotide on a chromosome is numbered 1, rather than 0), *left-most* (i.e., reads on the minus strand are defined to 'start' at the left-most coordinate, rather than the 5' coordinate), and *closed* (the start and end coordinates are included in the range; a range with identical start and end coordinates has width 1, a 0-width range is represented by the special construct where the end coordinate is one less than the start coordinate).

A complete definition of these genes as *GRanges* is:

```
> genes <- GRanges(seqnames=c("3R", "X"),
+                  ranges=IRanges(
+                      start=c(19967117, 18962306),
+                      end=c(19973212, 18962925)),
+                  strand=c("+", "-"),
+                  seqlengths=c(`3R`=27905053L, `X`=22422827L))
```

The components of a *GRanges* object are defined as vectors, e.g., of `seqnames`, much as one would define a *data.frame*. The start and end coordinates are grouped into an *IRanges* instance. The optional `seqlengths` argument specifies the maximum size of each sequence, in this case the lengths of chromosomes 3R and X in the 'dm2' build of the *D. melanogaster* genome. This data is displayed as

```
> genes
```

```
GRanges with 2 ranges and 0 elementMetadata cols:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
  [1]       3R [19967117, 19973212]      +
  [2]        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827
```

For the curious, the gene coordinates and sequence lengths are derived from the *org.Dm.eg.db* package for genes with Flybase identifiers FBgn0039155 and FBgn0085359, using the annotation facilities described in section 7.

The *GRanges* class has many useful methods defined on it. Consult the help page

```
> ?GRanges
```

and package vignettes (especially 'An Introduction to *GenomicRanges*')

```
> vignette(package="GenomicRanges")
```

for a comprehensive introduction. A *GRanges* instance can be subset, with accessors for getting and updating information.

```
> genes[2]
```

```
GRanges with 1 range and 0 elementMetadata cols:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
```

```
  [1]        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827

> strand(genes)

'factor' Rle of length 2 with 2 runs
  Lengths: 1 1
  Values : + -
Levels(3): + - *

> width(genes)

[1] 6096  620

> length(genes)

[1] 2

> names(genes) <- c("FBgn0039155", "FBgn0085359")
> genes  # now with names

GRanges with 2 ranges and 0 elementMetadata cols:
              seqnames               ranges strand
                 <Rle>            <IRanges>  <Rle>
  FBgn0039155       3R [19967117, 19973212]      +
  FBgn0085359        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827
```

`strand` returns the strand information in a compact representation called a *run-length encoding*, this is introduced in greater detail below. The 'names' could have been specified when the instance was constructed; once named, the *GRanges* instance can be subset by name like a regular vector.

As the `GRanges` function suggests, the *GRanges* class extends the *IRanges* class by adding information about `seqname`, `strand`, and other information particularly relevant to representing ranges that are on genomes. The *IRanges* class and related data structures (e.g., *RangedData*) are meant as a more general description of ranges defined in an arbitrary space. Many methods implemented on the *GRanges* class are 'aware' of the consequences of genomic location, for instance treating ranges on the minus strand differently (reflecting the 5' orientation imposed by DNA) from ranges on the plus strand.

Operations on ranges The *GRanges* class has many useful methods from the *IRanges* class; some of these methods are illustrated here. We use *IRanges* to illustrate these operations to avoid complexities associated with strand and seqname, but the operations are comparable on *GRanges*. We begin with a simple set of ranges:

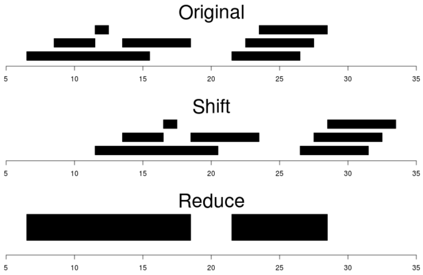


Figure 2: Ranges

```
> ir <- IRanges(start=c(7, 9, 12, 14, 22:24),
+               end=c(15, 11, 12, 18, 26, 27, 28))
```

These and some common operations are illustrated in the upper panel of Figure 2 and summarized in Table 5.

Methods on ranges can be grouped as follows:

Intra-range methods act on each range independently. These include `flank`, `narrow`, `reflect`, `resize`, `restrict`, and `shift`, among others. An illustration is `shift`, which translates each range by the amount specified by the `shift` argument. Positive values shift to the right, negative to the left; `shift` can be a vector, with each element of the vector shifting the corresponding element of the *IRanges* instance. Here we shift all ranges to the right by 5, with the result illustrated in the middle panel of Figure 2.

```
> shift(ir, 5)

IRanges of length 7
    start end width
[1]    12  20     9
[2]    14  16     3
[3]    17  17     1
[4]    19  23     5
[5]    27  31     5
[6]    28  32     5
[7]    29  33     5
```

Inter-range methods act on the collection of ranges as a whole. These include `disjoin`, `reduce`, `gaps`, and `range`. An illustration is `reduce`, which reduces overlapping ranges into a single range, as illustrated in the lower panel of Figure 2.

```
> reduce(ir)

IRanges of length 2
    start end width
[1]     7  18    12
[2]    22  28     7
```

`coverage` is an inter-range operation that calculates how many ranges overlap individual positions. Rather than returning ranges, `coverage` returns a compressed representation (run-length encoding)

```
> coverage(ir)
```

```
'integer' Rle of length 28 with 12 runs
  Lengths: 6 2 4 1 2 3 3 1 1 3 1 1
  Values : 0 1 2 1 2 1 0 1 2 3 2 1
```

The run-length encoding can be interpreted as 'a run of length 6 of nucleotides covered by 0 ranges, followed by a run of length 2 of nucleotides covered by 1 range…'.

Between methods act on two (or sometimes more) *IRanges* instances. These include `intersect`, `setdiff`, `union`, `pintersect`, `psetdiff`, and `punion`.

The `countOverlaps` and `findOverlaps` functions operate on two sets of ranges. `countOverlaps` takes its first argument (the `query`) and determines how many of the ranges in the second argument (the `subject`) each overlaps. The result is an integer vector with one element for each member of `query`. `findOverlaps` performs a similar operation but returns a more general matrix-like structure that identifies each pair of query / subject overlaps. Both arguments allow some flexibility in the definition of 'overlap'.

Common operations on ranges are summarized in Table 5.

`elementMetadata` (`values`) and `metadata` The *GRanges* class (actually, most of the data structures defined or extending those in the IRanges package) has two additional very useful data components. The `elementMetadata` function (or its synonym `values`) allows information on each range to be stored and manipulated (e.g., subset) along with the *GRanges* instance. The element metadata is represented as a *DataFrame*, defined in IRanges and acting like a standard *R data.frame* but with the ability to hold more complicated data structures as columns (and with element metadata of its own, providing an enhanced alternative to the Biobase class *AnnotatedDataFrame*).

```
> elementMetadata(genes) <-
+     DataFrame(EntrezId=c("42865", "2768869"),
+               Symbol=c("kal-1", "CG34330"))
```

`metadata` allows addition of information to the entire object. The information is in the form of a list; any data can be provided.

```
> metadata(genes) <-
+     list(CreatedBy="A. User", Date=date())
```

GRangesList The `GRanges` class is extremely useful for representing simple ranges. Some next-generation sequence data and genomic features are more hierarchically structured. A gene may be represented by several exons within it. An aligned read may be represented by discontinuous ranges of alignment to a reference. The *GRangesList* class represents this type of information. It is a

Table 5: Common operations on *IRanges*, *GRanges* and *GRangesList*.

Category	Function	Description
Accessors	`start`, `end`, `width`	Get or s et the starts, ends and widths
	`names`	Get or set the names
	`elementMetadata`, `metadata`	Get or set metadata on elements or object
	`length`	Number of ranges in the vector
	`range`	Range formed from min start and max end
Ordering	`<`, `<=`, `>`, `>=`, `==`, `!=`	Compare ranges, ordering by start then width
	`sort`, `order`, `rank`	Sort by the ordering
	`duplicated`	Find ranges with multiple instances
	`unique`	Find unique instances, removing duplicates
Arithmetic	`r + x`, `r - x`, `r * x`	Shrink or expand ranges `r` by number `x`
	`shift`	Move the ranges by specified amount
	`resize`	Change width, anchoring on start, end or mid
	`distance`	Separation between ranges (closest endpoints)
	`restrict`	Clamp ranges to within some start and end
	`flank`	Generate adjacent regions on start or end
Set operations	`reduce`	Merge overlapping and adjacent ranges
	`intersect`, `union`, `setdiff`	Set operations on reduced ranges
	`pintersect`, `punion`, `psetdiff`	Parallel set operations, on each `x[i]`, `y[i]`
	`gaps`, `pgap`	Find regions not covered by reduced ranges
	`disjoin`	Ranges formed from union of endpoints
Overlaps	`findOverlaps`	Find all overlaps for each `x` in `y`
	`countOverlaps`	Count overlaps of each `x` range in `y`
	`nearest`	Find nearest neighbors (closest endpoints)
	`precede`, `follow`	Find nearest `y` that `x` precedes or follows
	`x %in% y`	Find ranges in `x` that overlap range in `y`
Coverage	`coverage`	Count ranges covering each position
Extraction	`r[i]`	Get or set by logical or numeric index
	`r[[i]]`	Get integer sequence from `start[i]` to `end[i]`
	`subsetByOverlaps`	Subset `x` for those that overlap in `y`
	`head`, `tail`, `rev`, `rep`	Conventional R semantics
Split, combine	`split`	Split ranges by a factor into a *RangesList*
	`c`	Concatenate two or more range objects

list-like data structure, which each element of the list itself a *GRanges* instance. The gene FBgn0039155 contains several exons, and can be represented as a list of length 1, where the element of the list contains a *GRanges* object with 7 elements:

```
GRangesList of length 1:
$FBgn0039155
GRanges with 7 ranges and 2 elementMetadata cols:
      seqnames               ranges strand |   exon_id   exon_name
         <Rle>            <IRanges>  <Rle> | <integer> <character>
  [1]    chr3R [19967117, 19967382]      + |     64137        <NA>
  [2]    chr3R [19970915, 19971592]      + |     64138        <NA>
  [3]    chr3R [19971652, 19971770]      + |     64139        <NA>
  [4]    chr3R [19971831, 19972024]      + |     64140        <NA>
  [5]    chr3R [19972088, 19972461]      + |     64141        <NA>
  [6]    chr3R [19972523, 19972589]      + |     64142        <NA>
  [7]    chr3R [19972918, 19973212]      + |     64143        <NA>

---
seqlengths:
    chr3R
 27905053
```

The *GRangesList* object has methods one would expect for lists (e.g., `length`, subsetting). Many of the methods introduced for working with *IRanges* are also available, with the method applied element-wise.

The *GenomicFeatures* package Many public resources provide annotations about genomic features. For instance, the UCSC genome browser maintains the 'knownGene' track of established exons, transcripts, and coding sequences of many model organisms. The *GenomicFeatures* package provides a way to retrieve, save, and query these resources. The underlying representation is as sqlite data bases, but the data are available in *R* as *GRangesList* objects. The following exercise explores the *GenomicFeatures* package and some of the functionality for the *IRanges* family introduced above.

Exercise 5
Load the TxDb.Dmelanogaster.UCSC.dm3.ensGene annotation package, and create an alias `txdb` pointing to the TranscriptDb object this class defines.

Extract all exon coordinates, organized by gene, using `exonsBy`. What is the class of this object? How many elements are in the object? What does each element correspond to? And the elements of each element? Use `elementLengths` and `table` to summarize the number of exons in each gene, for instance, how many single-exon genes are there?

Select just those elements corresponding to flybase gene ids FBgn0002183, FBgn0003360, FBgn0025111, and FBgn0036449. Use `reduce` to simplify gene models, so that exons that overlap are considered 'the same'.

Solution:

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene # alias
> ex0 <- exonsBy(txdb, "gene")
> head(table(elementLengths(ex0)))

   1    2    3    4    5    6
3182 2608 2070 1628 1133  886

> ids <- c("FBgn0002183", "FBgn0003360", "FBgn0025111", "FBgn0036449")
> ex <- reduce(ex0[ids])
```

Exercise 6
(Independent) Create a TranscriptDb instance from UCSC, using `makeTranscriptDbFromUCSC`*.*

Solution:

```
> txdb <- makeTranscriptDbFromUCSC("dm3", "ensGene")
> saveDb(txdb, "my.dm3.ensGene.txdb.sqlite")
```

3.2 Working with strings

Underlying the ranges of alignments and features are DNA sequences. The *Biostrings* package provides tools for working with this data. The essential data structures are *DNAString* and *DNAStringSet*, for working with one or multiple DNA sequences. The *Biostrings* package contains additional classes for representing amino acid and general biological strings. The *BSgenome* and related packages (e.g., *BSgenome.Dmelanogaster.UCSC.dm3*) are used to represent whole-genome sequences. The following exercise explores these packages.

Exercise 7
The objective of this exercise is to calculate the GC content of the exons of a single gene, whose coordinates are specified by the `ex` *object of the previous exercise.*

Load the BSgenome.Dmelanogaster.UCSC.dm3 data package, containing the UCSC representation of D. melanogaster *genome assembly dm3.*

Extract the sequence name of the first gene of `ex`*. Use this to load the appropriate* D. melanogaster *chromosome.*

Use `Views` *to create views on to the chromosome that span the start and end coordinates of all exons.*

The SequenceAnalysis package defines a helper function `gcFunction` *(developed in a later exercise) to calculate GC content. Use this to calculate the GC content in each of the exons.*

Look at the helper function, and describe what it does.

Solution: Here we load the *D. melanogaster* genome, select a single chromosome, and create `Views` that reflect the ranges of the FBgn0002183.

```
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> nm <- as.character(unique(seqnames(ex[[1]])))
> chr <- Dmelanogaster[[nm]]
> v <- Views(chr, start=start(ex[[1]]), end=end(ex[[1]]))
```

Here is the helper function, available in the *SequenceAnalysis* package, to calculate GC content:

```
> gcFunction

function (x)
{
    alf <- alphabetFrequency(x, as.prob = TRUE)
    rowSums(alf[, c("G", "C")])
}
<environment: namespace:SequenceAnalysis>
```

The `gcFunction` is really straight-forward: it invokes the function `alphabetFrequency` from the *Biostrings* package. This returns a simple matrix of exon × nuclotiede probabilities. The row sums of the `G` and `C` columns of this matrix are the GC contents of each exon.

The subject GC content is

```
> subjectGC <- gcFunction(v)
```

3.3 Resources

There are extensive vignettes for *Biostrings* and *GenomicRanges* packages. A useful online resource is from Thomas Grike's group.

Table 6: Selected *Bioconductor* packages for sequence reads and alignments.

Package	Description
ShortRead	Defines the *ShortReadQ* class and functions for manipulating `fastq` files; these classes rely heavily on *Biostrings*.
GenomicRanges	*GappedAlignments* and *GappedAlignmentPairs* store single- and paired-end aligned reads.
Rsamtools	Provides access to BAM alignment and other large sequence-related files.
rtracklayer	Input and output of `bed`, `wig` and similar files

4 Reads and Alignments

The following sections introduce core tools for working with high-throughput sequence data; key packages for representing reads and alignments are summarized in Table 6. This section focus on the reads and alignments that are the raw material for analysis. Section 5 addresses statistical approaches to assessing differential representation in RNA-seq experiments. Section 6 outlines ChIP-seq analysis. Section 7 introduces resources for annotating sequences.

4.1 The *pasilla* data set

As a running example, we use the *pasilla* data set, derived from [2]. The authors investigate conservation of RNA regulation between *D. melanogaster* and mammals. Part of their study used RNAi and RNA-seq to identify exons regulated by Pasilla (*ps*), the *D. melanogaster* ortholog of mammalian NOVA1 and NOVA2. Briefly, their experiment compared gene expression as measured by RNAseq in S2-DRSC cells cultured with, or without, a 444bp dsRNA fragment corresponding to the *ps* mRNA sequence. Their assessment investigated differential exon use, but our worked example will focus on gene-level differences.

In this section we look at a subset of the *ps* data, corresponding to reads obtained from lanes of their RNA-seq experiment, and to the same reads aligned to a *D. melanogaster* reference genome. Reads were obtained from GEO and the Short Read Archive (SRA), and were aligned to the *D. melanogaster* reference genome *dm3* as described in the *pasilla* experiment data package.

4.2 Reads and the *ShortRead* package

Short read formats The Illumina GAII and HiSeq technologies generate sequences by measuring incorporation of florescent nucleotides over successive PCR cycles. These sequencers produce output in a variety of formats, but *FASTQ* is ubiquitous. Each read is represented by a record of four components:

```
@SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
+SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
```

The first and third lines (beginning with `@` and `+` respectively) are unique identifiers. The identifier produced by the sequencer typically includes a machine id followed by colon-separated information on the lane, tile, x, and y coordinate of the read. The example illustrated here also includes the SRA accession number, added when the data was submitted to the archive. The machine identifier could potentially be used to extract information about batch effects. The spatial coordinates (lane, tile, x, y) are often used to identify optical duplicates; spatial coordinates can also be used during quality assessment to identify artifacts of sequencing, e.g., uneven amplification across the flow cell, though these spatial effects are rarely pursued.

The second and fourth lines of the FASTQ record are the nucleotides and qualities of each cycle in the read. This information is given in 5' to 3' orientation as seen by the sequencer. A letter `N` in the sequence is used to signify bases that the sequencer was not able to call. The fourth line of the FASTQ record encodes the quality (confidence) of the corresponding base call. The quality score is encoded following one of several conventions, with the general notion being that letters later in the visible ASCII alphabet

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

are of lower quality; this is developed further below. Both the sequence and quality scores may span multiple lines.

Technologies other than Illumina use different formats to represent sequences. Roche 454 sequence data is generated by 'flowing' labeled nucleotides over samples, with greater intensity corresponding to longer runs of A, C, G, or T. This data is represented as a series of 'flow grams' (a kind of run-length encoding of the read) in Standard Flowgram Format (SFF). The *Bioconductor* package *R453Plus1Toolbox* has facilities for parsing SFF files, but after quality control steps the data are frequently represented (with some loss of information) as FASTQ. SOLiD technologies produce sequence data using a 'color space' model. This data is not easily read in to *R*, and much of the error-correcting benefit of the color space model is lost when converted to FASTQ; SOLiD sequences are not well-handled by *Bioconductor* packages.

Short reads in *R* FASTQ files can be read in to *R* using the `readFastq` function from the *ShortRead* package. Use this function by providing the path to a FASTQ file. There are sample data files available in the *SequenceAnalysisData* package, each consisting of 1 million reads from a lane of the Pasilla data set.

```
> fastqDir <- file.path(bigdata(), "fastq")
> fastqFiles <- dir(fastqDir, full=TRUE)
> fq <- readFastq(fastqFiles[1])
> fq
```

```
class: ShortReadQ
length: 1000000 reads; width: 37 cycles
```

The data are represented as an object of class *ShortReadQ*.

```
> head(sread(fq), 3)
```

```
  A DNAStringSet instance of length 3
    width seq
```

```
[1]    37 GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
[2]    37 GTTGTCGCATTCCTTACTCTCATTCGGGAATTCTGTT
[3]    37 GAATTTTTTGAGAGCGAAATGATAGCCGATGCCCTGA
```

```
> head(quality(fq), 3)
```

```
class: FastqQuality
quality:
  A BStringSet instance of length 3
    width seq
[1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
```

```
> head(id(fq), 3)
```

```
  A BStringSet instance of length 3
    width seq
[1]    58 SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
[2]    57 SRR031724.2 HWI-EAS299_4_30M2BAAXX:5:1:937:1157 length=37
[3]    58 SRR031724.4 HWI-EAS299_4_30M2BAAXX:5:1:1443:1122 length=37
```

The *ShortReadQ* class illustrates *class inheritance*. It extends the *ShortRead* class

```
> getClass("ShortReadQ")
```

```
Class "ShortReadQ" [package "ShortRead"]

Slots:

Name:       quality        sread           id
Class: QualityScore DNAStringSet   BStringSet

Extends:
Class "ShortRead", directly
Class ".ShortReadBase", by class "ShortRead", distance 2

Known Subclasses: "AlignedRead"
```

Methods defined on *ShortRead* are available for *ShortReadQ*.

```
> showMethods(class="ShortRead", where=getNamespace("ShortRead"))
```

For instance, the `width` can be used to demonstrate that all reads consist of 37 nucleotides.

```
> table(width(fq))
```

```
     37
1000000
```

The `alphabetByCycle` function summarizes use of nucleotides at each cycle in a (equal width) *ShortReadQ* or *DNAStringSet* instance.

```
> abc <- alphabetByCycle(sread(fq))
> abc[1:4, 1:8]

        cycle
alphabet   [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]
       A  78194 153156 200468 230120 283083 322913 162766 220205
       C 439302 265338 362839 251434 203787 220855 253245 287010
       G 397671 270342 258739 356003 301640 247090 227811 246684
       T  84833 311164 177954 162443 211490 209142 356178 246101
```

FASTQ files are getting larger. A very common reason for looking at data at this early stage in the processing pipeline is to explore sequence quality. In these circumstances it is often not necessary to parse the entire FASTQ file. Instead create a representative sample

```
> sampler <- FastqSampler(fastqFiles[1], 1000000)
> yield(sampler) # sample of 1000000 reads

class: ShortReadQ
length: 1000000 reads; width: 37 cycles
```

A second common scenario is to pre-process reads, e.g., trimming low-quality tails, adapter sequences, or artifacts of sample preparation. The *FastqStreamer* class can be used to 'stream' over the fastq files in chunks, processing each chunk independently.

ShortRead contains facilities for quality assessment of FASTQ files. Here we generate a report from a sample of 1 million reads from each of our files and display it in a web browser

```
> qas0 <- Map(function(fl, nm) {
+     fq <- FastqSampler(fl)
+     qa(yield(fq), nm)
+ }, fastqFiles,
+    sub("_subset.fastq", "", basename(fastqFiles)))
> qas <- do.call(rbind, qas0)
> rpt <- report(qas, dest=tempfile())
> browseURL(rpt)
```

A report from a larger subset of the experiment is available

```
> rpt <- system.file("GSM461176_81_qa_report", "index.html",
+                    package="SequenceAnalysis")
> browseURL(rpt)
```

Exercise 8
Use the helper function `bigdata` *(defined in the SequenceAnalysis package) and the* `file.path` *and* `dir` *functions to locate two fastq files from [2] (the files were obtained as described in the appendix and pasilla experiment data package.*

Input one of the fastq files using `readFastq` *from the ShortRead package.*

Use `alphabetFrequency` *to summarize the GC content of all reads (hint: use the* `sread` *accessor to extract the reads, and the* `collapse=TRUE` *argument to the* *`alphabetFrequency` function). Using the helper function `gcFunction` from the SequenceAnalysis package, draw a histogram of the distribution of GC frequencies across reads.*

Use `alphabetByCycle` to summarize the frequency of each nucleotide, at each cycle. Plot the results using `matplot`, from the graphics package.

As an advanced exercise, and if on Mac or Linux, use the parallel package and `mclapply` to read and summarize the GC content of reads in two files in parallel.

Solution: Discovery:

```
> dir(bigdata())
```

```
[1] "bam"   "fastq"
```

```
> fls <- dir(file.path(bigdata(), "fastq"), full=TRUE)
```

Input:

```
> fq <- readFastq(fls[1])
```

GC content:

```
> alf0 <- alphabetFrequency(sread(fq), as.prob=TRUE, collapse=TRUE)
> sum(alf0[c("G", "C")])
```

```
[1] 0.5457237
```

A histogram of the GC content of individual reads is obtained with

```
> gc <- gcFunction(sread(fq))
> hist(gc)
```

Alphabet by cycle:

```
> abc <- alphabetByCycle(sread(fq))
> matplot(t(abc[c("A", "C", "G", "T"),]), type="l")
```

Advanced (Mac, Linux only): processing on multiple cores.

```
> library(parallel)
> gc0 <- mclapply(fls, function(fl) {
+   fq <- readFastq(fl)
+   gc <- gcFunction(sread(fq))
+   table(cut(gc, seq(0, 1, .05)))
+ })
> ## simplify list of length 2 to 2-D array
> gc <- simplify2array(gc0)
> matplot(gc, type="s")
```

Exercise 9
Use `quality` *to extract the quality scores of the short reads. Interpret the encoding qualitatively.*

Convert the quality scores to a numeric matrix, using `as`*. Inspect the numeric matrix (e.g., using* `dim`*) and understand what it represents.*

Use `colMeans` *to summarize the average quality score by cycle. Use* `plot` *to visualize this.*

Solution:

```
> head(quality(fq))

class: FastqQuality
quality:
  A BStringSet instance of length 6
    width seq
[1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
[4]    37 IIIIIIIIIIIIIIIIIIIIIIII,II*E,&4HI++B
[5]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII&.$
[6]    37 III.IIIIIIIIIIIIIIIIIII%IIE(-EIH<IIII

> qual <- as(quality(fq), "matrix")
> dim(qual)

[1] 1000000      37

> plot(colMeans(qual), type="b")
```

4.3 Alignments and the *Rsamtools* package

Most down-stream analysis of short read sequences is based on reads aligned to reference genomes. There are many aligners available, including BWA [14, 13], Bowtie / Bowtie2 [12], and GSNAP; merits of these are discussed in the literature. There are also alignment algorithms implemented in *Bioconductor* (e.g., `matchPDict` in the *Biostrings* package, and the *Rsubread* package); `matchPDict` is particularly useful for flexible alignment of moderately sized subsets of data.

Alignment formats Most main-stream aligners produce output in SAM (text-based) or BAM format. A SAM file is a text file, with one line per aligned read, and fields separated by tabs. Here is an example of a single SAM line, split into fields.

```
> fl <- system.file("extdata", "ex1.sam", package="Rsamtools")
> strsplit(readLines(fl, 1), "\t")[[1]]

 [1] "B7_591:4:96:693:509"
 [2] "73"
 [3] "seq1"
```

Table 7: Fields in a SAM record. From http://samtools.sourceforge.net/samtools.shtml

Field	Name	Value
1	QNAME	Query (read) NAME
2	FLAG	Bitwise FLAG, e.g., strand of alignment
3	RNAME	Reference sequence NAME
4	POS	1-based leftmost POSition of sequence
5	MAPQ	MAPping Quality (Phred-scaled)
6	CIAGR	Extended CIGAR string
7	MRNM	Mate Reference sequence NaMe
8	MPOS	1-based Mate POSition
9	ISIZE	Inferred insert SIZE
10	SEQ	Query SEQuence on the reference strand
11	QUAL	Query QUALity
12+	OPT	OPTional fields, format TAG:VTYPE:VALUE

```
 [4] "1"
 [5] "99"
 [6] "36M"
 [7] "*"
 [8] "0"
 [9] "0"
[10] "CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG"
[11] "<<<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7"
[12] "MF:i:18"
[13] "Aq:i:73"
[14] "NM:i:0"
[15] "UQ:i:0"
[16] "H0:i:1"
[17] "H1:i:0"
```

Fields in a SAM file are summarized in Table 7. We recognize from the FASTQ file the identifier string, read sequence and quality. The alignment is to a chromosome 'seq1' starting at position 1. The strand of alignment is encoded in the 'flag' field. The alignment record also includes a measure of mapping quality, and a CIGAR string describing the nature of the alignment. In this case, the CIGAR is 36M, indicating that the alignment consisted of 36 `M`atches or mismatches, with no indels or gaps; indels are represented by `I` and `D`; gaps (e.g., from alignments spanning introns) by `N`.

BAM files encode the same information as SAM files, but in a format that is more efficiently parsed by software; BAM files are the primary way in which aligned reads are imported in to *R*.

Aligned reads in *R* The `readGappedAlignments` function from the *GenomicRanges* package reads essential information from a BAM file in to *R*. The result is an instance of the *GappedAlignments* class. The *GappedAlignments* class has been designed to allow useful manipulation of many reads (e.g., 20 million) under moderate memory requirements (e.g., 4 GB).

```
> alnFile <- system.file("extdata", "ex1.bam", package="Rsamtools")
> aln <- readGappedAlignments(alnFile)
> head(aln, 3)

GappedAlignments with 3 alignments and 0 elementMetadata cols:
      seqnames strand       cigar    qwidth     start       end     width
         <Rle>  <Rle> <character> <integer> <integer> <integer> <integer>
  [1]     seq1      +         36M        36         1        36        36
  [2]     seq1      +         35M        35         3        37        35
  [3]     seq1      +         35M        35         5        39        35
           ngap
      <integer>
  [1]         0
  [2]         0
  [3]         0
  ---
  seqlengths:
   seq1 seq2
   1575 1584
```

The `readGappedAlignments` function takes an additional argument, `param`, allowing the user to specify regions of the BAM file (e.g., known gene coordinates) from which to extract alignments.

A *GappedAlignments* instance is like a data frame, but with accessors as suggested by the column names. It is easy to query, e.g., the distribution of reads aligning to each strand, the width of reads, or the cigar strings

```
> table(strand(aln))

   +    -
1647 1624

> table(width(aln))

  30   31   32   33   34   35   36   38   40
   2   21    1    8   37 2804  285    1  112

> head(sort(table(cigar(aln)), decreasing=TRUE))

     35M      36M      40M      34M      33M 14M4I17M
    2804      283      112       37        6        4
```

Exercise 10
Use `bigdata`*,* `file.path` *and* `dir` *to obtain file paths to the BAM files. These are a subset of the aligned reads, overlapping just four genes.*

Input the aligned reads from one file using `readGappedAlignments`*. Explore the reads, e.g., using* `table` *or* `xtabs`*, to summarize which chromosome and strand the subset of reads is from.*

The object `ex` *created earlier contains coordinates of four genes. Use* `countOverlaps` *to first determine the number of genes an individual read aligns to, and then the number of uniquely aligning reads overlapping each gene. Since the RNAseq protocol was not strand-sensitive, set the strand of* `aln` *to* `*`*.*

Write the sequence of steps required to calculate counts as a simple function, and calculate counts on each file. On Mac or Linux, can you easily parallelize this operation?

Solution: We discover the location of files using standard *R* commands:

```
> fls <- dir(file.path(bigdata(), "bam"), ".bam$", full=TRUE)
> names(fls) <- sub("_.*", "", basename(fls))
```

Use `readGappedAlignments` to input data from one of the files, and standard *R* commands to explore the data.

```
> ## input
> aln <- readGappedAlignments(fls[1])
> xtabs(~seqnames + strand, as.data.frame(aln))
```

```
        strand
seqnames    -    +
   chr3L 5974 5402
   chrX  2283 2278
```

To count overlaps in regions defined in a previous exercise, load the regions.

```
> data(ex)              # from an earlier exercise
```

Many RNA-seq protocols are not strand aware, i.e., reads align to the plus or minus strand regardless of the strand on which the corresponding gene is encoded. Adjust the strand of the aligned reads to indicate that the strand is not known.

```
> strand(aln) <- "*"   # protocol not strand-aware
```

For simplicity, we are interested in reads that align to only a single gene. Count the number of genes a read aligns to...

```
> hits <- countOverlaps(aln, ex)
> table(hits)
```

```
hits
    0     1     2 
  772 15026   139 
```

and reverse the operation to count the number of times each region of interest aligns to a uniquely overlapping alignment.

```
> cnt <- countOverlaps(ex, aln[hits==1])
```

A simple function for counting reads is

```
> counter <-
+     function(filePath, range)
+ {
+     aln <- readGappedAlignments(filePath)
+     strand(aln) <- "*"
+     hits <- countOverlaps(aln, range)
+     countOverlaps(range, aln[hits==1])
+ }
```

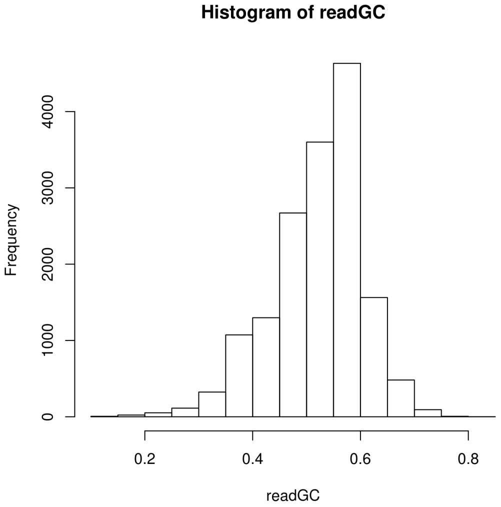


Figure 3: GC content in aligned reads

This can be applied to all files using `sapply`

```
> counts <- sapply(fls, counter, ex)
```

The counts in one BAM file are independent of counts in another BAM file. This encourages us to count reads in each BAM file in parallel, decreasing the length of time required to execute our program. On Linux and Mac OS, a straight-forward way to parallelize this operation is:

```
> if (require(parallel))
+     simplify2array(mclapply(fls, counter, ex))
```

The *GappedAlignments* class inputs only some of the fields of a BAM file, and may not be appropriate for all uses. In these cases the `scanBam` function in *Rsamtools* provides greater flexibility. The idea is to view BAM files as a kind of data base. Particular regions of interest can be selected, and the information in the selection restricted to particular fields. These operations are determined by the values of a *ScanBamParam* object, passed as the named `param` argument to `scanBam`.

Exercise 11
Consult the help page for `ScanBamParam`, and construct an object that restricts the information returned by a `scanBam` query to the aligned read DNA sequence. Your solution will use the `what` parameter to the `ScanBamParam` function.

Use the `ScanBamParam` object to query a BAM file, and calculate the GC content of all aligned reads. Summarize the GC content as a histogram (Figure 3).

Solution:

```
> param <- ScanBamParam(what="seq")
> seqs <- scanBam(fls[[1]], param=param)
> readGC <- gcFunction(seqs[[1]][["seq"]])
> hist(readGC)
```

5 RNA-seq

5.1 Varieties of RNA-seq

RNA-seq experiments typically ask about differences in trancription of genes or other features across experimental groups. The analysis of designed experiments is statistical, and hence an ideal task for *R*. The overall structure of the analysis, with tens of thousands of features and tens of samples, is reminiscent of microarray analysis; some insights from the microarray domain will apply, at least conceptually, to the analysis of RNA-seq experiments.

The most straight-forward RNA-seq experiments quantify abundance for known gene models. The known models are derived from reference databases, reflecting the accumulated knowledge of the community responsible for the data. The 'knownGenes' track of the UCSC genome browser represents one source of such data. A track like this describes, for each gene, the transcripts and exons that are expected based on current data. The *GenomicFeatures* package allows ready access to this information by creating a local database out of the track information. This data base of known genes is coupled with high throughput sequence data by counting reads overlapping known genes and modeling the relationship between treatment groups and counts.

A more ambitious approach to RNA-seq attempts to identify novel transcripts. This requires that sequenced reads be assembled into contigs that, presumably, correspond to expressed transcripts that are then located in the genome. Regions identified in this way may correspond to known transcripts, to novel arrangements of known exons (e.g., through alternative splicing), or to completely novel constructs. We will not address the identification of completely novel transcripts here, but will instead focus on the analysis of the designed experiments: do the transcript abundances, novel or otherwise, differ between experimental groups?

Bioconductor packages play a role in several stages of an RNA-seq analysis (Table 8; a more comprehensive list is under the RNAseq and HighThroughputSequencing BiocViews terms). The *GenomicRanges* infrastructure can be effectively employed to quantify known exon or transcript abundances. Quantified abundances are in essence a matrix of counts, with rows representing features and columns samples. The *edgeR* [19] and *DESeq* [1] packages facilitate analysis of this data in the context of designed experiments, and are appropriate when the questions of interest involve between-sample comparisons of relative abundance. The *DEXSeq* package extends the approach in *edgeR* and *DESeq* to ask about within-gene, between group differences in exon use, i.e., for a given gene, do groups differ in their exon use?

5.2 Differential expression with the *edgeR* package

RNA-seq differential representation experiments, like classical microarray experiments, consist of a single statistical design (e.g, comparing expression of samples assigned to 'Treatment' versus 'Control' groups) applied to each feature for which there are aligned reads. While one could naively perform simple tests (e.g., t-tests) on all features, it is much more informative to identify important aspects of RNAseq experiments, and to take a flexible route through this part of the work flow. Key steps involve formulation of a model matrix to cap-

Table 8: Selected *Bioconductor* packages for RNA-seq analysis.

Package	Description
EDASeq	Exploratory analysis and QA; also *qrqc*, *ShortRead*.
edgeR, *DESeq*	Generalized Linear Models using negative binomial error.
DEXSeq	Exon-level differential representation.
goseq	Gene set enrichment tailored to RNAseq count data; also *limma*'s `roast` or `camera` after transformation with `voom`.
easyRNASeq	Workflow; also *ArrayExpressHTS*, *rnaSeqMap*, *oneChannelGUI*.
Rsubread	Alignment (Linux only); also *Biostrings* `matchPDict` for special-purpose alignments.

ture the experimental design, estimation of a test static to describe differences between groups, and calculation of a P value or other measure as a statement of statistical significance.

Counting and filtering An essential step is to arrive at some measure of gene representation amongst the aligned reads. A straight-forward and commonly used approach is to count the number of times a read overlaps exons. Nuance arises when a read only partly overlaps an exon, when two exons overlap (and hence a read appears to be 'double counted'), when reads are aligned with gaps and the gaps are inconsistent with known exon boundaries, etc. The `summarizeOverlaps` function in the *GenomicRanges* package provides facilities for implementing different count strategies, using the argument `mode` to determine the counting strategy. The result of `summarizeOverlaps` can easily be used in subsequent steps of an RNA-seq analysis.

Software other than *R* can also be used to summarize count data. An important point is that the desired input for downstream analysis is often raw count data, rather than normalized (e.g., reads per kilobase of gene model per million mapped reads) values. This is because counts allow information about uncertainty of estimates to propagate to later stages in the analysis.

The following exercise illustrates key steps in counting and filtering reads overlapping known genes.

Exercise 12

The SequenceAnalysisData package contains a data set `counts` *with pre-computed count data. Use the* `data` *command to load it. Create a variable* `grp` *to define the groups associated with each column, using the column names as a proxy for more authoritative metadata.*

Create a `DGEList` *object (defined in the edgeR package) from the count matrix and group information. Calculate relative library sizes using the* `calcNormFactors` *function.*

A lesson from the microarray world is to discard genes that cannot be informative (e.g., because of lack of variation), regardless of statistical hypothesis under evaluation. Filter reads to remove those that are represented at less than 1 per million mapped reads, in fewer than 2 samples.

As a sanity check and to provide confidence that subsequent analysis is worthwhile, use `plotMDS` *on the filtered reads to perform multi-dimensional scaling. Interpret the resulting plot.*

Solution: Here we load the data (a matrix of counts) and create treatment group names from the column names of the counts matrix.

```
> data(counts)
> dim(counts)

[1] 14470     7

> grps <- factor(sub("[1-4].*", "", colnames(counts)),
+                levels=c("untreated", "treated"))
> pairs <- factor(c("single", "paired", "paired",
+                   "single", "single", "paired", "paired"))
> pData <- data.frame(Group=grps, PairType=pairs,
+                     row.names=colnames(counts))
```

We use the *edgeR* package, creating a *DGEList* object from the count and group data. The `calcNormFactors` function estimates relative library sizes for use as offsets in the generalized linear model.

```
> library(edgeR)
> dge <- DGEList(counts, group=pData$Group)
> dge <- calcNormFactors(dge)
```

To filter reads, we scale the counts by the library sizes and express the results on a per-million read scale. This is done using the `sweep` function, dividing each column by it's library size and multiplying by `1e6`. We require that the gene be represented at a frequency of at least 1 read per million mapped (`m > 1`, below) in two or more samples (`rowSums(m > 1) >= 2`), and use this criterion to subset the *DGEList* instance.

```
> m <- sweep(dge$counts, 2, 1e6 / dge$samples$lib.size, `*`)
> ridx <- rowSums(m > 1) >= 2
> table(ridx)                         # number filtered / retained

ridx
FALSE  TRUE
 6476  7994

> dge <- dge[ridx,]
```

Multi-dimensional scaling takes data in high dimensional space (in our case, the dimension is equal to the number of genes in the filtered *DGEList* instance) and reduces it to fewer (e.g., 2) dimensions, allowing easier visual assessment. The plot is shown in Figure 4; that the samples separate into distinct groups provides some reassurance that the data differ according to treatment. Nonetheless, there appears to be considerable heterogeneity within groups. Any guess, perhaps from looking at the quality report generated earlier, what the within-group differences are due to?

```
> plotMDS(dge)
```

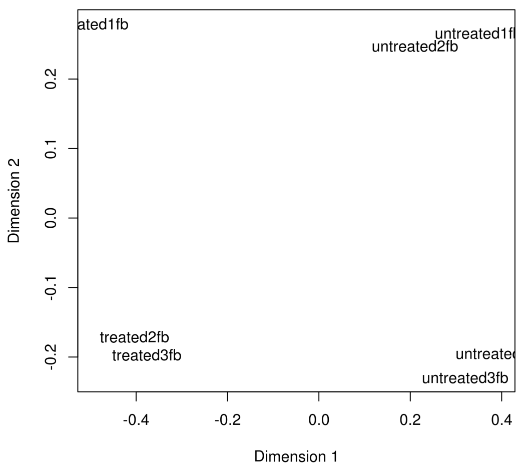


Figure 4: MDS plot of lanes from the Pasilla data set.

Experimental design In *R*, an experimental design is specified with the `model.matrix` function. The function takes as its first argument a `formula` describing the independent variables and their relationship to the response (counts), and as a second argument a `data.frame` containing the (phenotypic) data that the formula describes. A simple formula might read `~ 1 + Group`, which says that the response is a linear function involving an intercept (1) plus a term encoded in the variable `Group`. If (as in our case) `Group` is a factor, then the first coefficient (column) of the model matrix corresponds to the first level of `Group`, and subsequent terms correspond to *deviations* of each level from the first. If `Group` were *numeric* rather than *factor*, the formula would represent linear regressions with an intercept. Formulas are very flexible, allowing representation of models with one, two, or more factors as main effects, models with or without interaction, and with nested effects.

Exercise 13
To be more concrete, use the `model.matrix` *function and a formula involving* `Group` *to create the model matrix for our experiment.*

Solution: Here is the experimental design; it is worth discussing with your neighbor the interpretation of the `design` instance.

```
> (design <- model.matrix(~ Group, pData))

              (Intercept) Grouptreated
treated1fb              1            1
treated2fb              1            1
treated3fb              1            1
untreated1fb            1            0
untreated2fb            1            0
untreated3fb            1            0
```

```
untreated4fb            1            0
attr(,"assign")
[1] 0 1
attr(,"contrasts")
attr(,"contrasts")$Group
[1] "contr.treatment"
```

The coefficient (column) labeled 'Intercept' corresponds to the first level of `Group`, i.e., 'untreated'. The coefficient 'Grouptreated' represents the deviation of the treated group from untreated. Eventually, we will test whether the second coefficient is significantly different from zero, i.e., whether samples with a '1' in the second column are, on average, different from samples with a '0'. On the one hand, use of `model.matrix` to specify experimental design implies that the user is comfortable with something more than elementary statistical concepts, while on the other it provides great flexibility in the experimental design that can be analyzed.

Negative binomial error RNA-seq count data are often described by a negative binomial error model. This model includes a 'dispersion' parameter that describes biological variation beyond the expectation under a Poisson model. The simplest approach estimates a dispersion parameter from all the data. The estimate needs to be conducted in the context of the experimental design, so that variability between experimental factors is not mistaken for variability in counts. The square root of the estimated dispersion represents the coefficient of variation between biological samples. The following *edgeR* commands estimate dispersion.

```
> dge <- estimateTagwiseDisp(dge)
> mean(sqrt(dge$tagwise.dispersion))
```

```
[1] 0.18
```

This approach estimates a dispersion parameter that is specific to each tag. As another alternative, Anders and Huber [1] note that dispersion increases as the mean number of reads per gene decreases. One can estimate the relationship between dispersion and mean using `estimateGLMTrendedDisp` in *edgeR*, using a fitted relationship across all genes to estimate the dispersion of individual genes. Because in our case sample sizes (biological replicates) are small, gene-wise estimates of dispersion are likely imprecise. One approach is to moderate these estimates by calculating a weighted average of the gene-specific and common dispersion; `estimateGLMTagwiseDisp` performs this calculation, requiring that the user provides an *a priori* estimate of the weight between tag-wise and common dispersion.

Differential representation The final steps in estimating differential representation are to fit the full model; to perform the likelihood ratio test comparing the full model to a model in which one of the coefficients has been removed; and to summarize, from the likelihood ratio calculation, genes that are most differentially represented. The result is a 'top table' whose row names are the Flybase gene ids used to label the elements of the `ex` *GRangesList*.

Exercise 14
Use `glmFit` *to fit the general linear model. This function requires the input data* `dge`*, the experimental design* `design`*, and the estimate of dispersion.*

Use `glmLRT` *to form the likelihood ratio test. This requires the original data* `dge` *and the fitted model from the previous part of this question. Which coefficient of the design matrix do you wish to test?*

Create a 'top table' of differentially represented genes using `topTags`*.*

Solution: Here we fit a generalized linear model to our data and experimental design, using the tagwise dispersion estimate.

```
> fit <- glmFit(dge, design)
```

The fit can be used to calculate a likelihood ratio test, comparing the full model to a reduced version with the second coefficient removed. The second coefficient captures the difference between treated and untreated groups, and the likelihood ratio test asks whether this term contributes meaningfully to the overall fit.

```
> lrTest <- glmLRT(dge, fit, coef=2)
```

Here the `topTags` function summarizes results across the experiment.

```
> tt <- topTags(lrTest, n=10)
> tt[1:3,]
```

```
Coefficient:  Grouptreated
            logFC logCPM  LR   PValue      FDR
FBgn0039155  -4.7    6.0 542 5.9e-120 4.7e-116
FBgn0039827  -4.3    4.6 247  1.0e-55  4.0e-52
FBgn0029167  -2.2    8.2 211  6.6e-48  1.8e-44
```

As a 'sanity check', summarize the original data for the first several probes, confirming that the average counts of the treatment and control groups are substantially different.

```
> sapply(rownames(tt$table)[1:4],
+        function(x) tapply(counts[x,], pData$Group, mean))
```

```
          FBgn0039155 FBgn0039827 FBgn0029167 FBgn0034736
untreated        1576         554        6447         382
treated            64          31        1483          36
```

5.3 Additional steps in RNA-seq work flows

Annotation It is very easy to add annotations (mapping Entrez identifiers to gene names, KEGG or GO pathways, etc) to top tables; see Section 7.

Gene set enrichment Gene set enrichment approaches ask whether sets of genes are associated with a treatment group. Sets are typically defined to represent genes in particular biochemical or other pathways (e.g., from the KEGG or GO ontologies). Sets might be more generally defined, e.g., to represent chromosomal regions; the MSigDB (Molecular Signals data base) is one collection of diverse gene sets. Motivation for gene set analysis might be that pathways offer greater biological relevance or are more interpretabe than individual genes, or that the cumulative effect of several genes in a pathway may be statistically meaningful even though individual gene contributions are not.

Gene set enrichment was first developed for microarray data, where a variety of statistical approaches have been adopted; some of these are implemented in *Bioconductor*, e.g., under the 'Pathways' and 'GO' BiocViews terms, or in the *limma* package. One approach (implemented in the *GOstats* package) dichotomizes genes as 'significant' or not, and then uses a hypergeometric test (perhaps correcting for the hierarchical nature of some gene sets, e.g., in the GO classification) to assess whether significant genes are over-represented in each set. Another approach, developed in the *Category* package vignette, calculates the average of t-statistics of genes within a gene set and compares this with an appropriate null distribution.

Application of gene set enrichment approaches to studies of RNA-seq differential representation is conceptually similar to gene set enrichment in microarray analysis, but there are several important considerations [22]. Perhaps the most important is that long or highly expressed genes receive many hits, and hence are associated with greater statistical power. The *goseq* package uses a data base of known gene lengths to address this problem, as explored in the following exercise.

Exercise 15

Explore gene set analysis in the context of RNA-seq data through the goseq vignette. Can you think of alternative ways to address differences in statistical power associated with gene length or expression? Are there other nuances of RNA-seq data sets that should be taken into consideration?

In discussing gene set tests, Goeman and Buhlmann [6] distinguish between 'competitive' and 'self-contained' null hypotheses, and between 'gene' and 'subject' sampling to calculate P values.

A competitive test compares differential expression of a gene set by comparison to the complement of that gene set (e.g., hypergeommetric tests on $2x2$ tables of 'differentially expressed' versus 'not differentially expressed' genes and 'in gene set' versus 'not in gene set'). A self-contained test compares differential expression to a fixed standard independent of genes outside the gene set (e.g., sum of t statistics for genes in a set). Goeman and Buhlmann argue that self-contained tests have greater biological relevance and provide a natural extension of single-gene tests in a way that competitive tests do not.

Gene sampling assesses significance by permuting labels attached to genes (as in the hypergeommetric test), whereas subject sampling permutes subject labels and is much more consistent with standard statistical practice.

The *goseq* test is an example of a competitive test using gene sampling to assess significance. Alternative approaches for RNA-seq data are not well-developed. One possibility suggested by Smyth[1] is to use *limma*'s `voom` function to transform data to an approximate continuous scale, and use results in more familiar gene set analysis.

Exercise 16
As an advanced exercise, explore use of limma's `voom` *transformation and the* `roast` *and* `camera` *gene set test.*

Clustering and classification An obstacle to applying clustering and classification to sequence data is the choice of a distance metric appropriate for count data. The *edgeR* illustrates one approach. In the `plotMDS` method for objects of class *DGEList* uses tagwise dispersion estimates as a measure of biological variation. Tagwise dispersion across all samples is used to identify the most differentially expressed genes. The square root of the tagwise common dispersion of pairs of samples is then used to identify the most differentially expressed genes. The method is described on `?plotMDS.DGEList`; a similar distance metric could be used in other clustering and classification algorithms.

Specific applications may suggest more refined solutions. For instance, Holmes et al. [8] study microbial communities, where the raw data are read counts of taxonomic groups × biological samples. A useful approach is to fit a mixture model to the count data, where the mixture is of a finite number of Dirichlet probability vectors. This is illustrated in the *DirichletMultinomial* package.

Differential exon usage The RNA-seq analysis outlined here has focused on perhaps the most straight-forward research question – assessment of gene-level differential expression. Sequence data can deliver higher-resolution insight into gene expression. For instance, one might hope to gain understanding of transcript-level differential representation, or identify differential representation of novel transcripts. Novel transcript identification has received a great deal of attention, but poses significant challenges beyond the scope of this workshop. Approaches to transcript differential representation have often tried to combine estimation of transcript abundance with assessment of differential representation. The approach explored here is different and, in some ways, more straight-forward; it is based on the *DEXSeq* package.

DEXSeq starts with known gene models, rather than trying to quantify abundance of unknown transcripts. The gene models consist of exons grouped into transcripts, and transcripts grouped into genes; they can be retrieved from, e.g., the UCSC genome browser or ENSEMBL (via biomaRt). The gene models are simplified to represent disjoint (non-overlapping) exonic regions. Such regions may belong to one or several transcripts. Reads aligning to each region are counted, and the counts used in an analysis that is similar to the gene-level analysis of *edgeR* or *DESeq*. The analysis is modified, though, to incorporate the gene model. Specifically, one asks whether there is a significant interaction between treatment and exon use – do treatments differ in how exons are represented? An affirmative answer provides indirect evidence that, since a particular exon is also represented differently between treatments, the transcript to which the exon belongs is represented differently. The approach uses the same

[1] `https://stat.ethz.ch/pipermail/bioconductor/2012-April/044899.html`

statistical machinery as the *edgeR* or *DESeq* packages, so makes efficient use of available data with appropriate assumptions about the error model.

Exercise 17
Explore exon usage through the DEXSeq vignette. Compare the merits and challenges of this approach with, e.g., direct estimation of transcript abundance and differential representation. How straight-forward is it to interpret results of a DEXSeq analysis, in terms of differential transcript use? Under what experimental circumstances might this approach be most profitably employed? Are there any avenues for simplifying the analysis, e.g., in simplifying known gene models to capture just the splicing events differentiating transcripts (a tough question; see [21]).

The forthcoming *SpliceGraph* package takes this a step further by summarizing gene models into graphs, with 'bubbles' representing alternative splicing events; this reduces the number of statistical tests (increasing count per edge and statistical power) while providing meaningful insight into the types of events (e.g., 'exon skip', 'alternative acceptor') occurring.

5.4 Resources

The *edgeR*, *DESeq*, and *DEXSeq* package vignettes provide excellent, extensive discussion of issues and illustration of methods for RNA-seq differential expression analysis.

Table 9: Selected *Bioconductor* packages for RNA-seq analysis.

Package	Description
qrqc	Quality assessment; also *ShortRead*, *chipseq*.
PICS	Peak calling, also *mosaics*, *chipseq*, *ChIPseqR*, *BayesPeak*, *nucleR* (nucleosome positioning).
ChIPpeakAnno	Peak annotation.
DiffBind	Multiple-experiment analysis.
MotIV	Motif identification and validation; also *rGADEM*.

6 ChIP-seq

6.1 Varieties of ChIP-seq

ChIP-seq and similar experiments combine chromosome immuno-precipitation (ChIP) with sequence analysis. The idea is that the ChIP protocol enriches genomic DNA for regions of interest, e.g., sites to which transcription factors are bound. The regions of interest are then subject to high throughput sequencing, the reads aligned to a reference genome, and the location of mapped reads ('peaks') interpreted as indicators of the ChIP'ed regions. Reviews include those by Park and colleagues [17, 7]; there is a large collection of peak-calling software, some features of which are summarized in Pepke et al. [18].

Initial stages in a ChIP-seq analysis differ from RNA-seq in several important ways. The ChIP protocol is more complicated and idiosyncratic than RNA-seq protocols, and the targets of ChIP more variable in terms of sequence and other characteristics. While RNA- and ChIP-seq use reads aligned to a reference genome, ChIP-seq protocols require that the aligned reads be processed to identify peaks, rather than simply counted in known gene regions.

Many early ChIP-seq studies focused on characterizing one or a suite of transcription factor binding sites across a small number of samples from one or two groups. The main challenge was to develop efficient peak-calling software, often tailored to the characteristics of the peaks of interest (e.g., narrow and well-defined CTCF binding sites, vs. broad histone marks). More comprehensive studies draw from multiple samples, e.g., in the ENCODE project [10, 16]. Decreasing sequence costs and better experimental and data analytic protocols mean that these larger-scale studies are increasingly accessible to individual investigators. Peak-calling in this kind of study represents an initial step, but interpretting analyses derived from multiple samples present significant analytic challenges. *Bioconductor* packages play a role in several stages of a ChIP-seq analysis. (Table 9; a more comprehensive list is under the ChIPseq and HighThroughputSequencing BiocViews terms). The *ShortRead* package can provide a quality assessment report of reads. Following alignment, the *chipseq* package can be used, in conjunction with *ShortRead* and *GenomicRanges*, to identify enriched regions in a statistically informed and flexible way. *DiffBind* provides facilities for comprehensive analysis of experiments with multiple ChIP-seq samples. The *ChIPpeakAnno* package assists in annotating peaks in terms of known genes and other genomic features. Pattern matching in *Biostrings* and specialized packages such as *MotIV* can assist in motif identification.

Our attention is on analyzing multiple samples from a single experiment, and identifying and annotating peaks. We start with a typical work flow re-iterating key components in an exploration of data from the ENCODE project. An 'advanced' section then illustrates how exploratory or novel ChIP-seq algorithms can be implemented in *Bioconductor*. The chapter concludes with more downstream analysis, including motif discovery and annotation.

6.2 Initial Work Flow

We use data from GEO accession GSE30263, representing ENCODE CTCF binding sites. CTCF is a zinc finger transcription factor. It is a sequence specific DNA binding protein that functions as an insulator, blocking enhancer activity, and possibly the spread of chromatin structure. The original analysis involved Illumina ChIP-seq and matching 'input' lanes of 1 or 2 replicates from many cell lines. The GEO accession includes BAM files of aligned reads, in addition to tertiary files summarizing identified peaks. We focus on 15 cell lines aligned to hg19.

Quality assessment

Exercise 18
As a precursor to analysis, it is prudent to perform an overall quality assessment of the data; The SequenceAnalysisData package contains a quality assessment report generated from the BAM files. View this report. Are there indications of batch or other systematic effects in the data?

Solution: Here we visit the QA report.

```
> rpt <- system.file("GSE30263_qa_report", "index.html",
+               package="SequenceAnalysis")
> if (interactive())
+     browseURL(rpt)
```

Samples 1-15 correspond to replicate 1, 16-26 to replicate 2, and 27 through 41 the 'input' samples. Notice that overall nucleotide frequencies fall into three distinct groups, and that samples 1-11 differ from the other input samples. The 'Depth of Coverage' portion of the report is particularly relevant for an early assessment of ChIP-seq experiments.

Alignment and peak calling (3rd party) The main computational stages in the original work flow involved alignment using Bowtie, followed by peak identification using an algorithm ('HotSpots', [20]) originally developed for lower-throughput methodologies. We collated the output files from the original analysis with a goal of enumerating all peaks from all files, but collapsing the coordinates of sufficiently similar peaks to a common location. The *DiffBind* package provides a formalism with which to do these operations. Here we load the data as an *R* object `stam` (an abbreviation for the lab generating the data).

```
> stamFile <-
+     system.file("data", "stam.Rda", package="SequenceAnalysisData")
> load(stamFile)
> stam
```

```
class: SummarizedExperiment
dim: 369674 96
exptData(0):
assays(2): Tags PVals
rownames: NULL
rowData values names(0):
colnames(96): A549_1 A549_2 ... Wi38_1 Wi38_2
colData names(10): CellLine Replicate ... PeaksDate PeaksFile
```

Data exploration

Exercise 19
Explore `stam`*. Tabulate the number of peaks represented 1, 2, . . . , 96 times. We expect replicates to have similar patterns of peak representation; do they?*

Solution: Load the data and display the *SummarizedExperiment* instance. The `colData` summarizes information about each sample, the `rowData` about each peak. Use `xtabs` to summarize `Replicate` and `CellLine` representation within `colData(stam)`.

```
> head(colData(stam), 3)

DataFrame with 3 rows and 10 columns
             CellLine Replicate   TotTags  TotPeaks      Tags     Peaks
          <character>  <factor> <integer> <integer> <numeric> <numeric>
A549_1           A549         1   1857934     50144   1569215     43119
A549_2           A549         2   2994916     77355   2881475     73062
Ag04449_1     Ag04449         1   5041026     81855   4730232     75677
            FastqDate FastqSize  PeaksDate
               <Date> <numeric>     <Date>
A549_1     2011-06-25       463 2011-06-25
A549_2     2011-06-25       703 2011-06-25
Ag04449_1  2010-10-22       368 2010-10-22
                                                PeaksFile
                                              <character>
A549_1        wgEncodeUwTfbsA549CtcfStdPkRep1.narrowPeak.gz
A549_2        wgEncodeUwTfbsA549CtcfStdPkRep2.narrowPeak.gz
Ag04449_1 wgEncodeUwTfbsAg04449CtcfStdPkRep1.narrowPeak.gz

> head(rowData(stam), 3)

GRanges with 3 ranges and 0 elementMetadata cols:
      seqnames         ranges strand
         <Rle>      <IRanges>  <Rle>
  [1]     chr1 [10100, 10370]      *
  [2]     chr1 [15640, 15790]      *
  [3]     chr1 [16100, 16490]      *
  ---
  seqlengths:
        chr1      chr2      chr3      chr4 ...     chr22      chrX      chrY
   249250621 243199373 198022430 191154276 ...  51304566 155270560  59373566
```

```
> xtabs(~Replicate + CellLine, colData(stam))[,1:5]

         CellLine
Replicate A549 Ag04449 Ag04450 Ag09309 Ag09319
        1    1       1       1       1       1
        2    1       1       1       1       1
```

Extract the Tags matrix from the assays. This is a standard *R matrix*. Test which matrix elements are non-zero, tally these by row, and summarize the tallies. This is the number of times a peak is detected, across each of the samples

```
> m <- assays(stam)[["Tags"]] > 0 # peaks detected...
> peaksPerSample <- table(rowSums(m))
> head(peaksPerSample)

     1      2      3      4      5      6 
174574  35965  18939  12669   9143   7178 

> tail(peaksPerSample)

   91    92    93    94    95    96 
 1226  1285  1542  2082  2749 14695 
```

To explore similarity between replicates, extract the matrix of counts. Transform the counts using the asinh function (a log-like transform, except near 0; are there other methods for transformation?), and use the 'correlation' distance (cor.dist, from *bioDist*) to measure similarity. Cluster these using a hierarchical algorithm, via the hclust function.

```
> library(bioDist)                     # for cor.dist
> m <- asinh(assays(stam)[["Tags"]]) # transformed tag counts
> d <- cor.dist(t(m))                  # correlation distance
> h <- hclust(d)                       # hierarchical clustering
```

Plot the result, as in Figure 5.

```
> plot(h, cex=.8, ann=FALSE)
```

6.2.1 Peak calling with *R* / *Bioconductor* (advanced)

The following illustrates basic ChIP-seq in *R*. It is likely that these would be used either in an exploratory way, or as foundations for developing work flows tailored to particular ChIP experiments. We work through this section by developing functions for different parts of the work flow. Functions are applied to examples in an exercise at the end of this section.

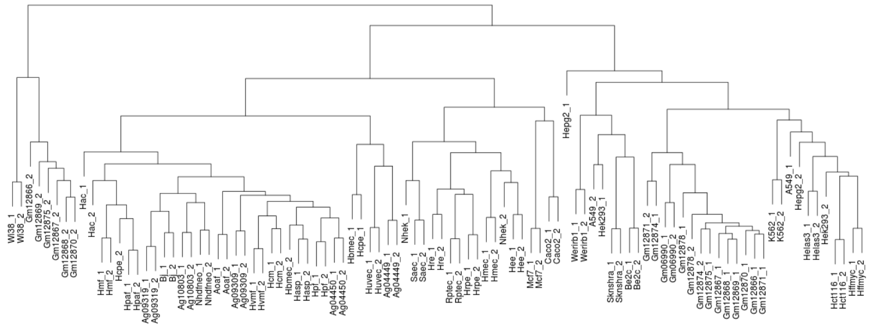


Figure 5: Hierarchical clustering of ENCODE samples.

Data input and pre-processing Here we develop our own function, named `chipPreprocess`, to pre-process a single sample lane. The work flow starts with data input. We suppose an available `bamFile`, with a *ScanBamParam* object `param` defined to select regions we are interested in; thee `bamFile` and `param` objects are defined later.

```
> aln <- readGappedAlignments(bamFile, param=param)
> seqlevels(aln) <-  names(bamWhich(param))
> aln <- as(aln, "GRanges")
```

We use `readGappedAlignments` followed by coercion to a *GRanges* object as a convenient way to retrieve a minimal amount of data from the BAM file, and to manage reads whose alignments include indels; `Rsamtools::scanBam` is a more flexible alternative. The `seqlevels` are adjusted to contain just the levels we are interested in, rather than all levels in the BAM file (the default returned by `readGappedAlignments`).

Sequence work flows typically filter reads to remove those that are optical duplicates or otherwise flagged as invalid by the manufacturer. Many work flows do not handle reads aligning to multiple locations in the genome. ChIP-seq experiments often eliminate reads that are duplicated in the sense that more than one read aligns to the same chromosome, strand, and start position; this acknowledges artifacts of sample preparation. These filters are handled by different stages in a typical work flow – flagging optical duplicates and otherwise suspect reads by the manufacturer or upstream software (illustrated in an exercise, below); discarding multiply aligning reads by the aligner (in our case, using the `-m` and `-n` options in *Bowtie*); and discarding duplicates as a pre-processing step. Simple alignment de-duplication is

```
> aln <- aln[!duplicated(aln)]
```

It is common to estimate fragment length (e.g., via the 'correlation' method [11], implemented in the *chipseq* package) and extend the 5' tags by the estimated length.

```
> fraglen <- estimate.mean.fraglen(aln, method="correlation")
> aln <- resize(aln, width=fraglen)
```

The end result can be summarized as a 'coverage vector' describing the number of (extended) reads at each location in the genome; a run length encoding is an efficient representation of this.

```
> coverage(aln)
```

These pre-processing steps can be summarized as a simple work-flow.

```
> chipPreprocess <- function(bamFile, param) {
+     aln <- readGappedAlignments(bamFile, param = param)
+     seqlevels(aln) <- names(bamWhich(param))
+     aln <- as(aln, "GRanges")
+     aln <- aln[!duplicated(aln)]
+     fraglen <- estimate.mean.fraglen(aln, method = "correlation")
+     aln <- resize(aln, width = fraglen)
+     coverage(aln)
+ }
```

Peak identification We now develop a function, `findPeaks`, to identify peaks. The coverage vector is a very useful representation of the data, and numerous peak discovery algorithms can be implemented on top of it. The *chipseq* package implements a straight-forward approach. The first step uses the distribution of singleton and doubleton islands to estimate a background Poisson noise distribution, and hence to identify a threshold island elevation above which peaks can be called at a specified false discovery rate.

```
> cutoff <- round(peakCutoff(cvg, fdr.cutoff=0.001))
```

Peaks are easily identified using `slice`

```
> slice(cvg, lower = cutoff)
```

resulting in a peak-finding work flow

```
> findPeaks <- function(cvg) {
+     cutoff <- round(peakCutoff(cvg, fdr.cutoff = 0.001))
+     slice(cvg, lower = cutoff)
+ }
```

Exercise 20
Walk through the work flow, from BAM file to called peaks, using the provided BAM files. These are from the Ag09319 cell line, CTCF replicate 1 and input lanes, filtered to include only reads from chromosome 6. Compare peaks found in the ChIP and Input lanes, and in the MACS analysis. It is possible to pick up the analysis after pre-processing by loading the `cvgs` *object. It can be very helpful to explore the data along the way; see the chipseq vignette for ideas.*

Solution: Specify the location of the BAM files, and the location where the coverage vectors will be saved.

```
> bamDir <- character()        # TODO: read BAM file from...
> cvgsSaveFile <- character()  # TODO: save coverage file to...
```

Storing the coverage vectors represents a check-pointing strategy, making it easy to resume an analysis if interrupted.

```
> library(GenomicRanges)
> bamFiles <- c(ChIP=file.path(bamDir,
+                 "wgEncodeUwTfbsAg09319CtcfStdAlnRep1.bam"),
+               Input=file.path(bamDir,
+                 "wgEncodeUwTfbsAg09319InputStdAlnRep1.bam"))
> stopifnot(all(file.exists(bamFiles)))
```

Create a `ScanBamParam` object specifying the regions of interest and other restrictions on reads to be input.

```
> chr6len <- scanBamHeader(bamFiles)[[1]][["targets"]][["chr6"]]
> param <- ScanBamParam(which=GRanges("chr6", IRanges(1, chr6len)),
+                       what=character(),
+                       flag=scanBamFlag(isDuplicate=FALSE,
+                         isValidVendorRead=TRUE))
```

Process each BAM file using `lapply`, and save the result.

```
> cvgs <- lapply(bamFiles, chipPreprocess, param)
> save(cvgs, cvgsSaveFile)
```

Load the saved coverage file, and find peaks using the simple approach outlined above.

```
> library(chipseq)
> cvgsFile <- system.file("data", "chipseq_chr6_cvgs.Rda",
+                         package="SequenceAnalysisData")
> stopifnot(file.exists(cvgsFile))
> load(cvgsFile)                          # previously saved
> peaks <- lapply(cvgs, findPeaks)
```

Compare the peaks using *GRanges* commands (e.g. convert the peaks to `IRanges` instances and use `countOverlaps` to identify peaks in common between the ChIP and Input lanes), and the `diffPeakSummary` function from the *chipseq* package. Compare the peaks to those found in the `stam` object.

```
> chip <- as(peaks[["ChIP"]][["chr6"]], "IRanges")
> inpt <- as(peaks[["Input"]][["chr6"]], "IRanges")
> table(countOverlaps(inpt, chip))

  0   1   2
635  19   3

> table(countOverlaps(chip, inpt))

   0    1    2
5534   23    1

> stamFile <- system.file("data", "stam.Rda",
+                 package="SequenceAnalysisData")
> load(stamFile)
> stam0 <- stam[,"Ag09319_1"]
> idx <- seqnames(rowData(stam0)) == "chr6" &
+            assays(stam0)[["Tags"]] != 0
> rng <- ranges(rowData(stam0))[as.logical(idx)]
> table(countOverlaps(chip, rng))
```

```
   0    1    2    3
 811 4689   56    2
```

Our naive analysis finds many of the peaks identified by a more comprehensive algorithm.

6.3 Comparison of multiple experiments: *DiffBind*

Exercise 21
Explore a ChIP-seq work flow through the DiffBind vignette.

6.4 Working with called peaks

Motifs Transcription factors and other common regulatory elements often target specific DNA sequences ('motifs'). These are often well-characterized, and can be used to help identify, *a priori*, regions in which binding is expected. Known binding motifs may also be used to identify promising peaks identified using *de novo* peak discovery methods like *MACS*. This section explores use of known binding motifs to characterize peaks; packages such as *MotIV* can assist in motif discovery.

The JASPAR data base curates known binding motifs obtained from the literature. A binding motif is summarized as a *position weight matrix* PWM or *position frequency matrix* PFM. Rows of a PWM correspond to nucleotides, columns to positions, and entries to the probability of the nucleotide at that position. Each start position in a reference sequence can be compared and scored for similarity to the PWM, and high-scoring positions retained. A PFM is a similar representation, but with entries corresponding to a count of the times a nucleotide is observed.

Exercise 22
The objective of this exercise is to identify occurrences of the CTCF motif on chromosome 1 of H. sapiens.

Load needed packages. Biostrings can represent a PWM and score a reference sequence. The BSgenome.Hsapiens.UCSC.hg19 package contains the hg19 build of H. sapiens, *retrieved from the UCSC genome browser. seqLogo and lattice are used for visualization.*

Retrieve the PWM for CTCF, with JASPAR id MA0139.1.pfm, using the helper function `getJASPAR` defined in the SequenceAnalysis package. Visualize the PWM using `seqLogo`.

Solution: Load the packages:

```
> library(Biostrings)
> library(BSgenome.Hsapiens.UCSC.hg19)
> library(seqLogo)
> library(lattice)
```

Retrieve the position weight matrix for CTCF, and display the PWM:

```
> pwm <- getJASPAR("MA0139.1.pfm") # SequenceAnalysis::getJASPAR
> seqLogo(scale(pwm, FALSE, colSums(pwm)))
```

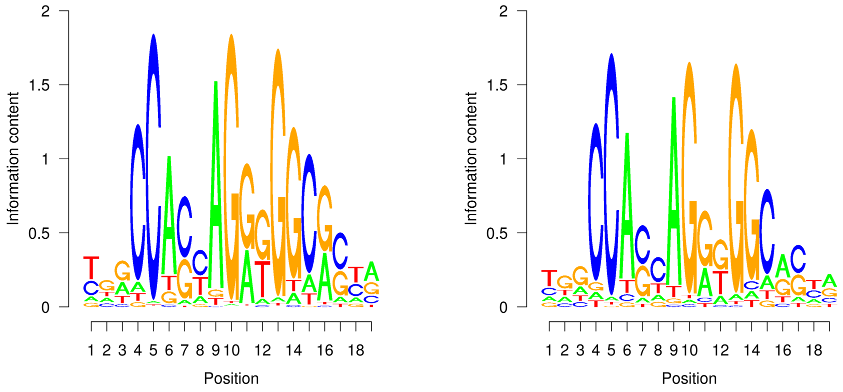


Figure 6: CTCF position weight matrix from JASPAR (left) and on the plus strand of chr1 (hits within 80% of maximum score, right).

Exercise 23

Use `matchPWM` to score the plus strand of chr1 for the CTCF PWM. Visualize the distribution of scores using, e.g., `densityplot`, and summarize the high-scoring matches (using `consensusMatrix`) as a `seqLogo`.

As an additional exercise, work up a short code segment to apply the PWM to both strands (see `?PWM` for some hints) and to all chromosomes.

Solution: Chromosome 1 can be loaded with `Hsapiens[["chr1"]]`; `matchPWM` returns a 'view' of the high-scoring locations matching the PWM. Scores are retrieved from the PWM and hits using `PWMscoreStartingAt`.

```
> chrid <- "chr1"
> hits <-matchPWM(pwm, Hsapiens[[chrid]]) # '+' strand
> scores <- PWMscoreStartingAt(pwm, subject(hits), start(hits))
```

The distribution of scores can be visualized with, e.g., `densityplot` from the *lattice* package.

```
> densityplot(scores, xlim=range(scores), pch="|")
```

`consensusMatrix` applied to the views in `hits` returns a position frequency matrix; this can be plotted as a logo, with the result in Figure 6. Reassuringly, the found sequences have a logo very similar to the expected.

```
> cm <- consensusMatrix(hits)[1:4,]
> seqLogo(makePWM(scale(cm, FALSE, colSums(cm))))
```

Exercise 24

We might expect that peaks found using de novo *techniques like MACS would be enriched for motifs identified for the known PWM. What fraction of our high-scoring positions are in the peaks in the `stam` object? What technical and biological issues might cloud this result?*

Solution:

```
> roi <- GRanges(chrid, ranges(hits), "+")
> seqinfo(roi) <- seqinfo(Hsapiens)
> sum(roi %in% rowData(stam)) / length(roi)
```

```
[1] 0.55
```

Annotation For an introduction to annotation resources in *Bioconductor*, see Section 7; the *ChIPpeakAnno* contributed package provides convenient ways to annotate ChIP-seq experiments.

Exercise 25
Annotating ChIP peaks is straight-forward. Load the ENCODE summary data, select the peaks found in all samples, and use the center of these peaks as a proxy for the true ChIP binding site. Use the transcript data base for the UCSC Known Genes track of hg19 as a source for transcripts and transcription start sites (TSS). Use `nearest` *to identify the TSS that is nearest each peak, and calculate the distance between the peak and TSS; measure distance taking account of the strand of the transcript, so that peaks 5' of the TSS have negative distance. Summarize the locations of the peaks relative to the TSS.*

Solution: Read in the ENCODE ChIP peaks for all cell lines.

```
> stamFile <-
+     system.file("data", "stam.Rda", package="SequenceAnalysisData")
> load(stamFile)
```

Identify the rows of `stam` that have non-zero counts for all cell lines, and extract the corresponding ranges:

```
> ridx <- rowSums(assays(stam)[["Tags"]] > 0) == ncol(stam)
> peak <- rowData(stam)[ridx]
```

Select the center of the ranges of these peaks, as a proxy for the ChIP binding site:

```
> peak <- resize(peak, width=1, fix="center")
```

Obtain the TSS from the *TxDb.Hsapiens.UCSC.hg19.knownGene* using the `transcripts` function to extract coordinates of each transcript, and `resize` to a width of 1 for the TSS; does this do the right thing for transcripts on the plus and on the minus strand?

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> tx <- transcripts(TxDb.Hsapiens.UCSC.hg19.knownGene)
> tss <- resize(tx, width=1)
```

The `nearest` function returns the index of the nearest `subject` to each `query` element; the distance between peak and nearest TSS is thus

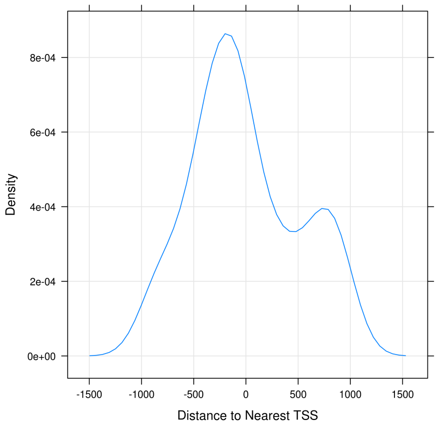


Figure 7: Distance to nearest TSS amongst conserved peaks

```
> idx <- nearest(peak, tss)
> sgn <- as.integer(ifelse(strand(tss)[idx] == "+", 1, -1))
> dist <- (start(peak) - start(tss)[idx]) * sgn
```

Here we summarize the distances as a simple table and density plot, focusing on binding sites within 1000 bases of a transcription start site; the density plot is in Figure 7.

```
> bound <- 1000
> ok <- abs(dist) < bound
> dist <- dist[ok]
> table(sign(dist))
```

```
 -1   1
669 417
```

```
> print(densityplot(dist[ok], type="g", xlab="Distance to Nearest TSS"))
```

The distance to transcript start site is a useful set of operations, so let's make it a re-usable function

```
> distToTss <-
+     function(peak, tx)
+ {
+     peak <- resize(peak, width=1, fix="center")
+     tss <- resize(tx, width=1)
+     idx <- nearest(peak, tss)
+     sgn <- as.numeric(ifelse(strand(tss)[idx] == "+", 1, -1))
+     (start(peak) - start(tss)[idx]) * sgn
+ }
```

Exercise 26
As an additional exercise, extract the sequences of all conserved peaks on 'chr6'. Do this using the BSgenome.Hsapiens.UCSC.hg19 package and `getSeq` *function. Use* `matchPWM` *to find sequences with a strong match to the JASPAR CTCF PWM motif, and plot the density of distances to nearest transcription start site for those with and without a match. What strategies are available for motif discovery?*

Solution: Here we select peaks on chromosome 6, and extract the DNA sequences corresponding to these peaks.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> ridx <- rowSums(assays(stam)[["Tags"]] > 0) == ncol(stam)
> ridx <- ridx & (seqnames(rowData(stam)) == "chr6")
> pk6 <- rowData(stam)[ridx]
> seqs <- getSeq(Hsapiens, pk6, as.character=FALSE)
> head(seqs, 3)

  A DNAStringSet instance of length 3
    width seq
[1]   311 CAGGGAGACTTGGGAAGGCTTCACGAAGGAGGGT...ACCCAACTCCTAAGCGTCACACATATAATCCTG
[2]   331 GCTAATAATTTACCATGAAGTAACAACTTTTCAC...TTTCCTAGGCAGCGAATTTAAGGGTAATGATCA
[3]   751 GTAAAGAATGGACTGACTTAAAGGCAGATGGAAT...AATCAAACAAGACAAAGAATCTTCGTGGCCACA
```

`matchPWM` operates on one DNA sequence at a time, so we arrange to search for the PWM on each sequence using `lapply`. We identify sequences with a match by testing the length of the returned object, and use this to create a density plot.

```
> hits <- lapply(seqs, matchPWM, pwm=pwm)
> hasPwmMatch <- sapply(hits, length) > 0
> dist <- distToTss(pk6, tx)
> ok <- abs(dist) < bound
> df <- data.frame(Distance = dist[ok], HasPwmMatch = hasPwmMatch[ok])
> print(densityplot(~Distance, group=HasPwmMatch, df,
+     plot.points=FALSE, type="g",
+     auto.key=list(
+       columns=2,
+       title="Has Position Weight Matrix?",
+       cex.title=1),
+     xlab="Distance to Nearest Tss"))
```

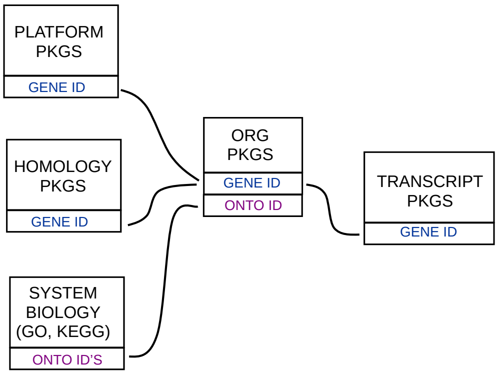


Figure 8: Annotation Packages: the big picture

7 Annotation

Bioconductor provides extensive annotation resources, summarized in Figure 8. These can be *gene*-, or *genome*-centric. Annotations can be provided in packages curated by *Bioconductor*, or obtained from web-based resources. Gene-centric *AnnotationDbi* packages include:

- Organism level: e.g. *org.Mm.eg.db*.
- Platform level: e.g. *hgu133plus2.db*, *hgu133plus2.probes*, *hgu133plus2.cdf*.
- Homology level: e.g. *hom.Dm.inp.db*.
- System biology level: *GO.db*, *KEGG.db*, *Reactome.db*.

Examples of genome-centric packages include:

- *GenomicFeatures*, to represent genomic features, including constructing reproducible feature or transcript data bases from file or web resources.
- Pre-built transcriptome packages, e.g. *TxDb.Hsapiens.UCSC.hg19.knownGene* based on the *H. sapiens* UCSC hg19 knownGenes track.
- *BSgenome* for whole genome sequence representation and manipulation.
- Pre-built genomes, e.g., *BSgenome.Hsapiens.UCSC.hg19* based on the *H. sapiens* UCSC hg19 build.

Web-based resources include

- *biomaRt* to query biomart resource for genes, sequence, SNPs, and etc.
- *rtracklayer* for interfacing with browser tracks, especially the UCSC genome browser.

7.1 Gene-centric annotations with *AnnotationDbi*

Organism-level ('org') packages contain mappings between a central identifier (e.g., Enterz gene ids) and other identifiers (e.g. GenBank or Uniprot accession number, RefSeq id, etc.). The name of an org package is always of the form *org.<Sp>.<id>.db* (e.g. *org.Sc.sgd.db*) where *<Sp>* is a 2-letter abbreviation of the organism (e.g. `Sc` for *Saccharomyces cerevisiae*) and *<id>* is an abbreviation (in lower-case) describing the type of central identifier (e.g. `sgd` for gene identifiers assigned by the *Saccharomyces* Genome Database, or `eg` for Entrez gene ids). The "How to use the '.db' annotation packages" vignette in the *AnnotationDbi* package (org packages are only one type of ".db" annotation packages) is a key reference. The '.db' and most other *Bioconductor* annotation packages are updated every 6 months.

Annotation packages contain an object named after the package itself. These objects are collectively called *AnnotationDb* objects, with more specific classes named *OrgDb*, *ChipDb* or *TranscriptDb* objects. Methods that can be applied to these objects include `cols`, `keys`, `keytypes` and `select`.

Exercise 27
What is the name of the org package for Drosophila*? Load it. Display the OrgDb object for the org.Dm.eg.db package. Use the `cols` method to discover which sorts of annotations can be extracted from it.*

Use the `keys` method to extract UNIPROT identifiers and then pass those keys in to the `select` method in such a way that you extract the SYMBOL (gene symbol) and KEGG pathway information for each.

Use `select` to retrieve the ENTREZ and SYMBOL identifiers of all genes in the KEGG pathway `00310`.

Solution: The *OrgDb* object is named `org.Dm.eg.db`.

```
> cols(org.Dm.eg.db)

 [1] "ENTREZID"     "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"
 [6] "CHRLOCEND"    "ENZYME"       "MAP"          "PATH"         "PMID"
[11] "REFSEQ"       "SYMBOL"       "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"
[16] "ENSEMBLTRANS" "GENENAME"     "UNIPROT"      "GO"           "EVIDENCE"
[21] "ONTOLOGY"     "FLYBASE"      "FLYBASECG"    "FLYBASEPROT"

> keytypes(org.Dm.eg.db)

 [1] "ENTREZID"     "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"
 [6] "CHRLOCEND"    "ENZYME"       "MAP"          "PATH"         "PMID"
[11] "REFSEQ"       "SYMBOL"       "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"
[16] "ENSEMBLTRANS" "GENENAME"     "UNIPROT"      "GO"           "EVIDENCE"
[21] "ONTOLOGY"     "FLYBASE"      "FLYBASECG"    "FLYBASEPROT"

> uniprotKeys <- head(keys(org.Dm.eg.db, keytype="UNIPROT"))
> cols <- c("SYMBOL", "PATH")
> select(org.Dm.eg.db, keys=uniprotKeys, cols=cols, keytype="UNIPROT")

  UNIPROT  SYMBOL  PATH
1  Q8IRZ0  CG3038  <NA>
2  Q95RP8  CG3038  <NA>
3  Q95RU8     G9a 00310
4  Q9W5H1 CG13377  <NA>
5  P39205     cin  <NA>
6  Q24312     ewg  <NA>
```

Selecting UNIPROT and SYMBOL ids of KEGG pathway 00310 is very similar:

```
> kegg <- select(org.Dm.eg.db, "00310", c("UNIPROT", "SYMBOL"), "PATH")
> nrow(kegg)

[1] 32

> head(kegg, 3)

   PATH UNIPROT  SYMBOL
1 00310  Q95RU8     G9a
2 00310  Q9W5E0 Suv4-20
3 00310  Q9W3N9 CG10932
```

Exercise 28
For convenience, `lrTest`, a DGEGLM object from the RNA-seq chapter, is included in the SequenceAnalysisData package. The following code loads this data and create a 'top table' of the ten most differentially represented genes. This top table is then cast as a `data.frame`.

```
> library(org.Dm.eg.db)
> data(lrTest)
> tt <- as.data.frame(topTags(lrTest))
```

Extract the Flybase gene identifiers (`FLYBASE`) from the row names of this table and map them to their corresponding Entrez gene (`ENTREZID`) and symbol ids (`SYMBOL`) using `select`. Use `merge` to add the results of `select` to the top table.

Solution:

```
> fbids <- rownames(tt)
> cols <- c("ENTREZID", "SYMBOL")
> anno <- select(org.Dm.eg.db, fbids, cols, "FLYBASE")
> ttanno <- merge(tt, anno, by.x=0, by.y="FLYBASE")
> dim(ttanno)

[1] 10  8

> head(ttanno, 3)

    Row.names logConc logFC LR.statistic  PValue     FDR ENTREZID  SYMBOL
1 FBgn0000071     -11   2.8          183 1.1e-41 1.1e-38    40831     Ama
2 FBgn0024288     -12  -4.7          179 7.1e-41 6.3e-38    45039 Sox100B
3 FBgn0033764     -12   3.5          188 6.8e-43 7.8e-40     <NA>    <NA>
```

7.2 Genome-centric annotations with *GenomicFeatures*

Genome-centric packages are very useful for annotations involving genomic coordinates. It is straight-forward, for instance, to discover the coordinates of coding sequences in regions of interest, and from these retrieve corresponding DNA or protein coding sequences. Other examples of the types of operations that are easy to perform with genome-centric annotations include defining regions of interest for counting aligned reads in RNA-seq experiments (Section 5) and retrieving DNA sequences underlying regions of interest in ChIP-seq analysis (Section 6), e.g., for motif characterization.

Exercise 29
Load the 'transcript.db' package relevant to the dm3 build of D. melanogaster*. Use* `select` *and friends to select the Flybase gene ids of the top table* `tt` *and the Flybase transcript names (TXNAME) and Entrez gene identifiers (GENEID).*

Use `cdsBy` *to extract all coding sequences, grouped by transcript. Subset the coding sequences to contain just the transcripts relevant to the top table. How many transcripts are there? What is the structure of the first transcript's coding sequence?*

Load the 'BSgenome' package for the dm3 build of D. melanogaster*. Use the coding sequences ranges of the previous part of this exercise to extract the underlying DNA sequence, using the* `extractTranscriptsFromGenome` *function. Use Biostrings'* `translate` *to convert DNA to amino acid sequences.*

Solution: The following loads the relevant Transcript.db package, and creates a more convenient alias to the *TranscriptDb* instance defined in the package.

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
```

We can discover available keys (using `keys`) and columns (`cols`) in `txdb`, and then use `select` to retrieve the transcripts associated with each differentially expressed gene. The mapping between gene and transcript is not one-to-one – some genes have more than one transcript.

```
> txnm <- select(txdb, fbids, "TXNAME", "GENEID")
> nrow(txnm)
```

```
[1] 19
```

```
> head(txnm, 3)
```

```
       GENEID      TXNAME
1 FBgn0039155 FBtr0084549
2 FBgn0039827 FBtr0085755
3 FBgn0039827 FBtr0085756
```

The *TranscriptDb* instances can be queried for data that is more structured than simple data frames, and in particular return *GRanges* or *GRangesList* instances to represent genomic coordinates. These queries are performed using `cdsBy` (coding sequence), `transcriptsBy` (transcripts), etc., where a function argument `by` specifies how coding sequences or transcripts are grouped. Here we extract the coding sequences grouped by transcript, returning the transcript names, and subset the resulting *GRangesList* to contain just the transcripts of interest to us. The first transcript is composed of 6 distinct coding sequence regions.

```
> cds <- cdsBy(txdb, "tx", use.names=TRUE)[txnm$TXNAME]
> length(cds)

[1] 19

> cds[1]

GRangesList of length 1:
$FBtr0084549
GRanges with 6 ranges and 3 elementMetadata cols:
      seqnames               ranges strand |    cds_id    cds_name exon_rank
         <Rle>            <IRanges>  <Rle> | <integer> <character> <integer>
  [1]    chr3R [19970946, 19971592]      + |     55167        <NA>         2
  [2]    chr3R [19971652, 19971770]      + |     55168        <NA>         3
  [3]    chr3R [19971831, 19972024]      + |     55169        <NA>         4
  [4]    chr3R [19972088, 19972461]      + |     55170        <NA>         5
  [5]    chr3R [19972523, 19972589]      + |     55171        <NA>         6
  [6]    chr3R [19972918, 19973094]      + |     55172        <NA>         7

---
seqlengths:
     chr2L  chr2LHet     chr2R  chr2RHet ...   chrXHet   chrYHet      chrM
  23011544    368872  21146708   3288761 ...    204112    347038     19517
```

The following code loads the appropriate BSgenome package; the `Dmelanogaster` object refers to the whole genome sequence represented in this package. The remaining steps extract the DNA sequence of each transcript, and translates these to amino acid sequences. Issues of strand are handled correctly.

```
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> txx <- extractTranscriptsFromGenome(Dmelanogaster, cds)
> length(txx)

[1] 19

> head(txx, 3)

  A DNAStringSet instance of length 3
    width seq                                               names
[1]  1578 ATGGGCAGCATGCAAGTGGCGCT...TGCAGATCAAGTGCAGCGACTAG FBtr0084549
[2]  2760 ATGCTGCGTTATCTGGCGCTTTC...TTGCTGCCCCATTCGAACTTTAG FBtr0085755
[3]  2217 ATGGCACTCAAGTTTCCCACAGT...TTGCTGCCCCATTCGAACTTTAG FBtr0085756

> head(translate(txx), 3)

  A AAStringSet instance of length 3
    width seq
[1]   526 MGSMQVALLALLVLGQLFPSAVANGSSSYSSTST...VLDDSRNVFTFTTPKCENFRKRFPKLQIKCSD*
[2]   920 MLRYLALSEAGIAKLPRPQSRCYHSEKGVWGYKP...YCGRCEAPTPATGIGKVHKREVDEIVAAPFEL*
[3]   739 MALKFPTVKRYGGEGAESMLAFFWQLLRDSVQAN...YCGRCEAPTPATGIGKVHKREVDEIVAAPFEL*
```

7.3 Using biomaRt

The *biomaRt* package offers access to the online biomart resource. this consists of several data base resources, referred to as 'marts'. Each mart allows access to multiple data sets; the *biomaRt* package provides methods for mart and data set discovery, and a standard method `getBM` to retrieve data.

Exercise 30
Load the biomaRt package and list the available marts. Choose the ensembl *mart and list the datasets for that mart. Set up a mart to use the* ensembl *mart and the* hsapiens_gene_ensembl *dataset.*

A biomaRt dataset can be accessed via `getBM`. In addition to the mart to be accessed, this function takes filters and attributes as arguments. Use `filterOptions` and `listAttributes` to discover values for these arguments. Call `getBM` using filters and attributes of your choosing.

Solution:

```
> library(biomaRt)
> head(listMarts(), 3)                      ## list the marts
> head(listDatasets(useMart("ensembl")), 3) ## mart datasets
> ensembl <-                                ## fully specified mart
+     useMart("ensembl", dataset = "hsapiens_gene_ensembl")
> head(listFilters(ensembl), 3)             ## filters
> myFilter <- "chromosome_name"
> head(filterOptions(myFilter, ensembl), 3) ## return values
> myValues <- c("21", "22")
> head(listAttributes(ensembl), 3)          ## attributes
> myAttributes <- c("ensembl_gene_id","chromosome_name")
> ## assemble and query the mart
> res <- getBM(attributes =  myAttributes, filters =  myFilter,
+              values =  myValues, mart = ensembl)
```

Use `head(res)` to see the results.

8 Annotation of Variants

A major product of DNASeq experiments are catalogs of called variants (e.g., SNPs, indels). We will use the *VariantAnnotation* package to explore this type of data. Sample data included in the package are a subset of chromosome 22 from the 1000 Genomes project. Variant Call Format (VCF; full description) text files contain meta-information lines, a header line with column names, data lines with information about a position in the genome, and optional genotype information on samples for each position.

8.1 Variant call format (VCF) files

Data are read from a VCF file and variants identified according to region such as `coding`, `intron`, `intergenic`, `spliceSite` etc. Amino acid coding changes are computed for the non-synonymous variants. SIFT and PolyPhen databases provide predictions of how severely the coding changes affect protein function.

Data exploration

Exercise 31
The objective of this exercise is to compare the quality of called SNPs that are located in dbSNP, versus those that are novel.

Locate the sample data in the file system. Explore the metadata (information about the content of the file) using `scanVcfHeader`*. Discover the 'info' fields* `VT` *(variant type), and* `RSQ` *(genotype imputation quality).*

Input sample data in using `readVcf`*. You'll need to specify the genome build (*`genome="hg19"`*) on which the variants are annotated. Take a peak at the* `rowData` *to see the genomic locations of each variant.*

dbSNP uses abbreviations such as `ch22` *to represent chromosome 22, whereas the VCF file uses 22. Use* `rowData` *and* `renameSeqlevels` *to extract the row data of the variants, and rename the chromosomes.*

The SNPlocs.Hsapiens.dbSNP.20101109 contains information about SNPs in a particular build of dbSNP. Load the package, use the `dbSNPFilter` *function to create a filter, and query the row data of the VCF file for membership.*

Create a data frame containing the dbSNP membership status and imputation quality of each SNP. Create a density plot to illustrate the results.

Solution: Explore the header:

```
> library(VariantAnnotation)
> fl <- system.file("extdata", "chr22.vcf.gz",
+                   package="VariantAnnotation")
> (hdr <- scanVcfHeader(fl))

class: VCFHeader
samples(5): HG00096 HG00097 HG00099 HG00100 HG00101
meta(1): fileformat
fixed(1): ALT
info(22): LDAF AVGPOST ... VT SNPSOURCE
geno(3): GT DS GL
```

```
> info(hdr)[c("VT", "RSQ"),]

DataFrame with 2 rows and 3 columns
         Number        Type                                         Description
    <character> <character>                                         <character>
VT            1      String indicates what type of variant the line represents
RSQ           1       Float      Genotype imputation quality from MaCH/Thunder
```

Input the data and peak at their locations:

```
> (vcf <- readVcf(fl, "hg19"))

class: VCF
dim: 10376 5
genome: hg19
exptData(1): header
fixed(4): REF ALT QUAL FILTER
info(22): LDAF AVGPOST ... VT SNPSOURCE
geno(3): GT DS GL
rownames(10376): rs7410291 rs147922003 ... rs144055359 rs114526001
rowData values names(1): paramRangeID
colnames(5): HG00096 HG00097 HG00099 HG00100 HG00101
colData names(1): Samples

> head(rowData(vcf), 3)

GRanges with 3 ranges and 1 elementMetadata col:
              seqnames               ranges strand | paramRangeID
                 <Rle>            <IRanges>  <Rle> |     <factor>
    rs7410291       22 [50300078, 50300078]      * |         <NA>
  rs147922003       22 [50300086, 50300086]      * |         <NA>
  rs114143073       22 [50300101, 50300101]      * |         <NA>
  ---
  seqlengths:
   22
   NA
```

Rename chromosome levels:

```
> rowData(vcf) <- renameSeqlevels(rowData(vcf), c("22"="ch22"))
```

Discover whether SNPs are located in dbSNP:

```
> library(SNPlocs.Hsapiens.dbSNP.20101109)
> snpFilt <- dbSNPFilter("SNPlocs.Hsapiens.dbSNP.20101109")
> inDbSNP <- snpFilt(rowData(vcf), subset=FALSE)
> table(inDbSNP)

inDbSNP
FALSE  TRUE
 6126  4250
```

Create a data frame summarizing SNP quality and dbSNP membership:

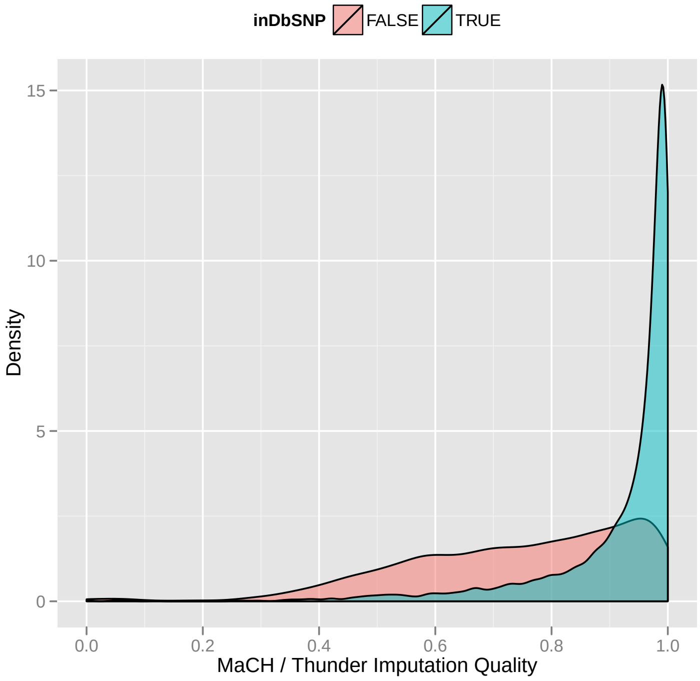


Figure 9: Quality scores of variants in dbSNP, compared to those not in dbSNP.

```
> metrics <-
+     data.frame(inDbSNP=inDbSNP, RSQ=values(info(vcf))$RSQ)
```

Finally, visualize the data, e.g., using *ggplot2* (Figure 9).

```
> library(ggplot2)
> ggplot(metrics, aes(RSQ, fill=inDbSNP)) +
+     geom_density(alpha=0.5) +
+     scale_x_continuous(name="MaCH / Thunder Imputation Quality") +
+     scale_y_continuous(name="Density") +
+     opts(legend.position="top")
```

8.2 Coding consequences

Locating variants in and around genes Variant location with respect to genes can be identified with the `locateVariants` function. Regions are specified in the `region` argument and can be one of the following constructors: `CodingVariants()`, `IntronVariants()`, `FiveUTRVariants()`, `ThreeUTRVariants()`, `IntergenicVariants()`, `SpliceSiteVariants()`, or `AllVariants()`. Location definitions are shown in Table 10.

Exercise 32
Load the TxDb.Hsapiens.UCSC.hg19.knownGene annotation package, and read in the `chr22.vcf.gz` example file from the VariantAnnotation package.

Table 10: Variant locations

Location	Details
`coding`	Within a coding region
`fiveUTR`	Within a 5' untranslated region
`threeUTR`	Within a 3' untranslated region
`intron`	Within an intron region
`intergenic`	Not within a transcript associated with a gene
`spliceSite`	Overlaps any of the first or last 2 nucleotides of an intron

Remembering to re-name sequence levels, use the `locateVariants` *function to identify coding variants.*

Summarize aspects of your data, e.g., did any coding variants match more than one gene? How many coding variants are there per gene ID?

Solution: Here we open the known genes data base, and read in the VCF file.

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
> fl <- system.file("extdata", "chr22.vcf.gz",
+                   package="VariantAnnotation")
> vcf <- readVcf(fl, "hg19")
> vcf <- renameSeqlevels(vcf, c("22"="chr22"))
```

The next lines locate coding variants.

```
> rd <- rowData(vcf)
> loc <- locateVariants(rd, txdb, CodingVariants())
> head(loc, 3)

GRanges with 3 ranges and 5 elementMetadata cols:
      seqnames               ranges strand | LOCATION   QUERYID      TXID
         <Rle>            <IRanges>  <Rle> | <factor> <integer> <integer>
  [1]    chr22 [50301422, 50301422]      * |   coding        24     76833
  [2]    chr22 [50301476, 50301476]      * |   coding        25     76833
  [3]    chr22 [50301488, 50301488]      * |   coding        26     76833
          CDSID      GENEID
      <integer> <character>
  [1]    225251       79087
  [2]    225251       79087
  [3]    225251       79087
  ---
  seqlengths:
   chr22
      NA
```

To answer gene-centric questions data can be summarized by gene regardless of transcript.

```
> ## Did any coding variants match more than one gene?
> splt <- split(values(loc)$GENEID, values(loc)$QUERYID)
> table(sapply(splt, function(x) length(unique(x)) > 1))
```

```
FALSE  TRUE
  956    15

> ## Summarize the number of coding variants by gene ID
> splt <- split(values(loc)$QUERYID, values(loc)$GENEID)
> head(sapply(splt, function(x) length(unique(x))), 3)

113730   1890  23209
    22     15     30
```

Amino acid coding changes `predictCoding` computes amino acid coding changes for non-synonymous variants. Only ranges in `query` that overlap with a coding region in `subject` are considered. Reference sequences are retrieved from either a `BSgenome` or fasta file specified in `seqSource`. Variant sequences are constructed by substituting, inserting or deleting values in the `varAllele` column into the reference sequence. Amino acid codes are computed for the variant codon sequence when the length is a multiple of 3.

The `query` argument to `predictCoding` can be a `GRanges` or `VCF`. When a `GRanges` is supplied the `varAllele` argument must be specified. In the case of a `VCF`, the alternate alleles are taken from `values(alt(<VCF>))$ALT` and the `varAllele` argument is not specified.

The result is a modified `query` containing only variants that fall within coding regions. Each row represents a variant-transcript match so more than one row per original variant is possible.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> coding <- predictCoding(vcf, txdb, seqSource=Hsapiens)
> coding[5:9]

GRanges with 5 ranges and 13 elementMetadata cols:
              seqnames               ranges strand | paramRangeID
                 <Rle>            <IRanges>  <Rle> |     <factor>
  22:50301584    chr22 [50301584, 50301584]      - |         <NA>
                   varAllele     CDSLOC             PROTEINLOC   QUERYID
              <DNAStringSet>  <IRanges> <CompressedIntegerList> <integer>
  22:50301584              A [777, 777]                     259        28
                     TXID     CDSID      GENEID  CONSEQUENCE       REFCODON
              <character> <integer> <character>     <factor> <DNAStringSet>
  22:50301584       76833    225251       79087   synonymous            CCG
                    VARCODON         REFAA         VARAA
              <DNAStringSet> <AAStringSet> <AAStringSet>
  22:50301584            CCA             P             P
 [ reached getOption("max.print") -- omitted 4 rows ]
  ---
  seqlengths:
   chr22
      NA
```

Using variant rs114264124 as an example, we see varAllele A has been substituted into the refCodon CGG to produce varCodon CAG. The refCodon is the sequence of codons necessary to make the variant allele substitution and therefore often includes more nucleotides than indicated in the range (i.e. the range is 50302962, 50302962, width of 1). Notice it is the second position in the refCodon that has been substituted. This position in the codon, the position of substitution, corresponds to genomic position 50302962. This genomic position maps to position 698 in coding region-based coordinates and to triplet 233 in the protein. This is a non-synonymous coding variant where the amino acid has changed from R (Arg) to Q (Gln).

When the resulting varCodon is not a multiple of 3 it cannot be translated. The consequence is considered a frameshift and varAA will be missing.

```
> coding[values(coding)$CONSEQUENCE == "frameshift"]

GRanges with 1 range and 13 elementMetadata cols:
              seqnames               ranges strand | paramRangeID
                 <Rle>            <IRanges>  <Rle> |     <factor>
  22:50317001    chr22 [50317001, 50317001]      + |         <NA>
                   varAllele     CDSLOC              PROTEINLOC   QUERYID
              <DNAStringSet>  <IRanges> <CompressedIntegerList> <integer>
  22:50317001          GCACT [808, 808]                     270       359
                     TXID     CDSID      GENEID CONSEQUENCE       REFCODON
              <character> <integer> <character>    <factor> <DNAStringSet>
  22:50317001       76834    225263       79174  frameshift            GCC
                    VARCODON         REFAA         VARAA
              <DNAStringSet> <AAStringSet> <AAStringSet>
  22:50317001            ACC             A
  ---
  seqlengths:
   chr22
      NA
```

SIFT and PolyPhen databases From predictCoding we identified the amino acid coding changes for the non-synonymous variants. For this subset we can retrieve predictions of how damaging these coding changes may be. SIFT (Sorting Intolerant From Tolerant) and PolyPhen (Polymorphism Phenotyping) are methods that predict the impact of amino acid substitution on a human protein. The SIFT method uses sequence homology and the physical properties of amino acids to make predictions about protein function. PolyPhen uses sequence-based features and structural information characterizing the substitution to make predictions about the structure and function of the protein.

Collated predictions for specific dbSNP builds are available as downloads from the SIFT and PolyPhen web sites. These results have been packaged into *SIFT.Hsapiens.dbSNP132.db* and *PolyPhen.Hapiens.dbSNP131.db* and are designed to be searched by rsid. Variants that are in dbSNP can be searched with these database packages. When working with novel variants, SIFT and PolyPhen must be called directly. See references for home pages.

Identify the non-synonymous variants and obtain the rsids.

```
> nms <- names(coding)
> idx <- values(coding)$CONSEQUENCE == "nonsynonymous"
> nonsyn <- coding[idx]
> names(nonsyn) <- nms[idx]
> rsids <- unique(names(nonsyn)[grep("rs", names(nonsyn), fixed=TRUE)])
```

Detailed descriptions of the database columns can be found with `?SIFTDbColumns` and `?PolyPhenDbColumns`. Variants in these databases often contain more than one row per variant. The variant may have been reported by multiple sources and therefore the source will differ as well as some of the other variables.

```
> library(SIFT.Hsapiens.dbSNP132)
> ## rsids in the package
> head(keys(SIFT.Hsapiens.dbSNP132), 3)

[1] "rs10000692" "rs10001580" "rs10002700"

> ## list available columns
> cols(SIFT.Hsapiens.dbSNP132)

 [1] "RSID"        "PROTEINID"   "AACHANGE"    "METHOD"      "AA"
 [6] "PREDICTION"  "SCORE"       "MEDIAN"      "POSTIONSEQS" "TOTALSEQS"

> ## select a subset of columns
> ## a warning is thrown when a key is not found in the database
> subst <- c("RSID", "PREDICTION", "SCORE", "AACHANGE", "PROTEINID")
> sift <- select(SIFT.Hsapiens.dbSNP132, keys=rsids, cols=subst)
> head(sift, 3)

         RSID PROTEINID AACHANGE PREDICTION SCORE
1 rs114264124 NP_077010    R233Q  TOLERATED  0.59
2 rs114264124 NP_077010    R233Q  TOLERATED  1.00
3 rs114264124 NP_077010    R233Q  TOLERATED  0.20
```

PolyPhen provides predictions using two different training datasets and has considerable information about 3D protein structure. See `?PolyPhenDbColumns` or the PolyPhen web site listed in the references for more details.

A Appendix: data retrieval

A.1 RNA-seq data retrieval

The following script was used to retrieve a portion of the Pasilla data set from the short read archive. The data is very large; extraction relies on installation of the SRA SDK, available from the Short Read Archive.

```
> library(RCurl)
> srasdk <- "/home/mtmorgan/bin/sra_sdk-2.0.1" # local installation
> sra <- "ftp://ftp-trace.ncbi.nih.gov/sra/sra-instant/reads/ByExp/sra"
> expt <- "SRX/SRX014/SRX014458/"
> url <- sprintf("%s/%s", sra, expt)
> acc <- strsplit(getURL(url, ftplistonly=TRUE), "\n")[[1]]
> urls <- sprintf("%s%s/%s.sra", url, acc, acc)
> for (fl in urls)
+     system(sprintf("wget %s",  fl), wait=FALSE, ignore.stdout=TRUE)
> app <- sprintf("%s/bin64/fastq-dump", srasdk)
> for (fl in file.path(wd, basename(urls)))
+      system(sprintf("%s %s", app, fl), wait=FALSE)
```

A.2 ChIP-seq data retrieval and MACS analysis

BAM and called peak files were obtained from http://hgdownload.cse.ucsc.edu/goldenPath/hg19/encodeDCC/wgEncodeUwTfbs. The script used to process called peak data into the `stam` object is at

```
> file.path("script", "chipseq-stam-called-peaks.R",
+           package="SequenceAnalysisData")
```

```
[1] "script/chipseq-stam-called-peaks.R/SequenceAnalysisData"
```

References

[1] S. Anders and W. Huber. Differential expression analysis for sequence count data. *Genome Biology*, 11:R106, 2010.

[2] A. N. Brooks, L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Research*, pages 193–202, 2011.

[3] J. M. Chambers. *Software for Data Analysis: Programming with R*. Springer, New York, 2008.

[4] P. Dalgaard. *Introductory Statistics with R*. Springer, 2nd edition, 2008.

[5] R. Gentleman. *R Programming for Bioinformatics*. Computer Science & Data Analysis. Chapman & Hall/CRC, Boca Raton, FL, 2008.

[6] J. J. Goeman and P. Buhlmann. Analyzing gene expression data in terms of gene sets: methodological issues. *Bioinformatics*, 23(8):980–987, Apr 2007.

[7] J. W. Ho, E. Bishop, P. V. Karchenko, N. Negre, K. P. White, and P. J. Park. ChIP-chip versus ChIP-seq: lessons for experimental design and data analysis. *BMC Genomics*, 12:134, 2011. [PubMed Central:PMC3053263] [DOI:10.1186/1471-2164-12-134] [PubMed:21356108].

[8] I. Holmes, K. Harris, and C. Quince. Dirichlet multinomial mixtures: Generative models for microbial metagenomics. *PLoS ONE*, 7(2):e30126, 02 2012.

[9] R. Kabacoff. *R in Action*. Manning, 2010.

[10] P. V. Kharchenko, A. A. Alekseyenko, Y. B. Schwartz, A. Minoda, N. C. Riddle, J. Ernst, P. J. Sabo, E. Larschan, A. A. Gorchakov, T. Gu, D. Linder-Basso, A. Plachetka, G. Shanower, M. Y. Tolstorukov, L. J. Luquette, R. Xi, Y. L. Jung, R. W. Park, E. P. Bishop, T. K. Canfield, R. Sandstrom, R. E. Thurman, D. M. MacAlpine, J. A. Stamatoyannopoulos, M. Kellis, S. C. Elgin, M. I. Kuroda, V. Pirrotta, G. H. Karpen, and P. J. Park. Comprehensive analysis of the chromatin landscape in Drosophila melanogaster. *Nature*, 471:480–485, Mar 2011. [PubMed Central:PMC3109908] [DOI:10.1038/nature09725] [PubMed:21179089].

[11] P. V. Kharchenko, M. Y. Tolstorukov, and P. J. Park. Design and analysis of ChIP-seq experiments for DNA-binding proteins. *Nat. Biotechnol.*, 26:1351–1359, Dec 2008. [PubMed Central:PMC2597701] [DOI:10.1038/nbt.1508] [PubMed:19029915].

[12] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, 10:R25, 2009.

[13] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25:1754–1760, Jul 2009.

[14] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26:589–595, Mar 2010.

[15] N. Matloff. *The Art of R Programming.* No Starch Pess, 2011.

[16] R. M. Myers, J. Stamatoyannopoulos, M. Snyder, ..., and P. J. Good. A user's guide to the encyclopedia of DNA elements (ENCODE). *PLoS Biol.*, 9:e1001046, Apr 2011. [PubMed Central:PMC3079585] [DOI:10.1371/journal.pbio.1001046] [PubMed:21526222].

[17] P. J. Park. ChIP-seq: advantages and challenges of a maturing technology. *Nat. Rev. Genet.*, 10:669–680, Oct 2009. [PubMed Central:PMC3191340] [DOI:10.1038/nrg2641] [PubMed:19736561].

[18] S. Pepke, B. Wold, and A. Mortazavi. Computation for ChIP-seq and RNA-seq studies. *Nat. Methods*, 6:22–32, Nov 2009. [DOI:10.1038/nmeth.1371] [PubMed:19844228].

[19] M. D. Robinson, D. J. McCarthy, and G. K. Smyth. edgeR: a Bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, 26:139–140, Jan 2010.

[20] P. J. Sabo, M. Hawrylycz, J. C. Wallace, R. Humbert, M. Yu, A. Shafer, J. Kawamoto, R. Hall, J. Mack, M. O. Dorschner, M. McArthur, and J. A. Stamatoyannopoulos. Discovery of functional noncoding elements by digital analysis of chromatin structure. *Proc. Natl. Acad. Sci. U.S.A.*, 101:16837–16842, Nov 2004.

[21] M. Sammeth. Complete alternative splicing events are bubbles in splicing graphs. *J. Comput. Biol.*, 16:1117–1140, Aug 2009.

[22] M. D. Young, M. J. Wakefield, G. K. Smyth, and A. Oshlack. Gene ontology analysis for rna-seq: accounting for selection bias. *Genome Biology*, 11:R14, 2010.

Chapter 1

INTRODUCTION TO MICROARRAY ANALYSIS

1.0 Introduction

Microarray analysis has emerged in the last few years as a flexible method for analyzing large numbers of nucleic acid fragments in parallel. Its origins can be traced to several different disciplines and techniques. Microarrays can be seen as a continued development of molecular biology hybridization methods, as an extension of the use of fluorescence microscopy in cell biology, as well as a diagnostic assay using capture to solid surface as a way to reduce the amount of analytes needed. The convergence of ideas and principles utilized in these fields, together with technological advancements in preparing miniaturized collections of nucleic acids on solid supports, have all contributed to the emergence of microarray and microchip technologies.

In molecular biology, analysis of nucleic acids by hybridization is a universally adopted key method for analysis. Filter-based dot blot analysis has been used for a long time as a convenient method for analyzing multiple samples by hybridization. Classical gene expression analysis methods such as Northern blotting, reverse transcriptase polymerase chain reaction (RT-PCR) and nuclease protection assays, are best suited for analyzing a limited number of genes and samples at a time. By reversing the Northern blotting principle so that the labelled moiety is derived from the mRNA sample and the immobilized fractions are the known sequences traditionally used as probes, filter-based gene expression analysis has enabled simultaneous determination of expression levels of thousands of genes in one experiment. Because of the ease of use of these filter-based methods and their compatibility with general lab equipment, these macroarrays have been widely adopted for gene expression studies (1). One disadvantage to using this method has been the relatively large size and the autofluorescence of the membrane, which prevents efficient use of multiplexed fluorescent probes and subsequently limits the number of samples that can be analyzed in each experiment (Fig 1).

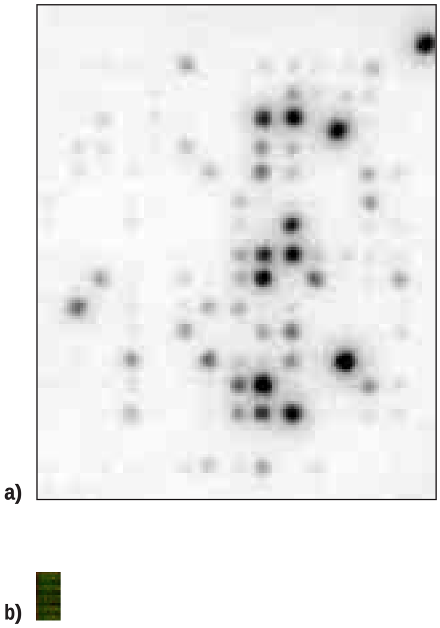


Fig 1. Comparison of a macroarray and microarray. A close-up of a filter macroarray (a) hybridized with ^{32}P-labelled cDNA probe and a microarray (b) hybridized with two different cDNA probes, one labelled with Cy™3 and the other with Cy5. The array images are shown approximately to the same scale.

Utilizing microscope slides to immobilize cells and chromosomes precedes filter-based gene expression analysis, as well as proven methods such as immunohistochemistry, immunocytochemistry and *in situ* hybridization. By combining fluorescence analysis of multiplexed probes with microscopy, fluorescent *in situ* hybridization (FISH) has enabled detection of nucleic acids within cells and chromosomes, and has been found useful in gene expression and genomic analysis.

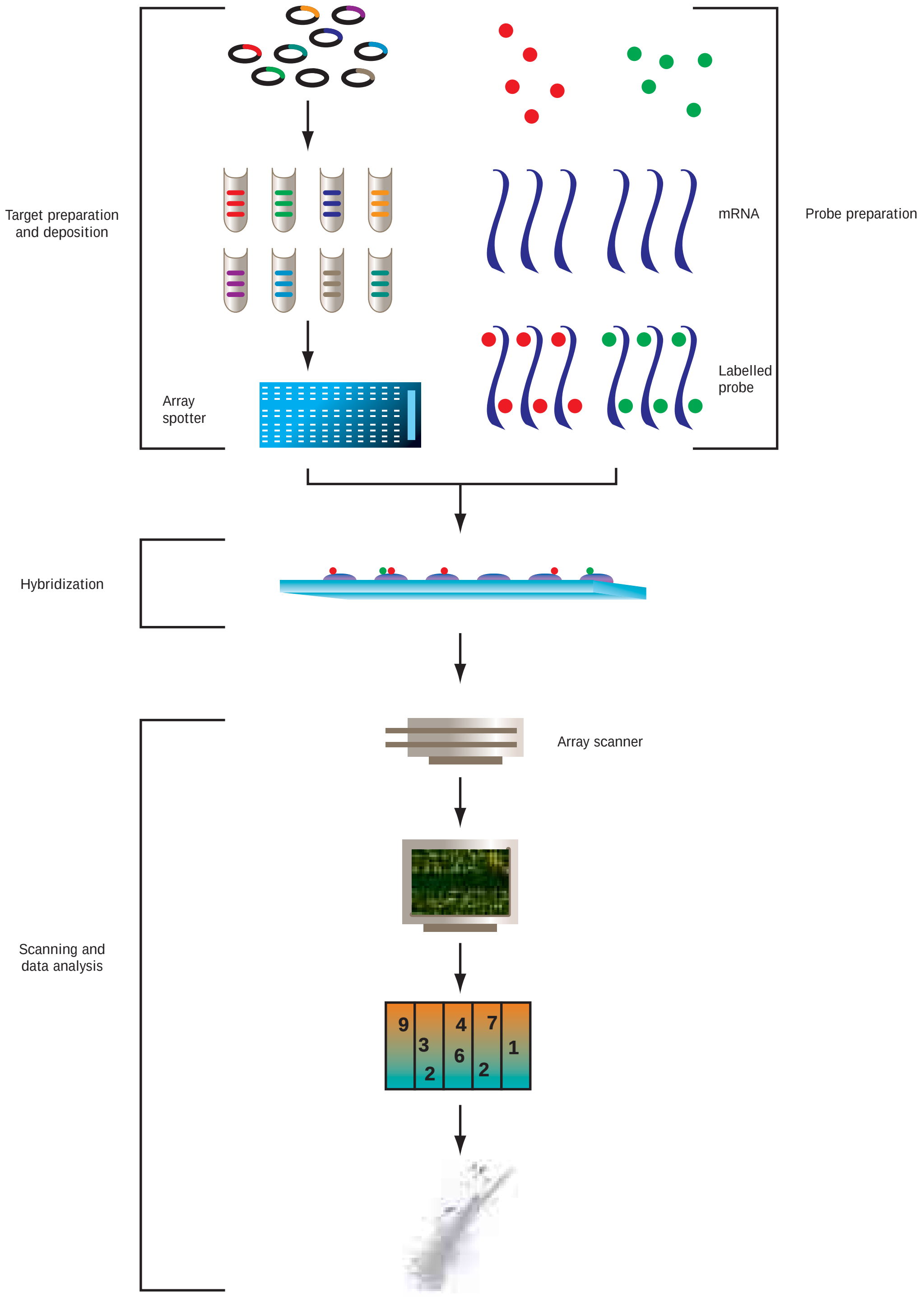
These two methods of analysis were brought together by advancements made in attaching nucleic acid sequences to a glass support. Borrowing a technique from semiconductor manufacturing, photolithography was used to synthesize oligonucleotides directly onto a glass support. Separately, a procedure called contact printing was used to deposit purified nucleic acid onto a slide surface (2). These methods have since made it possible to miniaturize the macroarray experiment, so to speak, by using microscope slides instead of membrane filters. In 1995 and 1996, the first papers in which the term ‘microarray’ was used in its current meaning were published by the laboratory of Pat Brown at Stanford University (3). The rapid adoption of this technique is illustrated by the publication in 2001 of over 900 papers on the use of microarray technique.

1.1 Principles of microarray analysis

Despite the variety of technical solutions that have been developed for performing microarray analysis, all are miniaturized hybridization assays for studying thousands of nucleic acid fragments simultaneously. All microarray systems (Fig 2) share the following key components:

- the array, which contains immobilized nucleic acid sequences, or ‘targets’
- one or more labelled samples or ‘probes’, that are hybridized with the microarray
- a detection system that quantitates the hybridization signal

Fig 2. Principles of microarrays.



1.1.1 Nomenclature for microarrays

The terms ‘probe’ and ‘target’ are sometimes used interchangeably to describe either the labelled sample or the immobilized nucleic acids. In this handbook the immobilized nucleic acid is referred to as the target and the labelled sample as the probe (Fig 3).

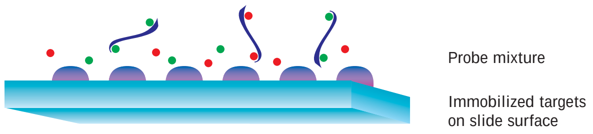


Fig 3. Target and probe. Targets are the immobilized nucleic acids on the slide surface. A probe consisting of two identical populations of nucleic acids labelled with different fluorescent dyes is shown.

1.1.2 Microarrays

Microarrays consist of a collection of nucleic acid sequences immobilized onto a solid support so that each unique sequence forms a tiny feature, called a ‘spot’ or ‘target’. These nucleic acids are obtained in numerous ways, and there are different methods for depositing them onto microarray slides (refer to chapter 3). The size of these spots varies from one system to another, but it is usually less than two hundred micrometers in diameter. A glass slide or glass wafer acts as the solid support onto which up to tens of thousands of spots can be arrayed in a total area of a few square centimeters (Fig 4).

Fig 4. A glass microarray. A standard and mirrored glass microscope slide that contains several thousands of immobilized cDNA fragments. Because of the small amounts of nucleic acid present and their tiny size, the target spots are not visible with the naked eye.

1.1.3 Probe labelling

The microarray sample that is being analyzed, whether it is mRNA for a gene expression study or DNA derived from genomic analysis, is converted to a labelled population of nucleic acids, the probe. These probes frequently consist of several thousands of different labelled nucleic acid fragments. The complexity of microarray hybridization—over 10 000 different labelled fragments interrogating up to 100 000 different immobilized sequences—is greater than that encountered in other routine molecular biology experiments. Therefore, this hybridization should be carried out under conditions that do not promote annealing of non-complementary fragments.

Fluorescent dyes, and especially the cyanine dyes Cy3 and Cy5, have been adopted as the predominant label in microarray analysis. Fluorescence has the advantage of permitting the detection of two or more different signals in one experiment. This has allowed investigators to perform comparative analysis of two or more samples on one microarray. It has also increased the accuracy and throughput of microarray analysis over filter-based macroarrays, in which only one radioactively labelled sample can be conveniently analyzed at a time.

1.1.4 Microarray hybridization

In a microarray hybridization, the labelled fragments in the probe are expected to form duplexes with their immobilized complementary targets. This requires that the nucleic acids are single-stranded and accessible to each other. The number of duplexes formed reflects the relative number of each specific fragment in the probe, as long as the amount of immobilized target nucleic acid is in excess and not limiting the kinetics of hybridization. Two or more samples labelled with different fluorescent dyes can be hybridized simultaneously, resulting in simultaneous hybridization taking place at each target spot. By measuring the different fluorescent signals associated with each spot, the relative abundance of specific sequences in each of the samples can be determined.

1.1.5 Scanning and data analysis

Microarray scanners typically contain two different lasers that emit light at wavelengths that are suitable for exciting the fluorescent dyes used as labels. A confocal microscope attached to a detector system records the emitted light from each of the microarray spots, allowing high-resolution detection of the hybridization signals.

Despite their small size, microarrays generate large quantities of data even from a single experiment. As a typical experiment will involve the use of several analyzed samples on replicate arrays, the use of computerized data processing is necessary in order to handle the amount of data generated and to gain maximum information from the experiment. This can be achieved by specialized software that extracts primary data from scanned microarray slide images, normalizes this data to remove the influence of experimental variation, and finally manipulates the data so that biologically meaningful conclusions can be made.

1.2 Applications of microarray analysis

The versatility of microarray analysis is confirmed by its rapid emergence as a general molecular biology analytical technique. Increasing numbers of researchers within academic institutions and industrial laboratories are now exploiting this technology in diverse biomedical disciplines. Microarrays have not become a replacement to established techniques, but more a novel, high-power approach to perform analyses that were previously time consuming.

By using information derived from the several complete or near complete genome sequences, including the human genome, it is now possible to perform genome-wide experiments using microarray technology. This has already been demonstrated for *S. cerevisiae* where all the expressed genes are known. As microarrays can contain thousands of targets, both characterized and uncharacterized, experiments can be conducted without prior hypotheses. This combined with the millions of data points that are possible to analyze in one experiment, microarray analysis has enabled global analysis of biological processes. Gene expression analysis, genome analysis, and drug discovery have been three of the main areas in which microarray analysis has been applied so far.

1.3 Gene expression analysis

Gene expression analysis examines the composition of cellular messenger RNA populations. The identity of transcripts that make up these populations and their expression levels are informative of cell state and activity of genes and, as the precursors of translated proteins, changes in mRNA levels are related to changes in the proteome.

1.3.1 Traditional techniques

Traditional gene expression analysis has used techniques such as Northern blotting, RT-PCR and nuclease protection assays. More advanced methods—some of these include differential display, subtractive hybridization, representational difference analysis, expressed sequence tags, cDNA fragment fingerprinting, and serial analysis of gene expression—have enabled the discovery of novel differentially expressed genes (4). However, the technical challenges of these methods still limit their use to the analysis of just a few samples at a time. Microarray analysis, in contrast, allows the analysis of thousands of genes in multiple samples with relative ease.

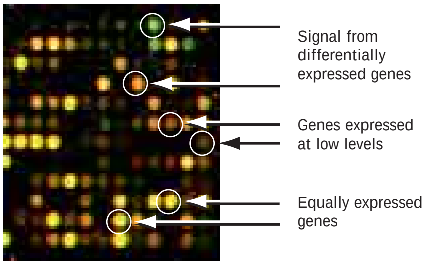

Signal from differentially expressed genes
Genes expressed at low levels
Equally expressed genes

Fig 5. Dual color differential microarray analysis. Dual color microarray hybridization signals are typically represented as false color images in which signals from one dye are presented in red (Cy5 in this case) and signals from the other dye in green (Cy3). If equal signal is obtained from a spot, it will appear yellow. Shades of green and red denote differences in relative abundance in favor of one or the other sample. As the screen appearance of microarray images can be easily manipulated, information gained from such images can be misleading.

1.3.2 Gene expression analysis with microarrays

A typical microarray gene expression analysis experiment compares the relative expression levels of specific transcripts in two samples. One of these samples is a control and the other is derived from cells whose response or status is being investigated. Each of these samples is labelled with a different fluorescent dye, and equal amounts of the labelled samples are combined and hybridized with the microarray. The fluorescent signals corresponding to the two dyes are measured independently from each spot after hybridization. After normalization, the intensity of the two hybridization signals can be compared. Equal signal from both samples suggests equal expression in both samples (Fig 5).

Microarray analysis does not give information about absolute gene expression levels in the samples. This is because the intensity of the fluorescent signals is not only proportional to the number of hybridized fragments but also to the length of these fragments and the number of fluorescent labels each fragment carries, i.e. labelling density. As these are determined by the unique nucleotide sequence of each gene and transcript, they will vary from gene to gene. If two samples have been labelled under similar conditions, the length and labelling density of specific transcripts will be similar in the two samples, making it possible to compare the relative abundance of the transcripts in the two samples. A strong hybridization signal from microarray analysis does not necessarily correspond to a highly expressed gene; it could be derived, for example, from a gene that is expressed at a relatively low level but yields long, highly-labelled probe fragments.

Gene expression analysis with microarrays has been applied to numerous mammalian tissues, plants, yeast, and bacteria alike (1, 5, 6, 7, 8). These studies have examined the effects of treating cells with chemicals, the consequences of over-expression of regulatory factors in transfected cells, and compared mutant strains with parental strains to delineate functional pathways. In cancer research microarrays have been used to find gene expression changes in transformed cells and metastases, to identify diagnostic markers, and to classify tumors based on their gene expression profiles (9, 10, 11).

1.4 Genomic analysis

Microarrays are proving to be useful tools for genomic analysis. Identification of new genes by examining nucleic acid sequences derived from open reading frames has proved to be an efficient way of annotating the human genome and facilitating the use of genomic information for experimental purposes (12). Understanding of gene regulation is advanced by elucidation of transcription factor gene interactions. For example, by combining immunoprecipitation of transcription factor-DNA complexes to microarray identification of DNA fragments on a genomic microarray, it was possible to identify functional regulatory elements in the yeast genome (13). Furthermore, microarrays can be used for predicting splice variants of transcripts and analyzing genomic fragments derived from genetic analysis methods, such as genomic mismatch scanning and representational difference analysis (14, 15). Oligonucleotide microarrays have been applied to analysis of known single nucleotide polymorphisms (SNPs) and mutations (16, 17). Samples can be sequenced using microarray hybridization (18), thus providing convenient means for identifying new genetic variants.

1.5 Drug discovery

As a typical drug discovery process takes several years and incurs high costs, and only a few drug candidates result in approved drugs, methods that increase the efficiency of the process and improve the chances of developing effective drugs have been welcomed.

Microarrays have been found to provide useful information in the different stages of the drug discovery process (8, 15, 19). Identification of potential drug targets can be aided by elucidating metabolic pathways by looking for co-expressed genes. The protein targets of drug treatments can be identified by finding a protein that causes the same changes as a drug when removed from cells. Once drug candidates have been identified and selected, microarrays can be used to define their toxic properties by examining expression profiles induced by drug treatments (20). On the other hand, different function modes of drugs were identified based on the gene expression changes they elicited (21).

Chapter 2

GENETIC CONTENT OF MICROARRAYS

2.0 Introduction

The genetic content of microarrays resides in the immobilized nucleic acid sequences on the microarray. The identity of these sequences determines what information can be obtained from array experiments and how reliable this information is. As microarrays enable simultaneous interrogation of up to tens or hundreds of thousands of targets with one or more labelled probes, generation of accurate data demands that only specific interactions result in detectable signals. Several strategies for preparing the immobilized target nucleic acids for microarrays exist (Fig 6). These nucleic acids can be synthesized directly on the microarray or they can be purified cDNA clones, other DNA fragments or oligonucleotides, which are deposited onto the array by a printing process. This flexibility of using either partially characterized sequences or defined oligonucleotides as targets has improved the application of microarray analysis to different biological problems in a number of species.

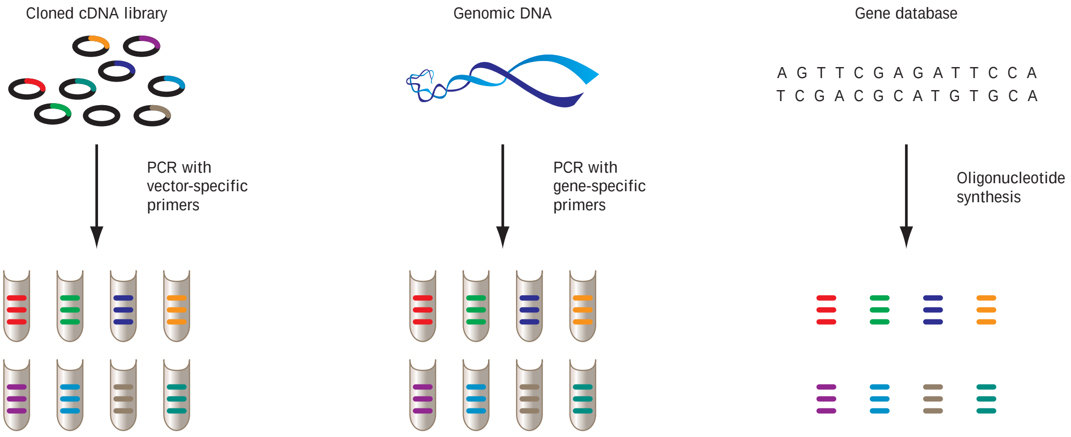

Cloned cDNA library
Genomic DNA
Gene database
A G T T C G A G A T T C C A
T C G A C G C A T G T G C A
PCR with vector-specific primers
PCR with gene-specific primers
Oligonucleotide synthesis

Fig 6. Sources of microarray target sequences. Some of the common strategies for obtaining targets for microarray analysis are illustrated.

2.1 Oligonucleotides as genetic content

2.1.1 Printed oligonucleotide arrays

Oligonucletides can be attached onto microarrays by depositing modified oligos onto a specially treated glass surface. The deposition can be achieved with a variety of contact and non-contact printing methods. See chapter 3, section 3.1.2 on common deposition methods for a detailed overview of this process.

Depending on the source of oligonucleotide content, the length of these oligonucleotides typically varies between 50–70 nucleotides (22, 23). Different microarrays can be easily prepared with the deposition method by choosing different sets of oligonucleotides for array printing. This method is increasingly advantageous because customized microarrays can be prepared in a researcher's own laboratory using microarray spotter equipment.

Regardless of the array fabrication method, the use of oligonucleotides requires that the nucleotide sequences of the intended targets are known. The publication of the human genome sequence as well as partial or complete sequences of several other organisms has facilitated this task. However, the accuracy of the information in the databases has a critical impact on the quality of the arrays. Errors in sequence entries can result in oligonucleotides that do not function in hybridization, because nucleotide mismatches can prevent efficient hybridization from taking place or non-complementary target strands are used by mistake. As more and more genes are identified and their sequences elucidated, the power of oligonucleotide arrays will increase.

2.1.2 Benefits of oligonucleotide arrays

Oligonucleotide targets have several benefits over cDNA targets.

- Different parts of the same gene can be represented on the array. This enables a more robust design of microarray experiments as the same gene can be probed independently for the same information in the same experiment.
- Oligonucleotides can be designed to distinguish between alternative splicing variants as well as different alleles. Oligonucleotides offer precise control over the genetic composition on the arrays. With a judicious choice of oligos, it is possible to discriminate between related gene sequences and study different members of gene families simultaneously.

- Oligonucleotide targets are readily available from commercial manufacturers or synthesized by researchers.
- The time and effort required to prepare oligonucleotides for array printing is less than that required for preparing cloned targets by molecular biology methods.

2.1.3 Design of oligonucleotides

The design of oligonucleotide targets should take into account factors that influence the specificity and strength of hybridization with labelled probes. The specificity can be estimated by comparing the oligonucleotide sequence with known gene sequences. Predicting the strength of hybridization is more difficult, however. Computer algorithms have been developed for selecting target oligonucleotides. Some general rules for oligonucleotide selection have been established:

- Repeat sequences should be avoided, including polynucleotide stretches, repetitive genomic elements, and palindromic sequences.
- The chosen sequences should not be homologous to other genes, but one short homologous stretch may still produce enough specificity in hybridization (22, 24, 25).
- The length of the oligonucleotide, its nucleotide sequence, as well as the positions of mismatches in the oligo, all influence the behavior of the oligo in hybridization.
- It is important to choose a fairly even distribution of all four nucleotides in the sequence.
- Testing of oligonucleotide targets before including them on arrays can help to eliminate sequences that will not perform well.
- The use of computer algorithms may also facilitate the selection of target oligonucleotides (23).

Target sequences may not be accessible to probe molecules near the attachment site on the solid support. mSpacer sequences can be used to increase hybridization efficiency. These are additional sequences added to the oligo sequence to move it further away from the solid support (2). 40-atom long spacers were found to result in up to 150-fold increase in hybridization signal on oligonucleotide arrays (26).

2.2 DNA fragments as genetic content

Development of mechanical microspotting methods and instruments, which can be used to deposit nucleic acid solutions onto glass surface, has enabled the use of cDNA clones and other DNA fragments as microarray targets (27). These methods allow quick and adaptable construction of microarrays that can be customized according to different experimental needs.

2.2.1 Sources of DNA targets

The nucleic acid fragments used for microarray construction can be derived from a number of sources. For gene expression microarrays the fragments are typically derived from either cDNA clones or amplified from exon sequences. Libraries of cDNA clones, expressed sequence tags, clones isolated from subtraction libraries in which the number of highly expressed sequences has been minimized, or PCR-amplified fragments corresponding to open reading frames in genomic DNA have been used as targets (28, 29). It is not always necessary to fully sequence the cDNA clones before using them on microarrays, nor have prior information about their expression in tissues. On the other hand, if the cDNA sequence is known, it is possible to select areas of cDNAs that hybridize with higher specificity to sequences derived from one gene only and which do not hybridize with other related sequences. Many 3' untranslated sequences can also contain repetitive genomic elements that will compromise hybridization specificity and should not be present in microarray targets.

As microarray analysis will usually involve the examination of thousands of fragments in one experiment, acquiring and maintaining large collections of nucleic acid fragments is labor-intensive and expensive. While access to genetic content has previously limited, to some extent, the adoption of microarray technology, the availability of ready-printed microarray slides from both commercial companies and academic consortiums has helped alleviate this problem.

2.2.2 Preparation of DNA targets

Before using for microarray spotting, DNA targets need to be amplified and purified. Typically, PCR amplification is used, and universal primers complementary for vector sequences simplify the process (30). It is possible to amplify the target sequences starting from bacterial cultures, purified plasmids, or RNA, if reverse transcription is performed before amplification. With PCR, it is possible to amplify only part of the DNA target or clone. This allows for the removal of sequences that might compromise hybridization specificity. The amplified DNA needs to be purified to remove enzymes, nucleotides, and buffer components, all of which can interfere with the microarray analysis if present in target solution. Column purification methods, such as GFX™ PCR DNA and Gel Band Purification Kit, can be used for this purpose. Whatever methods are used for amplification and purification, it is most important to verify that the amplified fragments are the right size, do not contain other contaminating sequences and that they are present in known quantities. Agarose gel electrophoresis is a convenient way of performing this analysis. The Ready-to-Run Electrophoresis System, which is capable of separating up to 96 samples in 5 min, is well suited for this task (Fig 7).

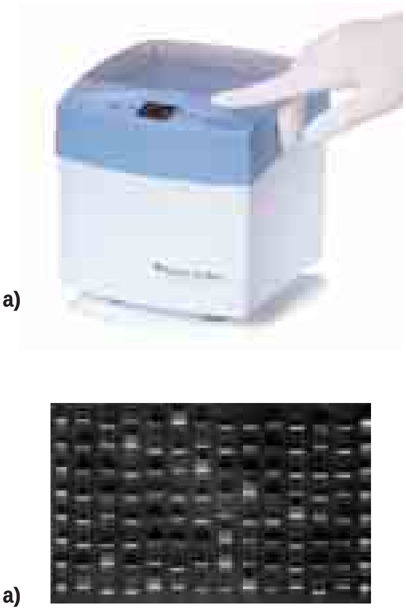


Fig 7. Separation of PCR* fragments with Ready-to-Run Electrophoresis System.

Special care is needed when large collections of nucleic acid fragments are handled simultaneously. It has been estimated that as much as 5–30% of clones in some collections are wrongly labelled or contaminated with other sequences (31, 32). It is important that the genes identified with microarray analysis are verified with other techniques.

2.2.3 Desired properties of DNA targets

An optimal length for DNA targets is between 300–800 nucleotides. Fragments of this length can be efficiently attached to the microarray slide surface, where they form specific and stable hybrids. Figure 8 shows that the retention of UV-immobilized double-stranded DNA targets on aminosilane-treated microarray slides increases slightly with increasing length of the molecules. With increasing length, however, the concentration of DNA required to guarantee the deposition of a sufficiently high number of target molecules within a spot increases. This creates practical problems for using long DNA sequences as targets, as it can become difficult to ensure that the targets are not limiting the hybridization reaction. In order to obtain accurate results from competitive microarray hybridization, the target molecules must be in excess of the corresponding labelled probe molecules. Otherwise, hybridization signals will be saturated.

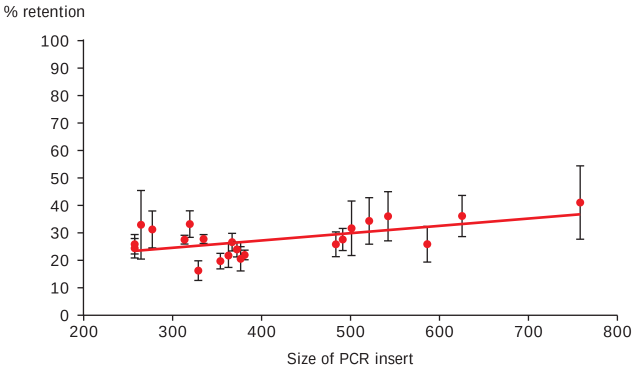


Fig 8. Retention of microarray targets of different lengths on aminosilane-treated microarray slide.

As was a requirement for oligonucleotide targets, DNA targets should not contain repetitive sequences, and they should contain sequences that are unique to one particular gene. Examination of potential cross hybridization between related sequences, such as those derived from a gene family, has revealed that more than 80% homology between targets results in hybridization signals that are not specific for one gene. However, even a lower degree of similarity was found to result in cross hybridization, suggesting that interpretation of microarray data must take the nature of the target sequences into account.

DNA targets do not need to be single-stranded. Spotting from denaturing solutions is enough to render even double-stranded targets available for hybridization. However, single-stranded DNA, which can be generated by asymmetric PCR or by exonuclease digestion of partially protected fragments (2), can also be used as targets in microarray analysis.

2.3 Control targets

Because microarray analysis is a complex process, there is a need for the use of effective controls during the whole process. For gene expression microarrays, the hybridization signal is influenced by a number of variable factors, including the number of specific transcripts in the labelled samples, the labelling method, the properties of microarray printer pens, hybridization conditions, and slide surface chemistry. Furthermore, variation in microarray signal is observed not only between different slides but also between different replica targets spotted onto different locations of the microarray slide. In this context, it is important to be able to draw conclusions from the validity of microarray results and to identify experiments that did not proceed optimally. Control sequences included on microarrays are the key factor to aid in these functions.

Different strategies have been devised for microarray control purposes. These include the use of spiked exogenous sequences of known quantities (25, 33) and housekeeping genes (34), i.e. genes whose expression is not expected to change under experimental conditions.

A control system that consists of both control targets and RNA spikes can monitor most aspects of the microarray process. A control strategy adopted in the Lucidea™ Universal ScoreCard™ combines the use of different types of control targets for spotting onto microarray slides and spikes added to samples before labelling. Together these elements cover aspects of slide printing, sample labelling, slide pretreatment, and hybridization. In a typical experiment, up to 24 replicas of the ScoreCard sequences would be included on a slide printed with 12 pens. This number of replicas allows calculation of quality indicators that report on variation between different pens and spot sets as well as the overall dynamic range and precision of signals.

In order to gain maximum information from the quality of microarrays, positive, negative, ratio, dynamic range controls, and normalization controls are typically used. Table 1 lists the properties and main utilities of these control types. As a microarray hybridization involves thousands of different nucleic acid fragments, it is important that sequences used as controls are selected and functionally tested to avoid unspecific or cross species hybridization. Negative controls, on the other hand, need to represent different nucleic acid sequences to be able to capture the occurrence of random hybridization events. Blank spots that contain no DNA are useful as negative controls too, but are not sufficient on their own.

The use of oligonucleotides as targets allows the use of mismatched sequences as controls. By comparing the signal from the correct sequence to that from the mismatched sequence, the reliability of each signal can be assessed individually (24).

Table 1. Control target types.

Control type	Composition	Purpose
Positive control	■ Pooled genomic DNA	■ Control for labelling and hybridization success
Negative control	■ DNA fragments derived from unrelated species	■ Specificity of hybridization ■ Detection limit
Ratio control	■ Two different sequences spiked into each sample before labelling at different amounts	■ Success of labelling and hybridization ■ Color discrimination
Dynamic range control	■ Different sequences spiked into samples before labelling at different molar amounts	■ Success of labelling and hybridization ■ Color balance ■ Dynamic range of detection ■ Detection limit and saturation of signal

Chapter 3

MANUFACTURING OF MICROARRAY SLIDES

3.0 Introduction

Microarray analysis is invariably performed on a glass slide, which enables the performance of high-throughput miniaturized hybridization assays with fluorescently labelled samples—a significant improvement over the use of membrane support.

Microarray manufacture requires three distinct components:

- production method
- microarray slide
- target genetic content

In this chapter the properties of deposition methods, instruments, and microarray slides are discussed.

3.1 Production methods

3.1.1 Oligo synthesis

Two parallel approaches have been developed for the production of microarray slides. Nucleic acid targets can either be synthesized directly onto the microarray slide, or purified targets can be deposited onto a solid surface that is capable of binding nucleic acids.

Adopting a photolithographic masking method used in the semiconductor industry, oligo synthesis is begun by attaching chemically modified linker groups, which contain photochemically removable protective groups, onto the glass surface (39). By masking different predefined positions of the glass at different steps, it is possible to synthesize different oligonucleotides at different locations. Target synthesis proceeds in a step-wise fashion using a different light-impermeable mask for each round. In each step, the unprotected areas are first activated with light which removes the light sensitive protective groups. Exposure of the activated areas to a nucleoside solution results in chemical attachment of the nucleoside to the activated positions. This process is then repeated by using a different mask and a new nucleotide until all nucleotides have been added to the oligo (35).

This method (Fig 9) produces arrays of small features that are anchored at their 3' ends to the array surface. Each feature is made up of oligonucleotides that all have the same nucleotide sequence. These arrays have a high density: an area of 1.6 cm can contain up to 400 000 features. Additionally, multiple arrays can be synthesized simultaneously onto a large glass wafer, further automating the manufacturing process. The wafers are then cut into individual arrays in preparation for use.

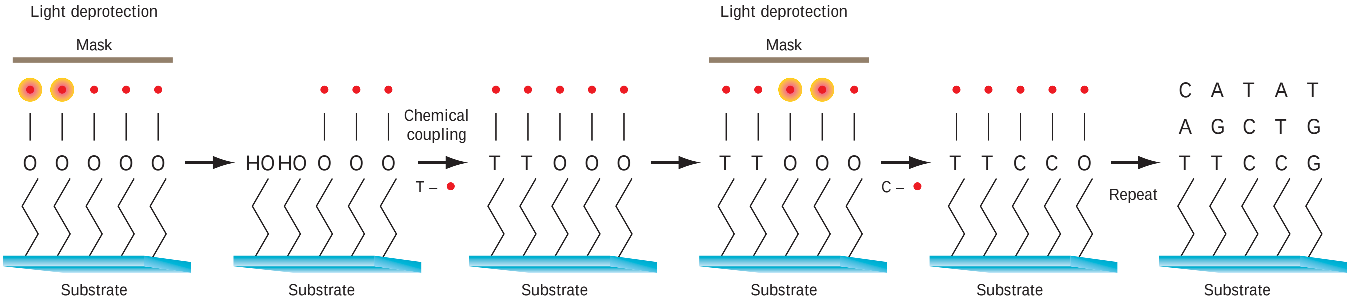


Fig 9. Microarray manufacturing using photolithography.

3.1.2 Deposition

Using common deposition methods, purified nucleic acids are attached to a modified glass slide. Typically, small volumes of nucleic acid solution—nanoliters or picoliters—are transferred onto the glass slide. Deposition methods are equally suitable for preparing microarrays containing oligonucleotides, cDNA sequences, as well as genomic DNA. Deposition methods are commonly used for preparing customized microarray slides.

The deposition chemistry involves a chemical reation between molecular groups on the glass surface and the oligo, resulting in the formation of covalent bonds that bind the ologonucleotide onto the array. There are many different suitable attachment chemistries. One is the coupling of amine-modified oligonucleotides to aldehyde slides. Another is the derivatization of 5' phosphate groups with imidazole, followed by reaction with the amine slide surface. A third is the use of bifunctional cross-linkers to couple aminated oligos to aminated glass (2).

Many different techniques have been developed for the deposition process, some of which are reviewed in this chapter. Regardless of the technique used, however, the manufacturing process should meet several criteria. Variation in the quantity of targets deposited, the shape of spots, the regularity of the array pattern, and the carryover of targets could all detrimentally affect the accuracy of microarray data.

3.1.3 Requirements of microarray spotting methods and instruments

Spot size and density

The microarray spots should be small and discernible from each other. The spots should be deposited in grid-like fashion, at equal distances from each other. It is important to immobilize the slides during printing, as even the slightest movement can distort the microarray pattern.

Spot reproducibility

The spots should be of uniform size and shape, and they should contain equal amounts of the target nucleic acids. This requires careful calibration and matching of individual printing pens.

Environmental control

Environmental conditions can have a significant effect on the quality of the spotted slides. Clean environment is important because dust particles can become fixed onto slides, causing background signals in microarray hybridization and spot finding problems during data analysis. Controlling the humidity helps to avoid changes in sample concentrations due to evaporation during printing and when spots are drying. High humidity levels may cause spots to smear whereas low humidity levels may cause evaporation from the sample plate. Under high temperature conditions targets will dry rapidly at the outer edges of the spot, thereby causing poor spot uniformity. This effect can cause a donut-shaped spot morphology. A humidity between 10–70% has been found to be most suitable for a microarray application.

Sample carryover

There must not be any carryover of previous target during the printing process.

Throughput

The printing process should be fast to allow timely printing of slides. The total time need for sample retrieval, printing, and washing of the printer pens needs to be considered.

3.1.4 Non-contact deposition

Non-contact deposition has been adapted for microarray manufacture from the modern ink jet printing industry. As the name implies, the printing heads do not touch the surface of the microarray. Piezoelectric printing and syringe-solenoid methods are the two common variations of this method.

- In piezoelectric printing (Fig 10) the target solution is drawn into a capillary that is in contact with a piezoelectric crystal. Application of voltage to the crystal results in a slight conformational change, squeezing the capillary. A small volume of sample is deposited onto the glass surface. This method allows for very rapid spotting times. Very small volumes can be delivered, as the distortion of the crystal shape can be accurately controlled. However, this deposition method is prone to problems caused by air bubbles, which can cause poor spot morphology.
- Syringe-solenoid deposition (Fig 11) uses a syringe pump positive displacement method to deposit nanoliter volumes onto a slide. A syringe that provides the pressure source is connected to a micro-solenoid valve. The sample is drawn up the dispensing tip via the syringe. The system is pressurized and the opening of the micro-solenoid valve allows small volumes of sample to be deposited onto the surface. This system is not as rapid as that of piezoelectric printing, and it is not able to deposit sub-nanoliter volumes; however, deposition volumes are very precise and reproducible.

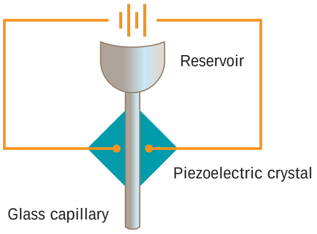


Fig 10. Diagram of a piezoelectric microarray printing system.

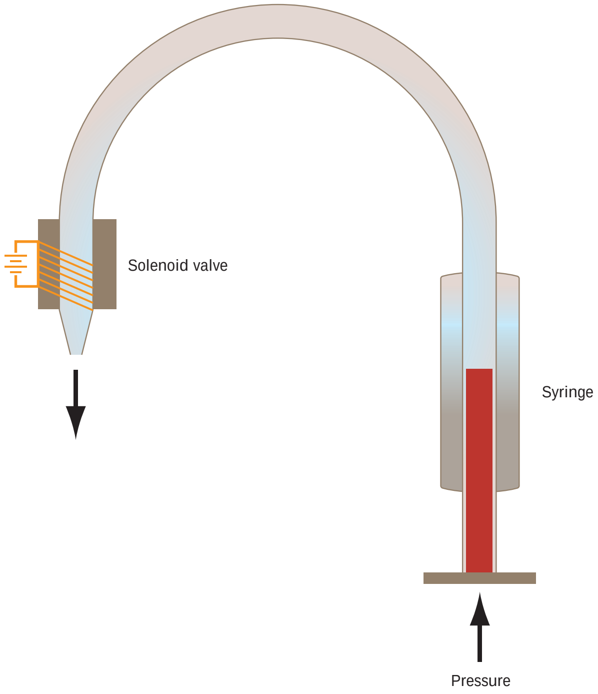


Fig 11. Diagram of a syringe-solenoid microarray printing system.

3.1.5 Contact deposition

In contact deposition, solid, hollow, or split-open pen designs are used to transfer target nucleic acid onto the slide surface. These pens are dipped into the target solution, a small volume of which adheres to the pen. When the pen comes into contact with the slide surface, a fraction of the nucleic acid solution on the pen is deposited onto the glass surface. One sample uptake of the pen allows for several spots to be printed. For achieving high throughput, several pens are used simultaneously, each of which typically deposits a different nucleic acid solution. Successful spot deposition is achieved by using pens that are quantitatively tested to ensure performance, such as those made by Amersham Biosciences.

Contact deposition requires less target nucleic acid solution than the non-contact methods and also results in smaller spots that can be packed more densely on the microarray surface.

Table 2. Comparison of characteristics of different microarray printing methods.

	a		
Microtiter plate well volume (microliters)	10–30	20–50	20–50
Uptake volume (microliters)	0.2–1.0	5–10	5–10
Spot volume (nanoliters)	0.5–2.5	5–100	0.1–10
Spot size (nanometers)	75–250	250–500	125–175

3.2 Lucidea Array Spotter

Lucidea Array Spotter (Fig 12) is a new contact deposition microarray spotter from Amersham Biosciences. The Lucidea Array Spotter is part of the Lucidea platform of products offered by Amersham Biosciences for microarray analysis. These products include Lucidea Array Spotter, Lucidea SlidePro Hybridizer (see chapter 9), and Lucidea Universal ScoreCard (see chapter 11). The key features of Lucidea Array Spotter are:

- Patent pending, stainless steel capillary pens that conserve sample and uniformly deposit picoliter volumes of target (Fig 13). From a single sample uptake of less than 200 nl, up to 150 spots can be spotted in duplicate, across each of 75 slides. The design of the pens (Fig 14) minimizes clogging with target solution and simplifies washing after each sample, resulting in no detectable carryover or mixing of samples during printing. To achieve good spot uniformity, the pens in each pen set are quantitatively tested during manufacturing to ensure performance.
- A newly designed five-step wash system eliminates the possibility of sample carryover as shown with dye-labelled DNA testing (Fig 15).
- Lucidea Array Spotter allows for monitoring and control of humidity and temperature monitoring during spotting (Fig 16).
- The target plates are kept in an area with minimized airflow to reduce evaporation while the printing is in progress.
- The spotting chamber is encased inside the enclosure of the instrument, thus limiting the access of particulates during slide printing.
- Several user-defined spotting modes are available to print arrays in up to four replicates per slide. Lucidea Spotting Pens can handle spotting fluids with significantly different viscosity.
- Control software integrates all aspects of spotter operation.

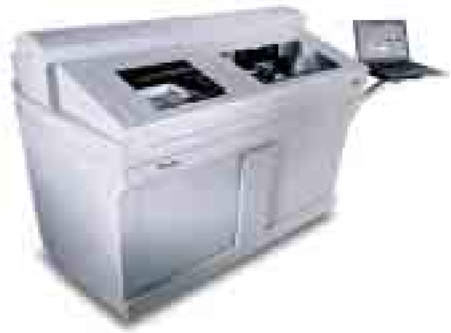


Fig 12. Lucidea Array Spotter.

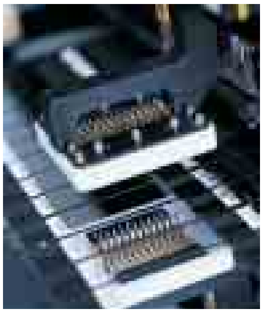


Fig 13. Lucidea Spotting Pen Set.

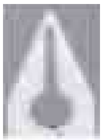


Fig 14. Side view of the tip of Lucidea Spotting Pen. Target solution is drawn by capillary action to the narrow opening in the tip of the pen.

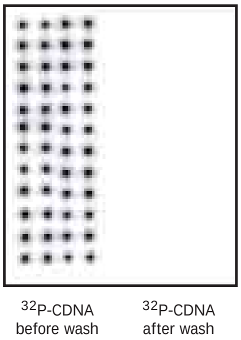
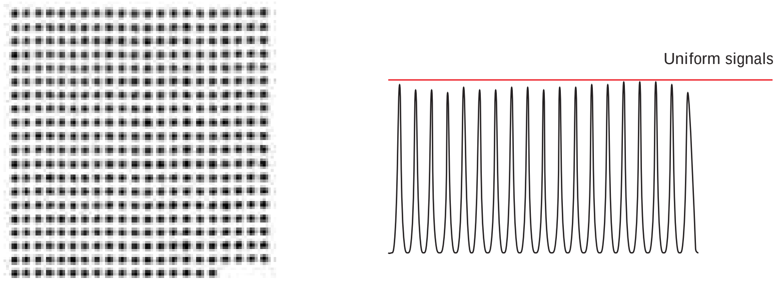


Fig 15. Wash system in Lucidea Array Spotter removes previous target solution efficiently, resulting in virtually no detectable carryover of sample during printing. The above experiment was performed with radiolabelled ^{32}P-cDNA spotted on a nylon membrane. The image on the right shows results after the pens have been washed, dipped into a sample blank, and then spotted.



Spots

Scanned

Fig 16. Controlled temperature and humidity result in relatively even spot intensities and morphology.

3.3 Microarray slides

At present the most commonly used support for microarrays are standard glass microscope slides that offer flat and rigid support with low intrinsic background fluorescence. However, there are quality differences between different manufacturer’s glass slides. Careful analysis of slides before they are used for microarray printing is recommended. Furthermore, it is very important to ensure that microarray slides are absolutely clean.

3.3.1 Slide surface chemistries

Nucleic acids will not attach efficiently to an untreated glass slide. Therefore, different surface chemistries have been developed to facilitate the attachment of targets to the slide. These treatments not only enable the binding of targets, but also determine the density of molecules that can be attached per surface unit.

The uniformity and thickness of the surface coating on the slide is critical for good quality microarray results, as this will influence spot uniformity and morphology, DNA binding, as well as background signals from microarray hybridization. Variation in slide coating can contribute to the variation in microarray signals and decrease the resolution of a microarray experiment. Uneven slide coating can also lead to poor attachment of deposited nucleic acid, which may come loose during microarray hybridization.

Commonly used slide surface modifications include the introduction of aldehyde, amino, or poly-lysine groups onto the slide surface. Aminosilane slides give highly consistent and reproducible data with high signal to noise values, and they are most favorable for use in microarray experiments.

3.3.2 Common slide types

Aldehyde slides

Amino-modified DNA can be attached to microarray slides that have been modified with aldehyde groups (Fig 17). The amino group can be introduced into DNA in a PCR amplification reaction using amino-modified oligonucleotides. The aliphatic amine on the amino-modified DNA acts as a nucleophile, attacking the carbon atom on reactive aldehydes covalently attached to the surface of the slide. An unstable intermediate is converted to a Schiff base through a dehydration reaction ($-H_2O$), and the DNA is bound to the surface. To minimize fluorescent background, the unreacted aldehyde groups are reduced to non-reactive primary alcohols by treatment with sodium borohydride ($NaBH_4$). Aromatic amines on the G, C, and A bases of naturally occurring DNA can also react with aldehyde groups. The efficiency of this side reaction is ~0.01% for short oligonucleotides and ~10% for double-stranded PCR products (36).

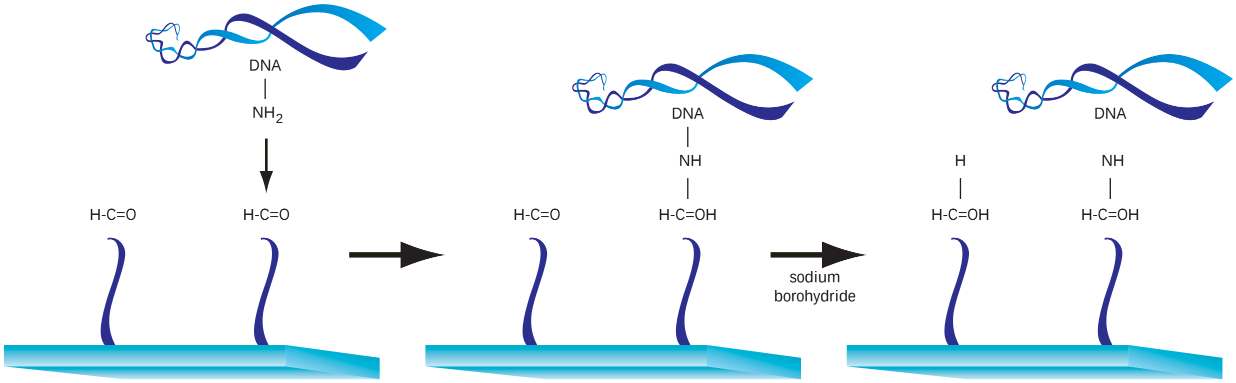


Fig 17. Attachment of amine-modified DNA to aldehyde slide.

Amine slides

Amine groups can be introduced onto microarray slides by treating cleaned glass with aminosilane, such as 3-aminopropyltrimethoxysilane (Fig 18). Vapor treatment of slides gives generally better results than deposition by a dipping method (37). Unmodified DNA can be attached to amine-modified slides, via interactions between negatively charged phosphate groups on the DNA and the positively charged slide surface. This interaction helps ensure denaturation of the DNA as well as increase its binding affinity to the slide surface. UV treatment can be used to further immobilize the DNA onto the slide surface. Attachment via electrostatic interactions is suitable for binding DNA fragments that are longer than 60–70 nucleotides. For attaching oligonucleotides to amine-modified glass, chemical coupling methods must be used (2).

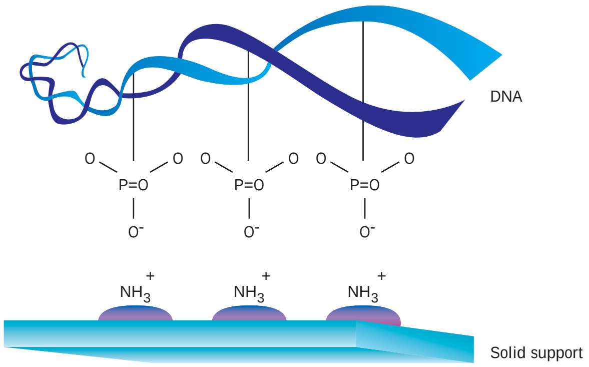


Fig 18. Attachment of unmodified DNA to an amine-modified slide surface. The DNA binds to the surface of the slide via an electrostatic interaction. The positive amines in the silane coating will attract the negative phosphate backbone of the DNA.

Poly-lysine slides

Treatment of the slide with poly-lysine creates a positively charged surface to which unmodified DNA can bind via ionic interactions (38).

3.3.3 Reflective slides

A large proportion of the fluorescent light emitted from the hybridized probe is scattered in all directions when using regular glass arrays. The introduction of a reflective surface below the spotting surface enables a significant amount of this scattered output to be directed towards the detector, hence increasing the amount of signal detected by the system. These reflective slides are constructed by adding a layer of aluminium above the glass surface. Figure 19 shows a diagram of a reflective slide.

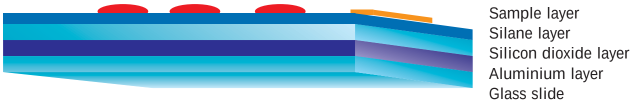


Fig 19. Diagram of the structure of a reflective microarray slide.

Signal enhancement is further achieved if an optimal thickness of silicon dioxide is used as a spacer on top of the reflective layer (Fig 20). It has been found that a thickness corresponding to $^1/_4$ of the excitation wavelength results in optimal signal enhancement for a particular dye. In a typical microarray experiment two different dyes, such as Cy3 and Cy5, are used. As these dyes have different excitation maxima, it is not possible to enhance the excitation of both dyes simultaneously. Since the fluorescence from Cy3 is already enhanced when the dye is bound to molecules, it is more critical to increase the fluorescent signal from Cy5-labelled molecules. Hence, Lucidea Reflective Slides have been designed to specifically enhance Cy5 signals from microarray experiments, resulting in better balanced signals from both dyes.

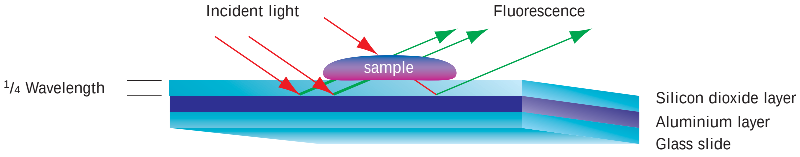


Fig 20. The enhancement of signal by a reflective slide.

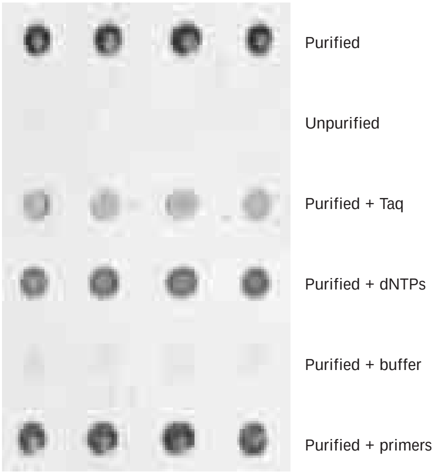


Fig 21. Purification of PCR-amplified targets. cDNA targets were amplified with PCR, then purified with column chromatography. The indicated reagents were added to the purified target DNA before spotting. Significantly decreased hybridization signals were observed from all targets containing impurities as compared with purified targets.

3.4 Target nucleic acids

The third critical component in microarray manufacturing is the target nucleic acid. Factors influencing the choice of target sequences are described in Chapter 2.

Microarray targets must be available in high enough concentration to allow a sufficient number of molecules to be deposited onto the slide. The purity of target solutions is important for both the efficient attachment of nucleic acids to the slide surface and the availability of the immobilized targets for hybridization. PCR-amplified targets must be purified to remove dNTPs, primers, DNA polymerase, buffer salts, and detergents. Column chromatography methods are suitable for this purpose. As shown in Figure 21, the presence of these compounds is detrimental to the success of microarray hybridization.

The targets, once attached to the microarray surface, are only available for hybridization when they are present in a denatured, single-stranded form. This can be achieved by spotting the targets under denaturing conditions. Typically, targets are dissolved in high salt solutions such as 3 × SSC, or in denaturing solvents such as DMSO.

3.5 Critical success factors for microarray preparation

- Always handle microarray slides in a clean environment.
- Never use gloves that contain powder as the powder will invariably get onto the slides and cause background signals.
- Never touch the array surface, only handle slides from sides.
- Only use low fluorescence microscope slides for microarray manufacturing.
- Clean microarray slides efficiently before applying slide surface treatment to them.
- Verify the purity and concentration of targets before using them for slide printing.
- Handle target plates with care to avoid drying out or mixing of targets.
- Follow the instructions provided with the microarray spotter carefully.
- Make sure that spotting is carried out under known and controlled temperature and humidity.
- Microarray slides have a limited shelf life, so prepare and use microarray slides in a timely fashion.
- Always store slides dessicated and protected from light.

Chapter 4

FLUORESCENT LABELS IN MICROARRAY ANALYSIS

4.0 Introduction

Most researchers performing microarray analysis prefer to use fluorescent dyes as labels in these experiments as their use offers high sensitivity of detection and enables detection of different dyes simultaneously. Furthermore, fluorescent dyes do not carry the hazards associated with radioactive markers. In this chapter, the general properties of fluorescence and CyDye™ fluorophores are discussed.

4.1 Definition of fluorescence

Fluorescence can be defined as the molecular absorption of light energy (photon) at one wavelength and its re-emission at another wavelength. Molecules that absorb light are known as chromophores. Molecules that both absorb and emit light are known as fluorochromes, or fluorophores.

Light is a high frequency electromagnetic wave, and the energy of the photon is inversely proportional to its wavelength (λ). Thus photons towards the blue end of the spectrum, i.e. light photons with shorter wavelengths, have a higher energy than those towards the red end of light spectrum (Fig 22).

The process of fluorescence is a three-phase one, consisting of excitation, the excited state, and emission.

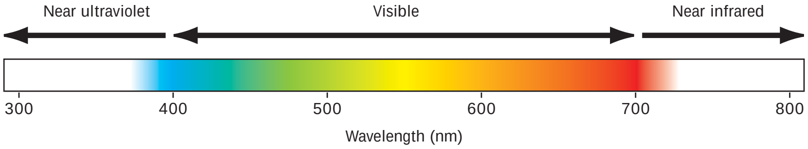


Fig 22. Light spectrum.

4.1.1 Excitation

Extinction coefficient (ε) is a measure for a fluorophore's ability to absorb light energy. When a photon of light energy (hv_{EX}) of the appropriate wavelength is absorbed by a fluorophore, an electron is boosted to a higher, unstable excited energy state. The difference between the ground state (S_0) and the higher energy state (S_n) is a property of the fluorophore; it is equivalent to the energy of the photon absorbed. Because photons with shorter wavelengths have higher energy, the shorter the wavelength of the absorbed photon, the higher the excited energy state reached by the fluorophore. The wavelength at which the fluorophore has maximum excitation is determined by the structural properties of that fluorophore.

4.1.2 The excited state

The excited state typically lasts a fraction of a second. During this state some of the energy absorbed may be dissipated in the form of vibrational and rotational energy, often resulting in localized heating. The fluorophore thus loses some of the energy it has absorbed from excitation, prior to any fluorescent emission taking place. It is for this reason that the quantum yield (ϕ) of a particular fluorophore (ratio of the number of photons emitted to the number of photons absorbed) is between 0 and 1.0. The quantum yield of a fluorophore can be greatly influenced by the medium in which it resides. For example, unconjugated Cy3 in phosphate buffered saline solution (PBS) has a quantum yield of 0.04; thus a large proportion of the energy absorbed by each dye molecule is lost to its surrounding solution. However, in glycerol the quantum yield increases more than ten-fold to 0.52.

4.1.3 Emission

Once the fluorophore has reached the lowest vibrational energy level within the electronic excited state (S_1), the electron falls from the excited state to the ground state (S_0). It is at this point in the decay process that light is emitted at a specific wavelength (hv_{EM}). Because some energy between excitation and emission has already been lost, the emitted photon has less energy than the original photon absorbed by the fluorophore (Fig 23). Therefore, the emitted light has a longer wavelength. The difference between the maximum excitation wavelength and the maximum emission wavelength is known as the Stokes shift ($hv_{EX} – hv_{EM}$).

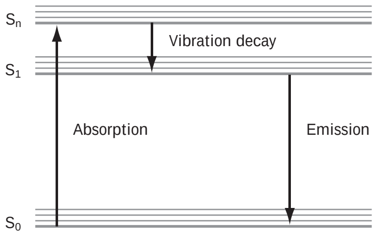


Fig 23. Diagram illustrating the energy levels of the fluorescence process.

4.1.4 Photobleaching

The fluorescent process is rapid (10^{-8} seconds) and cyclical, enabling the fluorophore molecule to be excited repeatedly. It must be considered, however, that the excited state of a fluorophore is generally more chemically reactive than the ground state. In conditions of intense light, the fluor may gradually lose its fluorescent properties, a phenomenon known as "photobleaching". This results in lower fluorescent output from the fluorophore after prolonged exposure to light or repeated excitation. For more photosensitive dyes, such as fluorescein, photobleaching may be a significant factor when using instrumentation with high laser power. In contrast, as seen in Figure 24, CyDye fluors are more resistant to photobleaching, which makes them more suitable for multiple applications.

An excellent approach to reduce photobleaching is to maximize detection sensitivity so that the excitation intensity can be reduced.

The brightness (intensity of output) of a fluorophore is proportional to both the extinction coefficient (ε, the molecule’s ability to absorb light energy) and the quantum yield (ϕ, the molecule’s efficiency to re-emit light). Both of these are constants under specific static environmental conditions. Consequently, fluorophores with very different characteristics may give a comparable signal brightness. Fluorescein, for example, has a molar extinction coefficient of ~70 000 and a quantum yield of ~0.9, whereas Cy5 has values of ~200 000 and 0.3 respectively. However both are of similar brightness.

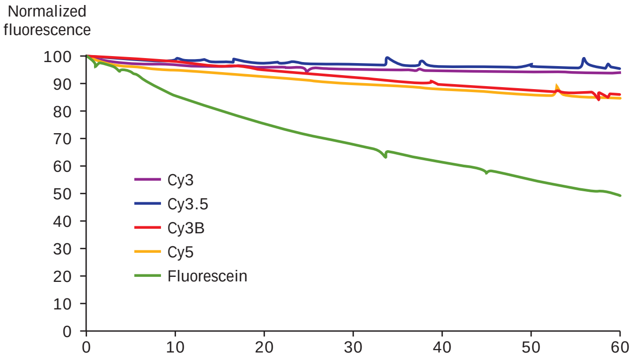


Fig 24. Photostability of fluorophores under "natural" light conditions.

4.2 Applications of fluorescence

The fluorescent process, combined with appropriate imaging instrumentation, enables a sensitive and quantitative detection method for microarray expression analysis.

4.2.1 Benefits of fluorescent labelling

The particular Stokes shift properties of individual fluorophores make it possible to separate excitation light from emission light with the use of optical filters. Good spectral separation enables high sensitivity of detection and yields low background.

By choosing fluorophores that have different pairs of excitation and detection wavelengths, it is possible to excite and detect multiple dyes in the same sample. This method enables multiplexing with dyes—labelling two or more samples with different dyes that have different absorption and emission spectra—and makes it possible to analyze several samples simultaneously. Figure 25 shows the absorbance and emission spectra of Cy3 and Cy5, the most widely used pair of fluorescent dyes in microarray analysis. The minimal overlap between the Cy3 and Cy5 spectra demonstrates how it is possible to detect both dyes with minimal cross talk (overlap) between their respective signals.

Another advantage of using fluorescent dyes as labels is that they are less hazardous than radioactive compounds and offer significantly increased stability, or longer shelf life. The availability of fluorophores conjugated to many different chemical groups enables the labelling of nucleic acids, proteins, lipids, and carbohydrate molecules. Furthermore, fluorescent dyes can also be used as reporters to detect changes in pH, ion concentrations, or dye environment.

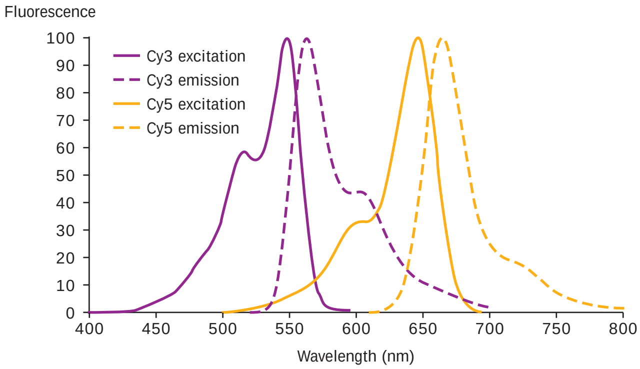


Fig 25. Excitation and emission spectra of Cy3 and Cy5 dyes.

4.2.2 Fluorescent quenching

Fluorescent quenching (Fig 26) causes decreased fluorescent signals. It occurs when two or more fluorophores are in close proximity to each other and the excitation energy is dissipated in interactions between the adjacent dye molecules, rather than emitted as fluorescent light. Chemical structures as well as photochemical properties of dyes determine the distance at which two fluors will quench.

Quenching can occur when samples have been labelled too densely or when too much labelled sample is used in hybridization. Over-labelling not only results in the loss of linearity between fluorescent signal emitted and the number of fluors present, but will, in extreme cases, reduce the signal to levels that cannot be observed. In microarray analysis, quenching may also occur when two probe strands come into close proximity of each other. This is likely to be most apparent in the presence of highly expressed transcripts, where a very large number of labelled molecules are bound densely at a precise location. See chapter 6, section 6.2.4 for further overview of this process.

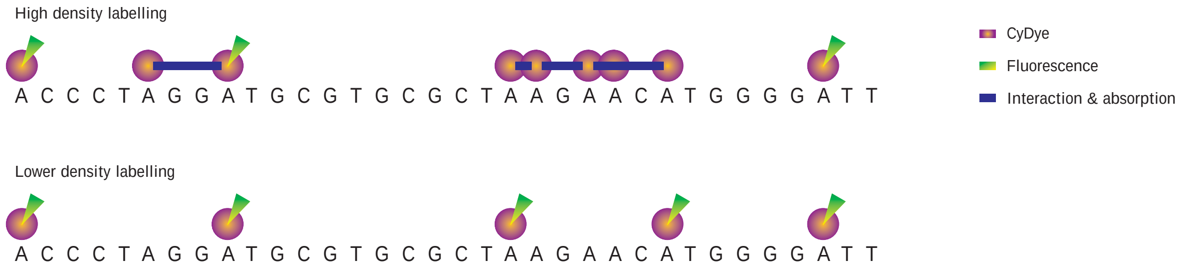

High density labelling
A C C C T A G G A T G C G T G C G C T A A G A A C A T G G G G A T T
Lower density labelling
A C C C T A G G A T G C G T G C G C T A A G A A C A T G G G G A T T
CyDye
Fluorescence
Interaction & absorption

Fig 26. A schematic representation of the effect of quenching under varying labelling conditions. Fluorescent signal from the same nucleic acid fragment labelled with high and low densities is depicted. Green shows fluorescent emission, whereas purple denotes dissipation of energy between closely spaced fluors (pink circles).

4.3 CyDye fluorophores

4.3.1 Chemical structure of cyanine dyes

The cyanine, or CyDye, family of fluorescent dyes were first used in the photographic industry as film sensitizers. They were subsequently discovered for use in molecular biology applications when a CyDye was coupled to a N-succinimide ester, to form the first dUTP (40, 41).

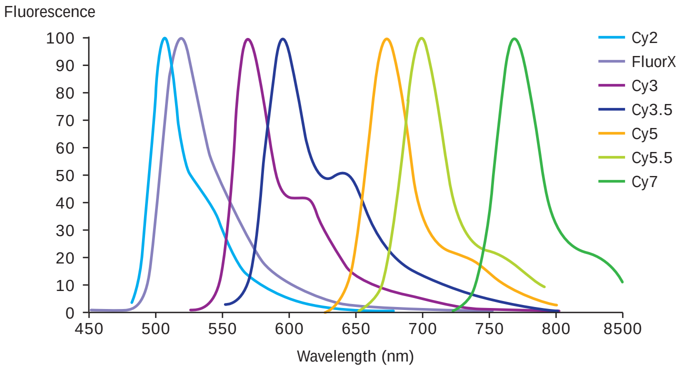
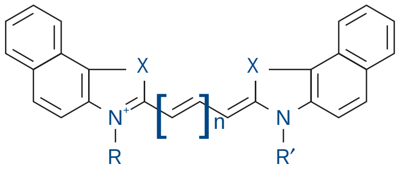


Fig 27. The emission spectra of some CyDye fluors.

This family of fluors consists of a chemically related group of dyes whose emission spectra span the spectrum of visible light (Fig 27). CyDye fluors share a core structure consisting of two heterocyclic indocyanine ring structures joined by a polymethine bridge (Fig 28). Each dye differs in the structure of this bridge. Adding pairs of conjugated C atoms to the polymethine chain results in a wavelength shift of ~100 nm, for example Cy3 (550 nm) and Cy5 (650 nm). An important modification of Cyanine dyes is sulfonation of the indocyanine rings, as shown in Figure 29. The sulfonate acid groups increase the solubility of the dyes. In addition they reduce aggregation of dye molecules, as the introduction of negative charge makes the dye molecules repel each other. This results in a decrease of fluorescence quenching.



X = O, NR, $C(CH_3)_2$, S, Se
R and R′ = alkyl or aryl
n = 0, 1, 2, 3, 4 or 5

Fig 28. Core structure of the cyanine dyes.

The multiplexing properties of CyDye fluors were further increased by synthesizing Cy3.5 and Cy5.5 (Fig 30). The addition of benzene rings shifts their absorbance and emission spectra up by approximately 30 nm to the red. Two additional sulfonate groups are needed to increase solubility in order to overcome aggregation due to benzene rings.

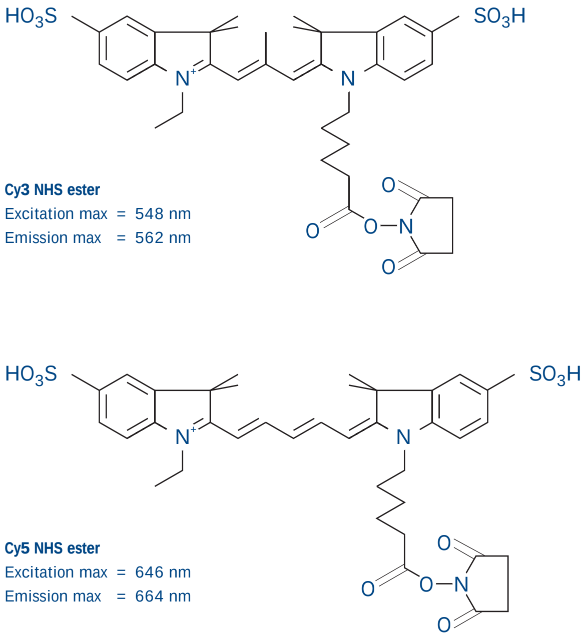


Fig 29. Structure of Cy3 NHS ester (3 carbon bridge) and Cy5 NHS ester (5 carbon bridge).

Cy3 NHS ester
Excitation max = 548 nm
Emission max = 562 nm

Cy5 NHS ester
Excitation max = 646 nm
Emission max = 664 nm

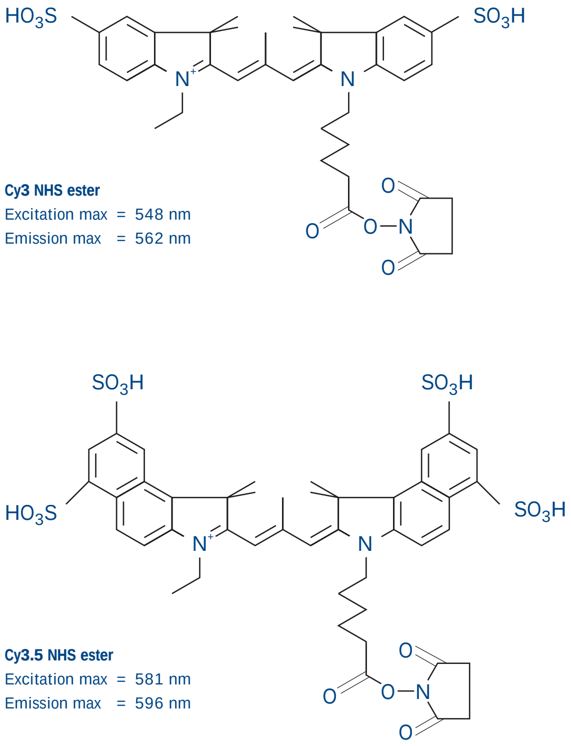


Fig 30. Structures of Cy3 and Cy3.5 NHS esters.

Cy3 NHS ester
Excitation max = 548 nm
Emission max = 562 nm

Cy3.5 NHS ester
Excitation max = 581 nm
Emission max = 596 nm

4.3.2 Fluorescent and chemical properties of CyDyes

Cy3 and Cy5 have become the most commonly used pair of CyDyes. This can be attributed to the following factors:

- The photostability of Cy3 and Cy5 is higher than that of other widely used dyes (Fig 24), making the use of CyDye fluors more practical.
- Cy3 and Cy5 are bright dyes that give strong fluorescent signals.
- The good spectral separation of Cy3 and Cy5 means that each can be excited at a different wavelength and their emissions can be detected separately. This requires the use of two different lasers, typically a 532 nm and 633 nm laser for Cy3 and Cy5, respectively. By choosing optical filters that only collect emitted light from part of the spectra, Cy3 and Cy5 signals can be measured with minimal overlap.
- The fluorescence of Cy3 and Cy5 is minimally affected by factors such as pH and the presence of DMSO. Additionally, these dyes can withstand temperatures and conditions normally encountered in molecular biology applications.

4.3.3 Handling of CyDye fluors

Correct handling of CyDye and CyDye compounds helps to conserve their fluorescent properties. When handling these substances, observe the following precautions:

- Minimize the exposure of CyDye compounds to all light sources.
- Store CyDye in amber tubes, in light-safe containers, or wrapped in aluminium foil to protect them from light during storage.
- Take CyDye out of their protective container only when ready to use and return to dark immediately after use.
- If using CyDye nucleotides, prepare single use aliquots to avoid freeze thaw.
- Protect CyDye NHS esters and other CyDye labelling conjugates such as CyDye Direct™ from moisture by storing dessicated.
- Do not store CyDye in solutions containing concentrated amines. Phosphate buffer or water is preferred over Tris-buffers.

Table 3. Photochemical properties of selected CyDye fluors.

	b a	a	a
Cy2	489	506	~150 000
Cy3	550	570	150 000
Cy3.5	581	596	150 000
Cy5	649	670	250 000
Cy5.5	675	694	250 000
Cy7	743	767	~250 000

4.4 Use of fluorescent dyes in microarray analysis

Multiple labelling strategies have been developed for incorporating fluorescent labels into microarray probes. These techniques are discussed in chapter 6. The photochemical properties of fluorescent dyes, especially the positions of their excitation and emission peaks, determine the specification of scanning instruments, the laser type, and the choice of emission filters that are required to separate and detect fluorescent signals from particular dyes. The popularity of pairing Cy3 and Cy5 as labels has led to the development of microarray scanning instruments that are optimally suited for detection of these dyes.

General requirements for detecting fluorescent signals in microarray analysis are:

- High enough resolution to image a large area (2 × 6 cm) in a short period of time.
- At least two fluorescent spectra must be distinguished to accommodate differential gene expression experiments using two fluorescent dyes.
- The wide range of message abundance levels requires an instrument with a low fluorescent detection threshold to allow detection of rare messages and wide linear dynamic range to measure the more abundant messages.
- The entire area of the microarray must be scanned uniformly to ensure reproducibility.

4.4.1 Characteristics of fluorescent detection

The energy of the emitted fluorescent light is a statistical function of the available energy levels in the fluorochrome, but it is independent of the intensity of the light used to excite the fluorophore. In contrast, the intensity of the emitted fluorescent light varies with the intensity and wavelength of incident light and the brightness and concentration of the fluorochrome. When more intense light is used to illuminate a sample, more fluorochrome molecules are excited, and the number of photons emitted, i.e. the number of electrons falling to the ground state, increases. If the illumination is very intense, all the fluorochrome molecules are in the excited state most of the time—this is called saturation.

When the illumination wavelength and intensity are held constant, as with the use of a controlled laser light source, the number of photons emitted is a linear function of the number of fluorochrome molecules present. At very high fluorochrome concentrations, the signal becomes non-linear because the fluorochrome molecules are so dense that excitation occurs only at or near the surface of the sample. Additionally, some of the emitted light is reabsorbed by other fluorochrome molecules (self-absorption).

The amount of light emitted by a given number of fluorochrome molecules can be increased by repeated cycles of excitation. In practice, however, if the excitation light intensity and fluorochrome concentration are held constant, the total emitted light becomes a function of how long the excitation beam continues to illuminate those fluorochrome molecules (dwell time). If the dwell time is long relative to the lifetime of the excited state, each fluorochrome molecule can undergo many excitation and emission cycles.

Measuring fluorescent light intensity (emitted photons) can be accomplished with any photosensitive device. For example, for detection of low-intensity light, a photomultiplier tube (PMT) can be used. This is simply a photoelectric cell with a built-in amplifier. When light of sufficient energy hits the photocathode in the PMT, electrons are emitted, and the resulting current is amplified. The strength of the current is proportional to the intensity of the light detected. The light intensity is usually reported in arbitrary units, such as relative fluorescence units (RFU).

If fluorescent samples are detected with a system that uses an intense light source to excite the dyes, photobleaching can occur. This irreversible destruction of an excited fluorophore will result in a loss of brightness, or emission light intensity. As photobleaching is a consequence of excitation, altering the characteristics of detection, such as increasing the voltage of PMT to allow more sensitive detection, will not cause it. Microarray slides hybridized with Cy3- or Cy5-labelled probes can be scanned several times with commercial microarray scanners without considerable loss of fluorescent emission.

Chapter 5

PREPARATION OF RNA SAMPLES FOR MICROARRAY ANALYSIS

5.0 Introduction

There are multiple steps involved in isolating and preparing RNA samples to be used for microarray analysis. Discussed in this chapter are factors affecting the quality of analyzed RNA, protecting RNA samples from contamination and degradation, purifying RNA, and finally, characterizing the purified RNA.

5.1 Factors affecting RNA quality

The quality of information obtained from microarray experiments is primarily dependent upon the quality of RNA analyzed. Ideally, the RNA should be devoid of DNA, protein, carbohydrates, lipids, and other compounds. The presence of these substances will not only make it difficult to correctly estimate the amount of RNA present in the sample, but can contribute to fluorescent background signals in the array hybridization. Degradation of RNA, whether by enzymatic or chemical means, results in the loss of gene expression information from the labelled samples. Furthermore, if the quantity or quality of the two samples being compared differ, misleading conclusions can be made.

Compared with DNA, RNA is relatively unstable and can be degraded either enzymatically, chemically or physically. Ribonuclease enzymes (RNAses) degrade RNA into short oligonucleotides in a rapid reaction. They are present in all cells and can be derived from a variety of environmental sources, such as the hands, skin, and hair; bacterial or viral contamination of solutions; or remnants of previous reagents in lab glassware. Inactivation of these enzymes is difficult; therefore it is essential that precautions are taken to ensure that RNA degradation is minimized during isolation, purification, and storage.

5.2 Protecting RNA from degradation

RNA can be degraded if it comes into contact with any source of RNAse. Discussed in this section are the many ways in which RNA samples can become contaminated and how to protect them from subsequent degradation if contamination occurs.

5.2.1 Protecting RNA from contamination by environment

An easy way of accidentally contaminating RNA preparations is by transferring nucleases from the investigator's hands, skin, or hair to the sample. The transfer of RNAses can also take place via equipment, bench surfaces, and door handles. Common molecular biology protocols, such as plasmid preparation, involve the use of high amounts of RNAses and can lead to RNAse contamination if handled in the same location. In order to protect RNA samples from contamination, the following precautions are recommended:

- Wear disposable gloves while handling RNA samples and while preparing other solutions for RNA work. Use a clean pair of gloves if potential contamination of any kind occurs.
- Perform RNA work in a separate area of the laboratory where no RNAses are allowed. Before RNA handling, clean the bench surface with a detergent, such as RNAse ZAP™ (Ambion).
- Lab equipment, such as tissue homogenizers, non-disposable centrifuge tubes, gel tanks, and trays that can come into contact with RNA, should be reserved for RNA work. If this is not possible, large equipment or those items made from materials that do not withstand autoclaving temperatures should be cleaned with RNAse inactivating detergents.

5.2.2 Protecting RNA during preparation of reagents

In order to obtain good quality RNA and to maintain its integrity during subsequent analysis, all reagents should be prepared so that they do not contain any traces of RNAses.

- Use only disposable plastic tubes and pipette tips for RNA work. Clean plastic-ware should be baked at 120 °C for at least 20 min to reduce RNAse (and other nuclease) contamination. Before baking, pack centrifuge tubes into glass beakers and cover them with aluminium foil. Additionally, wrap tip racks in foil to keep the baked items free of contamination after treatment.
- Set aside reagents for use in RNA work only. Always use baked or disposable spatulas and weighing trays for measuring out reagents.

- Treatment with diethyl pyrocarbonate (DEPC) renders RNAses inactive and can be used to clean solutions and labware before use in protocols involving RNA. As DEPC is toxic it should be handled with appropriate care. Prepare 0.2% (v/v) solution of DEPC in water and soak clean equipment and glassware in the solution for at least 1 h. Rinse the treated equipment with sterile water and let dry in a clean place where no further contamination is encountered. Finally, autoclave the equipment/labware in sealed autoclave bags. This is necessary as autoclaves become contaminated by RNAses from other autoclaved materials or from the water used in the autoclaving process.
- Reagent solutions can be treated by adding 0.2% (v/v) DEPC into the solution. After the solution has been left to stand for a couple of hours, it can be autoclaved to remove DEPC by heat degradation. Solutions containing Tris-buffers cannot be prepared in this way. However, water that has been treated with DEPC and then autoclaved, can be used to make up any Tris-buffers.
- Use DEPC-treated water in all RNA protocols.
- Store sterilized equipment and solutions unopened in a clean environment, away from any potential sources of contamination.

5.2.3 Protecting RNA during isolation process

Isolation of RNA requires disruption of cellular structures, which leads to the release of RNAses. Rapid degradation of RNA follows if these RNAses come in contact with RNA during cell homogenization. The following protocols are recommended:

- Use strong denaturing agents such as guanidium isothiocyanate (GITC) or guanidium hydrochloride in RNA extraction protocols as they will denature RNAses (and other enzymes) efficiently and quickly.
- RNAse inhibitor, which is a placental protein that inactivates RNAses by binding to them, can also be used during RNA isolation. However, the effectiveness of the RNAse inhibitor can be affected by the composition of extraction solutions, and denaturation of the inhibitor can release active RNAses.
- The longer the time elapsed between collecting cells or tissue and preparing a denatured homogenate, the more chance there is for RNA degradation to take place. Furthermore, during this time, changes in gene expression can also take place as cells react to the change in their environment. Therefore it is good practice to work as quickly as possible when preparing biological samples for RNA extraction.

- Harvest cultured cells directly into a solution that contains GITC to ensure minimal degradation. Pipette cell lysis solution directly onto a washed cell monolayer. This will lead to immediate cell lysis. Scrape the cells off and transfer into a centrifuge tube. The lysate can be homogenized by drawing it several times through a needle with a syringe or by vigorous mixing until the lysate becomes clear and homogeneous. Cell lysates prepared in this way can be stored frozen, preferably at -70 °C, until RNA isolation protocol is completed.

- Freeze tissue samples rapidly in liquid nitrogen, as this helps to minimize RNA degradation. It is advisable to cut large samples into small pieces before freezing them as this will greatly facilitate their subsequent use for RNA extraction. These samples must be stored frozen, preferably in liquid nitrogen. Thawing of the samples, at any stage, will result in RNA degradation as the freezing process causes some cellular damage. For best results, mechanically pulverize frozen tissue samples while they are kept cold with liquid nitrogen. This can be done with a pestle and mortar. The cold powder should then be dissolved in lysis buffer containing GITC to prepare a clear homogenate.

5.2.4 Protecting RNA during storage and handling

RNA should be protected during storage and handling. The following protocols are recommended:

- RNA is not stable at alkaline pH. This property can be exploited to degrade RNA selectively from DNA. However, if the aim is to maintain RNA intact, all solutions that come into contact with RNA should be neutral or mildly acidic. As the pH of laboratory water can vary, using dilute buffers, such as TE pH 7.6, is recommended.

- RNA degradation is more rapid at high temperatures. For long term storage, storing RNA solutions at -70 °C is recommended. All RNA solutions should be stored on ice while working with them and kept thawed for the minimum time needed. As with other nucleic acids, avoiding freeze-thaw cycles is important. If large amounts of RNA are prepared at one time, it is recommended that the purified nucleic acid is aliquoted for storage and only the required number of aliquots are thawed at any time.

5.3 Choosing an RNA isolation method

5.3.1 Methods for purifying total RNA

Numerous RNA isolation methods have been published and a variety of RNA isolation kits are available (42, 43). The key criteria in choosing a method should be to achieve a high yield of intact and pure RNA. Obtaining long RNA molecules can be problematic, and the choice of purification method will influence results (44).

5.3.2 Critical factors in RNA isolation

The main factors in isolating good RNA are the composition of the cell lysis buffer, the method of cell disruption, and the method used for separating RNA from protein, DNA, and other compounds. The nature of the biological sample is also relevant as some RNA isolation methods may be more suitable for certain tissues. Whereas soft tissues or cultured cells disrupt quickly and efficiently, and methods using mild cell lysis buffers can give good results, harder tissues containing large amounts of connective tissue, such as muscle, will require the use of strong chaotropic agents such as GITC. Amersham Biosciences RNA extraction kits, such as QuickPrep™ Total RNA Extraction Kit, RNA Extraction Kit, and QuickPrep *Micro* mRNA Purification Kit, all contain either guanidium hydrochloride or guanidium isothiocyanate in the lysis buffer, and are suitable for use with a wide variety of cells and tissues. In general, RNA preparations that use chaotropic agents in the lysis buffer tend to give the best results. However, the protocols are more laborious, involve the use of toxic chemicals, and take longer.

Choosing an efficient cell/tissue disruption method for RNA extraction is important. If cell lysis is not complete, the yield of RNA will be compromised. Mechanical disruption using tissue homogenizers, vigorous vortexing, needle and syringe, sonication, pestle and mortar, and bead milling are among commonly used methods. The main considerations in choosing a disruption method are the amount of each sample and the time that is needed for preparation of a cell lysate.

A variety of techniques can be used to differentially separate RNA from other cellular compounds. Precipitation with lithium chloride, acid extraction phenol/chloroform, binding to an absorbent matrix or cesium chloride gradient centrifugation can be used successfully to purify RNA. However, the quality and quantity of RNA obtained with these methods can vary.

The presence of contaminating DNA in total RNA samples can cause problems in microarray analysis. Most labelling methods will label both RNA and DNA with equal efficiency. Labelled DNA can hybridize with microarray targets and can lead to high level hybridization signals that are not derived from transcripts. RNA cannot be quantitated separately from DNA, so an accurate estimation of the amount of RNA in contaminated RNA preparations is impossible. Therefore, it is advisable to treat total RNA preparations to remove DNA contaminants before using the RNA for labelling. This can be achieved with DNase I treatment or by using CsCl gradient centrifugation to separate RNA from DNA.

5.3.3 Isolating RNA from difficult samples

The nature of some biological samples may necessitate the use of modified RNA extraction strategies to avoid contamination of RNA samples with other compounds. For example, plant tissues can contain polyphenols and polysaccharides. Precipitation with polyvinyl pyrrolidone can be used to remove these substances from RNA preparations. The hardness of cell walls and outer protective structures can also pose a problem. Freezing the samples followed by mechanical grinding may be necessary to efficiently disrupt cell walls and to release cellular RNA. In some cases, as with yeast that has cell walls that can form capsules, disruption of cellular structures increases access to RNA. Digestion with enzymes, such as zymolase, can be used to weaken the cell walls before mechanical disruption with bead milling to lyse the cells. Similarly, isolation of bacterial RNA benefits from the use of enzymes that digest and weaken outer supportive structures. Lysozyme treatment followed by mechanical bead milling is a suitable approach for disrupting bacterial cells for RNA extraction.

5.3.4 Purification of eukaryotic mRNA

Most eukaryotic transcripts contain a poly-A tail, and this property can be exploited to separate transcripts from other RNA molecules. Incubation of total RNA with oligonucleotides containing a poly-T sequence, otherwise called oligo(dT), will result in the hybridization between the poly-A tail of transcripts and the oligonucleotides. By attaching the oligo(dT) to a solid support, it is possible to specifically separate transcripts away from other RNA molecules. QuickPrep *Micro* mRNA Purification Kit uses oligo(dT) cellulose for extraction of mRNA.

Although purification of mRNA lengthens the sample preparation protocols, it provides several benefits for microarray analysis:

- Probes prepared from mRNA usually give higher signal to noise values on arrays than probes prepared from similar amounts of total RNA.
- Total RNA preparations are more likely to contain compounds other than RNA, which can interfere with the labelling or hybridization steps.
- The yield of labelled cDNA is higher from mRNA than from total RNA, because alternative priming strategies that use oligo(dT) can be used.
- It is easier to prepare labelled probes corresponding to the 5' ends of transcripts from mRNA populations than from total RNA.

5.3.5 Purification of prokaryotic mRNA

Purification of mRNA from prokaryotes is difficult as most transcripts lack poly-A tails. However, strategies have been developed to polyadenylate 3' ends of bacterial transcripts in crude extracts.

Enrichment for bacterial mRNA can also be achieved by selective degradation of ribosomal RNA. By synthesizing first-strand cDNA selectively from ribosomal RNA with the use of specific primers, RNAse H can be used to degrade the RNA strand in the resulting double-stranded hybrid. DNAse I digestion can then remove the DNA strand, resulting in the enrichment of transcripts. Up to 80% enrichment can be achieved with this method (45).

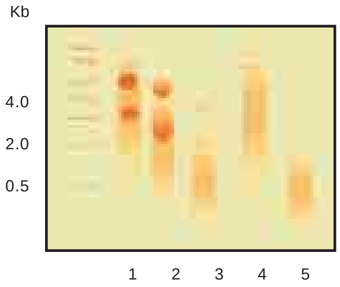


Fig 31. Schematic illustration of typical results obtained from analyzing the quality of RNA with a denaturing agarose gel. Denatured RNA samples were electrophoretically separated in an agarose gel and visualized with UV light after ethidium bromide staining. Lane 1. Intact total RNA with both ribosomal RNA bands present as sharp and bright bands. High abundance transcripts can be discerned as distinct bands. Lane 2. Partially degraded total RNA. Although ribosomal bands are still visible, the average size of RNA is smaller, and no distinct transcript bands are visible. Lane 3. Badly degraded total RNA. Most of the RNA is shorter than 1 kb, and ribosomal RNA bands appear as smears. Lane 4. Intact mRNA. Fragments longer than 4 kb should be visible and abundant transcripts should appear as distinct bands. Lane 5. Degraded mRNA. The transcripts appear as a fast migrating smear.

5.4 Characterization of purified RNA

Microarray gene expression data is derived from a comparison of hybridization signals obtained from two samples. Accurate results are only obtained when the two samples are of the same quality. Comparison of a partially degraded RNA sample to an intact sample can artificially show some genes as being more highly expressed in the better quality sample. Likewise, differences in the amount of RNA in different samples can also give biased data. Therefore, it is highly advisable to verify both the quantity and quality of the RNA or mRNA sample before their use in microarray analysis.

5.4.1 Measuring the amount of RNA

The amount of RNA can be quantified by measuring the absorbance of RNA solution at 260 nm. Pure RNA solution that contains 40 µg of RNA per mL will give absorbance of 1 AU. This method works well with clean RNA samples that are devoid of other contaminating substances. Unfortunately, this is rarely the case with total RNA samples purified with simple methods. Proteins and DNA or other compounds, such as those released from some affinity chromatography columns, will absorb at 260 nm. This absorbance is indistinguishable from that of RNA and will give an artificially high estimate for the amount of RNA in the sample. By measuring absorption spectra from 200 to 350 nm, some conclusions on the purity of RNA can be made. However, the presence of DNA in the sample can still go undetected. Therefore, it is important to use an RNA isolation method that specifically removes DNA from the purified sample.

5.4.2 Verifying the quality of RNA

Agarose gel electrophoresis performed under denaturing conditions can be used to analyze the quality of RNA. Gels containing formaldehyde have been traditionally used for this purpose, but denaturation of RNA hairpins by glyoxal has gained popularity, as this method does not involve the use of large quantities of harmful chemicals (42, 43, 46). Both methods, followed by staining of the gels with nucleic acid binding stains, such as ethidium bromide or Vistra Green™, are useful for observing overall differences in RNA quality. It is relatively easy to see if a sample is badly degraded as the ribosomal bands appear as smears. However, it may be difficult to detect more subtle degradation of transcripts unless large amounts of sample are used. Figure 31 shows a schematic illustration of typical results from RNA gel electrophoresis. Transferring the gel onto a membrane for Northern blotting analysis can give more precise information, as the status of specific transcripts in different samples can be analyzed.

Reverse transcriptase PCR (RT-PCR) offers a convenient, fast, and versatile method for obtaining information regarding the quality of RNA preparations from small sample amounts. First-strand cDNA synthesized with oligo(dT) priming can be used as PCR template.

By choosing specific primer pairs, it is possible to determine whether different transcripts are intact in the samples. Performing PCR amplification for a limited number of cycles, or with real time detection, makes it possible to estimate the relative amount of specific transcripts in different samples. By amplifying transcripts derived from genes whose expression is not expected to vary under experimental conditions, it is also possible to compare the amount of mRNA in the different samples. By choosing several pairs of primers from one preferably long transcript, and targeting its 5' central and 3' regions, it may be possible to observe partial degradation of samples. Performing PCR with primers that are derived from gene introns can reveal the presence of genomic DNA.

5.5 General recommendations for preparing RNA samples for microarray analysis

In conclusion, the following general recommendations for preparing RNA for microarray analysis are given:

- Minimize the degradation of RNA at all handling stages.
- Choose an RNA purification method that gives good yields of pure and intact RNA from your samples, even if this means using a complicated protocol.
- Measure the amount of RNA before using it for microarray labelling.
- Verify the quality of the RNA before using it for microarray labelling.
- If possible, purify mRNA for use in microarray analysis.
- Prepare all the samples for microarray analysis with the same protocol.

Chapter 6

SAMPLE LABELLING FOR GENE EXPRESSION ANALYSIS

6.0 Introduction

In differential gene expression analysis two or more RNA samples are compared to identify differences in the abundance and identity of the transcripts they contain. In order to convert the information contained in the transcript populations into a form that can be hybridized with microarrays and subsequently detected, the transcript populations need to be labelled. This can be achieved using different methods; an ideal method retains both the information carried by the identity of the transcripts as well as their relative abundance in the sample.

6.1 The diversity of transcript populations

Messenger RNA molecules, otherwise called transcripts, carry the genetic information encoded in genes. In most cells these transcripts constitute only a small proportion of the total RNA, whereas ribosomal and transfer RNA account for more than 98%. In any cell type, the transcript population typically consists of thousands of distinct transcripts, most of which are transcribed from different genes (although splice variants of genes exist too). These transcripts can be present in widely varying amounts ranging from just a few copies per cell to thousands of copies. Furthermore, the relative levels of transcripts are constantly changing as the cell responds to different environmental signals. The amount of transcripts is estimated to follow a normal distribution in which a small number of genes are expressed at high or very low levels. The majority of the genes are expressed at intermediate levels.

Genes come in different sizes, with different numbers and sizes of exons. The size of transcripts reflects this by varying from a few hundred nucleotides to up to about 20 000 nucleotides. Average length of transcripts is estimated to be between 1.5–1.7 kb.

6.2 Requirements of labelling methods

6.2.1 Retaining gene expression information

The labelling methods used in microarray analysis must cope with the inherent diversity of transcript sequences and create representations that contain all the information present in the original transcript population. Thus an ideal labelling system is neither biased towards any nucleotide sequences, nor does it label differently transcripts of different sizes or sequences that are expressed at different levels.

In reality, existing labelling methods do not convert all information into labelled form. Enzymatic methods are limited to copying certain nucleic acid sequences, whereas the instability of some transcripts is a general problem for all methods.

6.2.2 Length of labelled fragments

Accurate information about gene expression can only be deduced from microarray experiments if the labelled nucleic acids can hybridize efficiently and with specificity to their complementary targets. The length of the labelled fragment is an important factor in determining these parameters. Fragments longer than 100 nucleotides can hybridize strongly enough with their target sequences to withstand stringent hybridization and wash conditions. The hybridization kinetics of shorter fragments is faster. For optimal hybridization, probes consisting of fragments of 200–500 nucleotides long are recommended. Longer fragments may not find their targets as efficiently as shorter fragments, but will produce a higher signal when hybridized as they carry more labelled molecules.

6.2.3 Yield of labelled probe

The amount of labelled probe prepared by the labelling method is important to the sensitivity of microarray experiments. This is because the efficiency of the labelling process is critical in determining the lowest amount of mRNA that can be used to generate detectable signals from microarrays. With higher amounts of mRNA available, differences in probe yield from different labelling methods can make the difference between being able to hybridize one or several slides with one probe. Ideally, the labelling method should transform each transcript into a labelled fragment, without any bias towards more highly expressed sequences. If the labelling method results in net amplification of nucleic acid in the labelling process, the amplification process should be linear, i.e. the original ratios of expression levels within the sample should not be changed in the amplification process.

6.2.4 Optimum labelling density

The fluorescent labels used in microarray analysis bring their own restrictions to labelling protocols. If two or more fluorescent molecules are in close proximity of each other, a significant portion of the absorbed light energy can be spent on interactions between different molecules and dissipated as heat. This will result in less than the expected amount of fluorescence being emitted from the sample, and moreover, the amount of fluorescence is no longer directly proportional to the number of fluors in the sample. This phenomenon is called "quenching", and it is an inherent property of fluorescent molecules. Each fluorophore has slightly different quenching properties that are determined by its chemical structure. In practical terms this means that for each fluor there is an optimal labelling density, or distance between attached labels, which will produce maximum fluorescence from a labelled nucleic acid fragment. Exceeding this optimum labelling density results in decreased fluorescent signal (Fig 32). Therefore, in order to achieve highest sensitivity of detection, the labelling method used in microarray experiments should be optimized to yield fragments that are labelled at the maximum density as determined by the labelling fluors. See chapter 4, section 4.2.2 for additional information.

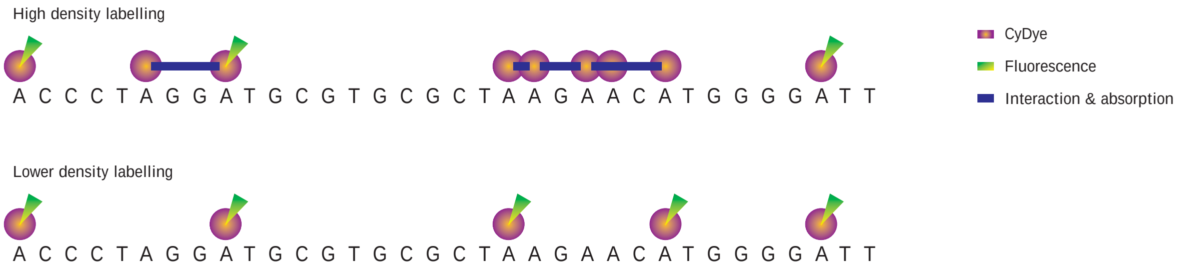

High density labelling
A C C C T A G G A T G C G T G C G C T A A G A A C A T G G G G A T T
Lower density labelling
A C C C T A G G A T G C G T G C G C T A A G A A C A T G G G G A T T
CyDye
Fluorescence
Interaction & absorption

Fig 32. Quenching and labelling density. Fluorescent output from two identical nucleic acid strands labelled to different labelling densities is depicted. Green denotes fluorescent emission whereas purple shows energy being lost in intermolecular interactions between adjacent fluorophores. Lower labelling density results in higher fluorescent signal.

6.2.5 Equal labelling with different fluors

The purpose of differential gene expression analysis is to detect relative differences in the number of specific transcripts between two or more samples. This requires that the two samples hybridize competitively with the immobilized targets and differences in relative signals primarily reflect changes in the number of the transcripts. From a technical point of view, this is best achieved when equal numbers of equally labelled nucleic acid fragments are compared. As two different fluors are used in two-color analysis, no imbalance due to the properties of the fluors should be present in the labelled populations. In extreme cases such imbalances can lead to false positive signals from gene expression microarrays. The labelling method should produce the same labelling density and size distribution of labelled fragments, regardless of the fluorescent dye label used.

6.2.6 Nucleotide sequence preferences

As all labelling methods attach the fluorescent label in a specific manner, usually via certain nucleotides, the nucleotide sequence of the molecules being labelled can have a significant effect on labelling density and also on the length of labelled fragments (Fig 33). For example, incorporation of CyDye dUTP instead of CyDye dCTP into cDNA that is C-rich is less likely to result in quenching because the likelihood of incorporating two CyDye dCTP into close proximity will be lower. Conversely, highly A-rich sequences are best labelled using CyDye dCTP as label.

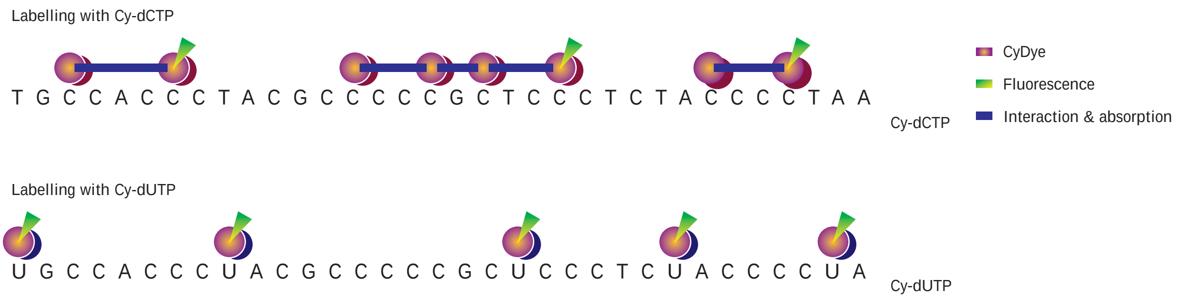

Labelling with Cy-dCTP
T G C C A C C C T A C G C C C C C G C T C C C T C T A C C C C T A A
Cy-dCTP
CyDye
Fluorescence
Interaction & absorption
Labelling with Cy-dUTP
U G C C A C C C U A C G C C C C C G C U C C C T C U A C C C C U A
Cy-dUTP

Fig 33. The effect of sequence on labelling density and fluorescent signal. The same nucleotide fragment has been labelled with Cy-dCTP and Cy-dUTP, using a similar ratio of label to cold nucleotide. As the fragment is GC-rich, the use of Cy-dCTP results in much higher labelling density on this fragment than the use of Cy-dUTP, which can only be incorporated in a few positions. However, the fluorescent signal from the Cy-dUTP labelled fragment is greater than from the more densely labelled fragment, because quenching becomes a significant factor in the situation of high labelling density.

As probes used in microarray analysis are complex mixtures of nucleic acid sequences, careful optimization of labelling methods is needed to ensure that quenching is not a practical problem for the majority of transcripts. However, it may not be possible to create labelling conditions that would be perfect for all sequences. Performing a ‘yellow experiment’, in which the same sample is labelled with both fluors and then used in microarray hybridization, can help to identify the targets that do not give balanced signals, as well as to optimize the microarray system in general. By choosing target sequences that do not show high nucleotide sequence bias, it is also possible to control this aspect of labelling.

Increasing labelling density is not the best solution for increasing signal from microarrays. Using a lower labelling density is more likely to result in higher signal. As fluorescent molecules tend to be fairly large, high labelling densities can lead to changes in the hybridization kinetics of labelled molecules (47). The melting temperature (Tm) of highly substituted nucleic acids are lower than those of unlabelled nucleic acids and can result in lower hybridization signal.

6.3 Labelling strategies

6.3.1 mRNA vs total RNA

Only a small proportion, about 1.5–2.5%, of cellular RNA is mRNA. Prior to hybridization, mRNA must be purified and labelled. Since most of the cellular RNA is ribosomal RNA, specific protocols are used to separate mRNA from ribosomal RNA.

Eukaryotic transcripts usually have a poly-dA tail at their 3' ends. This property can be exploited by using a complementary poly-dT sequence to capture polyadenylated transcripts away from other RNA species, as well as from other molecules and impurities. Because of the additional purification steps involved in the preparation of mRNA, mRNA samples tend to give higher signal to noise values on microarrays than total RNA samples.

6.3.2 Priming with oligo(dT)

In the labelling reaction, mRNA can be selected from total RNA for use as the labelling template by using oligo(dT) primers that will hybridize with the poly-A tail in transcripts. Addition of an anchoring base to the 3' end of these primers directs cDNA synthesis to the beginning of the poly-A stretch. This has the advantage of producing labelled fragments that are devoid of most of the repetitive sequence. This priming method will result in only one copy of cDNA that contains primarily 3' sequences synthesized from each transcript. If the targets on the microarray are derived from 5' ends of long cDNAs, probes labelled directly in cDNA synthesis using oligo(dT) priming may not produce complementary fragments to these targets, resulting in absence of signal in hybridization.

6.3.3 Random priming

Random priming, in which a mixture of oligonucleotides comprised of all sequence variants of a short sequence of defined length are used as primers, can be used to produce probes that contain sequences derived from all parts of transcripts. Typically, each transcript is copied into several non-overlapping probe fragments. Because of their longer length and ability to form more stable duplexes, nonamers are preferred over hexamers and give higher yields of cDNA. Random priming is only compatible with mRNA templates, as random primers can anneal to all RNA molecules. cDNA synthesis from total RNA with random priming will produce a large quantity of short fragments that lack specificity in hybridization and usually give rise to high background signals. As the proportion of label incorporated into cDNA derived from mRNA is going to be very small under these conditions, the specific signals from microarray spots will be low. Conversely, as most of the fluorescent label is incorporated into sequences derived from ribosomal RNA, unspecific hybridization can become a problem.

6.3.4 Other priming strategies

Highest yield of cDNA from mRNA (without probe amplification) is achieved with the use of both oligo(dT) and random primers together (Fig 34). This strategy has the highest likelihood of copying all parts of transcripts into probe, and therefore, is suitable for use with target sequences that are derived from varying parts of genes.

It is also possible to use specific primers to copy transcripts into probe. As each sequence requires the synthesis of a specific primer, this approach can be costly and require a new set of primers to be prepared for each different microarray. The advantage of this approach is that only those sequences that are analyzed on the microarray are labelled. It is also possible to use total RNA as a sample.

6.3.5 Amount of primer

Regardless of the type of primer used, its concentration should be in excess of the number of possible binding sites on transcripts so that its availability is not limiting cDNA synthesis. cDNA synthesis with general priming strategies (oligo[dT] and random nonamers) may be biased towards highly expressed transcripts if too much mRNA is used. The specific priming strategies may be biased against high-expressing transcripts if the amount of each primer is not sufficient to cover the whole expression range.

6.3.6 Labelling bacterial RNA

Bacterial mRNA lacks poly-A tails and selecting transcripts for labelling from total RNA is not as easy as with eukaryotic RNA. It is possible to remove ribosomal RNA sequences by converting them into cDNA: RNA hybrids, followed by digestion with RNAse H and DNAse I to selectively remove the double-stranded sequences. Random priming strategies can be used successfully with bacterial RNA, if the stringency of hybridization is controlled carefully to counteract the high proportion of label associated with ribosomal RNA sequences. Alternatively, priming strategies that utilize gene specific primers or short primers that are able to prime from several genes have been used (48, 49). As gene specific primers prime cDNA synthesis only from those genes that are being studied on the array, they can help to increase specificity of hybridization and signal to noise values.

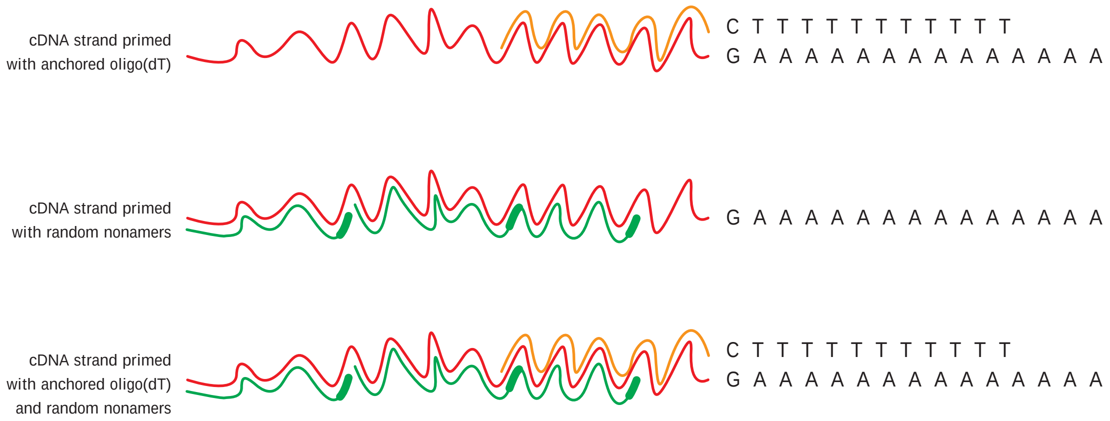

cDNA strand primed with anchored oligo(dT)
C T T T T T T T T T T T
G A A A A A A A A A A A A A A
cDNA strand primed with random nonamers
G A A A A A A A A A A A A A A
cDNA strand primed with anchored oligo(dT) and random nonamers
C T T T T T T T T T T T
G A A A A A A A A A A A A A A

Fig 34. The use of different primers in cDNA synthesis. cDNA strand synthesized with anchored oligo(dT) priming is shown in yellow and those primed with nonamers in green. The combined use of both primers results in the probe being prepared from all parts of the transcript.

6.4 Enzymatic labelling methods

6.4.1 Labelling strategies

Several strategies based on molecular biology or chemical reactions have been developed for labelling samples for gene expression microarray analysis. The availability of fluorescent labels in different reactive forms has contributed to the diversity of labelling methods. All these strategies have in common that they start with an RNA population (Fig 35). Molecular biology strategies rely on the use of enzymes to convert mRNA into new populations of nucleic acids, either DNA or RNA. Combining two or more enzymatic reactions into one protocol widens the choice further. Using more than one enzyme for labelling, however, has the disadvantage that the information carried by the original population is likely to change more than by using a single enzyme. This is because some information is lost in each enzymatic conversion step, and as the lost information is dependent on the sequence of the transcripts and the properties of the enzyme, the representations synthesized by each enzyme will be different. Chemical methods have the advantage that no copying of nucleic acid to another form takes place; instead the labelling moiety reacts with the nucleic acids to form covalently modified, labelled probe population.

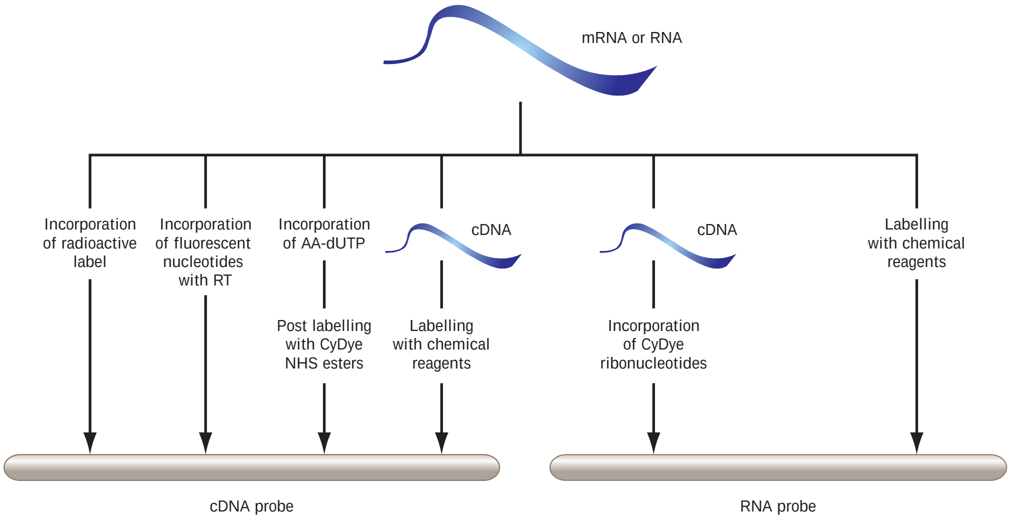

mRNA or RNA
Incorporation of radioactive label
Incorporation of fluorescent nucleotides with RT
Incorporation of AA-dUTP
cDNA
Post labelling with CyDye NHS esters
Labelling with chemical reagents
cDNA
Incorporation of CyDye ribonucleotides
Labelling with chemical reagents
cDNA probe
RNA probe

Fig 35. Labelling strategies for gene expression microarrays.

6.4.2 Fluorescent labels

Fluorescent dyes, especially the cyanine dyes Cy3 and Cy5, are the most popular choice for dual color microarray analysis. The main benefit of using CyDye fluors in particular is that two dyes can be excited and detected from the same slide. CyDye fluors also produce bright signals and have a wide dynamic range of detection, so both weak and strong signals can be detected in the same experiment. Fluorescent dyes can be directly incorporated into nucleic acid by either enzymatic or chemical methods.

6.5 Labelling in first-strand synthesis

6.5.1 Principle

One of the simplest and most popular labelling strategies is to convert mRNA population into a labelled first-strand cDNA population. This is achieved by copying the transcripts into cDNA molecules with a reverse transcriptase while incorporating a modified CyDye nucleotide. The cDNA synthesis can be primed with a choice of primers including random primers, anchored oligo(dT) as well as gene specific primers. This allows the use of both mRNA and total RNA as sample.

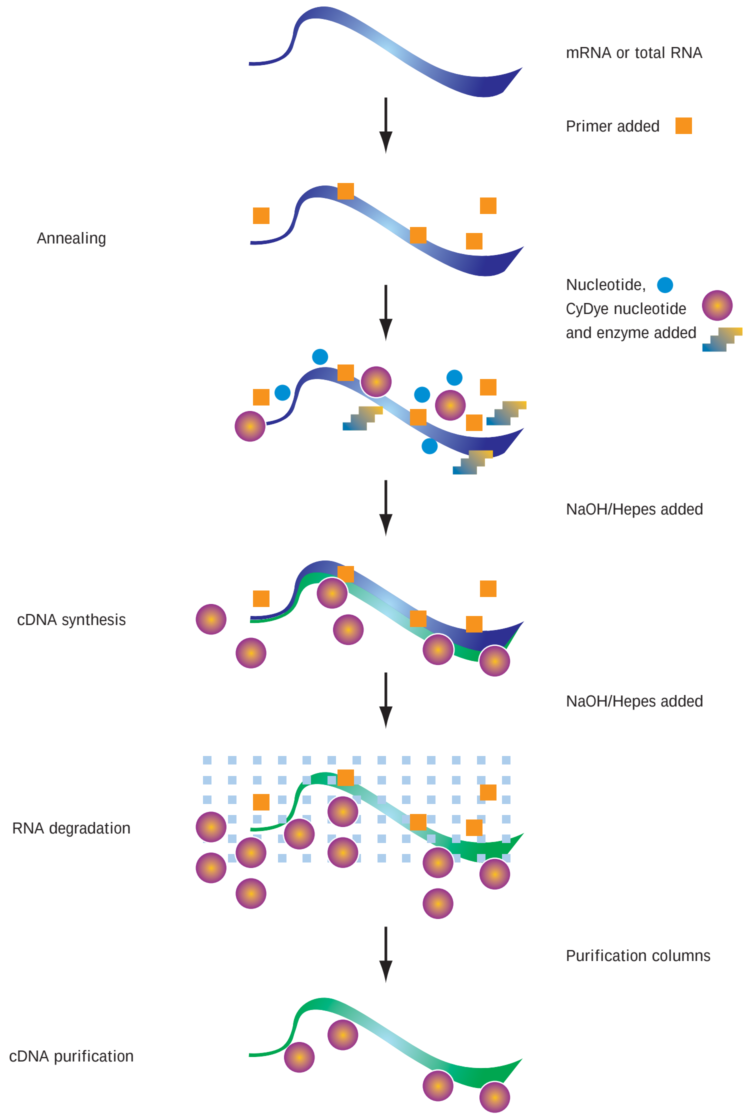


Fig 36. Principle of labelling in first-strand cDNA synthesis. Fluorescent nucleotide (pink circles) is incorporated into first-strand cDNA by a reverse transcriptase. After degradation of the mRNA template strand, labelled single-stranded cDNA probe can be purified.

6.5.2 Optimization

The incorporation of fluorescently labelled nucleotides is the rate-limiting step of this labelling method. This is because all polymerase enzymes, both DNA and RNA dependent polymerases, incorporate unlabelled nucleotides more efficiently than larger fluorescent dye nucleotides. The incorporation kinetics are very much dependent on the identity of the enzyme, and even homologous enzymes can have very different properties. The ratio between the fluorescently labelled nucleotide and the corresponding unlabelled nucleotide in the labelling reaction determines the incorporation of fluors into cDNA. As different fluorescent dyes and nucleotides have different structures, it is necessary to optimize this ratio separately for each combination of dye and nucleotide. Furthermore, optimized ratios determined for one enzyme will not necessarily give optimal results with other enzymes.

Labelling in first-strand synthesis does not produce long cDNA molecules. This is because elongation of the nucleotide chain is dependent on the previous nucleotides having been incorporated. The incorporation of more than one fluorescent label consecutively into cDNA is not favored by polymerases. If the ratio of fluorescent nucleotide to unlabelled nucleotide is high, a highly labelled cDNA can be transcribed, but cDNA synthesis will stall if the mRNA sequence requires several labelled nucleotides to be incorporated in a row. Only short fragments will be made, resulting in low yield of cDNA. Lowering the nucleotide ratio can increase the yield of cDNA, but this will compromise labelling density and the brightness of the probe. A balance between these two factors and the consequences of quenching at high labelling densities must be attained for optimal results. In practice this requires evaluation of fluorescent signals on microarrays from probes labelled at different nucleotide ratios, and this demands considerable effort.

6.6 cDNA Post Labelling

6.6.1 Principle

The shortcomings of the first-strand cDNA labelling method and the availability of CyDye as reactive N-hydroxyl succinimidal dyes (NHS-dyes) have led to the development of cDNA post-labelling method.

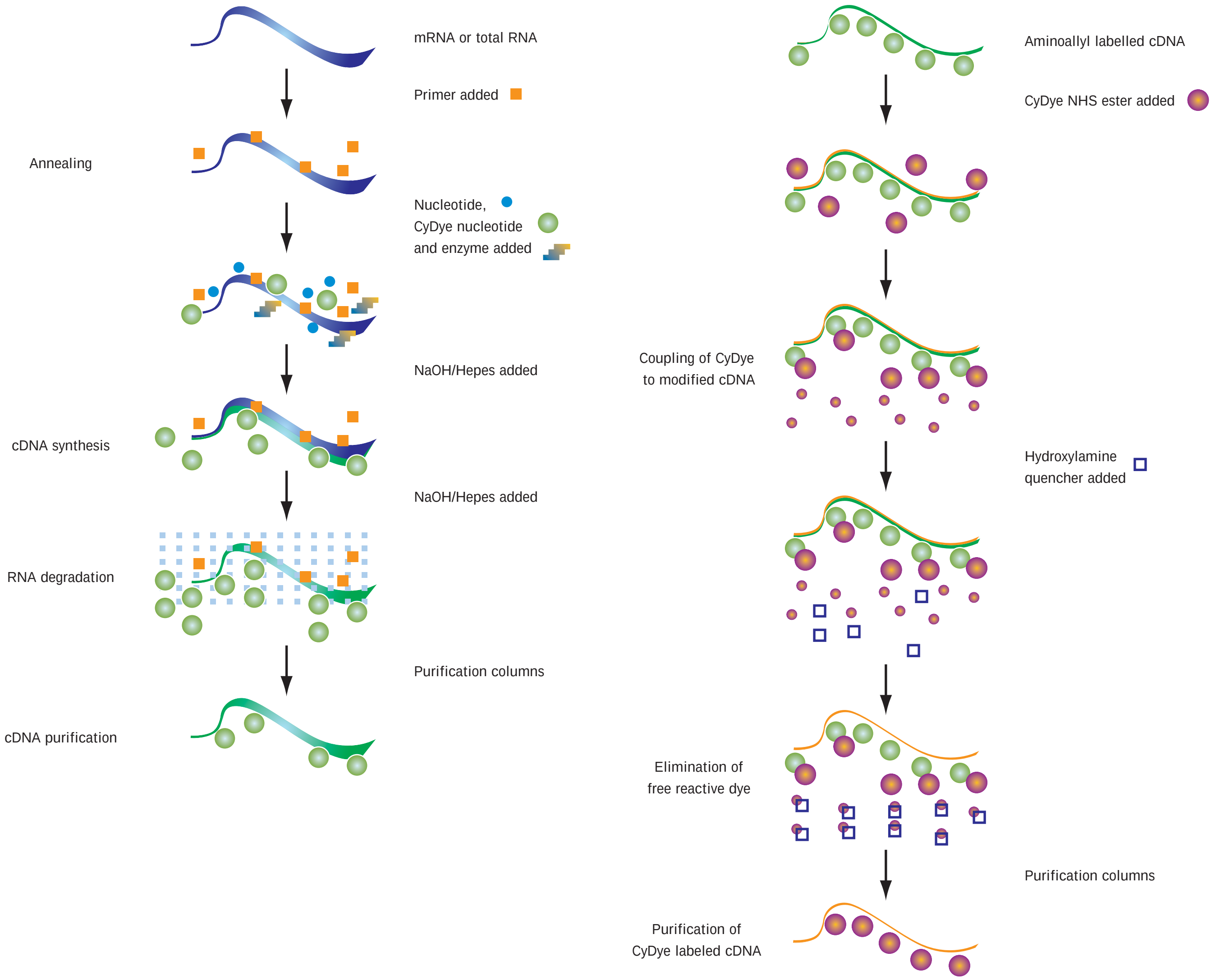


Fig 37. The principle of post-labelling is illustrated. mRNA is converted into first-strand cDNA that contains aminoallyl-dUTP. After elimination of mRNA template, the amine groups on cDNA are reacted with CyDye-NHS ester, resulting in the generation of fluorescently labelled cDNA. Excess of NHS-ester can be neutralized with hydroxylamine, and labelled cDNA is purified for use.

Amersham Biosciences has developed CyScribe™ Post-Labelling Kit (see section 6.14) for this method. In this method, amine-modified cDNA is first synthesized by incorporating aminoallyl-modified nucleotide in first-strand cDNA by a reverse transcriptase. After removal of RNA template and purification of the amine-modified cDNA, chemical labelling with N-hydroxyl succinimidyl-ester derivative of CyDye is performed. NHS-esters react with amine groups to form a covalent bond between CyDye and the amine group while the NHS-ester group is released (Fig 38). A high excess of CyDye NHS-ester is required for efficient reaction, and any non-reacted label is neutralized with an excess of small amine such as hydroxylamine. The cDNA needs to be re-purified after labelling to remove CyDye that is not incorporated into labelled cDNA.

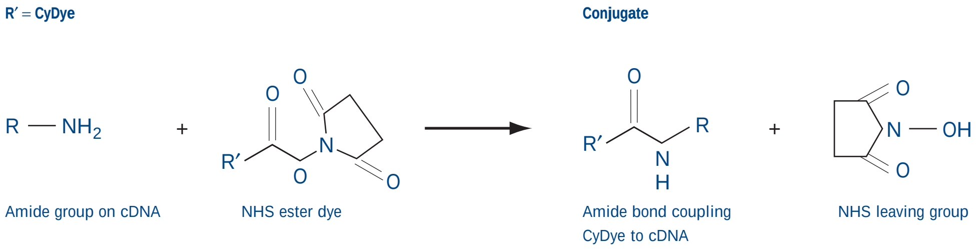


Fig 38. The use of CyDye NHS-ester in labelling amine-modified cDNA

6.6.2 Benefits

The main benefits of this method over the first-strand cDNA labelling method are derived from the more efficient incorporation of the smaller aminoallyl nucleotide compared with the bulkier CyDye nucleotides. As a consequence, the yield of cDNA is considerably higher than with the first-strand labelling method. The cDNA fragments synthesized in the post-labelling method are also longer. Further benefits of the post-labelling method stem from the chemical labelling step itself. The number of amine groups on cDNA is the main factor influencing labelling density. The sequences of the cDNAs being labelled do not have a major impact on labelling outcome. Because of this, more random attachment of labels is achieved than with the first-strand cDNA labelling method. Furthermore, as the labelling process is not dependent on the structure of different fluorescent dyes, it is easier to achieve equal labelling with both Cy3 and Cy5, and the labelling method introduces less variation into microarray analysis. This is illustrated in tighter scatter plots derived from microarray hybridizations in which the mRNA samples were labelled with the post-labelling method (Fig 39). As the extent of experimental variation is reduced, the detection of smaller changes in gene expression between two samples is improved.

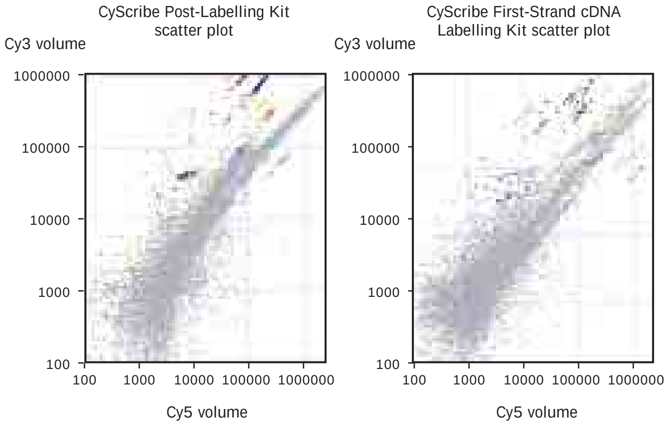


Fig 39. Scatterplots of gene expression microarray analyses comparing skeletal muscle and placental mRNA samples. Identical mRNA samples were labelled with the post-labelling method or with first-strand cDNA labelling. Identical slides were hybridized with equal amounts of all probes.

6.6.3 Chemical considerations

Coupling of CyDye NHS-ester to amine-modified cDNA requires mildly alkaline pH, which is provided by the coupling buffer. However, if they are not carefully removed, buffer components or acidic residues from preceding steps can alter the pH of this buffer. Furthermore, any amine groups present on other compounds will compete with the amine-modified cDNA for CyDye incorporation. Hence the free aminoallyl-dUTP, Tris-buffer, and reverse transcriptase enzyme must be removed from the cDNA preparation before labelling. This can be best achieved with affinity column chromatography methods, such as GFX columns, in which amine-modified cDNA is bound to the matrix, other compounds are washed away, and cDNA is finally eluted with water. Alternatively, standard ethanol precipitation also works well and provides the added benefit of concentrating the cDNA ready for CyDye coupling.

NHS-esters are readily hydrolyzed with water, even with the small amount of moisture present in laboratory air. Because of this, aliquots of CyDye NHS-esters must always be stored desiccated and protected from light. Storing CyDye NHS-esters in solution can lead to rapid loss of reactivity. As these reactive dyes were originally developed for protein labelling, the quantities provided commercially were adjusted for this application. This necessitated aliquoting of reactive dyes before their use for microarray sample labelling and frequently resulted in decreased activity as a consequence of handling and storage. The availability of individually foil-packed, pre-dispensed, and freeze-dried aliquots of Cy3 and Cy5 NHS-esters for microarray analysis has removed this problem. Furthermore, the quality of the CyDye NHS-ester in CyDye Post-Labelling Reactive Dye Packs is higher than available otherwise.

6.7 RNA amplification and labelling in RNA synthesis

6.7.1 RNA amplification

The amount of RNA sample can be a limiting factor for microarray analysis, and it may be necessary to amplify RNA before analysis. In the most commonly used protocol, the mRNA population is first converted into a double-stranded cDNA that contains a promoter sequence for viral RNA polymerase, such as T7, T3, or SP6 polymerase (Fig 40).
This can be achieved by using a modified oligo(dT) primer containing a 5' extension coding for the viral promoter. Each resulting cDNA molecule will contain one RNA polymerase promoter sequence. By including the corresponding RNA polymerase and ribonucleotides in the reaction, several RNA copies can be synthesized from each template.

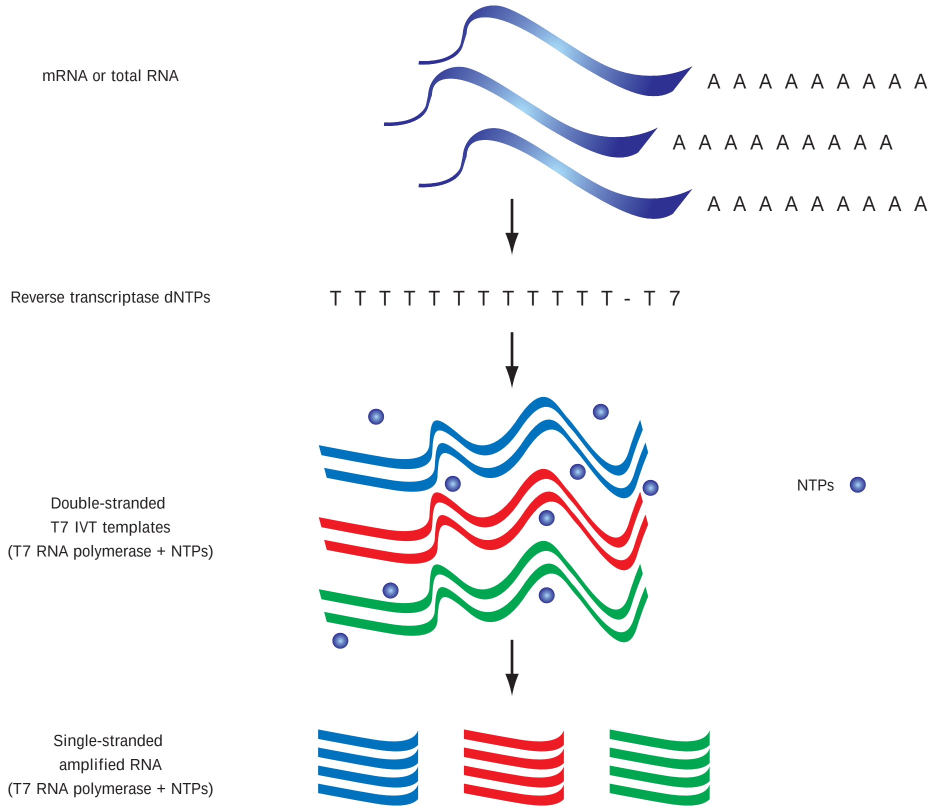

mRNA or total RNA
A A A A A A A A A
A A A A A A A A A
A A A A A A A A A
Reverse transcriptase dNTPs
T T T T T T T T T T T T - T 7
Double-stranded T7 IVT templates (T7 RNA polymerase + NTPs)
NTPs ●
Single-stranded amplified RNA (T7 RNA polymerase + NTPs)

Fig 40. Principle of RNA amplification with RNA polymerases.

This results in a linear amplification of the cDNA pool into RNA without altering the relative abundance of sequences in the mixture significantly. However, the length of synthesized fragments will be shorter than the original templates. 2000-fold amplification of starting RNA can be achieved with this method, in one round of amplification (50, 51). This RNA amplification strategy can be used on its own, and the amplified RNA population can be labelled separately using other methods. Alternatively, labelling and amplification can be performed together by including a CyDye ribonucleotide in the reaction (Fig 41). In some cases when only a few cells are available for the preparation of RNA sample, such as when laser capture micro (LCM) dissected samples are analyzed, several rounds of RNA amplification can be performed to acquire enough RNA for labelling (52).

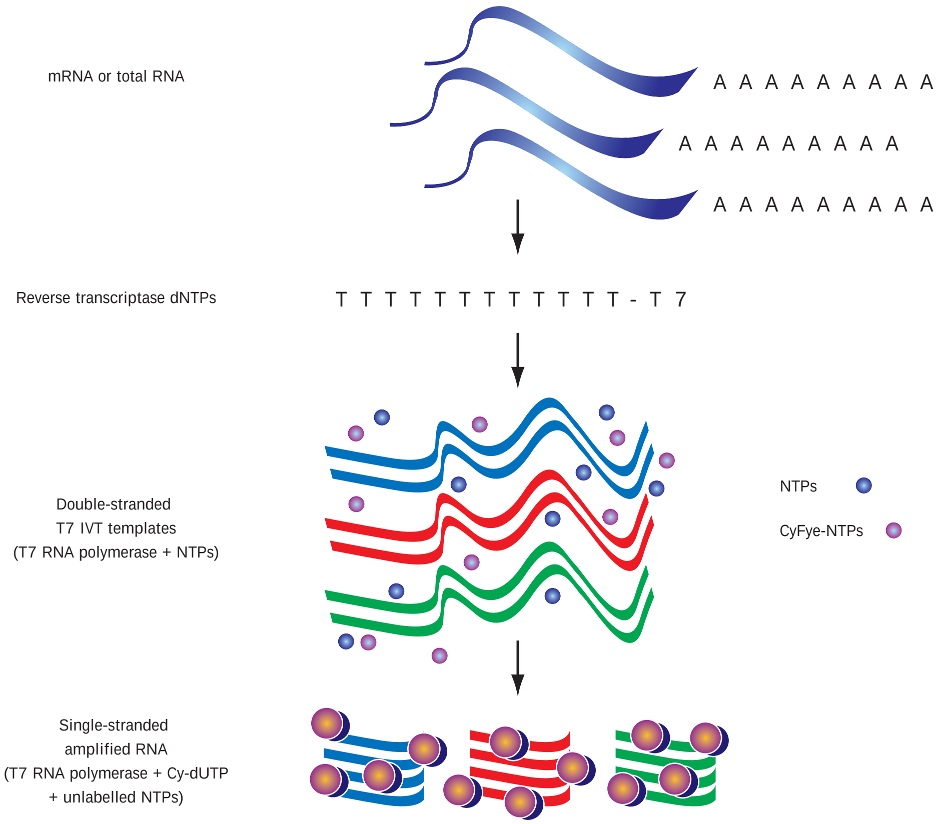


Fig 41. Principle of preparation of fluorescently labelled RNA probe with RNA polymerase amplification.

6.7.2 Other amplification methods

Amplification of RNA can also be achieved by other means, including using limited numbers of PCR cycles to amplify double-stranded cDNA (53). However, PCR-based methods have not gained wide popularity in microarray analysis, because of the problems associated in logarithmic amplification of complex nucleic acid mixtures. Sequence and size differences in nucleic acid fragments can influence their amplification rate and result in selection of sub-populations of sequences (50). Regardless of the RNA amplification strategy, the method should neither alter the relative abundance of different transcripts in the sample nor result in the creation of sequence chimeras in which sequences from two or more transcripts are joined together. Linear amplification strategies avoid these pitfalls and are favored in microarray analysis.

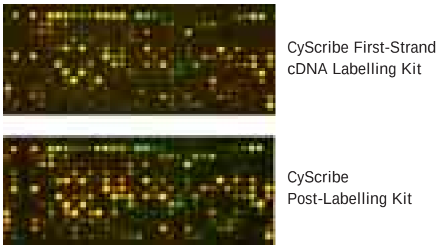
6.7.3 Labelling with RNA amplification

Synthetic RNA can be labelled in synthesis reactions by incorporating a CyDye ribonucleotide. The main factor determining both labelling density and yield of labelled RNA in this method is the ratio of CyDye ribonucleotide to the corresponding unlabelled ribonucleotide. As is the case for DNA polymerases, RNA polymerases incorporate the unlabelled nucleotide more efficiently, and a compromise between labelling density and yield of RNA is necessary. This requires careful optimization of the labelling conditions, combined with the analysis of fluorescent signal derived from the probes.

RNA labelling can also be performed by using hapten-labelled ribonucleotide, such as biotinylated ribonucleotides. This strategy has the advantage of giving higher yields of cDNA, as the incorporation of large dye nucleotide is not a rate-limiting factor. However, the need to perform additional detection steps adds to the length of the protocol and makes the use of more than one color at a time more difficult.

6.7.4 Fragmentation of RNA probes

The secondary structure of RNA can interfere with hybridization to targets. Fragmentation of RNA probe into smaller fragments of 50–200 nucleotides can be performed to overcome this. This can be achieved in controlled fashion by exposure of RNA to potassium and magnesium ions (24). Careful handling of RNA is necessary at all stages to minimize uncontrolled degradation of RNA to nucleotides and short fragments. If RNA probes are used in hybridization, precautions must be taken throughout the whole microarray procedure.



CyScribe First-Strand cDNA Labelling Kit

CyScribe Post-Labelling Kit

Fig 42. Expression patterns obtained from gene expression microarrays comparing skeletal muscle and placental mRNA samples. Identical mRNA samples were labelled with the post-labelling method and first-strand cDNA labelling methods using CyScribe Labelling Kits. Equal amounts of all probes were hybridized with identical microarrays. Although the overall hybridization patterns are very similar, some significant differences in signal intensities of individual spots can be seen. These reflect the size differences of labelled fragments obtained with the two labelling methods. Despite the different appearance of the two microarrays, both gave similar information on differential gene expression.

6.8 Expression patterns

The intensity of hybridization signals from microarrays is determined not only by the number of hybridized probe fragments but also by their length and the number of fluorescent labels that are associated with them, or the labelling density. Similar labelling densities can be achieved with different labelling methods. However, with different methods, the average length of the labelled fragments varies. In practice, the intensity of signals derived from two identical samples labelled with different methods will be different. Hence, gene expression microarrays do not give quantitative information about gene expression. However, if two samples are labelled by the same method using two different fluorescent labels, the relative intensity of the signals will reflect the relative abundance of the specific transcripts in those two samples. Information about differential gene expression can therefore be gained (Fig 42). Labelling the two samples to be compared under identical conditions is therefore extremely important for guaranteeing that experimental artifacts do no lead to wrong conclusions from microarray results.

6.9 Random prime labelling of DNA

A modified random prime labelling method can be used to label DNA with CyDye. In this method, Klenow polymerase incorporates fluorescently labelled nucleotides in a DNA synthesis reaction, which is primed with random nonamer or hexamer primers. This method is a practical solution for genomic microarrays, although direct chemical labelling methods can also be used. Random prime labelling methods are not recommended because two different enzymes are needed to convert mRNA into labelled form.

6.10 Direct chemical labelling of mRNA

When an enzyme is used to convert a mRNA population into another nucleic acid population, some information is lost because the ability of different enzymes to copy through different nucleic acid sequences varies. By using a chemical labelling method, it is possible to label mRNA directly by using a chemical reaction, coupling modified CyDye reagent to RNA molecules. These methods are simple to perform, as no modification of RNA is required before labelling. Furthermore, these methods are less prone to discrimination against certain nucleotide sequences that are difficult templates for polymerase-based labelling systems. It is important to note that any chemical that avidly reacts with nucleic acid molecules is potentially toxic and will require careful handling and adequate safety measures to be taken.

6.11 Purification of labelled probes

Regardless of the labelling strategy, it is necessary to purify the labelled nucleic acid after labelling, as the amount of incorporated fluorescent dye is typically only a small fraction of all the dye present in the sample. The recovery of labelled nucleic acid from purification is a major limiting factor for most labelling methods. Products such as AutoSeq™ G-50, GFX, and QIAquick™ spin columns have been developed for the purification of double-stranded cDNA. They can also be used to remove unincorporated CyDye nucleotides away from fluorescently labelled single-stranded cDNA, but their use does not result in optimal recovery of labelled material. Typically, less than 40% of the labelled probe is recovered, and the recovery can vary considerably between different samples. This variation not only reduces the amount of data that can be generated with microarray analysis, but also significantly contributes to poor quality of results if the amount of probes are not adjusted before hybridization. When small amounts of template are labelled, the loss of labelled cDNA tends to be higher, and as little as 5% of probe may be recovered.

All of these methods, however, are relatively successful in removing the free dye nucleotide. Ethanol precipitation is not an option for use with most labelling methods, as it can result in the formation of dye aggregates that will produce intense speckled background in array hybridization.

Appreciation of these problems has lead to the development of a novel GFX purification system, CyScribe GFX Purification Kit, which has been specifically tailored for purification of single-stranded CyDye labelled cDNA. As this purification system is based on binding of the labelled nucleic acid to a customized GFX matrix, it is also suitable for use with the post-labelling method. These CyScribe GFX columns give excellent yield of purified cDNA from different synthesis scales, including small-scale samples. Therefore, the use of this purification system downstream of optimized labelling kits can improve the sensitivity of microarray analysis, enable the use of smaller amounts of RNA samples, or increase the number of replica slides that can be hybridized with one sample.

6.12 The CyScribe family of labelling kits

The CyScribe family of labelling kits from Amersham Biosciences has been developed to offer a range of optimized labelling products for producing CyDye labelled microarray probes. These kits enable flexible choice of different labelling methods to suit the different needs of researchers.

6.13 CyScribe First-Strand cDNA Labelling Kit

6.13.1 Features of the kit

CyScribe First-Strand cDNA Labelling Kit has been developed for preparing highly fluorescent CyDye labelled probes for microarray analysis using first-strand cDNA labelling. The kit has been optimized for the use of either CyDye dCTP or CyDye dUTP nucleotides as labels. Two nucleotide mixes are provided in the kit to provide optimal nucleotide ratios for each type of nucleotide when 1 nmol of CyDye nucleotide is used per reaction. The compositions of these solutions have been optimized so that labelled cDNA will contain an attached CyDye fluor to every 12–25 nucleotides synthesized, regardless of the nucleotide used.

CyScribe First-Strand cDNA Labelling Kit contains both anchored oligo(dT) primers and random nonamer primers, allowing flexible choice of templates. As little as 50 ng of mRNA and 2.5 µg of total RNA can be labelled per reaction with the kit and used successfully in microarray hybridization on one slide (Fig 43). Recommended highest amounts of

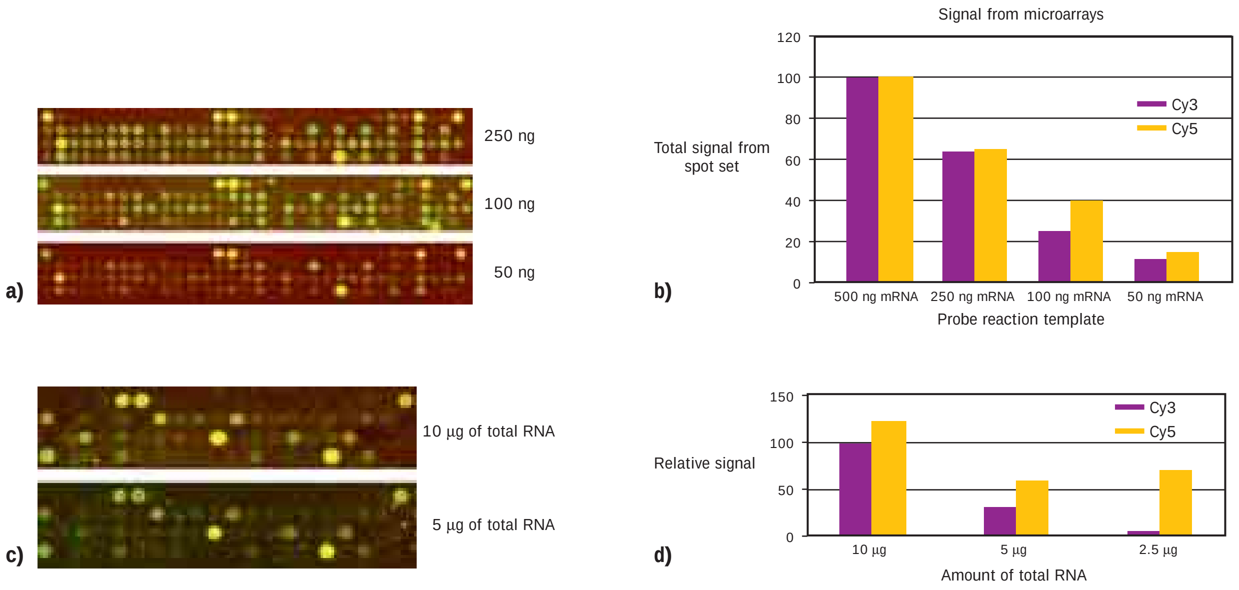


Fig 43. The use of CyScribe First-Strand cDNA Labelling Kit with small template amounts. The indicated amounts of mRNA and total RNA templates were labelled with Cy3 and Cy5 in duplicate reactions, purified, and all of the recovered probe was used in a microarray hybridization with duplicate slides. Panels A and C show part of microarray images obtained with mRNA probes and total RNA probes, respectively. Panels B and D show quantified Cy3 and Cy5 signals from these arrays. As the amounts of probes were not adjusted before hybridization, a high amount of variation in signal intensities was observed in this experiment.

template are 500 ng of mRNA and 25 µg of total RNA per reaction. These amounts of template will generate enough probe for several microarray hybridizations. Highest yield of labelled probe is obtained with dual priming, when both oligo(dT) and random primers are used together. This priming strategy is only compatible with the use of mRNA templates. The choice or amount of RNA template does not affect the labelling density achieved with the kit.

6.13.2 Degradation of RNA template

Degradation of the RNA template after cDNA synthesis is necessary to prevent the labelled probe from hybridizing with the original template in solution instead of the microarray targets during microarray hybridization. The removal of RNA can be performed enzymatically, by using RNAse H enzyme to digest the RNA component of RNA DNA heterohybrid molecules. A simpler option is to degrade the RNA strands by raising the pH of the probe solution. The CyScribe Kits contain an efficient RNA degradation protocol which has been developed to minimize pH fluctuation observed in earlier protocols in which small volumes of concentrated alkali and acids were used. In this improved protocol, RNA is degraded by the addition of 2 µl of 2.5 M sodium hydroxide, and after incubation the solution is neutralized with 10 µl of 2 M Hepes free acid.

The CyScribe First-Strand cDNA Labelling Kit also contains a mixture of synthetic mRNA molecules, as control RNA template, that can be used to gain familiarity of the labelling technique, or to troubleshoot problems. Because of its synthetic nature this control RNA is not suitable for microarray hybridization.

6.13.3 Critical success factors for CyScribe First-Strand cDNA Labelling Kit

- Only label RNA that is intact, clean, and in known quantity.
- If possible, purify mRNA for best labelling results and highest yield of labelled probe.
- Do not exceed the recommendations for template amount.
- Do not alter the amount of CyDye in reaction.
- Pipette all volumes exactly.
- Protect CyDye from light during all handling and storage.
- Do not alter the RNA degradation protocol.
- Monitor the success of purification.
- Measure the amount of purified probe before performing microarray hybridization.

6.14 CyScribe Post-Labelling Kit

6.14.1 Features of the kit

CyScribe Post-Labelling Kit has been developed to offer an optimal and convenient solution for using post-labelling methods in microarray analysis. Each kit provides reagents for performing 12 cDNA synthesis and labelling reactions with both Cy3 and Cy5 fluors. CyScript reverse transcriptase, which is a highly efficient enzyme and gives high yields of cDNA, is used to synthesize amine-modified cDNA by incorporation of aminoallyl-dUTP into first-strand cDNA. The amount of this modified nucleotide in the cDNA synthesis reaction has been adjusted to give an optimal labelling density with CyDye fluors.

The kit includes a protocol for an improved RNA degradation method and two alternative methods for removing amine-containing impurities from amine-modified cDNA. Purified cDNA is reacted with an amount of reactive CyDye NHS-ester that has been chosen to give high and reproducible labelling density, similar to that achieved with the CyScribe First-Strand cDNA Labelling Kit. Protocols for reacting excess NHS-esters with hydroxylamine and subsequent purification of labelled cDNA with column chromatography are also provided.

A practical difficulty in performing CyDye post-labelling of microarray samples has been the necessity for aliquoting and storing CyDye NHS-esters when traditional reagents developed originally for protein labelling have been used. Because of the instability of NHS-esters in moist conditions, their storage in standard laboratory conditions can result in significant loss of reactivity in just a few weeks. CyScribe Post-Labelling Kit solves this problem by providing ready-to-use CyDye NHS-esters in individually dispensed aliquots. These dyes have been sealed in foil to protect them from light and contain desiccant for extra protection. The reactive dyes in these aliquots are also guaranteed to contain over 75% reactive dye content, thus providing the highest quality reagents for microarray labelling.

CyScribe Post-Labelling Kit includes both oligo(dT) and random nonamer primers, offering flexible use of both total and messenger RNA as template. Because this kit yields high amounts of cDNA, it is recommended that 500 ng or less of mRNA and 25 µg or less of total RNA are used per reaction. Highest yield of cDNA is obtained from mRNA using both types of primers together.

6.14.2 Comparison of performance with CyScribe First-Strand cDNA Labelling Kit

Properties of labelled probe:

- The CyScribe Post-Labelling Kit synthesizes about three times as much cDNA than CyScribe First-Strand cDNA Labelling Kit from an equal amount of template. However, because this kit involves two purification steps in which some of the cDNA is lost, approximately two-fold increase in final probe yield is achieved.
- Both kits have been optimized to give similar labelling densities: on average a CyDye fluor is attached at every 12–25 nucleotides.
- As the average length of cDNAs synthesized with the post-labelling kit is longer, higher signals can be obtained from some targets (ones with long transcripts) with this kit.
- The post-labelling method is not influenced by individual nucleotide sequences to the same extent as the first-strand cDNA labelling method is. Therefore, this method can cope better with sequences that contain repeated nucleotide stretches, which can lead to chain termination in first-strand labelling.
- Because of the chemical nature of the labelling process, more random distribution of CyDye fluors over labelled cDNAs is obtained than with the first-strand labelling method in which labelling is modified by sequence-specific events.

6.14.3 Identification of differential gene expression with CyScribe kits

The performance of the two CyScribe labelling kits in identifying differential gene expression was investigated in a model experiment in which human skeletal muscle and placental mRNA populations were compared. Replicate labelling reactions were performed with both systems, purified probes were pooled, and replica slides were hybridized with 25 pmol of each probe. Examples of the expression patterns produced by the two CyScribe Labelling Kits from these

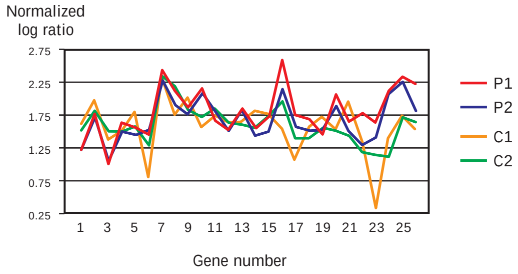


Fig 44. Identification of muscle-specific gene expression with CyScribe Labelling Kits. Data is shown from two replica slides hybridized with skeletal muscle and placental cDNA probes labelled with CyScribe First-Strand cDNA Labelling Kit and CyScribe-Post Labelling Kit.

experiments, as shown in Figure 44, highlight the different intensity of several gene-specific signals. However, when muscle-specific gene expression was examined, essentially the same genes were identified with both methods as being more highly expressed in muscle tissue.

In order to compare the performance of the CyScribe Labelling Kits in analyzing gene expression over a wide range of expression values, 31 random gene sequences were selected for analysis. In skeletal muscle mRNA, the expression levels of these genes cover the whole dynamic range of values (data not shown).

Figure 45 depicts normalized gene expression log ratios for the 31 selected gene sequences. Despite the differences in the intensity of the observed Cy3 and Cy5 signals generated by the two labelling systems (data not shown), the extent of differential gene expression revealed by the two methods is concordant for most of the genes. On the whole, variation between replica slides is greater than variation between the two labelling methods. Some genes show slightly different (within 0.5 log units) values for differential gene expression and may indicate the presence of gene sequence-specific labelling events. However, this data does not support the conclusion that either of the methods is labelling with a systematic bias towards one of the fluors. Rather, a particular labelling method may be more suitable for extracting information from certain gene sequences, and combining data from experiments using different labelling principles could enhance chances of identifying significant difference in gene expression between two samples.

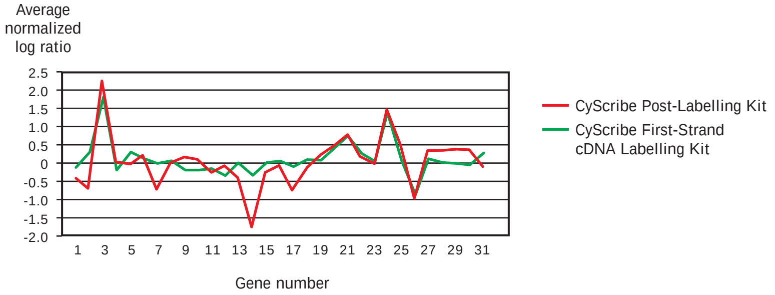


Fig 45. Differential gene expression determined with CyScribe First-Strand cDNA Labelling Kit and CyScribe Post-Labelling Kit for 31 randomly chosen genes in skeletal muscle and placental mRNA samples. Data from two replica slides, each of which contained two identical spot sets for each of the labelling methods, is shown.

6.14.4 Critical success factors for CyScribe Post-Labelling Kit

- Only label RNA that is intact, clean, and in known quantity.
- If possible, purify mRNA for best labelling results and highest yield of labelled probe.
- Do not exceed the recommendations for template amount.
- Do not use random nonamers with total RNA.
- Purification of amine-modified cDNA is critical for labelling success: other amines and acidic ions should be removed.
- Do not alter the RNA degradation protocol.
- Only dissolve CyDye-NHS esters immediately before use.
- Do not reuse CyDye-NHS ester solutions.
- Pipette all volumes exactly.
- Protect CyDye from light during all handling and storage.
- Monitor the success of purification.

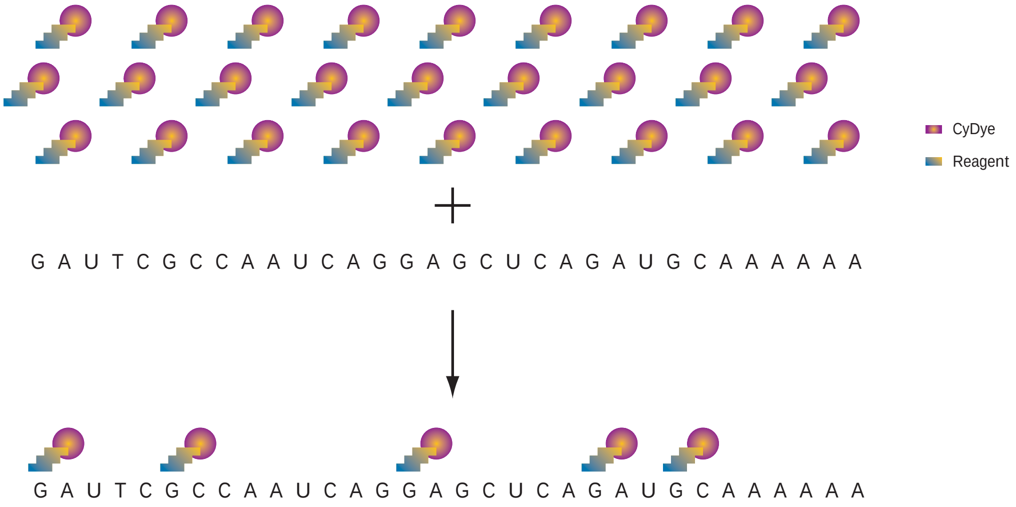

CyDye
Reagent
+
G A U T C G C C A A U C A G G A G C U C A G A U G C A A A A A A
G A U T C G C C A A U C A G G A G C U C A G A U G C A A A A A A

Fig 46. Principle of mRNA labelling with CyScribe Direct mRNA Labelling Kit.

6.15 CyScribe Direct mRNA Labelling Kit

6.15.1 Principle

CyScribe Direct™ mRNA Labelling Kit utilizes Cy3 Direct and Cy5 Direct labelling reagents to generate highly labelled mRNA probes for use in microarray applications. These labelling reagents contain a CyDye fluor attached to a chemical group which can react efficiently with guanine residues, resulting in covalent attachment of CyDye to mRNA (Fig 46). Because of this high reactivity with nucleic acids, Cy Direct™ reagent is hazardous and requires careful handling. When in contact with water, the reagent hydrolyzes and loses its reactivity. As seen in Figure 47, the whole labelling process, including removal of excess dyes, can be performed in less than 2 h.

Because of the chemical nature of the labelling, attachment of both Cy3 and Cy5 is equally efficient and even. Furthermore, the reaction does not require any enzymatic modification of the mRNA prior to labelling. The attachment of CyDye to mRNA does not interfere with subsequent hybridization with DNA targets; thus mRNA labelled with CyScribe Direct mRNA Labelling Kit is suitable for use as microarray probe. Because mRNA can be used directly as the probe, no loss of information because of incomplete or biased transcription process occurs.

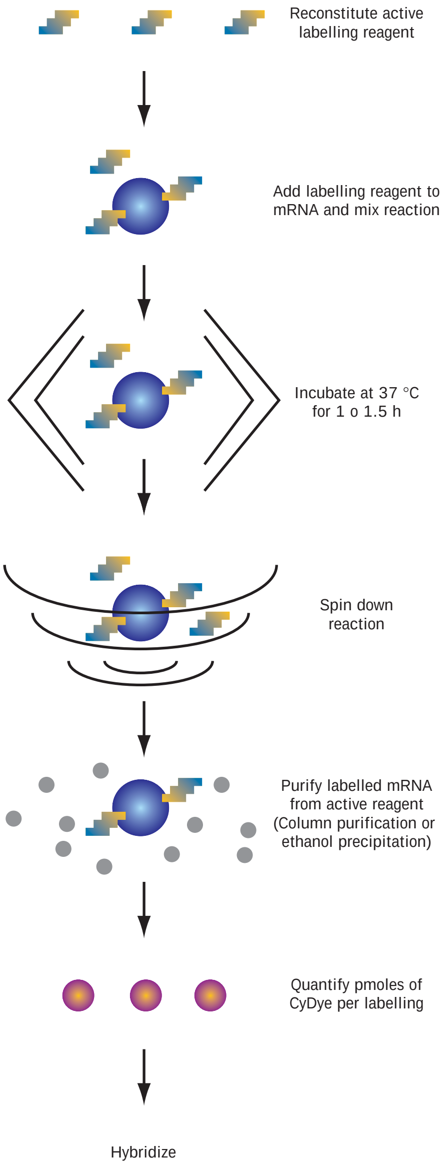

Reconstitute active labelling reagent
Add labelling reagent to mRNA and mix reaction
Incubate at 37 °C for 1 o 1.5 h
Spin down reaction
Purify labelled mRNA from active reagent (Column purification or ethanol precipitation)
Quantify pmoles of CyDye per labelling
Hybridize

Fig 47. The method for labelling mRNA with CyScribe Direct mRNA Labelling Kit.

6.15.2 Application

Labelling reactions performed with CyScribe Direct mRNA Labelling Kit can be scaled up or down to accommodate different amounts of template: as little as 250 ng or as much as 5 µg of mRNA can be labelled in a single reaction. The ratio of the CyDye Direct Labelling Reagent to mRNA determines the labelling density achieved with the kit. Extending the labelling time beyond 1 h can result in higher labelling efficiency than shorter times. The length of transcripts is an important factor in determining the intensity of hybridization signals from mRNA probes labelled with the CyScribe Direct mRNA Labelling Kit. As seen in Figure 48, the expression patterns obtained with the direct labelling method differ considerably from those generated with the first-strand labelling method. This evidence suggests that the direct labelling method may be advantageous for analyzing gene expression from transcripts that are difficult templates for reverse transcription.

The CyScribe Direct mRNA Labelling Kit has been developed for use with purified mRNA that is free from contaminating DNA, proteins, or nucleotides. The kit requires the use of purified mRNA as template because the chemical labelling reaction cannot discriminate between transcripts and other species of RNA. Labelling of total RNA will result in low signal to noise values from microarray experiment.

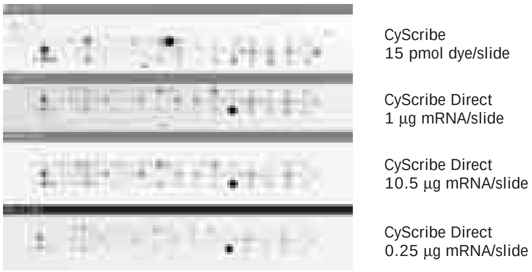


Fig 48. Hybridization of microarrays with varying amounts of mRNA labelled with Cy3 using CyScribe Direct mRNA Labelling Kit. Results from a DNA probe generated with CyScribe First-Strand cDNA Labelling Kit are shown for comparison.

6.16 Critical success factors for sample labelling

- Observe safety precautions while performing the labelling reaction.
- It is imperative that the mRNA is free from contaminating ribosomal RNA, DNA, and proteins.
- Handle RNA so that degradation is avoided. See Chapter 5 for advice. Protect the CyDye Direct labelling reagent from water or any moisture, as it can be inactivated on contact.
- Store the reagent vial and reaction tubes sealed from the environment.
- Protect all reactions and labelling reagents from light when storing and handling.
- Minimize RNAse contamination of mRNA probes during microarray hybridization.

Table 4. Choosing the right CyScribe Kit.

Feature	CyScribe Direct mRNA Labelling Kit	CyScribe First Strand cDNA Labelling Kit	CyScribe Post Labelling Kit
Brightness of signals	●	●	🟠
Even incorporation of Cy3 and Cy5	🟠	○	🟠
Starting material	mRNA	mRNA or total RNA	mRNA or total RNA
Quantity of starting material using total RNA	—	2.5 – 25 µg	2.5 – 25 µg
Quantity of starting material using mRNA	250 ng – 1 µg	50 ng – 1 µg	100 – 500 ng
Possible to prepare a batch of unlabelled cDNA and store	no	no	yes
Simplicity of protocol	🟠	🟠	○
Time from RNA to probe	2 h	3 h	5.5 h
Suitable for less experienced users	🟠	🟠	○
Labelling density	20 – 35 nuc	12 – 25 nuc	12 – 25 nuc

🟠 = Highly recommended   ● = Recommended   ○ = Suitable   — = Not suitable

Chapter 7

CHARACTERIZATION OF LABELLING MICROARRAY PROBES

7.0 Introduction

When performing gene expression microarrays, it is important to characterize the labelled probes in order to avoid experimental error derived from variation between the probes. Even under carefully controlled conditions, some differences in the amounts of nucleic acid samples, labelling success, and recovery of material from the purification system will occur. In order to account for these artifacts and to ensure that these variations are not carried through to hybridization, it is highly recommended that the properties of the labelled probes are determined before microarray hybridization.

Several methods can be used for characterizing the properties of the labelled probes. As a minimum we recommend routinely measuring the amount of CyDye in the labelled and purified sample. If problems do occur in microarray hybridizations, the other methods described below can be used for troubleshooting purposes.

- The quantity of CyDye and nucleic acid in the labelled probe can be determined using spectrophotometry.
- Radioactive spiking can be used to derive information about the amount of nucleic acid in the sample and the success of purification.
- Gel electrophoresis and fluorescence scanning can be used to analyze the molecular distribution of the probe, its purity, and relative fluorescence.

7.1 Determination of the amount of CyDye in a labelled sample with spectrophotometry

Spectrophotometry can be used to determine the amount of CyDye incorporated into labelled nucleic acid. This can be achieved by measuring the absorbance of the solution containing the nucleic acid at the absorption maximum for Cy3 and Cy5. These wavelengths are 550 nm for Cy3 and 650 nm for Cy5. From the known extinction coefficients corresponding to these wavelengths, the concentration and amount of CyDye in the sample can be calculated. The amount of CyDye in the purified sample can be used as a guide to optimize the amount of probe in the hybridization. Best results are achieved when the amounts of Cy3 and Cy5 in dual color hybridization are equal.

It is necessary to purify the labelled nucleic acid before performing the spectrophotometry analysis, as any residual, unincorporated CyDye labelled nucleotides will interfere with the detection of CyDye labelled cDNA.

7.1.1 Measuring the amount of CyDye in the probe with spectrophotometry

Follow the steps below to measure the amount of incorporated CyDye:

❶

Sample preparation

- Dilute an aliquot of purified probe with water. The required volume depends on the size of available measuring cells. It is recommended to use the smallest cells possible. Small volume disposable measuring cells are available, and they offer the added advantage that a separate cell can be used for each sample, thus minimizing cross contamination from one sample to another. However, it is also possible to use regular 100 µl glass cells, but these need to be cleaned thoroughly by rinsing with sterile water between samples.

❷

Measurement

- Measure the absorbance spectrum of the sample from 200 to 700 nm against a blank. Although only the absorbance values at the absorbance peaks are needed to calculate the amount of CyDye in the sample, the shape of the absorption spectra can give additional information about the quality of the labelled sample. The absorption spectrum of CyDye contains two peaks, of which the second should be of higher intensity (Fig 49). The first peak indicates the presence of intermolecular interactions between different dye molecules. If both peaks are of nearly similar intensity, this is a sign of quenching, i.e. loss of fluorescent signal because absorbed light energy is spent on intermolecular interactions. This is usually associated with over-labelling of the sample.

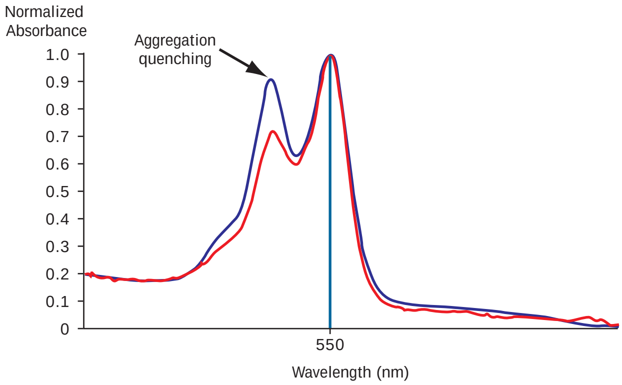


Fig 49. Quantification of the amount of CyDye in a labelled probe with spectrophotometry: examples of absorption spectra from Cy3-labelled samples. The presence of two absorption peaks of near identical intensity (gray line) is a sign of intermolecular interactions and aggregation, which can reduce fluorescent signal from the probe. The red line shows a typical profile of Cy3 absorption spectrum.

- Measure absorbance at 550 nm for Cy3 and at 650 nm for Cy5 using cuvettes with 1 cm path length to determine the amount of CyDye in the sample. The observed absorbance values depend on how much the sample was diluted before measurement, how much RNA was used in the labelling, the labelling method used, and the efficiency of labelling. Typically, values around 0.050 would be expected from first-strand cDNA labelling reactions in which 1 µg of mRNA is used as a template, when the sample is diluted to 100 µl before measurement. Choose a dilution that will give a measurement that is clearly discernible from background values. Sometimes it is necessary to use all of the labelled sample for analysis.
- Recover the measured sample for future use. It may be necessary to concentrate the sample before using for microarray hybridization. This can be performed by drying down the sample in a vacuum concentrator or by letting sample that has been heated to 60 °C evaporate to dryness. It is important to protect the sample from light during all handling.

❸

Calculation

- The amounts of Cy3 and Cy5 incorporated into probes can be calculated from their respective extinction coefficients and using the following equations:

Extinction coefficients

150 000 M^{-1} cm^{-1} at 550 nm for Cy3 and
250 000 M^{-1} cm^{-1} at 650 nm for Cy5

Calculation equations

pmol Cy3 in purified sample =
$(A_{550}$ / 150 000) × dilution factor × (z µl) × (w cm) × 1012
where
A_{550} = absorbance at 550 nm
z µl = the volume of sample after purification
w cm = optical path in cuvette

pmol Cy5 in measured sample =
$(A_{650}$ / 250 000) × dilution factor × (z µl) × (w cm) × 10^{12}
where
A_{650} = absorbance at 650 nm
z µl = the volume of sample after purification
w cm = optical path in cuvette

7.2 Determination of the amount of labelled nucleic acid in the sample

7.2.1 UV spectrophotometry

Two methods can be used to determine the amount of labelled nucleic acid in the sample: spectrophotometry and radioactive spiking. Nucleic acids absorb at 260 nm, and absorption measurement at this wavelength can be used for quantitative purposes. However, several purification systems release other materials absorbing near or at this wavelength and, depending on the amounts of these compounds, most of the absorbance at 260 nm will be derived from something other than nucleic acid. Therefore absorption spectra from 200 to 400 nm should be measured, and only if the peak at 260 nm is clearly distinguishable from absorption at near wavelengths, should estimation of nucleic acid amount be made (Fig 50).

The following approximations can be used to calculate the amount of nucleic acid in the probe:

1 A_{260} unit of double-stranded DNA corresponds to 50 µg/ml

1 A_{260} unit of single-stranded DNA corresponds to 37 µg/ml

1 A_{260} unit of single-stranded RNA corresponds to 40 µg/ml

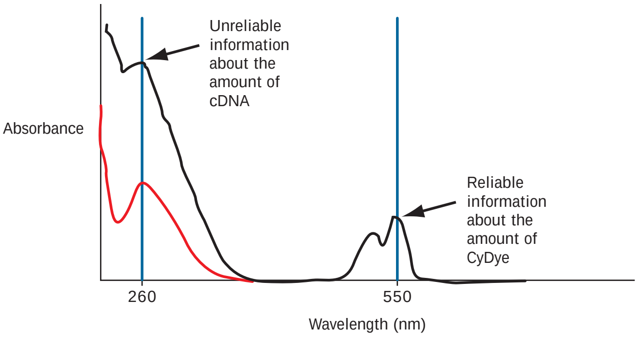


Fig 50. Quantification of the amount of nucleic acid in probe with UV spectrophotometry. The sample, from which the black spectrum was generated, contains impurities absorbing near 260 nm and does not give reliable information about the amount of nucleic acid in the sample. The sample from which the red spectrum was measured is cleaner and the A_{260} can be used for quantification. In both cases, the absorbance spectra for Cy3 at 550 nm gives usable information.

7.2.2 Spiking with radioactive nucleotide

If more accurate quantification of the labelled nucleic acid is needed, the labelling reactions can be spiked with a small amount of a radioactive nucleotide. This enables the determination of the amount of nucleic acid synthesized as well as the amount of sample recovered from purification. This approach can be used with all labelling methods in which new nucleic acid is synthesized. For labelling in cDNA synthesis, a deoxynucleotide must be used, and for labelling of synthetic RNA, a ribonucleotide is needed. In order to minimize interference with the labelling process, the nucleotides used as spike and label should be different. For example, dATP is suitable for use as a spike with both CyDye-dCTPs and CyDye-dUTPs. Only small amounts of the radioactive nucleotide are needed—2 µCi per sample or less is adequate—and isotopes of lower energy, such as ^{33}P, can be used. Note that the absolute amount of the radioactive spike is not critical, as long as the radioactivity can be measured accurately. The low concentration of radioactive nucleotide solutions means that the spiked nucleotide will not contribute significantly to the total amount of that nucleotide in the reaction. The amount of the radioactive isotope incorporated into labelled sample is relatively small and does not interfere with the detection of fluorescence.

The information derived from the radioactive spike can be used to estimate the amount of nucleic acid synthesized. It is reasonable to assume that the radioactive nucleotide and unlabelled nucleotide are incorporated by enzymes at a similar rate. Therefore, if the percentage of the radioactive nucleotide incorporated into the synthesized product is known, it can be concluded that the same percentage of the unlabelled nucleotide is incorporated as well. If the total amount of this nucleotide in the labelling mix is known, and it is assumed that all four nucleotides are incorporated with equal efficiency (this is probably not absolutely true), it is possible to calculate the amount of nucleic acid synthesized. This method only estimates the amount of nucleic acid synthesized, but in most cases gives more accurate data than UV spectrophotometry.

In order to quantify the amount of nucleic acid synthesized by spiking, a small amount of radioactive nucleotide is added to the labelling reaction. In order to keep the total reaction volume unchanged, the amount of water pipetted into the reaction must be adjusted. The radioactive nucleotide should not contain any colored marker or stabilization agents, as these compounds are fluorescent in the same regions of visible spectrum as CyDye and can interfere with determination of CyDye amount.

Note: Because of the radioactivity present in the spike, necessary precautions for working with radioactive materials must be followed.

No other special modifications are needed for the labelling reactions. After the labelling reaction has been completed and before the sample is purified, the incorporation of the radioactive nucleotide into the synthesized nucleic acid needs to be determined. The incorporated radioactive nucleotide can be easily separated from free nucleotides by thin layer chromatography on PEI cellulose chromatography plates (Fig 51).

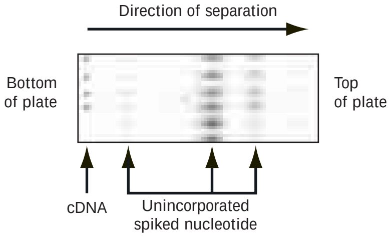


Fig 51. Thin layer chromatography analysis of the incorporation of radioactive spike into labelled cDNA sample. The labelled nucleic acid does not migrate far from the sample application line, whereas the unincorporated radioactive nucleotide moves up toward the top of the plate.

7.2.3 Thin layer chromatography analysis of spike incorporation

To perform thin layer chromatography, follow the steps outlined below:

❶

Preparation

- Prepare a 20 × 20 cm glass-backed PEI cellulose chromatography plate (available from Merck) for use by cutting a thin linear groove into the PEI cellulose layer at 1 cm distance from the top edge of the plate. Mark sample positions along a line that is 3 cm from the bottom edge of the plate.

❷

Titration

- Pipet 0.5 µl samples of labelling reactions in duplicate along the marked line. The sample spots should be about 1 cm apart. Do not damage the PEI cellulose layer with the tip.

❸

Separation

- Place the plate in a rectangular chromatography tank so that the bottom of the plate is immersed 2 cm deep in 1 M K_2HPO_4. Make sure that the level of the buffer is below the level of the marked sample line. Cover the tank and let sample separation take place. Newly synthesized nucleic acid will not move far from the sample line whereas free nucleotides will move progressively towards the top of the plate.
- When the buffer has reached the top groove, i.e. when the samples have migrated the full available length of the 20 cm plate, remove the plate from the tank and let it air dry.

❹

Imaging

- Wrap the plate in cling-film and expose it to a phosphor screen for 1–6 h. Take care not to overexpose the phosphor screen as it will saturate the signal. Some trial and error may be needed to determine correct exposure time.
- Scan the phosphor screen on a Typhoon™ Variable Mode Imager, using the recommended settings. As an alternative to using phosphor screens, any instrument that can accommodate chromatography plates and measure radioactivity quantitatively can be used.

❺

Analysis

- Calculate the proportion of the radioactive nucleotide that is associated with the synthesized nucleic acid. This is the incorporation percentage. For example, the ImageQuant™ software can be used for this purpose.
- Calculate the yield of nucleic acid as follows:

Yield of nucleic acid =

(mol of cold nucleotide in labelling mix) × (incorporation percentage) × 4 × 330 g/mol

This formula assumes that all four nucleotides are incorporated in equal proportions (hence, times 4) and that the molecular weight of average nucleotide is 330 g/mol.

For example first-strand cDNA labelling was performed with 2 nmol of dATP in the reaction. 20% of $[\alpha\text{-}^{33}P]$dATP was incorporated into cDNA.

Yield of cDNA =
2×10^{-9} mol × 20% × 4 × 330 g/mol = 528 ng

7.3 Calculation of labelling density

Labelling density can be defined as the amount of CyDye incorporated into a known amount of nucleic acid. It can be expressed as pmol of CyDye incorporated into µg of nucleic acid, or can be converted to the number of CyDye molecules per 100 nucleotides. Labelling density is a measure of the average distance between CyDye fluors on the labelled nucleic acid. Samples labelled successfully, with an optimized protocol, will usually have similar labelling densities, but problems with the labelling reagents or in performing the protocol can result in variation in the incorporation of CyDye into cDNA. For example, in the cDNA post-labelling method, exposure of the CyDye-NHS esters to moisture during storage would result in low labelling density without affecting the amount of cDNA synthesized.

Once the amount of CyDye incorporated into nucleic acid and the amount of the nucleic acid are known, labelling density can be calculated.

$$\text{Labelling density} = \frac{\text{pmol CyDye in labelled sample}}{\mu\text{g nucleic acid in labelled sample}}$$

1 µg of cDNA contains approximately 1×10^{-6} /330 = 3030 pmol of nucleotides.

Hence labelling density of 100 pmol/µg equals 100 pmol/3030 pmol of nucleotides = 3.3 CyDye nucleotides per 100 nucleotides = 3030 pmol of nucleotides.

7.4 Recovering labelled nucleic acid after purification

The recovery of CyDye labelled nucleic acids from purification systems can be variable. In order to draw conclusions about the performance of the purification process, two aspects need to be considered. First, the recovery of the labelled material needs to be determined. Poor recovery can limit the amount of slides that can be hybridized with the sample, and the true benefits of labelling methods may not be realized. Second, the presence of free CyDye needs to be assessed. Free CyDye in the purified probe will affect the determination of CyDye amount in the labelled nucleic acid and can result in too little probe being used in hybridization. Free CyDye, especially free CyDye-

$$\text{recovery \%} = \frac{(\text{cpm after}) \times (\text{volume recovered}) \times 100\%}{(\text{cpm before}) \times (\text{incorporation \%/100}) \times (\text{volume purified})}$$

nucleotide, can also contribute to hybridization background and give rise to speckled images that are difficult to quantify. Gel electrophoresis is a convenient method for determining whether the purified probes are free of unincorporated dye.

Radioactive spiking, as explained above, can be used to obtain information about the purification process. Scintillation counting of a small aliquot (0.5–1 μl) of labelling reaction sampled before and after purification can be used to calculate the proportion of cDNA (or other nucleic acid) recovered. For this calculation it is necessary to know what proportion of the radioactive nucleotide was incorporated into the nucleic acid. This can be determined from the unpurified labelling reaction with thin layer chromatography analysis as explained above.

7.5 Analysis of the composition and fluorescence of labelled sample

Gel electrophoresis can be used to investigate the quality of the labelled sample. Separation of unincorporated CyDye from labelled nucleic acids can be achieved with denaturing polyacrylamide gel electrophoresis (PAGE) or agarose gel electrophoresis. After electrophoretic separation, the gel can be scanned to detect Cy3 and/or Cy5 fluorescence using a multipurpose scanner, such as Typhoon 9410 Variable Mode Imager. From the scanned image it is possible to detect how much free CyDye there is present in the sample. If PAGE gels are used, the size range of the labelled nucleic acid population can be examined. This can give an indication of the quality of the starting population of RNA and success of the labelling reactions. If unpurified sample is separated alongside purified samples, it is possible to estimate the recovery of probe from purification. By comparing the relative fluorescence of different labelled samples, the relative amount of CyDye in each sample can be estimated (Fig 52).

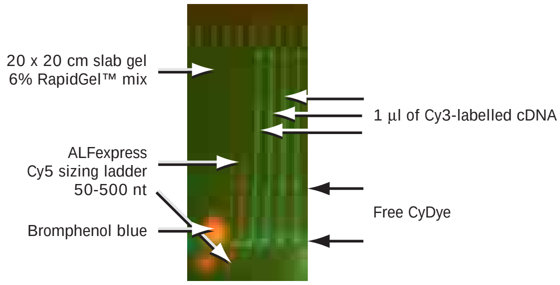


Fig 52. PAGE analysis of Cy3-labelled cDNA probes. 1 μl samples of Cy3-labelled and purified cDNA probes prepared with cDNA post-labelling method were separated in 6% sequencing gel. The gel was scanned for Cy3 and Cy5 fluorescence with Typhoon scanner to detect ALFexpress™ Sizer and the labelled probes The cDNA probe consists of fragments longer than 150 nucleotides and some unincorporated Cy3-reactive dye is present, as indicated. All four samples show similar relative fluorescence and quality.

7.5.1 Analyzing CyDye labelled probes with PAGE

In order to analyze CyDye labelled probes using PAGE, follow the procedures detailed below:

❶

Preparation

- PAGE gels suitable for analysis of CyDye labelled samples can be prepared from standard 6% or 8% (w/v) sequencing gel mix such as RapidGel-XL-6%. The gel can be cast using equipment designed for sequencing, but since single base pair resolution is not required, slab gel instruments can be used. These provide the added benefit of thicker and deeper wells that simplify sample loading.
- 0.1–1 µl of purified labelling reaction is enough for detection of CyDye labelled nucleic acid by fluorescence scanning. Add 2 µl of formamide and 8 µl of water to each sample. Do not use normal loading/denaturation buffers which contain dyes such as bromophenol blue or xylene cyanol, as these will interfere with the detection of CyDye fluorescence.
- Dilute 4 µl of fluorescent markers such as ALFexpress Sizer 50-500 with 4 µl of water and 2 µl of formamide. This sizer contains a Cy5-labelled DNA marker ladder.

❷

Denature

- Denature all samples by boiling for 2 min at 95 °C. Snap cool on ice before loading on to the gel.

❸

Electrophoresis

- Load samples to a gel that has been pre-electrophoresed for 15–30 min. Perform electrophoresis according to the instructions provided with your equipment. Use 1× TBE as buffer. Protect the samples from light during the electrophoresis.
- In order to help monitor the progress of electrophoresis, you can load a small aliquot of loading buffer (1 µl) containing bromphenol blue into a side well of the gel that is well separated from the wells containing labelled cDNAs. Stop the electrophoresis when the bromphenol dye is still well within the gel. Unincorporated CyDye will migrate faster than bromphenol blue.
- Remove one of the gel plates before scanning and make sure that the back of the remaining plate is clean. Do not let the gel dry before scanning.

❹

Imaging

- Scan the gel on a Typhoon Variable Mode Imager. Detect Cy3 by excitation with 532 nm laser and using emission filter 555 BP 20. Detect Cy5 by excitation with 633 nm laser and using emission filter 670 BP 30. Set PMT to 800 V, focal plane to +3 mm, and use normal sensitivity. The PMT values may need to be adjusted to account for different amounts of sample in the gel.

Agarose gel electrophoresis can be used instead of PAGE. Single-stranded CyDye labelled nucleic acid fragments will not migrate true to their size in standard agarose gels, but valid information about the purity and fluorescence of labelled sample can be obtained. 0.1–1 µl of labelling reaction can be diluted by adding 2 µl of 50% (v/v) glycerol and water to 10 µl. No loading dyes should be used. Denaturation of samples is not necessary before gel loading. Run the gel in 1× TBE or 1× TAE.

Chapter 8

MICROARRAY HYBRIDIZATION

8.0 Introduction

The process of hybridization is typically performed in order to identify and quantitate nucleic acids within a larger sample. Generally, it involves annealing a single-stranded nucleic acid to a target complementary strand. Southern blotting is one well-established hybridization method. In this technique, samples of genomic DNA—the target—are attached to a membrane and incubated within a solution of fluorescently labelled DNA—the probe (54). Binding of probe molecules to the sample on the membrane highlights complementary sequences, and the intensity of signal is proportional to the amount of immobilized sample.

The microarray hybridization technique works in a very similar way to that of Southern blotting, except that it is carried out in reverse, and the target is first attached to a slide instead of a membrane.

There are several critical factors to performing a successful microarray hybridization. The following are discussed in detail in this chapter:

- Pre-hybridization
- Hybridization conditions
- Hybridization buffer
- Stringency washes

8.1 Overview of the microarray hybridization process

❶

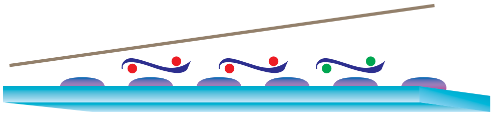
Attachment

Genes of interest are spotted onto a solid surface by the array spotter. These are known as the targets. Attachment chemistry will often be required to ensure that the DNA remains attached to the slide surface throughout the hybridization process.

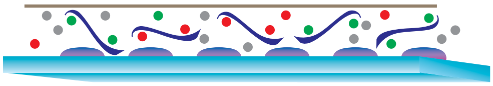
❷

Hybridization

Hybridization buffer containing a known amount of labelled sample DNA—often referred to as probe— is then placed on the slide surface. A coverslip can then be carefully placed on top of the slide.



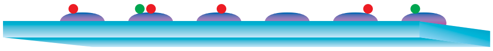
The slide is then incubated in a humid environment for up to 16 h. During this time the labelled probe is in contact with the targets on the slide. If the sequence homology is good then the probe will adhere to the target.



❸

Washing

Once the hybridization is complete, the slides are washed, and buffer and probe of little or no homology to the target will be washed away, leaving the labelled probe of high homology attached to the target and available for detection.



8.2 Pre-hybridization

Pre-hybridization consists of incubating the spotted slide in a buffer in the absence of probe. Different slide chemistries require slightly different pre-hybridization protocols that vary in the type of buffer used. Consult manufacturers' recommendations to find out what is the best procedure for each slide. Pre-hybridization prepares the microarray for hybridization in the following two ways:

- Badly adhered target is washed away during pre-treatment. If this step is omitted, this target will often wash off the slide surface during the hybridization and will hybridize to the probe in solution, thus competing with the immobilized targets. This can decrease hybridization signal.
- Pre-treatment ensures that the target is available for hybridization. The target is normally double-stranded DNA and, although targets are frequently spotted in a denaturing solution such as DMSO, most microarray protocols do not contain a specific denaturization step. Pre-hybridization may also act to block any sites on the slide surface that are capable of binding the probe nonspecifically, thus improving the backgrounds.

8.3 Hybridization

There are several widely used methods for carrying out the hybridization, either using automated instruments or performing the procedure manually. In this chapter we describe the manual hybridization method, which is the most widely used. General properties of manual hybridization will be discussed, followed by advice for choosing a suitable hybridization method for different microarray slide types.

8.3.1 Coverslip method

In the coverslip method, hybridization buffer containing the probe is incubated on the microarray under a coverslip. This way only a small volume, typically about 30 µl, of buffer is needed. The coverslip method is a very convenient one in that it requires no special equipment. However, a microarray slide under coverslip is prone to drying out, especially around the edges, causing most of the array to become unusable. Originally, coverslips were sealed on the slide. This prevented drying but made the coverslip difficult to remove prior to detection. A more practical approach is to carry out the hybridization in a humid environment, thus preventing evaporation of the hybridization buffer from beneath the coverslip. This can be achieved with anything from a humid chamber with slide trays and a reservoir, to a plastic box lined with wet tissues at the base (Fig 53). In either case, the atmosphere within the chamber must remain at >95% humidity throughout the 16-h hybridization. Equally important is not to allow the slide to come in contact with the water, which may dilute the probe or cause water marks on the slide surface.

a)

b)

Fig 53. Practical solutions for ensuring that a humid environment is maintained during manual microarray hybridization. A plastic box (a) containing a platform raised above moistened tissues is sufficient for hybridizing a few slides. Commercially available humid chamber (b) holds up to 40 slides on removable trays and fits into most lab ovens.

8.3.2 Hybridization buffer

The hybridization buffer and conditions used are vital for successful results. Hybridization buffers vary considerably but will normally contain the following components:

- a buffering component that acts to stabilize variations in pH
- a detergent that acts to lower the surface tension and allow the buffer to flow easily under a coverslip
- compounds that act as rate enhancers, volume excluders, or to speed up the hybridization and lower the T_m

Melting temperature (T_m) is the temperature at which 50% of the probe is denatured. This temperature will be affected by both the size and G-C content of the probe fragments, but the effect is minimized by optimizing the salt content and formulation of commercial hybridization buffers, thus making them suitable for use with most probes without optimization. Formamide is a denaturing reagent that is often used to lower the T_m of the probe and hence the temperature of hybridization. The optimum hybridization temperature for microarrays, in aqueous buffers, will be high (65–75 °C). At these high temperatures drying out of the slide becomes more of a problem; the probe is also more likely to degrade. The addition of formamide to a buffer decreases the T_m by 0.65 °C for every 1% concentration; therefore, the addition of 50% formamide to the hybridization lowers the optimum temperature to a more reasonable 42 °C (55). However, hybridizations carried out in formamide should be left for 16 h, unless the probe concentration is increased.

8.3.3 Probe blocking

Most manufacturers recommend some type of probe blocking either prior to or during the hybridization to prevent nonspecific hybridization of probe to common genetic elements. One common blocking agent is poly-dA oligo, which hybridizes to poly-dT tails (formed during the cDNA probe synthesis by the poly-dA oligo, and prevents probe from hybridizing with poly-A sequences often present in targets. Other types provide more general forms of blocking, such as salmon sperm and yeast tRNA to block non-specific binding and the inclusion of Cot1 DNA™ mop up repetitive sequences. The blocking agents are normally added to the labelled probe/hybridization buffer solution prior to applying to the slide surface. The solution can then be heated to denature any double-stranded DNA and to allow the blocking to take place, before setting up the hybridization reaction.

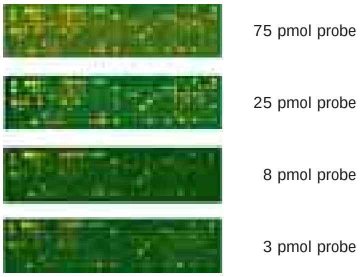


Fig 54. Cy3- and Cy5-labelled probes are hybridized on Amersham Biosciences reflective slides at 75, 25, 8 and 3 pmol of each dye per slide. Although hybridization signals can be detected using as little as 3 pmol of each probe, the intensity of signals is greatly increased by using 25 pmol of each probe, thus allowing the rarer messages to be visualized.

8.3.4 Probe concentration

The amount of probe to add to a hybridization will vary, depending on the samples used, the slide type, and what information is expected to be gained. Slide manufacturers will recommend optimum probe concentrations to use in hybridizations with their slides. If two or more colors are being used, it is important that exactly the same amount of probe labelled with each dye is added so as not to skew the results in favor of one of the probes. For glass slides 30 pmol of each labelled dye is sufficient for most systems, but this should be reduced by as much as half when using mirrored slides, which contain a reflective layer capable of enhancing signal intensities. Increasing the amount of probe used will increase the result obtained but only up to a certain point (Fig 54), beyond which the increase in background levels will actually decrease the amount of background-corrected signal (Fig 55).

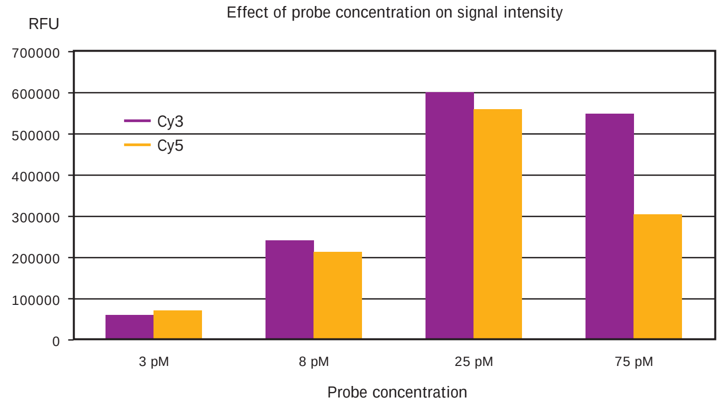


Fig 55. When increased amounts of labelled probe are used, there is an increase in background levels. This decreases the amount of background-corrected signal detected from the microarray slide.

8.3.5 Probe depletion and target saturation

As in most manual hybridizations the reaction is carried out under a coverslip, which means that there is sufficient solution under the coverslip for complete coverage of the array but no room for any movement or flow of the buffer under the coverslip during the hybridization. Whether a labelled probe fragment finds its complementary target on the microarray is therefore simply relying on diffusion of the probe. Research has shown that a 20 bp oligo diffuses a distance of 3.6 mm in an 18 h period, therefore a labelled cDNA sample is hardly going to move during an overnight hybridization. This means that if there were several replicate target spots concentrated in a small area on an array, these spots would be competing for a limited amount of complementary sequences within diffusion distance. This is called probe depletion, and it can limit the signal obtained from microarray. This will be most relevant for those transcripts that are present in low numbers in the labelled samples, as the signal from these spots may fall below detection sensitivity of the microarray system.

The sensitivity of the microarray system is determined by several factors including the amount of label attached to the probe molecules, the level of background signal, and the sensitivity of the scanner. Furthermore, the rate at which the probe molecules find their targets is a more critical determinant of sensitivity than the amount of spotted target. For most low- to medium-abundance genes, the amount of spotted target is in a huge excess over the probe molecules. For a high abundance gene, the amount of probe in solution starts to approach the amount of target present, which can lead to target saturation. Target saturation will be determined by factors such as the amount of target initially spotted on the slide and the amount retained on that slide after pre-treatment, as well as the percentage of that target available for hybridization and the efficiency of hybridization. Together, the sensitivity of detection and target saturation determine the dynamic range of the microarray experiment.

8.3.6 Use of hybridization chambers

Hybridization chambers are used in order to overcome any problem arising from the amount of buffer used under coverslips and possible probe depletion. These are plastic chambers that hold an individual slide and a larger volume of hybridization buffer. The amount of labelled probe added remains the same. These are then incubated overnight in a water bath or oven. In order to introduce a significant amount of mixing, using an automated hybridizer is recommended.

8.3.7 Practical tips for setting up coverslip hybridization

This protocol gives instructions how to set up a microarray hybridization using coverslips. It should be performed with clean slides and coverslips, preferably in a clean room or under a hood. It is recommended to use plastic coverslips that have been packed in plastic film. Do not use gloves that contain powder, as this can easily get onto microarray slides and cause background problems.

❶

Pre-hybridization

- Store spotted slides in a desiccator until use.
- Read through the pre-treatment and hybridization protocols thoroughly before use, as buffers often need preparing and preheating before use.
- Pretreat the number of slides required for the experiment. Some manufacturers say that pre-treated slides can be stored before use, but not all, so it is worth checking before doing a large batch.
- During the pre-treatment stage, prepare the probes. This will often involve drying down equivalent amounts of the two probes of interest together and reconstituting them in the manufacturer’s recommended buffer. Some protocols require heating the probe before use; the probe prepared by reverse transcription will be single-stranded and therefore should not require denaturing before use.

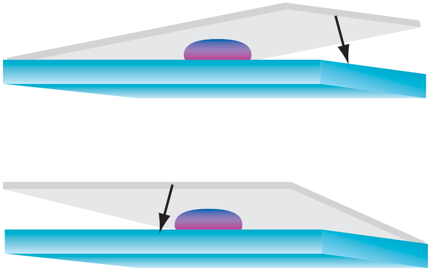


Fig 56. Lowering the coverslip.

❷

Hybridization

- Once the probe is prepared, lay the spotted slide, DNA side up, on a clean surface. Absorbent tissue that does not release any fibers is a good choice of surface. This is important, as most slides are glass, and dirt on the rear of the slide will affect the result on the front of the slide (this is obviously not an issue with opaque or mirrored slides).
- Using a pipette, transfer the required amount of hybridization buffer/probe mixture onto the slide. Avoid touching the slide surface with the pipette tip. Try to deposit the mixture along the short side of the slide, away from the spotted area.
- Take a clean coverslip and place it on the slide near the probe mixture, and allow surface tension to speed the buffer along the coverslip. Then gently lower the coverslip, avoiding trapping air bubbles underneath. There are several ways of lowering the coverslip, two of which are illustrated in Figure 56.
- If air bubbles become trapped beneath a coverslip (Fig 57), do not move the coverslip to try and remove them. Movement of the coverslip will result in damage to the targets themselves. Most small air bubbles will disperse once the slide is transferred to hybridization temperature. Larger bubbles can be 'encouraged' to move by gently pressing on the surface of the coverslip with a pipette tip.
- The probe mixture is light sensitive, so once the coverslip is on, place the slide in the humid chamber and incubate overnight in the dark.

Note: If using RNA probes, it is important to take appropriate precautions to protect all reagents from nucleases (see Chapter 5 on RNA handling).

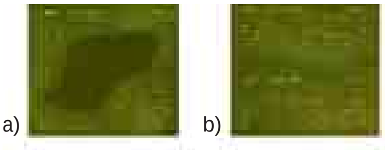


Fig 57. Typical problems encountered in microarray hybridization. Trapping of air bubbles (a) beneath the coverslip will lead to areas on the array that fail to hybridize at all. Allowing the slide to dry out during the hybridization will lead to high patchy backgrounds (b) that may cause difficulty at the analysis stage.

8.4 Stringency washes

The purpose of the post-hybridization washes is to remove all unattached and loosely bound probe molecules. This prevents false positive signals and removes all components of the hybridization buffer, preventing background noise in the form of smearing and speckles. Again, as the slides are light sensitive at this stage, the washing steps should be carried out in the dark so as to minimize signal loss due to bleaching of the fluorescent dyes. Once the slides have been washed, they should immediately be dried by centrifugation or nitrogen steam to prevent smearing while drying. The slides should then be stored in the dark in a desiccator and scanned as soon as possible. If, once scanned, it is found that the slides have high background or low stringency, it is worth re-washing the slide and re-scanning.

The stringency washes will affect the amount of labelled probe retained on the slide for the final analysis. While it is obviously important to remove all the loosely bound probe, it is important to not strip the bound probe. Generally, stringency washes are carried out in a SSC/SDS solutions of different concentrations, with the primary washes often being carried out at the same temperature as the hybridization. Primary wash solutions have a high salt content (typically 1–2× SSC/0.1% SDS buffer), and they remove most of the hybridization buffer components. The secondary washes are performed with low salt buffer (typically 0.1× SSC/0.1% SDS), and they will remove the loosely bound probe from the blot. It also serves to remove any remaining salt from the primary washes. Failing to warm solutions thoroughly before use will lower their effectiveness and may lead to increases in background noise. Conversely, warming solutions too much (as often happens if a microwave oven is used) or using too low a salt concentration in the buffers, will strip precious signal from the blot. Check the manufacturer's protocols for exact wash dilution volumes. Manufacturers often suggest a water dip prior to drying the slides to prevent smearing. Check the protocols provided with the buffer components for instructions on performing the water dip.

8.5 Microarray slides

8.5.1 Choosing the right hybridization protocol for different slide types

Most manufacturers of microarray slides will provide a hybridization protocol that they have optimized for their system. The following table lists some of the more commonly used slides and a brief summary of tested hybridization protocols for them. This in not an exhaustive list and the protocols are only for reference. Please refer to the manufacturer's own protocols prior to use.

Table 5. Microarray slide types and their characteristics.

Slide type	Manufacturer	Spotting chemistry	Hybridization buffer	Pretreatment protocol	Hybridization
Lucidea Reflective Slides	Amersham Biosciences	50% DMSO	Version 2 (4×)-formamide based	None required	42 °C overnight
CMT-GAPS	Corning	50% DMSO or 3× SSC	25% formamide, 5× SSC, 0.1% SDS	25% formamide, 5× SSC, 0.1% SDS for 45 min at 42 °C	42 °C overnight
SigmaScreen™	Sigma-Aldrich	3× SSC	ArrayHyb	1% SDS for 2 min, water rinse. Boiling water for 2 min, ethanol dip	50 °C 6 h overnight
Type I	Clontech	150 mM Na phosphate	GlassHyb	Optional: 70 mM succinic anhydride in 315 ml 1-methyl-2-pyrrolidinone and 35 ml Na borate pH 8–15 min at RT. Boiling water for 2 min. Ethanol dip.	50 °C overnight
Type II	Clontech	150 mM Na phosphate	GlassHyb	70 mM succinic anhydride in 315 ml 1-methyl-2-pyrrolidinone and 35 ml Na borate pH 8–15 min at RT. Boiling water for for 2 min. Ethanol dip.	50 °C overnight
Super Amine	Telechem Int.	5× SSC	UniHyb (1.25×)	0.1% SDS twice for 2 min at RT followed by a water rinse. Incubate in bolling water for 3 min before drying.	42–65 °C overnight

Chapter 9

LUCIDEA SLIDEPRO HYBRIDIZER

9.0 Introduction

A complex system, microarray analysis is affected by a number of experimental factors, including target preparation, physical deposition of the targets, slide chemistry, probe chemistry, hybridization, and detection of the fluorescent signal (37). In order to improve the efficiency of microarray analysis, each source of variation must be eliminated or minimized. Control of environmental conditions during hybridization in particular, is critical in producing and maintaining a consistent fluorescent signal. Lucidea SlidePro was designed to overcome the problems associated with hybridization variability.

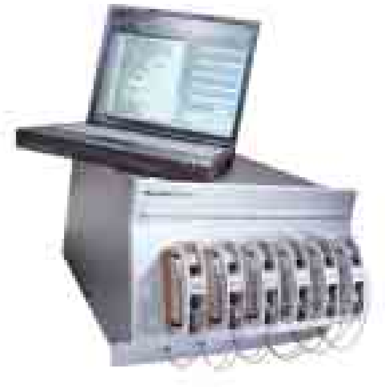


Fig 58. Lucidea SlidePro Hybridizer.

9.1 Features of Lucidea SlidePro Hybridizer

Lucidea SlidePro Hybridizer (Fig 58) has a modular format that consists of a base control unit, up to four additional modules, and control software on a laptop computer. Each unit contains six individually temperature-controlled chambers, each of which holds a standard microscope slide. With all five modules a total of 30 slides can be processed. The multi-module protocol software allows each module to be started at different times and different experiments can be conducted on each module, thereby increasing user flexibility. Also, each module has its own pump, which allows a faster processing time.

Lucidea SlidePro is capable of automating a variety of chemical and biochemical techniques in which incubation and wash steps are performed at varying temperatures. It is primarily used in microarray analysis to automate the pre-hybridization, hybridization, and washing of microarray slides. It has been designed to be used in conjunction with the Lucidea range of microarray instruments and reagents.

In Lucidea SlidePro, each slide is held in a chamber sealed with a patented O-ring. Pre-treatment and wash solutions are drawn into the chamber, from up to five reservoirs, and deposited into a waste bottle. Each module can run off one set of wash botttles, or multiple modules can run off the same set of wash bottles. Hybridization samples are injected through a septum port at the lower end of the chamber (Fig 59).

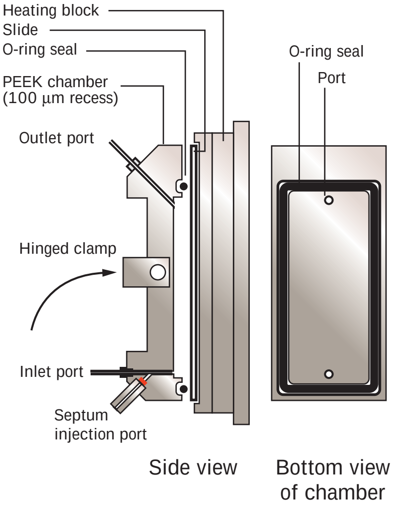


Fig 59. Schematic of individual slide chamber.

9.2 Benefits of Lucidea SlidePro Hybridizer

The key benefits of Lucidea SlidePro include:

- Improved uniformity of signals and Cy3/Cy5 ratios within and between slides.
- Temperature and mixing controls for each chamber. Small volumes are drawn in and out of the chamber to provide continuous mixing. The volume, speed, and length of mixing can also be individually controlled, making optimization easier.
- Temperature and wash solution controls for each chamber. This improves the reproducibility of signals both within and between slides and from user to user. Greater reproducibility can increase the accuracy of results with fewer numbers of replicates.
- Enhanced detection of rare messages. The mixing of probe during hybridization results in enhanced signals, without the need to use increased amounts of probe as compared with manual hybridization. Increased signal strength compared to background enables detection of low signals from rare messages.
- Rapid start-up time and ease of use. Experiments to determine optimal hybridization parameters, such as temperature and washing, are performed with ease. Standard protocols are provided to decrease time to optimize experimental procedures. Different conditions can be tested within a single run, using up to 30 slides with five modules. The software has a help feature for fast start-up and troubleshooting.
- Facilitates reuse of probe. Probe can be removed from the hybridization chamber via the injection port, allowing samples to be reused multiple times. The instrument is paused after hybridization and the probe removed with a syringe. The instrument continues with washing and drying of slides. While overall signals decrease with probe reuse, the Cy3/Cy5 ratios are not significantly altered.
- Reduced demands on user time. The user needs only to load slides and inject the probe. Following automated hybridization, washing, and drying, the slides are removed from the instrument ready to scan.

9.3 Validation in microarray hybridization

We describe here how a number of hybridization factors—including time, probe reuse and probe concentration—affect the signal-to-noise values detected from microarray experiments. Despite the effects of these factors on overall signal strength, the Cy3/Cy5 ratio remains constant, suggesting that differential expression may be determined under conditions which do not provide an optimal signal. Three experimental applications were used to demonstrate the utility of Lucidea SlidePro.

9.3.1 Comparison of automated and manual hybridization

Twenty-four standard silanized glass slides were spotted with p53 cDNA. Twelve slides were hybridized manually (approximately 30 µl of hybridization sample was placed on a microarray slide, under a coverslip and incubated in a humidified container in a hybridization oven) and twelve were hybridized in Lucidea SlidePro. All were hybridized using Cy3-labelled human skeletal muscle and Cy5-labelled human skeletal muscle.

Hybridization in Lucidea SlidePro produced increased signal intensity and more consistent Cy3/Cy5 ratios with very low variability compared to the manual method. Analysis of variance (ANOVA) comparison of the Cy3/Cy5 ratios showed significantly less variation in the Lucidea SlidePro processed slides compared to manually processed slides (Fig 60).

Coefficient of variation		
Type 5	Lucidea SlidePro	Manual
Within slide	8.6%	35.9%
Between slides	4.1%	10.7%
Total	12.7%	46.6%

Fig 60. Lucidea SlidePro vs manual ANOVA coefficient of variation values for glass slides hybridized in Lucidea SlidePro or manually.

Results suggest that hybridization efficiency and data reproducibility could be improved using Lucidea SlidePro as compared to the manual methodology (Fig 61-62).

9.3.2 The effect of probe mixing

Since diffusion rates on solid surfaces are much lower than those in solution, compensation for localized depletion of probe may not occur within the time frame of a hybridization (37, 56). Lucidea SlidePro provides mixing during hybridization, which ensures a constant probe concentration and thereby eliminates depletion effects.

Initial experiments were designed to determine whether mixing could enhance hybridization efficiency. The relative signals of a serial dilution of target hybridized under static or mixing conditions were assessed (Fig 63). Despite greater than 300-fold dilution of probe, mixing during hybridization enhanced the signal detected around 5-fold at the highest concentration of target. This result suggests that localized probe depletion effects could be reduced by mixing the sample.

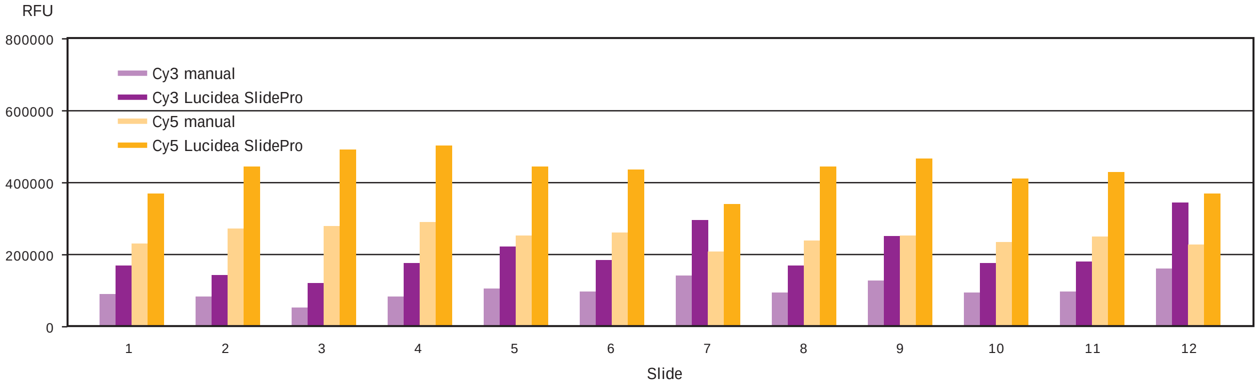


Fig 61. Lucidea SlidePro vs manual mean signal intensities.

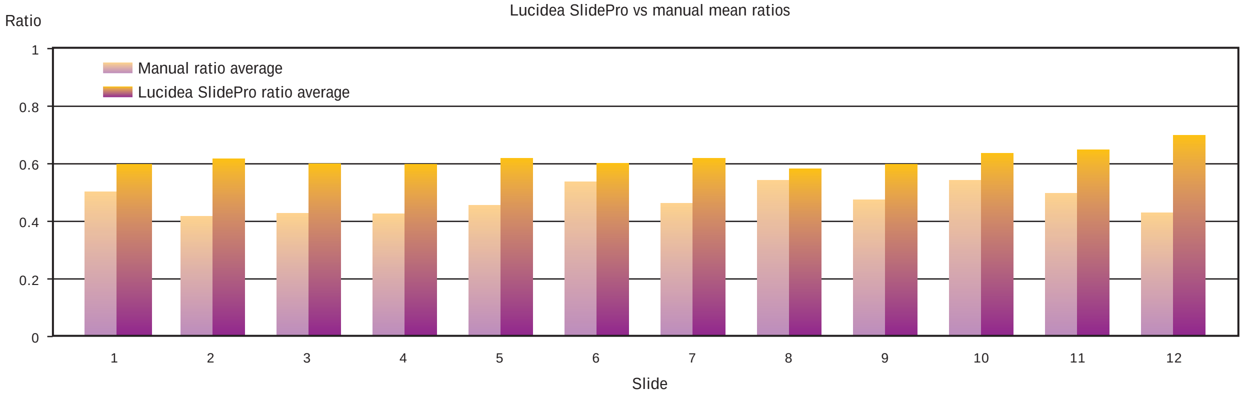


Fig 62. Lucidea SlidePro vs manual mean Cy3/Cy5 ratios.

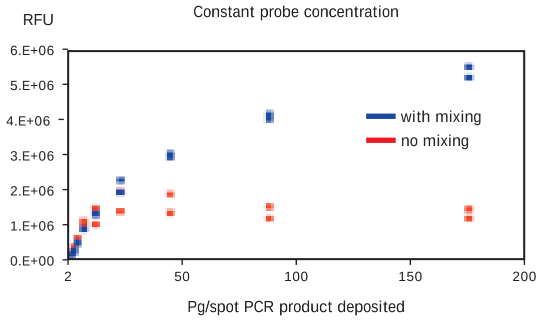
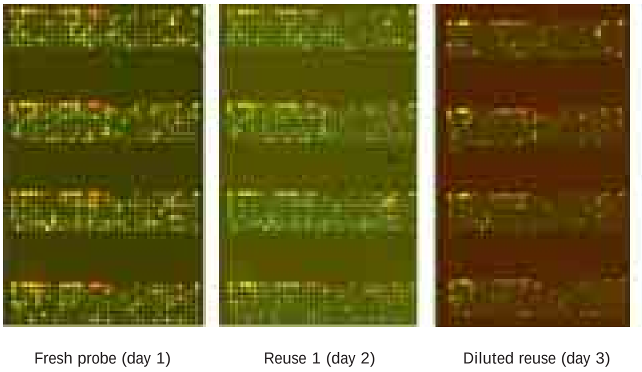


Fig 63. The effect of mixing on hybridization signal.

9.3.3 Reuse of probe

Since probe is often generated from mRNA samples that are in short supply, it may be desirable to use a labelled probe for multiple rounds of hybridization. This is possible using Lucidea SlidePro, since probe can be removed directly through the injection port following hybridization. The effect of probe reuse on overall signal strength and Cy3/Cy5 ratios was assessed. Duplicate reflective slides were hybridized in Lucidea SlidePro with 200 µl of Cy3-labelled skeletal muscle and Cy5-labelled placenta cDNA (40 pmol total) per chamber. Following hybridization and immediately prior to washing the slides, probe was recovered from all chambers using a syringe/needle through the injection port. Approximately 60% of the injection volume (120 µl) was recovered from each chamber. Each probe was then reconstituted to the original volume of 200 µl in 1× version 2 hybridization buffer and reinjected into chambers containing fresh slides. This constituted the first reuse of probe. The procedure was repeated for the second reuse.

Although the signal strength decreased over multiple hybridizations (Fig 64), the Cy3/Cy5 ratios were relatively unaffected (Fig 65). This suggests that the individual Cy3/Cy5 ratios for each gene should remain constant, provided the signal is strong enough to be detected above background. Furthermore, overall background was reduced with multiple rounds of hybridization, providing an additional benefit.


Fresh probe (day 1)

Reuse 1 (day 2)

Diluted reuse (day 3)

Fig 64. Reuse of probe. Scanned images showing arrays following overnight hybridization in Lucidea SlidePro with fresh probe, first r euse, and second reuse.

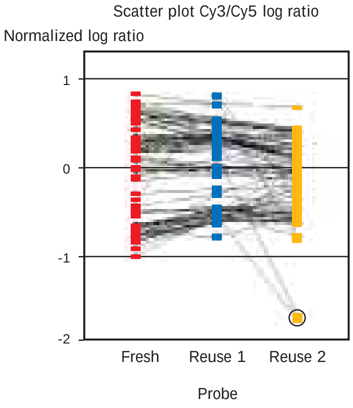


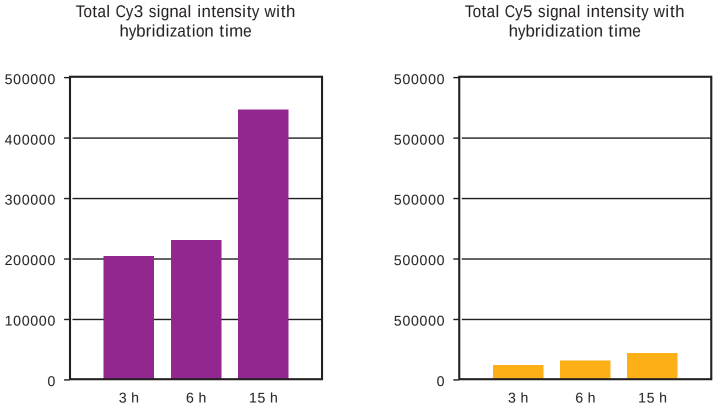
Fig 65. Reuse of probe. Cy3/Cy5 ratios of selected genes following hybridization with fresh probe, reuse 1, and reuse 2.

9.3.4 Length of hybridization

Standard microarray protocols call for overnight hybridization (12–18 h). Time course studies suggest that effective hybridization can take much longer than this (data not shown). In order to assess whether Lucidea SlidePro enhances the kinetics of hybridization, the relative signal strength following different hybridization times was investigated. Arrayed reflective slides were hybridized with 20 pmol each of Cy3 skeletal muscle and Cy5 placenta probe per slide. Total mean signal intensities (Fig 66) increased with time up to 16 h hybridization, but Cy3/Cy5 ratios (Fig 67) remained consistent between the 3-h, 6-h, and 15-h timepoints.

Another set of microarray slides were spotted with Lucidea Universal ScoreCard dynamic range controls (control elements that are used to evaluate the dynamic range and sensitivity of the system), and the amount of probe was increased to 60 pmol labelled probe per slide. Slides were hybridized for 3, 6, 9, 12, and 15 h in Lucidea SlidePro. Mean signal intensities were similar (Fig 68) for all hybridization times with dynamic ranges representing 2000 copies (DR 2), 200 copies (DR 3), and 20 copies (DR 5). These results suggest that shorter hybridization times may be used for higher throughput, provided that sufficient signal is obtained to detect low expressing genes of interest. Furthermore, increased probe concentrations may be used to decrease the hybridization time required for signal detection provided that the mean signal above background is not compromised.

Fig 66. Effect of hybridization time. Total mean Cy3 and Cy5 signals following 3-h, 6-h, and 15-h hybridization in Lucidea SlidePro.



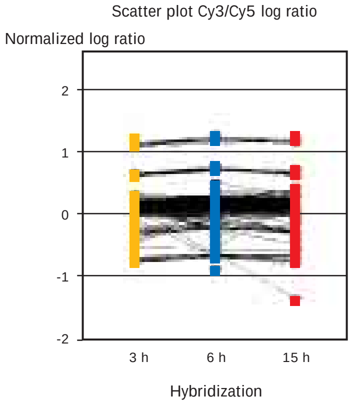


Fig 67. Effect of hybridization time. Cy3/Cy5 ratios of selected genes following 3-h, 6-h, and 15-h hybridization in Lucidea SlidePro.

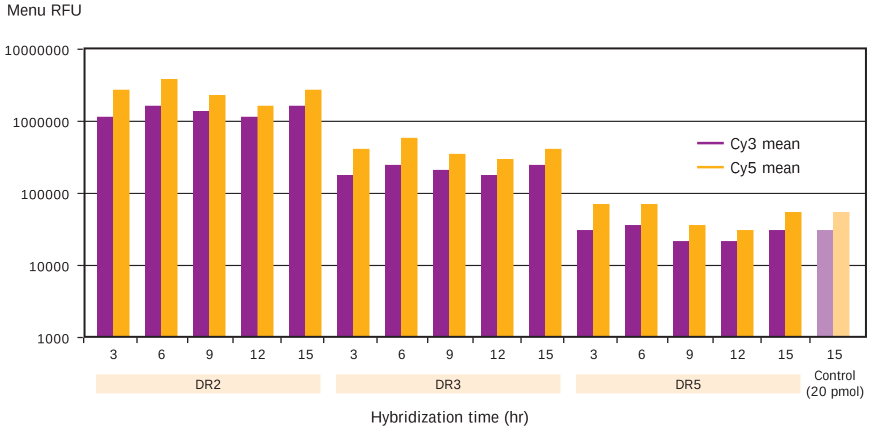
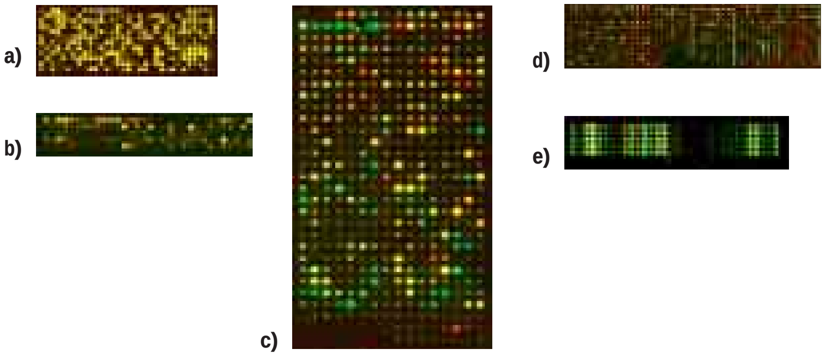


Fig 68. Effect of hybridization time. Total mean Cy3 and Cy5 signals of dynamic range controls.

9.4 Using different types of microarray slides

The dimensions of the slide chamber are specific and only standard microscope slides—25–25.5 mm × 75.5–76.0 mm, and 0.95–1.15 mm thick—may be used with Lucidea SlidePro. The hybridization chamber will fit arrays of up to 20 × 59 mm, allowing for a barcode on one end of the slide. The array should be spotted at least 2.5 mm from the edge. It is important that the bar code does not exceed 10 mm in width, or it will lie under the O-ring seal and cause leakage.

Lucidea SlidePro has been optimized for use with Amersham Biosciences version 2 hybridization buffer and microarray slides. Lucidea SlidePro can also be used to hybridize other manufacturer's slides (Fig 69). These protocols can be found on the Amersham Biosciences web site (www.amershambiosciences.com).



T28 direct labeled
Cy3 heart / Cy5 muscle

Fig 69. Lucidea SlidePro hybridization with various commercially available slides.

Chapter 10

FLUORESCENCE IMAGING SYSTEMS IN MICROARRAY ANALYSIS

10.0 Introduction

All fluorescence imaging systems require the following key elements (Fig 70):

- Excitation source
- Light delivery optics
- Light collection optics
- Filtration of the emitted light
- Detection, amplification and digitization of the emitted light

In this chapter, various types of scanner systems are discussed. Their light delivery and light collection mechanisms, signal detection and amplification, and overall performance are detailed, as well as criteria for selecting appropriate fluorochromes and filters for use with the scanner.

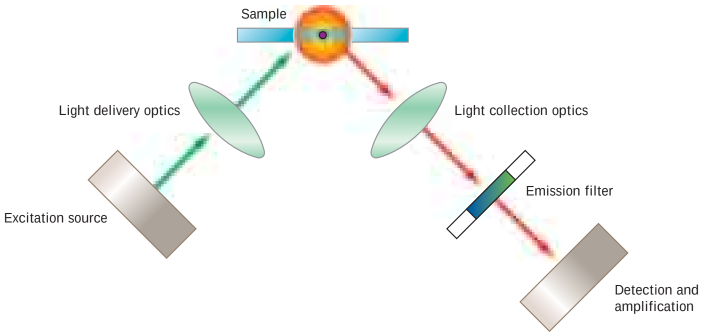


Fig 70. Components of a general fluorescence imaging system.

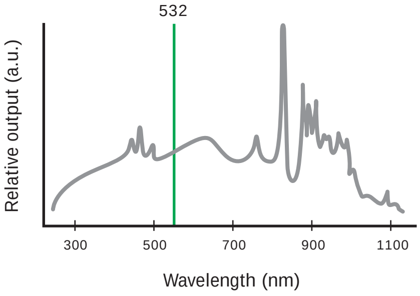


Fig 71. Spectral output of light from a xenon lamp and Nd:YAG laser. The "relative output" axis is scaled arbitrarily for the two light sources. The 532-nm line of the Nd:YAG laser is shown in green.

10.1 Requirements of a fluorescence imaging system

10.1.1 Excitation sources and light delivery optics

Light energy is essential to fluorescence. Light sources fall into two broad categories—wide-area, broad-wavelength sources, such as UV and xenon arc lamps, and line sources with discrete wavelengths, such as lasers (Fig 71). Broad-wavelength excitation sources are used in fluorescence spectrometers and camera imaging systems. Although the spectral output of a lamp is broad, it can be tuned to a narrow band of excitation light with the use of gratings or filters. In contrast, lasers deliver a narrow beam of collimated light that is predominantly monochromatic. In most camera systems, excitation light is delivered to the sample by direct illumination of the imaging field, with the excitation source positioned either above, below, or to the side of the sample.

Laser-based imaging systems, on the other hand, use more sophisticated optical paths, comprising mirrors and lenses, to direct the excitation beam to the sample. Some filtering of the laser light may also be required before the excitation beam is directed to the sample. For microarray applications, laser-based instruments are substantially favored, therefore CCD scanners will not be discussed.

10.1.2 Light collection optics

High-quality optical elements, such as lenses, mirrors, and filters, are integral components of any efficient imaging system. Optical filters are typically made from laminates of multiple glass elements. Filters can be coated to selectively absorb or reflect different wavelengths of light, thus creating the best combination of wavelength selection, linearity, and transmission properties.

10.1.3 Filtration of the emitted light

Although emitted fluorescent light radiates from a fluorochrome in all directions, it is typically collected from only a relatively small cone angle on one side of the sample. For this reason, light collection optics must be as efficient as possible. Any laser light that is reflected or scattered by the sample must be rejected from the collection pathway by a series of optical filters. Emitted light can also be filtered to select only the range or band of wavelengths that is of interest to the user. Systems that employ more than one detector require additional beam splitter filters to separate and direct the emitted light along separate paths to the individual detectors.

10.1.4 Detection, amplification, and digitization

For detection and quantification of emitted light, either a photomultiplier tube (PMT) or a charge-coupled device (CCD) can be used. In both cases, photon energy from emitted fluorescent light is converted into electrical energy, thereby producing a measurable signal that is proportional to the number of photons detected. After the emitted light is detected and amplified, the analog signal from a PMT or CCD detector is converted to a digital signal. The process of digitization turns a measured continuous analog signal into discrete numbers representing intensity levels. The number of intensity levels available is based on the digital resolution of the instrument, which is usually given as a number of bits, which increases exponentially by two. 8-bit, 12-bit, and 16-bit digital files correspond to the number of intensity levels allocated within that image file (256, 4096 and 65 536, respectively).

Digital resolution defines the ability to resolve two signals with similar intensities. Since only a limited number of intensity levels are available, it is unavoidable that this conversion process introduces a certain amount of error. To allow ample discrimination between similar signals and to keep the error as low as possible, the distribution of the available intensity levels should correspond well to the linear dynamic range of a detector. There are two methods of distributing intensity levels. A linear (even) distribution has the same spacing for all the intensity levels, allowing measurement across the dynamic range with the same absolute accuracy. However, relative digitization error increases as signals become smaller. A non-linear distribution (e.g. logarithmic or square root functions) divides the lower end of the signal range into more levels while combining the high end signals into fewer intensity levels. Thus, the absolute accuracy decreases with higher signals, but the relative digitization error remains more constant across the dynamic range.

10.2 Scanner systems

10.2.1 Excitation sources

Most fluorescence scanner devices used in life science research employ laser light for excitation. A laser source produces a narrow beam of highly monochromatic, coherent, and collimated light. The combination of focused energy and narrow beam-width contributes to the excellent sensitivity and resolution possible with a laser scanner. The active medium of a laser—the material that is made to emit light—is commonly a solid state (glass, crystal), liquid, or gas (57). Gas lasers and solid-state lasers both provide a wide range of specific wavelength choices for different imaging needs. Other light sources used in imaging systems include light emitting diodes (LEDs), which are more compact and less expensive than lasers, but produce a wide-band, low-power output.

Lasers

There are several commonly used types of lasers.

- Argon ion lasers produce a variety of wavelengths including 457 nm, 488 nm and 514 nm that are useful for excitation of many common fluorochromes, such as fluorescein and Cy2.
- Helium neon or HeNe lasers, which generate a single wavelength of light (633 nm), are popular in many laser scanners, and can be used to excite Cy5.
- Neodymium:Yttrium Aluminum Garnet (Nd:YAG) solid-state lasers, when frequency-doubled, generate a strong line at 532 nm which can be used to excite Cy3.
- Diode lasers (or semiconductor diode lasers) are compact lasers. Because of their small size and light weight, these light sources can be integrated directly into the scanning mechanism of a fluorescence imager. Diode lasers are inexpensive and are generally limited to wavelengths above 635 nm.

Light Emitting Diodes (LEDs)

As an alternative to lasers, the LED produces an output with a much wider bandwidth (over 60 nm) and a wide range of power from low to moderate output. Because LED light emissions are doughnut-shaped, and not collimated, the source must be mounted very close to the sample using lenses to tightly focus the light. LEDs are considerably smaller, lighter, and less expensive than lasers. They are available in the visible wavelength range above 430 nm.

10.3 Excitation light delivery

Because light from a laser is well-collimated and of sufficient power, delivery of excitation light to the sample is relatively straightforward, with only negligible losses incurred during the process. For lasers that produce multiple wavelengths of light, the desired line(s) can be selected by using filters that exclude unwanted wavelengths, while allowing the selected line to pass at a very high transmission percentage. Excitation filters are also necessary with single-line lasers, as their output is not 100% pure. Optical lenses are used to align the laser beam, and mirrors can be used to redirect the beam within the instrument. One of the main considerations in delivering light using a laser scanning system is that the light source is a point, while the sample typically occupies a relatively large two-dimensional space. Effective sample coverage can be achieved by rapidly moving the excitation beam across the sample in two dimensions. There are two ways to move and spread the point source across the sample, which are discussed below.

10.3.1 Galvanometer-based systems

Galvanometer-based systems use a small, rapidly oscillating mirror to deflect the laser beam, effectively creating a line source (Fig 72). By using relatively simple optics, the beam can be deflected very quickly, resulting in a short scan time. Compared to confocal systems, galvanometer-based scanners are useful for imaging thick samples due to the ability to collect more fluorescent signal in the vertical dimension. However, since the

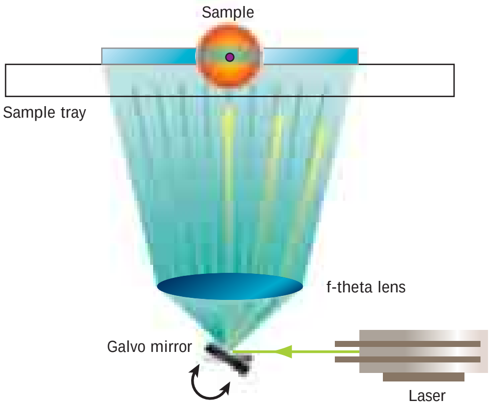


Fig 72. Galvanometer-controlled scanning mechanism. Light is emitted from the laser in a single, straight line. The galvanometer mirror moves rapidly back and forth redirecting the laser beam and illuminating the sample across its entire width (X-axis). The f-theta lens reduces the angle of the excitation beam delivered to the sample. The entire sample is illuminated either by the galvanometer mechanism moving along the length of the sample (Y-axis) or the sample moving relative to the scanning mechanism.

excitation beam does not illuminate the sample from the same angle in every position, a parallax effect can result. The term parallax here refers to the shift in apparent position of targets, predominately at the outer boundaries of the scan area. Additionally, the arc of excitation light created by the galvanometer mirror produces some variations in the effective excitation energy reaching the sample at different points across the arc. These effects can be minimized with an f-theta lens, but when the angle of incident excitation light varies over the imaging field, some spatial distortion can still occur in the resulting image.

10.3.2 Moving-head scanners

Moving-head scanners use an optical mechanism that is equidistant from the sample. This means that the angle and path length of the excitation beam is identical at any point on the sample (Fig 73). This eliminates variations in power density and spatial distortion common with galvanometer-based systems. Although scan times are longer with a moving-head design, the benefits of uniformity in both light delivery and collection of fluorescence are indispensable for accurate signal quantification. For microarray scanners an alternative method is to move the stage that contains the microarray slide. In some scanners the stage is moved in one direction while the galvanometer moves the laser beam across in the second dimension.

10.4 Light collection

The light collection optics in a scanner system must be designed to efficiently collect as much of the emitted fluorescent light as possible. Laser light that is reflected or scattered by the sample is generally

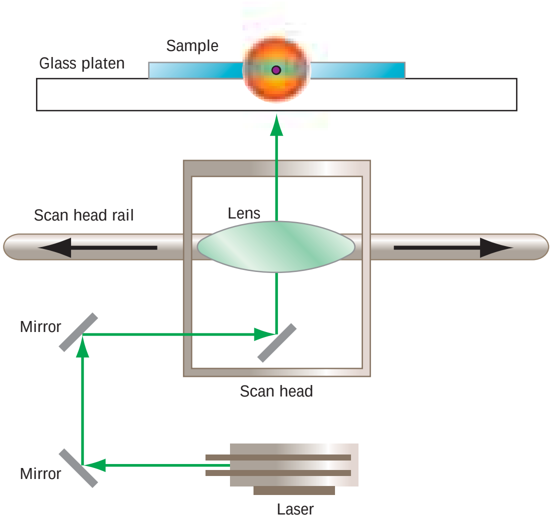


Fig 73. Moving-head scanning mechanism. The light beam from the laser is folded by a series of mirrors and ultimately reflected onto the sample. The sample is illuminated across its width as the scan head moves along the scan head rail (X-axis). The entire sample is illuminated by the scan head, laser, and mirrors tracking along the length of the sample (Y-axis).

rejected from the collection pathway by a laser-blocking filter, the design of which is to exclude the light produced by the laser source, while passing all other emitted light. Light collection schemes vary depending on the nature of the excitation system. With galvanometer systems, the emitted fluorescence must be gathered in a wide line across the sample. This is usually achieved with a linear lens (fiber bundle or light bar), positioned beneath the sample, that tracks with the excitation line, collecting fluorescence independently at each pixel. Although this system is effective, it can produce image artifacts. At the edges of the scan area where the angle of the excitation beam, relative to the sample, is farthest from perpendicular, some spatial distortion may occur. Where very high signal levels are present, stimulation of fluorescence from sample areas that are adjacent to the pixel under investigation can result in an inaccurate signal measurement from that pixel, an artifact known as flaring or blooming.

With moving-head systems, emitted light is collected directly below the point of sample excitation. Again, it is important to collect as much of the emitted light as possible to maintain high sensitivity. This can be achieved by using large collection lenses, or lenses with large numerical apertures (NA). Since the NA is directly related to the full angle of the cone of light rays that a lens can collect, the higher the NA, the greater the signal resolution and brightness (58). Moving-head designs can also include confocal optical elements that detect light from only a narrow vertical plane in the sample. This improves sensitivity by focusing and collecting emission light from the point of interest while reducing the background signal and noise from out-of-focus regions in the sample (Fig 74). Additionally, the parallel motion of moving head designs removes other artifacts associated with galvanometer-based systems, such as spatial distortion and the flaring or blooming associated with high activity samples.

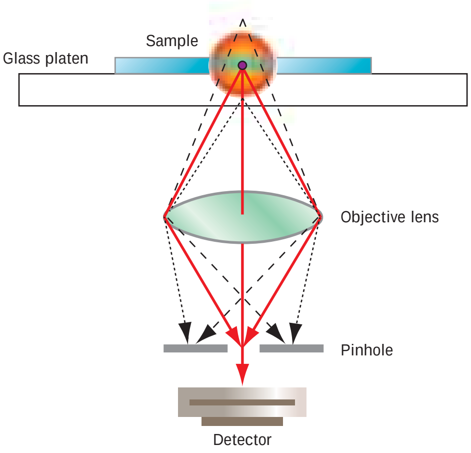


Fig 74. Illustration of confocal optics. Fluorescence from the sample is collected by an objective lens and directed toward a pinhole aperture. The pinhole allows the emitted light from a narrow focal plane (red solid lines) to pass to the detector, while blocking most of the out-of-focus light (black dashed lines).

10.5 Signal detection and amplification

The first stage in fluorescent signal detection is selection of only the desired emission wavelengths from the label or dye. In single-channel or single-label experiments, emission filters are designed to allow only a well-defined spectrum of emitted light to reach the detector. Any remaining stray excitation or scattered light is rejected. Because the intensity of the laser light is many orders of magnitude greater than the emitted light, even a small fraction of laser light reaching the detector will significantly increase background. Filtration is also used to reduce background fluorescence or inherent autofluorescence originating from either the sample itself or the sample matrix gel, membrane, or microplate. In multichannel or multi-label experiments using instrumentation with dual detectors, additional filtering is required upstream of the previously described emission filter. During the initial stage of collection in these experiments, fluorescence from two different labels within the same sample is collected simultaneously as a mixed signal. A dichroic beam splitter must be included to spectrally resolve the contribution from each label and then direct the light to appropriate emission filters (Fig 75). At a specified wavelength, the beam splitter partitions the incident fluorescent light beam into two beams, passing one and reflecting the other. The reflected light creates a second channel that is filtered independently and detected by a separate detector. In this way, the fluorescent signal from each label is determined accurately in both spatial and quantitative terms.

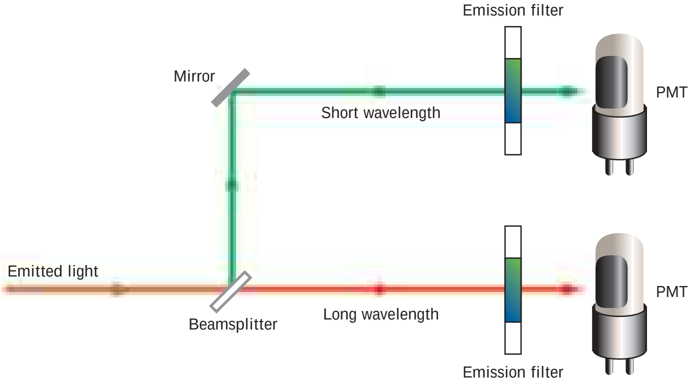


Fig 75. Use of a beam splitter or dichroic filter with two separate PMTs. Light from a dual color sample enters the emission optics as a combination of wavelengths. A dichroic beam splitter distinguishes light on the basis of wavelength. Wavelengths above the beam splitter range pass through, those below are reflected. In this way two channels are created. These two channels can then be filtered and detected independently.

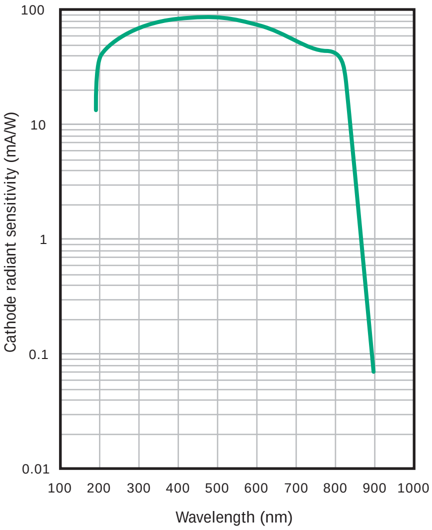


Fig 76. An example of the response of a PMT versus wavelength.

After the fluorescent emission has been filtered and only the desired wavelengths remain, the light is detected and quantified. Because the intensity of light at this stage is very small, a PMT must be used to detect it. In the PMT, photons of light hit a photocathode and are converted into electrons which are then accelerated in a voltage gradient and multiplied from 10^6 to 10^7 times. This produces a measurable electrical signal that is proportional to the number of photons detected. The response of a PMT is typically useful over a wavelength range of 300–800 nm (Fig 76). High-performance PMTs extend this range to 200–900 nm.

10.6 System performance

The performance of a laser scanner system is described in terms of system resolution, linearity, uniformity, and sensitivity.

Resolution can be defined in terms of both spatial and amplitude resolution. Spatial resolution of an instrument refers to its ability to distinguish between two very closely positioned objects. It is a function of the diameter of the light beam when it reaches the sample and the distance between adjacent measurements. Spatial resolution is dependent on, but not equivalent to, the pixel size of the image. Spatial resolution improves as pixel size is reduced. Systems with higher spatial resolution can not only detect smaller objects, but can also discriminate more accurately between closely spaced targets. However, an image with a 100-µm pixel size will not have a spatial resolution of 100 µm. The pixel size refers to the collection sampling interval of the image. According to a fundamental sampling principle, the Nyquist Criterion, the smallest resolvable object in an image is no better than twice the sampling interval (59). Thus, to resolve a 100-µm sample, the sampling interval must be at most 50 µm. Amplitude resolution, or gray-level quantification, describes the minimum difference that is distinguishable between levels of light intensity (or fluorescence) detected from the sample (60). For example, an imaging system with 16-bit digitization can resolve and accurately quantify 65 536 different values of light intensity from a fluorescent sample.

Linearity of a laser scanner is the signal range over which the instrument yields a linear response to fluorochrome concentration and is therefore useful for accurate quantification. The linear dynamic range can be defined in at least 3 ways:

1) the electronic dynamic range of the scanner

2) the chemical dynamic range of the fluorescent dyes used

3) the biological dynamic range of the system under study

A scanning system with a wide dynamic range can detect and accurately quantify signals from both very low- and very high-intensity targets in the same scan. The linear dynamic range of most laser scanner instruments is between 10^4 and 10^5.

Uniformity across the entire scan area is critical for reliable quantitation. A given fluorescent signal should yield the same measurement at any position within the imaging field. Moving-head scanners, in particular, deliver flat-field illumination and uniform collection of fluorescent emissions across the entire scan area.

Detection limit is the minimum amount of sample that can be detected by an instrument at a known confidence level. From an economical standpoint, instruments with better detection limits are more cost-effective because they require less fluorescent sample for analysis.

10.7 Fluorochrome and Filter Selection

To generate fluorescence, excitation light delivered to the sample must be within the absorption spectrum of the fluorochrome. Generally, the closer the excitation wavelength is to the peak absorption wavelength of the fluorochrome, the greater the excitation efficiency. Appropriate filters are usually built into scanner instruments for laser line selection and elimination of unwanted background light. Fixed or interchangeable optical filters that are suitable for the emission profile of the fluorochromes are then used to refine the emitted fluorescence, such that only the desired wavelengths are passed to the detector. Matching a fluorochrome label with a suitable excitation source and emission filter is the key to optimal detection efficiency.

10.7.1 Types of emission filters

The composition of emission filters used in fluorescence scanners and cameras ranges from simple colored glass to glass laminates coated with thin interference films. Coated interference filters generally deliver excellent performance through their selective reflection and transmission effects. Three types of optical emission filters are commonly used.

Long-pass (LP) filters pass light that is longer than a specified wavelength and reject all shorter wavelengths. A good quality long-pass filter is characterized by a steep transition between rejected and transmitted wavelengths (Fig 77a). Long-pass filters are named for the wavelength at the midpoint of the transition between the rejected and transmitted light (cutoff point). For example, the cutoff point in the transmission spectrum of a 560 LP filter is 560 nm, where 50% of the maximum transmittance is rejected. The name of a long-pass filter may also include other designations, such as OG (orange glass), RG (red glass), E (emission), LP (long-pass), or EFLP (edge filter long-pass). OG and RG are colored glass absorption filters, whereas E, LP, and EFLP filters are coated interference filters. Colored glass filters are less expensive and have more gradual transition slopes than coated interference filters.

Short-pass (SP) filters reject wavelengths that are longer than a specified value and pass shorter wavelengths. Like long-pass filters, short-pass filters are named according to their cutoff point. For example, a 526 SP filter rejects 50% of the maximum transmittance at 526 nm (Fig 77b).

Band-pass (BP) filters allow a band of selected wavelengths to pass through, while rejecting all shorter and longer wavelengths. Band-pass filters provides very sharp cutoffs with very little transmission of the rejected wavelengths. High-performance band-pass filters are also referred to as Discriminating Filters (DF). The name of a band-pass filter is typically made up of two parts:

- the wavelength of the band center (the 670 BP 30 filter passes a band of light centered at 670 nm [Fig 78]);
- the full-width at half-maximum transmission (FWHM) (the 670 BP 30 filter passes light over a wavelength range of 30 nm [655–685 nm] with an efficiency equal to or greater than half the maximum transmittance of the filter).

Band-pass filters with an FWHM of 20–30 nm are optimal for most fluorescence applications, including multi-label experiments. Filters with FWHMs greater than 30 nm allow collection of light at more wavelengths and give a higher total signal; however, they are less able to discriminate between closely spaced, overlapping emission spectra in multichannel experiments. Filters with FWHMs narrower than 20 nm transmit less signal and are most useful with fluorochromes with very narrow emission spectra.

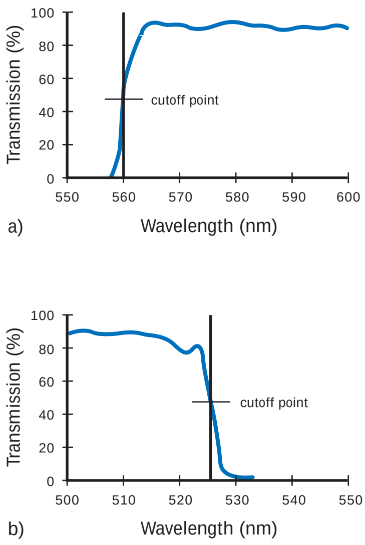


Fig 77. Transmission profiles for a (a) 560-nm long-pass and a (b) 526-nm short-pass filter. The cutoff points are noted.

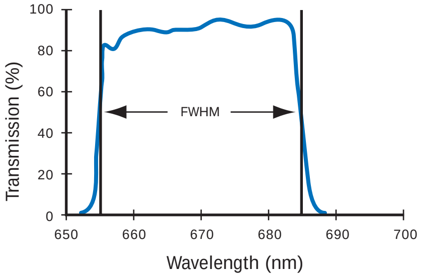


Fig 78. Transmission profile for a band-pass (670 BP 30) filter. The full-width at half-maximum (FWHM) transmission of 30 nm is indicated by the arrows.

10.8 Using emission filters to improve sensitivity and linearity range

When selectable emission filters are available in an imaging system, filter choice will influence the sensitivity and dynamic range of an assay. In general, if image background signal is high, adding an interchangeable filter may improve the sensitivity and dynamic range of the assay. The background signal from some matrices (gels and membranes) has a broad, relatively flat spectrum. In such cases, a band-pass filter can remove the portion of the background signal comprising wavelengths that are longer or shorter than the fluorochrome emissions. By selecting a filter that transmits a band at or near the emission peak of the fluorochrome of interest, the background signal is typically reduced with only slight attenuation of the signal from the fluorochrome. Therefore, the use of an appropriate band-pass filter should improve the overall signal-to-noise ratio (S/N).

To determine if a filter is needed, scans should be performed with and without the filter while other conditions remain constant. The resulting S/N values should then be compared to determine the more efficient configuration. Interchangeable filters can also be used in fluorescence scanners to attenuate the sample signal itself so that it falls within the linear range of the system. Although scanning the sample at a reduced PMT voltage can attenuate the signal, the response of the PMT may not be linear if the voltage is set below the instrument manufacturer's recommendation. If further attenuation is necessary to prevent saturation of the PMT, the addition of an appropriate emission filter can decrease the signal reaching the detector.

10.9 General guidelines for selecting fluorochromes and filters

10.9.1 Single-color imaging

Excitation efficiency is usually highest when the fluorochrome's absorption maximum correlates closely with the excitation wavelength of the imaging system. However, the absorption profiles of most fluorochromes are rather broad, and some fluorochromes have a second (or additional) absorption peak or a long "tail" in their spectra. It is not mandatory that the fluorochrome's major absorption peak matches exactly the available excitation wavelength for efficient excitation. For example, the absorption maxima of the fluorescein and Cy3 fluorochromes are 490 nm and 552 nm respectively (Fig 79). Excitation of either dye using the 532-nm wavelength line of the Nd:YAG laser may seem to be inefficient, since the laser produces light that is 40 nm above the absorption peak of fluorescein and 20 nm below that of Cy3. In practice, however, delivery of a high level of excitation energy at 532 nm does efficiently excite both fluorochromes. For emission, selecting a filter that transmits a band at or near the emission peak of the fluorochrome generally improves the sensitivity and linear range of the measurement. Figure 80 shows collection of Cy3 fluorescence using either a 580 BP 30 or a 560 LP emission filter.

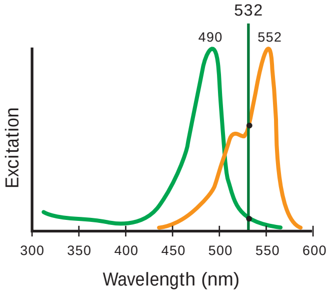


Fig 79. Excitation of fluorescein (green) and Cy3 (orange) using 532-nm laser light. The absorption spectra of Cy3 and fluorescein are overlaid with the 532-nm wavelength line of the Nd:YAG laser.

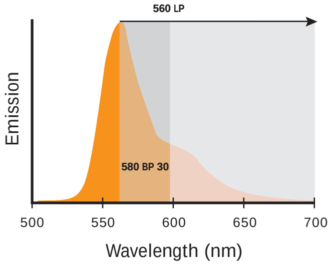


Fig 80. Filtering of Cy3 fluorescence using either a 580 BP 30 (dark gray area) or a 560 LP filter (light and dark gray areas).

10.9.2 Multicolor imaging

Multicolor imaging allows detection and resolution of multiple targets using fluorescent labels with different spectral properties. The ability to multiplex or detect multiple labels in the same experiment is both time and cost-effective and improves accuracy for some assays. Analysis using a single label can require a set of experiments or many repetitions of the same experiment to generate one set of data. For example, single-label analysis of gene expression from two different tissues requires two separate hybridization to different gene arrays, or consecutive hybridization to the same array with stripping and re-probing. With a dual-label approach, however, the DNA probes from the two tissue types are labelled with different fluorochromes and used simultaneously with the same gene array. In this way, experimental error is reduced because only one array is used, and hybridization conditions for the two probes are identical. Additionally, by using a two-channel scan, expression data is rapidly collected from both tissues, thus streamlining analysis.

The process for multicolor image acquisition varies depending on the imaging system. An imager with a single detector acquires consecutive images using different emission filters and, in some cases, different excitation light. When two detectors are available, the combined or mixed fluorescence from two different labels is collected at the same time and then resolved by filtering before the signal reaches the detectors. Implementation of dual detection requires a beam splitter filter to spectrally split the mixed fluorescent signal, directing the resulting two emission beams to separate emission filters (optimal for each fluorochrome), and finally to the detectors. A beam splitter, or dichroic reflector, is specified to function as either a short-pass or long-pass filter relative to the desired transition wavelength. For example, a beam splitter that reflects light shorter than the transition wavelength and passes longer wavelengths is effectively acting as a long-pass filter (Fig 75).

10.9.3 Fluorochrome selection in multicolor experiments

When designing multicolor experiments, two key elements must be considered: the fluorochromes used and the emission filters available. As with any fluorescence experiment, the excitation wavelength of the scanner must fall within the absorption spectrum of the fluorochromes used. Additionally, the emission spectra of different fluorochromes selected for an experiment should be relatively well resolved from each other. However, some spectral overlap between emission profiles is almost unavoidable. To minimize cross-contamination, fluorochromes with well-separated emission peaks should be chosen along with emission filters that allow reasonable spectral discrimination between the fluorochrome emission profiles. Figure 86 shows the emission overlap between two common fluorochromes and the use of band-pass filters to discriminate the spectra. For best results, fluorochromes with emission peaks at least 30 nm apart should be chosen. A fluorescence scanner is most useful for multicolor experiments when it provides selectable emission filters suitable for a variety of labels. A range of narrow band-pass filters that match the peak emission wavelengths of commonly used fluorochrome labels will address most multicolor imaging needs.

10.10 Amersham Biosciences imaging systems

Amersham Biosciences offers a variety of imaging instruments that are well suited for use in microarray analysis. For more information, please consult Amersham Biosciences web site at www.amershambiosciences.com.

Fig 81. Typhoon Variable Mode Imager.

10.10.1 Typhoon 9210: High performance laser scanning system

Excitation sources: 532-nm Nd:YAG and 633-nm HeNe lasers

Filters: 6 emission filters and 3 beamsplitters (up to 13 emission filter positions)

Detection: 2 high sensitivity PMTs

Imaging modes: 4 modes. 2 modes for fluorescence detection, chemiluminescence, storage phosphor

Scanning area: 35 × 43 cm

Maximum resolution: 10 µm

Sample types: microarrays, gel sandwiches, agarose and polyacrylamide gels, blots, microplates, TLC plates, and macroarrays

10.10.2 Typhoon 9410: High performance laser scanning system

Excitation sources: 532-nm Nd:YAG, 633-nm HeNe, and 457-nm and 488-nm Argon lasers

Filters: 7 emission filters and 3 beamsplitters (up to 13 emission filter positions)

Detection: 2 high sensitivity PMTs

Imaging modes: 5 modes. 3 modes for fluorescence detection, chemiluminescence, storage phosphor, chemifluorescence

Scanning area: 35 × 43 cm

Maximum resolution: 10 µm

Sample types: microarrays, gel sandwiches, agarose and polyacrylamide gels, blots, microplates, TLC plates, and macroarrays

Notes: Versatile fluorescence and radioactive imager that can scan microarrays but also contains an extra blue laser

Chapter 11

DESIGN, CONTROLS AND DATA ANALYSIS OF MICROARRAY EXPERIMENTS

11.0 Introduction

Methods for the analysis of microarray data are still evolving, and there is no standard experimental design or method of data analysis for microarray experiments at the present time. However, some efforts are being made to set a common annotation and standards for microarray data in order to create public databases for microarray results (61, 62, 63). Meanwhile, in this chapter some important considerations for analyzing microarray data are discussed.

11.1 Experimental design

Data analysis begins with experimental design. When planning a microarray experiment it is important to consider sources of variation within the experiment. These can arise from the samples reflecting differences in gene expression between individual animals or different tissue culture plates. Furthermore, time-dependent variation in gene expression levels resulting from circadian rhythms can also be a factor. Experimental variations may also occur due to variation within the experiment itself. In order to ensure that these experimental errors can be identified, slides should contain replicate spots of each target and replicate slides should be analyzed with pooled or multiple mRNA samples. This replication enables the use of statistical tools such as averages and standard deviations to monitor the extent of experimental variation (64).

Lucidea Universal ScoreCard has been developed by Amersham Biosciences to address the need for controls. It is a set of controls used to validate and normalize microarray experimental data. It is further described in section 6 of this chapter.

Another prudent measure is to perform reverse color or ‘flip-flop’ experiments. In these experiments the two mRNA samples being compared in a microarray experiment are labelled separately with both Cy3 and Cy5. Replica slides are hybridized with both combinations of probes. By comparing the signal ratios from the reversed slides, it is possible to identify data that is affected more by the labelling process or quality of mRNA than by changes in gene expression levels.

Once the mRNA extraction, labelling, hybridization, and scanning are complete, the final stage in the microarray experiment is data analysis. This is a complex multi-step process and is illustrated in Figure 82. The steps of data analysis are described in further detail in this chapter.

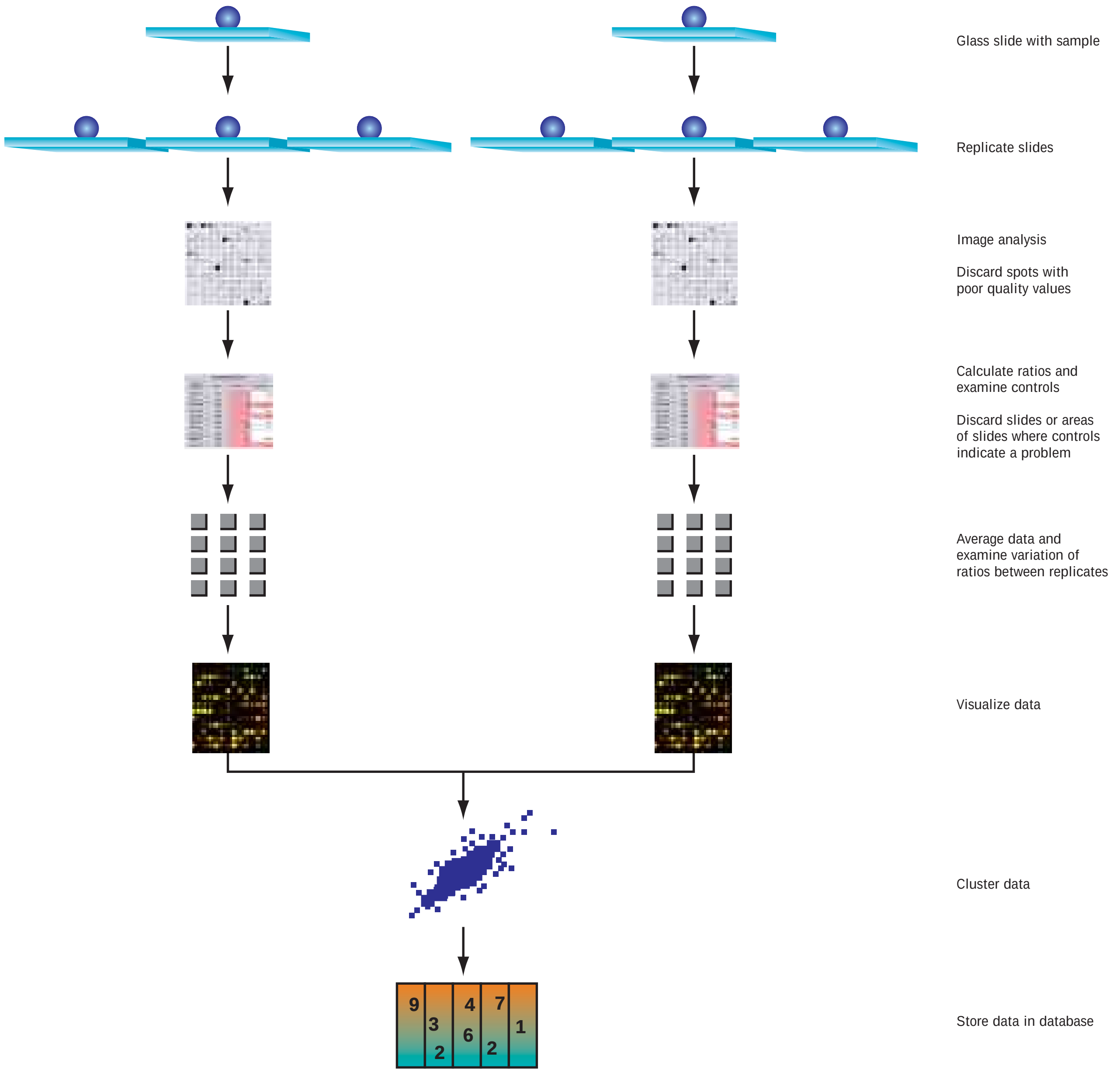

Glass slide with sample
Replicate slides
Image analysis
Discard spots with poor quality values
Calculate ratios and examine controls
Discard slides or areas of slides where controls indicate a problem
Average data and examine variation of ratios between replicates
Visualize data
Cluster data
Store data in database
9
3
2
4
6
7
2
1

Fig 82. Stages of microarray data analysis.

11.2 Overview of microarray data analysis

Microarray data analysis consists of four main steps:

- Image analysis
- Examination of controls
- Data normalization
- Visualization and clustering

Image analysis uses a dedicated software to quantitate the fluorescent intensity at each spot. Normally, this involves a process called spotfinding. The second step is to examine the controls on the arrays. Normalization is performed next, followed by calculation of mean ratios. The data can then be visualized in a graphical form, and clustered, such that meaningful trends can be found among data from multiple slides and experiments.

The amount of data obtained from microarray experiments is vast and can be generated very rapidly. However, it is important to know the quality of the data. There are three types of quality values that can be used. It is recommended that all three are used within a microarray experiment. The types of values are:

- A series of metrics reported by the image analysis software to ensure that the spots that have been quantitated appear to be good spots, for example, regularly shaped.
- A series of controls on your microarray to ensure the hybridization has occurred correctly. These will indicate how specific and efficient the hybridization has been.
- Analysis of the data from replicate targets. Replicates are critical for indicating how good the overall data is and whether the results obtained are statistically meaningful (64).

11.3 Image analysis

A scanned microarray image records the fluorescent intensities of all pixels in the image area, including pixels from within (signal) and outside (background) the DNA spots. The first step in the microarray workflow is to locate the spots. Consider the following objectives:

- accurately define the positions of every DNA spot in the image
- provide appropriate measurement of fluorescence intensity for each spot by quantifying the intensities of pixels within and outside the DNA spots
- provide quality metrics that give estimation of the accuracy of the intensity measurement

The first step, alternately called "gridding", can usually be performed using dedicated software, such as ArrayVision or Lucidea Automated Spotfinder.

The process of spotfinding begins by defining a grid, or an array of circles, that indicates the expected size of each spot, how far away they are spaced, and how they are arranged in an array, all regardless of the intensities of individual spots. This information can be measured in pixel or micron units. Once a grid is defined, it is overlaid onto the scanned image such that the circles are nearly exactly aligned with the spots on the microarray image. This spotfinding process can be automated using spotfinding software, which serves to eliminate the tedious task of manual alignment. In addition to the following descriptions, see the spotfinding software help guide for instructions how to most effectively use this tool.

- **Manual**: This method involves first dividing the grid into several subgrids, and then visually aligning each smaller subgrid with the corresponding area of the image by adjusting the position of circles.
- **Semi-automated**: This is where the software algorithm finds the spots, but some user intervention is required.
- **Automated**: This is where minimal user intervention is required. These software packages can automatically analyze multiple images while eliminating the need for supervision.

The next step in image analysis is to determine the signal present inside and outside the spot. The background signal is then subtracted from the spot signal to give background-corrected signal. Whereas spot signal is calculated from within the positioned grid, the background signal is determined by calculating the average pixel intensity in a user-defined region. Although mean or median background signal can be used, median values are more resistant to variation in background caused, for example, by fluorescent speckles. Some of the various regions in which the background can be calculated are illustrated in Figure 83. It is important to use a background correction method such that any pixels from the spot do not get included in the background. This can easily occur if the spots are close together.

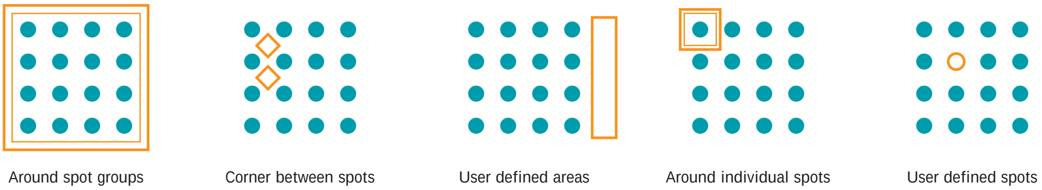


Fig 83. Some of the background region options available to the user in the ArrayVision™ Image Analysis Software. The green represents the spots enclosed by the grid while the blue encloses the background region. Image analysis quality metrics calculated by analysis software are increasingly used to highlight data that may be unreliable and should be omitted from further analysis. Typical causes of suspect data include dust speckles over spots, poor spot morphology, very low or very high signal.

11.4 Spotfinding software offered by Amersham Biosciences

11.4.1 Lucidea Automated Spotfinder

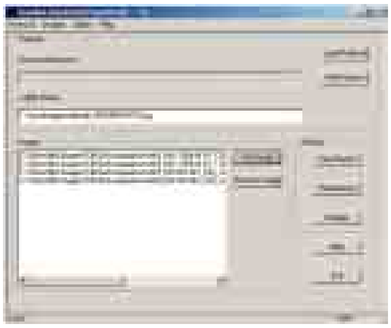


Fig 84. Lucidea Automated Spotfinder has a simple, intuitive user interface to initiate the automatic processing of microarray images. In this example, four images are loaded for analysis.

Lucidea Automated Spotfinder processes microarray images by performing spotfinding and data extraction in an automated fashion with virtually no manual intervention (Fig 84). The output from Lucidea Automated Spotfinder includes the signal intensity for each spot, plus quality metrics to assess individual spots as well as the overall image. The software is compatible with images produced by commercially available scanners. Several images can be analyzed at once in a batch mode, without manual inspection or image manipulation. Lucidea Automated Spotfinder features include:

- fully automated spot finding and data extraction
- multiple reporting options and data export
- user-defined templates for analyzing single or multiple images
- metrics for assessing data quality
- background subtraction

11.4.2 ArrayVision

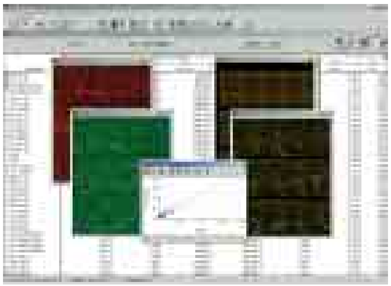


Fig 85. ArrayVision software provides automated analysis of radioisotopic or fluorescent macro- and microarrays.

ArrayVision software is a semi-automated software used for performing image analysis (Fig 85). Some of its features include:

- automated alignment of quantification grid over array
- choice of methods for background signal removal
- quality metrics
- reporting tools and data export
- visualization tools for viewing array images

11.5 Use of controls in microarray experiments

As with all experiments, microarrays should contain a series of controls to ensure that the data obtained from the arrays is accurate. Therefore, included on microarrays should be some cDNA or oligos which are expected to give a negative result, and some which should give a positive result. The types of controls that should be included are discussed below.

11.5.1 Negative controls

Negative controls are spotted DNA sequences that should not hybridize with any labelled probe. The negative controls used should ideally come from organisms that are only distantly related to the organism being studied in the experiment. For example with human microarrays, sequences from bacterial genes, intergenic regions, plant genes, or double-stranded poly-dA are often used for this purpose (25, 65, 66). Under optimal analysis conditions negative control spots should not give any signal at all. However, if the stringency of the hybridization is not high enough, non-specific hybridization between labelled cDNA molecules in the probe and unrelated target sequences on the array may take place, resulting in detectable signals from negative control spots. The higher the signal from negative controls, the less reliable is the data from the whole slide. These negative controls are particularly important to include if the spotted array consists of oligonucleotides because in these types of arrays, a lower hybridization stringency may be used. Negative controls can also be used to detect contamination between targets during spotting. Placing negative control targets after positive control targets that are always expected to give signal can do this.

11.5.2 Poly-adenylated DNA and CotI DNA

When using oligo(dT) to prime first-strand cDNA synthesis, it is possible that the oligo(dT) will prime within the poly-A tail of the mRNA. If this occurs there will be a string of dT bases within the probe. It is possible that the targets spotted may also contain a similar string of poly-A sequences, particularly if the targets were derived originally from an EST library which had been made by the use of oligo(dT) primer. In order to prevent cross reactivity of the poly-dT sequences within the probe with potential poly-dA sequences in the targets, a poly-dA oligo of 80 bases can be included in the hybridization to block the poly-dT (65). In order to ensure that this process has occurred correctly, it is good to include as a negative control, a poly-dA sequence spotted on the microarray. If these spots are negative this suggests that the blocking has occurred effectively.

Another sequence that may cause problems within the probe is derived from repetitive sequences such as Alu-repeat sequences. These sequences can be blocked by the inclusion of Cot-1 DNA in the hybridization. To ensure that this blocking has occurred correctly, the spotting of Cot-1 DNA as a negative control is recommended.

11.5.3 Positive controls

Labelled DNA

DNA can be labelled with CyDye fluors using polymerase chain reaction (PCR), or in the case of oligos, during the synthesis of the oligos (many oligo manufacturers offer this service). When the labelled DNA is spotted onto the array, the DNA will be fluorescent and serve as a useful positive control for verifying that the target DNA is binding effectively to the slide surface during the hybridization and washes. Total genomic DNA can also be used as a positive target. Positive controls placed on different locations of the slide can help in the spotfinding process by providing clearly detectable signals in known positions, regardless of the type of probe used.

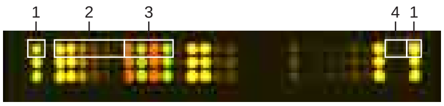


Fig 86. To validate, normalize, and filter microarray data, four different types of controls are supplied:

1. Calibration controls: the signal intensities of ten individual controls span 4.5 orders of magnitude for both Cy3 and Cy5 channels. These controls can be used to generate a calibration curve.
2. Ratio controls: eight ratio controls are provided at both low and high expression levels and are used to evaluate precision of ratios.
3. Negative controls: two controls are used to estimate non-specific hybridization and potential carryover with the microarray system.
4. Utility controls: three individual controls can be used to troubleshoot and examine sample preparations, or they can be used as additional ratio or calibration controls.

Spikes for determining sensitivity and dynamic range

Using some of the negative control genes discussed above can be used to make a different set of positive controls. RNA can be synthesized by *in vitro* transcription from plasmids carrying these negative control sequences. This synthetic mRNA can then be included in the Cy3 and Cy5 labelling reactions at known concentrations. This process is known as spiking, and the synthetic mRNA are the spikes. Several different mRNA spikes can be used and spiked in at different concentrations, resulting in a set of controls that can give the researcher a value for the linear dynamic range and sensitivity of the assay. These controls are known as dynamic range controls. In addition, different spikes can be spiked into the Cy3 and Cy5 labelling reactions at different concentrations. These types of controls are known as ratio controls. After hybridization of the probe to the slide, washing, and scanning, the Cy3 and Cy5 signals obtained from the ratio controls can be compared with the known amount of mRNA spiked into the labelling reactions and the theoretical known ratios. Therefore a series of spikes can determine how sensitive the hybridization has been and how accurate the data obtained is.

11.5.4 Housekeeping gene controls

Some genes are expressed relatively consistently within many different cell types. These are called housekeeping genes. Examples of such genes are actin, glyceraldehyde 3-phosphate dehydrogenase (GAPDH), and tubulin. These housekeeping genes can be included in microarray experiments as controls to ensure that the hybridization has occurred, and they can also act as a normalization factor. However, the expression of these genes can vary under experimental conditions, and relying on the use of one or few housekeeping genes can result in skewed data.

11.5.5 Controls for measuring pen-to-pen variation

Housekeeping gene, spikes, or positive controls can be spotted in replicate across the slide. If a different pen on the microarray spotter spots each of these replicates, this may provide some information on the pen-to-pen variation of the spotter. It should be considered that the variation of each of these controls will be dependent not only on pen-to-pen variation but also on the variation in the slide surface and any variation in the hybridization. Therefore, if a high pen-to-pen variation is seen, a pen test on the spotter should be performed.

11.6 Control products offered by Amersham Biosciences

11.6.1 Lucidea Universal Scorecard

Lucidea Universal ScoreCard contains a set of 23 artificial genes that serve as analytical controls to validate and normalize microarray data. The controls are composed of DNA sequences from yeast intergenic regions, and their performance has been shown to be independent of a wide variety of species. This system can be used as a universal reference for validating and normalizing microarray data as well as for creating a calibration curve for determining limit of detection, linear range, and saturation of microarray experiment (Fig 86).

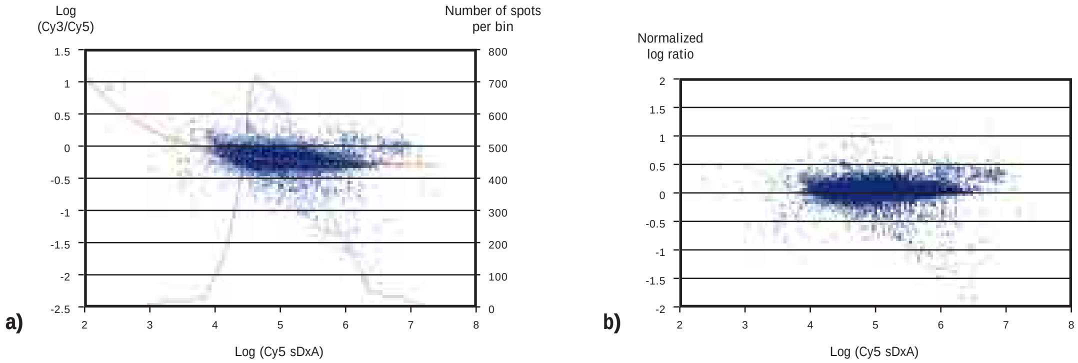


Fig 87. Applying non-linear normalization to microarray data. Panel A shows distribution of log(Cy3/Cy5) values plotted against Cy5 signal values (blue crosses) from a typical microarray experiment. The orange line denotes the average relationship between these log ratios as a function of Cy5 signals. As can be seen from the shape of the curve, this relationship is non-linear over the distribution of Cy5 signals. The blue line shows distribution of Cy5 signal intensities. Panel B shows the same data after it has been normalized using the non-linear normalization algorithm.

11.7 Normalization

In order to compare ratio data from one microarray slide to another microarray slide, the ratio data needs to be normalized to correct for experimental variation. The reason for this is that from one slide to another there will be differences between the relative Cy3 and Cy5 signals due to one or more of the following:

- the amounts of mRNA used in the Cy3 and Cy5 labelling reaction
- efficiency of detection of the Cy3 and Cy5 by the detection system within the scanner
- relative incorporation differences of the Cy3 and Cy5 reverse transcriptases

11.7.1 Linear normalization

Linear normalization assumes that there is a single normalization factor required over the whole signal range. There are three methods to obtain this factor:

- use signal ratios from housekeeping genes
- use signal ratios from spikes which have been added to the labelling reaction in equal amounts
- use total signal, which is the summation of all the Cy3 signals and all the Cy5 signals

In the housekeeping gene or spike gene methods, it is assumed that the spots corresponding to these targets give a ratio of 1. The total signal method assumes that addition of all the signals in the Cy3 and the Cy5 channel should give a Cy3/Cy5 ratio of 1. The normalization factor can be calculated from the observed ratios for the housekeeping genes, spikes, or total signals to give a conversion factor that results in the expected ratio for the control spots.

For example, a housekeeping gene in experiment A gives a ratio of 2, while the gene of interest has a Cy3/Cy5 ratio of 5. Therefore the normalized ratio for the gene of interest is 5/2 = 2.5. In experiment C, the housekeeping gene has a ratio of 3, while the gene of interest has a Cy3/Cy5 ratio of 7.5. The normalized ratio is 7.5/3 = 2.5.

The three methods discussed above assume that the normalization factor is constant over the whole signal range, which in most cases is not.

11.7.2 Non-linear normalization

It has been found that linear normalization is not necessarily accurate (67). Examining the data shown in Figure 87a can show this. In this experiment, Cy3-labelled muscle and Cy5-labelled muscle probes (identical mRNA labelled with two different fluors) were hybridized to a single slide. The results of this experiment are plotted below as a plot of log Cy3/Cy5 ratio against the log Cy5 signal (Fig 87b). As the same mRNA was used, it would be expected that the log Cy3/Cy5 ratio should be constant over all Cy5 signals. However, as can be seen from the graph, the ratio is higher at low Cy5 signal levels compared to the figure at high Cy5. Therefore, the normalization factor used for spots with a low Cy5 signal should be different from the normalization factor that is used at high Cy5 signals.

There are two principal non-linear methods of calculating the normalization factor. One method is to rank all the data points according to their Cy3+Cy5 signal. Then for each 50 genes calculate the normalization factor for those 50, in the same way as total normalization is carried out. A more precise way is to fit a curve to the data so that the normalization factor for each point can be calculated. Software packages are commercially available that can perform this kind of normalization. A non-linear normalization generally results in a more accurate normalization than linear normalization.

11.7.3 Post normalization

Once the normalization procedure has been carried out for all the data points, the behavior of controls is examined next. The negative controls should have a signal-to-noise ratio of less than 3 [(SNR = [average signal – average background]/standard deviation of background). Any higher SNR than 3 suggests that the data obtained from this experiment may not be accurate.

If a series of dynamic range controls have been included, this is one way to estimate sensitivity. A control which has a SNR above 3 would be regarded as having been detected. If spikes have been included, such that the spikes have been placed in the Cy3 reaction at a different amount compared to the Cy5 reaction, then the theoretical ratio can be compared to the actual ratio. Finally, controls such as pen-to-pen variation controls may suggest other potential problems within the experiment.

If the data from the slide meets the criteria set by the researcher then the next step is to look at the variation between replicates. It is recommended that each experiment be repeated several times, and there are statistical criteria as to how many replicates should be carried out to give a certain degree of confidence in your results (64). Replicates could be within multiple spots on the same slide, but can also be carried out using several different slides. Typically log ratios are calculated so that ratios less than 1 appear as a negative value. The coefficient of variation (CV = [Standard deviation of the ratio] / [mean ratio]) can also be calculated and provides a simple measure of the value of a particular data point. Data points with high CVs can be highlighted or discarded. Often ratios with high CVs are due to low signal in one or both of the channels. A standard practice by some researchers is therefore to discard spots which have a signal in both channels with a SNR below 3. If there is a low signal (below a SNR of 3) in one channel, then the ratio could be considered to be an arbitrary fixed value to avoid very large ratios (to prevent, for example if Cy5 signal is zero, the Cy3/Cy5 would be infinite). In addition, it should be remembered that there may be significant biological variation that must be taken into account when designing experiments.

11.8 Visualization and clustering

After microarray data is normalized to account for differences in Cy3 and Cy5 signals, it can subsequently be exported to any number of data visualization software for further analysis. These software products can be used to mine the data for significant changes in gene expression. The process of visualization can significantly enhance data analysis. It can provide helpful features, such as data integration, customized query devices, and pattern recognition. Clustering data points, or genes, that show similar responses on microarray analysis can be used to identify genes that have similar gene expression patterns and which possibly belong to the same pathway.

11.9 Visualization software products offered by Amersham Biosciences

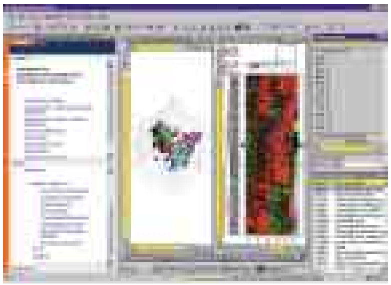


Fig 88. Exploring gene expression data using hierarchical clustering, and principal component analysis with Spotfire DecisionSite for Functional Genomics.

11.9.1 Spotfire DecisionSite for Functional Genomics

This software combines the core capabilities of Spotfire™ DecisionSite™ with specialized tools for interrogating and extracting information from microarray data. In addition to simplified access to data and information, Spotfire DecisionSite for Functional Genomics provides researchers with leading analytical methods used in gene expression analysis. Dynamic visualizations and interaction with computational results help researchers in validating and prioritizing target genes (Fig 88).

Some of the features of this software include:

- easy access to information, in any format, wherever it resides
- visually interactive representations of enriched data sets for enhanced analysis
- publication and sharing of results for collaborative decision-making
- idenification of key patterns with distinction calculation, hierarchical, bi-directional hierarchical, and K-means cluster analysis
- preparation of data for analysis with standard array and gene based normalization
- identification of entities exhibiting characteristic or signature profiles with ad hoc profile search and analysis

11.10 Scierra Microarray Laboratory Workflow Systems: Information Management Systems for Microarray Laboratories

Microarray technology is rapidly becoming the mainstream platform for high-throughput gene expression analysis. As microarray experiments generate vast amounts of experimental and biological data, an urgent need is created for informatics tools that can manage the microarray workflow process more effectively and efficiently. Scierra™ Microarray Laboratory Workflow System, as a part of a larger integrated laboratory information managment system, is designed to address this need.

11.10.1 Scierra Laboratory Workflow Systems

Scierra Laboratory Workflow Systems (LWS) are a series of bioinformatics solutions to aid in the collection, annotation, collation, curation, and analysis of biological data. The platform includes four products:

- Scierra Sequencing LWS System
- Scierra Genotyping LWS System
- Scierra Microarray LWS System
- Scierra Proteomics LWS System

These Scierra LWS products are built on a common software framework that manages and tracks all aspects of an experiment. Each system mirrors the natural workflow found within the laboratory, and succeeds in linking manual processes, instrumentation, software, and reagent use into one system. This integration enables the collection and comparison of each type of biological information. The open framework design allows the introduction of new instruments and reagents, thereby providing a scalable system that will grow as needs increase.

11.10.2 Scierra LWS architecture

Scierra LWS is a three-tiered application comprised of an Oracle™ database, a middleware application server, and multiple clients. Scierra LWS can accept data from most available network client sources, including computers running Windows™ 2000 or Windows NT™. Work is easily requested and results readily accessed through a standard browser-based user interface that communicates to the middleware application server.

11.10.3 Scierra Microarray Laboratory Workflow System (MA-LWS)

A typical microarray workflow involves array content preparation, array production, sample preparation and labelling, array hybridization, scanning, and image analysis.

Scierra MA-LWS has been designed to mirror the workflow of the typical microarray lab (Fig 89). It allows users to perform many tasks. First, users can organize large numbers of samples and experiments based on projects. Users can also manage and track a large variety of samples and reagents, including:

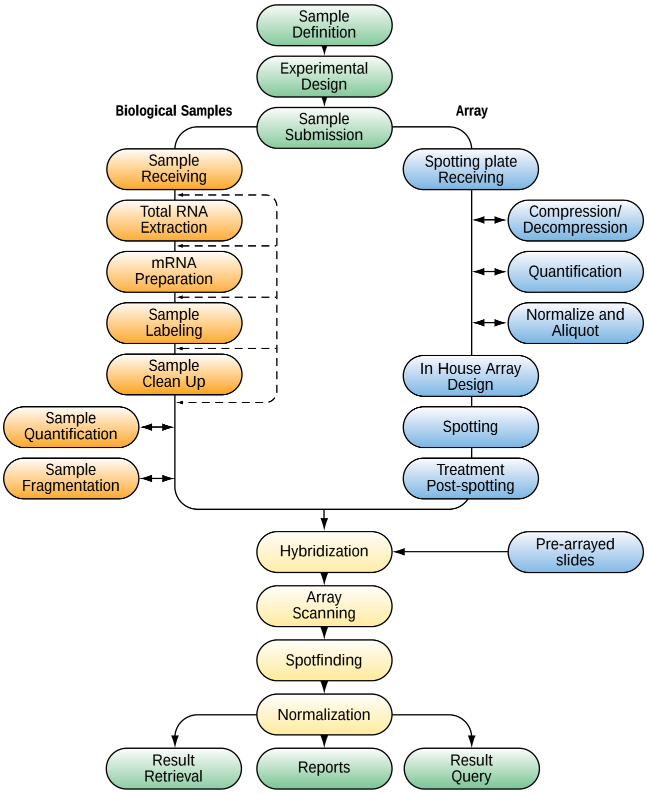
- crude biological samples, including blood, tissue, and cells
- total RNA, mRNA, and labelled cDNA
- spotting plates and spotting sample types, including cDNA and oligo
- spotting substrates, chemistry, and protocol
- arrayed slides

Scierra MA-LWS allows users to manage and track every activity in the microarray workflow, including:

- array preparation—spotting custom arrays and pre-arrayed slides
- sample preparation and labelling
- hybridization, scanning, and image analysis

Scierra MA-LWS makes it possible to integrate components of different microarray systems, including spotters and scanners, and image analysis software. Users can store and effectively retrieve large amounts of information. Flexible reporting tools provide for standard and user-defined queries across different activities. With Scierra MA-LWS, the precious microarray data is completely captured and securely stored in the database for further analysis.

Fig 89. Scierra MA-LWS mirrors the microarray workflow.



Chapter 12

TROUBLESHOOTING MICROARRAY EXPERIMENTS

12.0 Introduction

Microarray analysis is a complicated multistep process consisting of discrete steps as shown in Figure 90. Each of these steps is critical in determining whether the experiment is successful, and problems encountered at any stage will be detrimental to the quality of data obtained. Troubleshooting microarray experiments needs to consider all these steps.

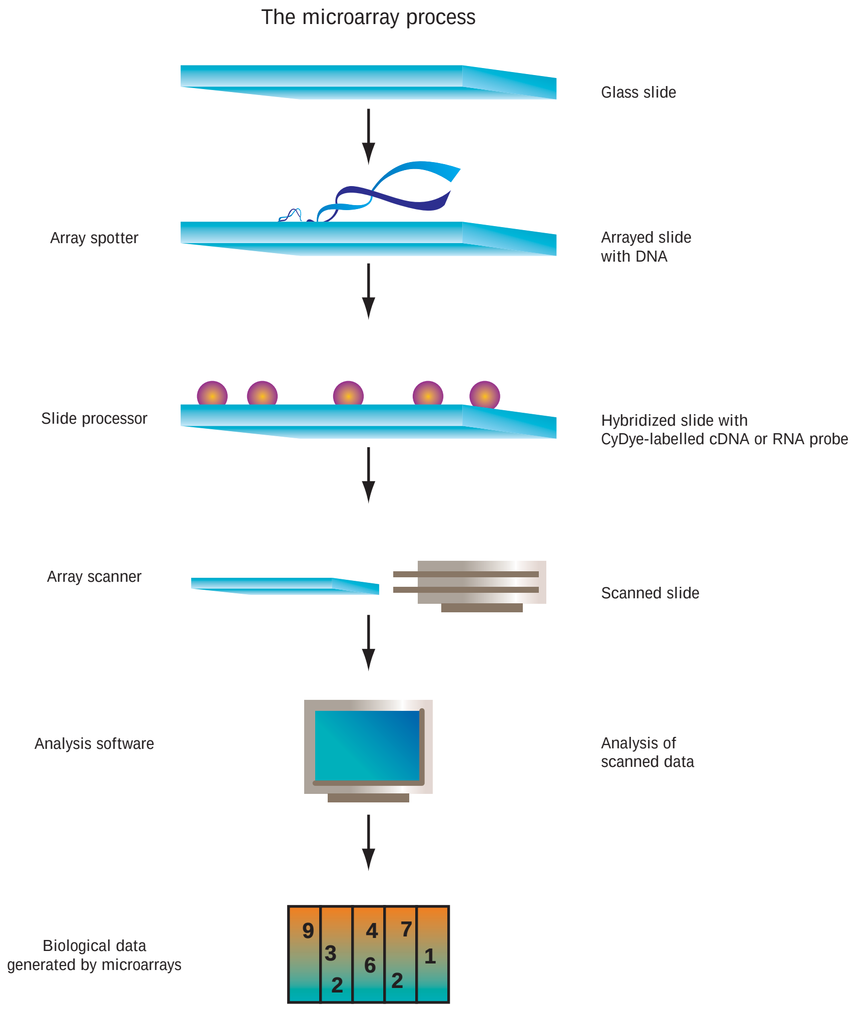


Fig 90. Flowchart of microarray experiment.

The output from a microarray experiment is the intensity of hybridization signals, which reflect the expression levels of the corresponding genes in the analyzed samples. However, other experimental factors also have significant influence on the magnitude of these signals. Some of these factors are listed in Figure 91.

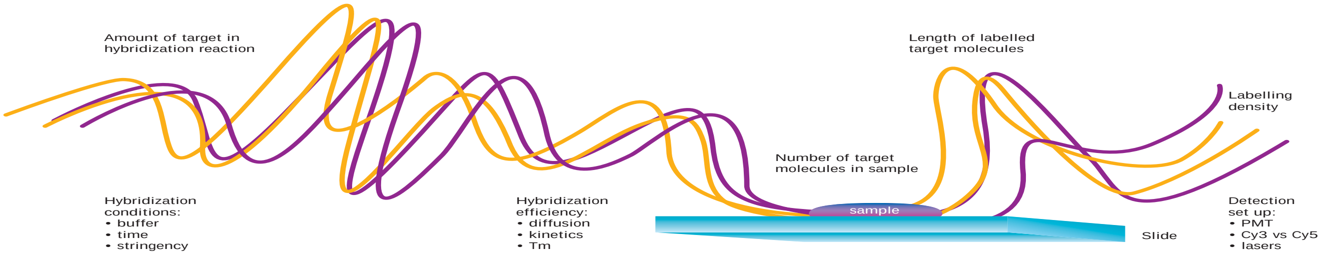

Amount of target in hybridization reaction
Length of labelled target molecules
Labelling density
Number of target molecules in sample
sample
Hybridization conditions:
• buffer
• time
• stringency
Hybridization efficiency:
• diffusion
• kinetics
• Tm
Slide
Detection set up:
• PMT
• Cy3 vs Cy5
• lasers

Fig 91. Factors influencing the intensity of observed microarray signals.

12.1 Optimization of the microarray system

As microarray analysis is not trivial to perform, careful optimization of the microarray system is recommended. Due to the wide choice of reagents, consumables and instruments from various manufacturers, it is important to optimize the selected combination of materials and protocols before starting real experiments. Some reagents may not work well with other reagents; for example, spotting buffer may not be compatible with all different slide surfaces, and different hybridization buffers may give very different results on identical slides.

System optimization should test all microarray components together using target sequences and probes that are as close to real samples as possible. Such experiments should incorporate controls, as discussed in Chapter 11. Combining the use of realistic sets of targets and control reagents, as illustrated in Figure 92, gives the best results. The aim of

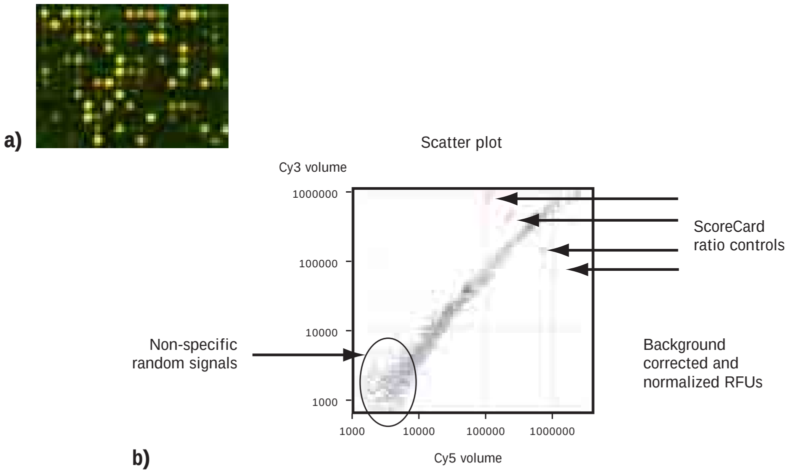


Fig 92. Panel A shows a scanned microarray image from a yellow experiment in which skeletal muscle mRNA was labelled with Cy3 and Cy5. Only a proportion of the microarray slide is shown. Panel B shows a scatterplot derived from a yellow experiment. The microarray used in this experiment contained ratio control targets in addition to cDNA clones and mRNA corresponding to these control sequences was spiked into the mRNA before labelling. These ratio control spots appear as green and red dots on the array image and are also shown on the scatterplot, where they fall above and below the scatter line.

system optimization is to find the best overall protocol for the system and to determine what is the standard performance of the system.

12.1.1 The yellow experiment

The yellow experiment is an efficient tool for microarray system optimization. In this experiment, the same RNA sample is labelled with both fluorescent dyes, typically with Cy3 and Cy5 fluors. Hybridization of equal amounts of both probes onto a microarray should produce equal hybridization signals from both colors. In a false color array image, all target spots should appear as yellow dots. As computer screen images of microarray data can be misleading, analyzing numerical data from a yellow experiment as a scatter plot is more informative. As no differential gene expression is expected, normalized Cy3 signals plotted against normalized Cy5 signals should appear as a straight line. Figure 92 shows an example of microarray data generated from a typical yellow experiment. Because RNA isolated from different cell cultures or individuals is likely to contain slightly different levels of some transcripts, it is recommended that pooled RNA obtained from several RNA isolations is used for system optimization. Alternatively, purified RNA is also available commercially.

Yellow experiments only require one type of RNA sample, but provide information on all aspects of the microarray process. In contrast, experiments in which two different RNA species are used have the added complication that differential gene expression will be present in unknown quantities. In reverse-color experiments, in which both samples are labelled separately with two colors and hybridizations are performed with both possible combinations, any inherent variation between the quality of the two arrays can complicate the optimization process and result in wrong conclusions. For example, high and uneven Cy3 background on one of the slides in a reverse-color experiment can give the appearance of unbalanced labelling with one fluorescent dye. For ‘real’ experiments in which information about gene expression is being sought, reverse color experiments are useful.

12.2 Experimental design and execution

12.2.1 Experimental variation

Experimental variation must be taken into account when designing and carrying out microarray experiments (Fig 93). No two microarray experiments, even if replicas of each other, will give exactly the same results. Each step of the process contributes to this variation, which may mask the presence of differential gene expression, leading to false negative results. If the amount of variation is not known, false positive results can also be obtained if randomly varying results are taken at face value.

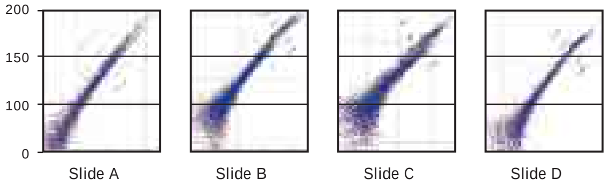


Fig 93. Experimental variation. Four identical microarray slides were hybridized simultaneously with equal aliquots of the same probe. Scatterplots of normalized gene expression data are shown. Variation in background fluorescence arising from uneven slide surfaces was the major contributing factor to experimental variation in this case.

12.2.2 Replication in microarray analysis

Replication is the key to identifying and quantifying variation in microarray experiments. It has been found that performing three replica microarray hybridizations with different slides reduced the misclassification of gene expression compared with performing single hybridizations (64). Data calculated and pooled from all these replicas enables statistical determination of experimental variation in terms of standard deviation and coefficient of variance (CV).

Microarray experiments should contain the following:

- Each target should be present in at least two, preferably more, copies on the microarray.
- Multiple slides should be hybridized with each probe pair.
- Multiple RNA samples should be obtained for each experimental condition.

12.2.3 Step controls

Separately monitoring the various steps in microarray analysis, while lengthening the protocols, provides quality control information. Step controls also offer the following benefits:

- Researchers' resources are used most efficiently, with the least amount of waste.
- Problems can be identified on a step-by-step basis, and most likely causes obtained.
- Information from intermediate steps allows conclusions about the overall validity of microarray results to be drawn.

Ideal step controls provide numerical or visual information that unequivocally characterizes the success of that step. Recommended control procedures are listed in Table 6. **The control measures indicated in bold should be included in every microarray experiment**. Additional controls are recommended to be performed when new protocols or reagents are being tested. It is advisable to prepare some extra reagents if control procedures are performed. Greatest benefits from step controls are derived when the results are evaluated before continuing with the experiment.

12.3 Performing microarray analysis

There are several critical points to performing a successful microarray hybridization experiment. Please take note of the following:

- Follow all protocols precisely.
- Be consistent across experiments when several separate hybridizations are involved.
- Maintain precise technique when performing the microarray experiment and analyzing the results.
- Always use appropriate reagents for each protocol; alterations can be a source of error in final data analysis.
- Keep record of individual reagents used in each experiment as it can be useful in identifying causes of problems.

Table 7 lists some typical problems in microarray analysis and their likely causes. Most of these problems can be identified and avoided by following the recommended quality control measures listed in Table 7, which can also aid in identifying the cause of the problem. Often there can be several contributing causes to any problem.

Table 6. Quality control measures for microarray analysis.

Step	Control mearsures
Preparation of cDNA targets by PCR	■ **Verify presence of only one band in agarose electrophoresis.** ■ **Determine quantity of target DNA.** ■ Sequence target to verify identity.
Microarray slide coating	■ Scan to detect presence of background fluorescence or dirt particles. ■ Perform additional surface tests.
Microarray printing	■ Use fluorescent DNA as control for printing quality. ■ Use DNA-binding dyes to detect printing DNA on slide. ■ **Perform a test yellow experiment with well-characterized RNA preparation for each printing batch and especially when new targets are introduced.** ■ **Incude plenty of control targets on slide.** ■ **Note down temperature and humidity of printing chamber.** ■ **Keep track of how many times targets have been used.** ■ **Keep track of when slides were printed.**
RNA isolation	■ **Check purity and integrity of RNA by gel electrophoresis or RT-PCR.** ■ **Determine quantity of RNA.**
Sample labelling and purification	■ **Spike in synthetic control mRNA.** ■ Perform control labelling reaction with control RNA. ■ **Determine incorporation of CyDye into labelled sample by spectrophotometry.** ■ **Determine whether purified sample contains free CyDye by gel electrophoresis.** ■ Determine size of labelled nucleic acid fragments with gel electrophoresis. ■ Determine the amount of labelled nucleic acid. ■ Determine the amount of CyDye per microgram of labelled nucleic acid (labelling density). ■ Determine the recovery of labelled cDNA from purification step by radioactive spiking or gel analysis.
Hybridization and stringency washes	■ **Use equal and optimal amounts of Cy3- and Cy5-labelled probes in the hybridization.** ■ **Include positive and negative hybridization controls.** ■ Perform hybridization without probe.
Scanning	■ Test scanner performance in order to verify correct functioning. ■ **Perform scans at different settings to ensure optimal data collection.** ■ **Visually inspect scanned images to detect any obvious blemishes or areas of poor data.**
Data analysis	■ **Background correct and normalize data before drawing conclusions.** ■ **Examine how well normalization worked.** ■ **Determine the amount of variation in experiment.** ■ **Never trust data from one slide.**
Verification of microarray results	■ **Use independent analytical techniques to verify whether the results obtained from microarray analysis are reproducible and biologically significant.**

Table 7. Troubleshooting microarray experiments,

Symptom	Possible cause	Remedy
No hybridization signal.	■ Target concentration too low.	■ Determine target concentration before slide spotting.
	■ Targets not clean enough.	■ Remove PCR components from targets before slide spotting.
	■ Poor retention of targets on slide.	■ Prepare new microarray slides. Check that spotting buffer and protocol are compatible with slide type.
	■ No transcripts in RNA sample.	■ Obtain new RNA/mRNA sample and test it before labelling.
	■ Failed labelling reaction.	■ Always check the success of labelling reaction before using it in hybridization.
	■ Faulty component in labelling reaction.	■ Test components of labelling reaction against new reagents. Use control RNA.
	■ Loss of probe in purification step.	■ Check success of probe purification before use.
	■ Poor hybridization.	■ Check that hybridization buffer and protocol are compatible with slide type.
	■ Failure of scanning instrument.	■ Test performance of scanner with known amounts of fluors.
	■ Detection sensitivity too low.	■ Adjust detection sensitivity.
	■ CyDye have been exposed to light during handling.	■ Protect CyDye from light.
	■ Target genes not expressed in examined tissue.	■ Use housekeeping genes and positive controls to ascertain proper functioning of the system.
	■ Human error at some stage.	■ Repeat experiment and use step controls to monitor progress.
Low or undetectable Cy3 and/or Cy5 signal.	■ Poor retention of targets on slide. Identical probes were hybridized with two different types of slide that contained the same targets (Fig 94n, 94o).	■ Check purity and concentration of targets. Use slides of different batch. Use different slide type.
	■ Targets are old.	■ Prepare new targets for spotting.
	■ Poor labelling reaction with one dye. One or more components faulty in labelling reaction.	■ Check success of labelling reaction. Check performance of all components of labelling reaction such as nucleotides, enzyme and fluorescent nucleotides or reactive dyes.
	■ Loss of probe in purification.	■ Check performance of purification. Do not purify Cy3 and Cy5 probes together.
	■ Unequal amount of Cy3 and Cy5.	■ Use equal amounts of probes in hybridization.
	■ Too little probe in hybridization.	■ Measure the amount of probe before hybridization. Use more RNA to prepare probe.
	■ Free CyDye in probe.	■ Optimize probe purification.
	■ Poor quality RNA sample or samples.	■ Test RNA before labelling.
	■ Too much or too little quantity of RNA.	■ Measure amount of RNA before labelling.
	■ RNA contaminated by DNA.	■ Use DNAse I to remove DNA.

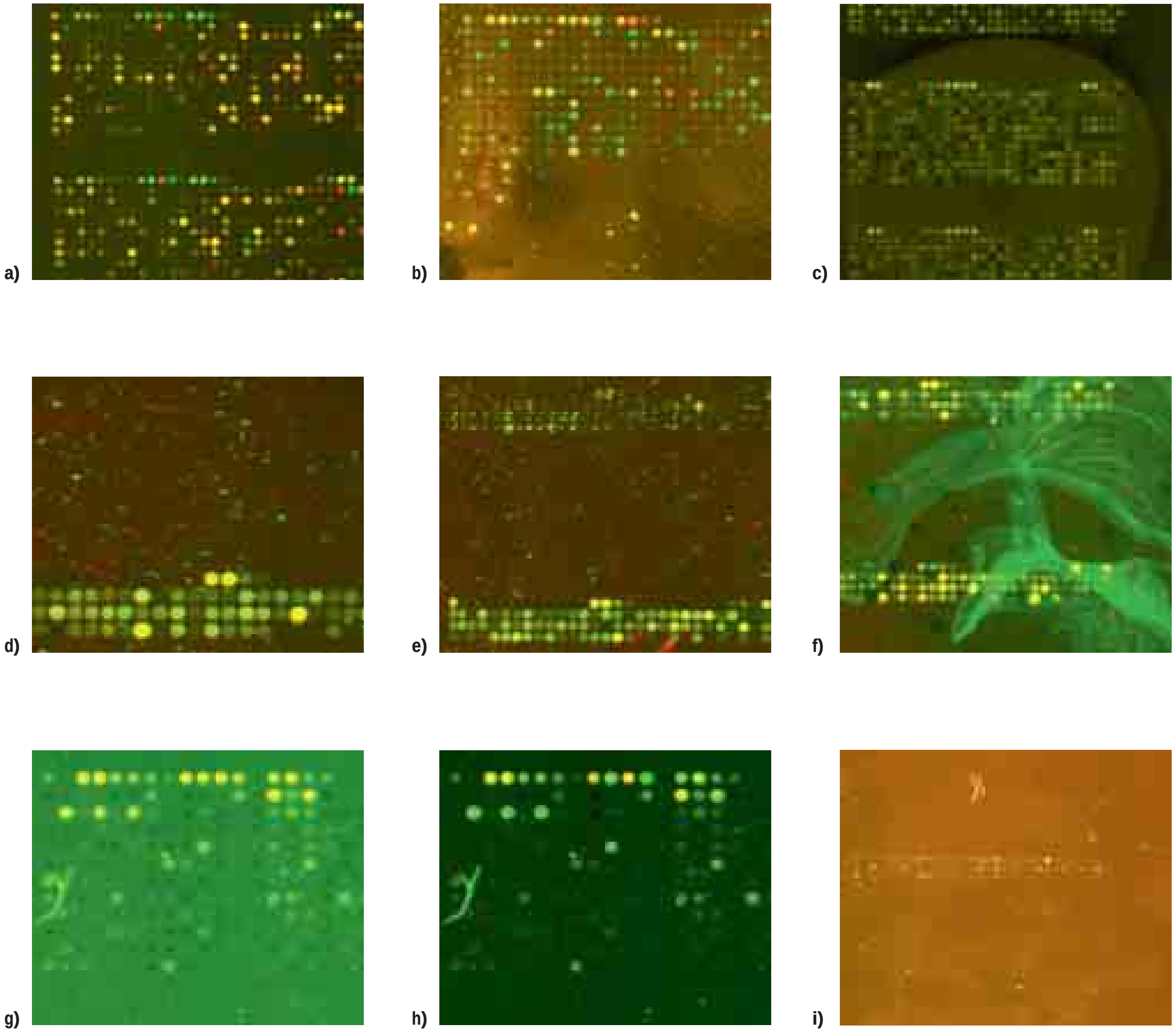
Table 7 cont'd. Troubleshooting microarray experiments.

Symptom	Possible cause	Remedy
	Hybridization conditions not optimal.	Check compatibility of hybridization buffer and slide surface. Use lower hybridization stringency.
	High background in hybridization.	Use slides of a different batch. Optimize hybridization and wash protocol.
	Overexposure of Cy3 and Cy5 to light during storage and handling.	Protect CyDye from light always.
	Laser source not working optimally.	Check laser performance.
	Detection sensitivity not optimal.	Optimize laser power and detection sensitivity settings.
	Detection sensitivity not optimal.	Optimize laser power and detection sensitivity settings.
	Human error.	Repeat experiment with step controls.
Unbalanced Cy3 and Cy5 signals.	Too high labelling density leading to quenching of one fluorescent dye.	Label to a lower density.
	Too much CyDye nucleotide in labelling reaction.	Use less CyDye nucleotide.
	Too much of one probe in hybridization leading to quenching.	Optimize the amount of probe in hybridization. Measure the amount of probe in hybridization.
	Some nucleotide sequences label poorly.	Use a different labelled nucleotide. Use a different labelling method.
	Poor or variable quality of RNA sample/samples. Biological variation in RNA samples.	Use good quality RNA for labelling. Use pooled RNA samples.
	High fluorescent background in . one color.	See separate entry.
	Normalization method is not adequate.	Use a different normalization method.
	High amount of variation in experiment.	Optimize and standardize experimental conditions to reduce amount of variation.
High background, weak specific signals.	Poor labelling reaction.	Check success of labelling reaction before hybridization.
	Random nonamers used for labelling total RNA.	Prepare probe with oligo(dT) primer.
	Probe fragments very short.	Use good quality RNA. Check purity of RNA. Re-purify RNA.
Uneven fluorescent background on slide.	Poor slide quality with an uneven coating. Often background is higher on one side of slide (Fig 94a).	Use slides of a different batch. Optimize slide surface treatment protocol. Use different source of microscope slides.
	Fluorescent background from pre-hybridization or hybridization solution (Fig 94f).	Optimize wash protocol. Include water dip at the end. Dry slides quickly.

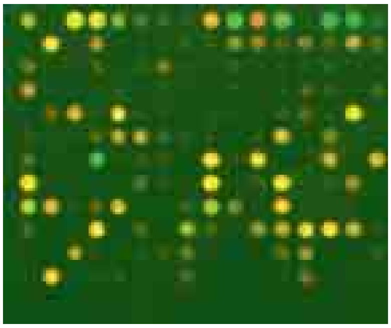
Table 7 cont'd. Troubleshooting microarray experiments.

Symptom	Possible cause	Remedy
	■ Salts in wash buffer dried onto dried onto slide (Fig 94b).	■ Dip slide into water before drying.
	■ Powder from lab gloves adheres to slide.	■ Handle slides using powder-free gloves.
	■ Edge of coverslip has dried during manual hybridization (Fig 94m).	■ Perform hybridization under humid conditions.
Most spots give high uniform signal.	■ High amount of unspecific signals.	■ Increase stringency of hybridization and washes.
Even fluorescent background.	■ Poor slide quality (Fig 94g, 94i).	■ Use slides from different batch. Use different source of microscope slides.
	■ Too much probe used in hybridization.	■ Quantify probe before use.
	■ Over labelling of sample.	■ Optimize amount of probe to use. Optimize labelling density.
Speckled background on slide.	■ CyDye nucleotides remain in probe (Fig 94d).	■ Optimize purification of probe. Check probe for presence of free CyDye.
Particles seen on slide.	■ Dust particles have been fixed onto slide.	■ Always handle slides in clean environment. Use air stream to remove any dust particles from dry slides before spotting and use.
	■ Slide surface is scratched (Fig 94p).	■ Handle slides with care using forceps.
	■ Finger prints seen on slide.	■ Never touch slides with bare hands.
Bubble effect on slide.	■ Air has been trapped under coverslip (Fig 94c).	■ Remove air bubbles from hybridization.
Spots appear as comets with tails.	■ Hybridized probe is coming loose during low stringency wash/water dip (Fig 94j).	■ Optimize wash conditions. Dry slides quickly after washes and water dip.
Deformed spots.	■ Doughnut spots (Fig 94l).	■ Control humidity of spotting process.
	■ Tiny spots.(Fig 94e).	■ Test printing pen performance.
	■ Variably sized spots (Fig 94k).	■ Wrong spotting buffer for slide chemistry.
	■ Negative spots caused by slide background which is higher than hybridization signals (Fig 94h).	■ Use slides of a different batch.

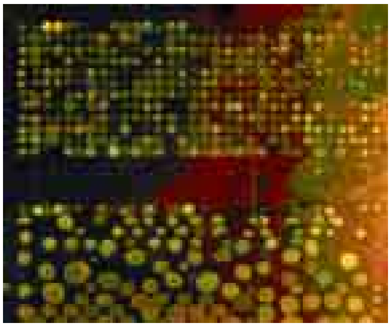
Fig 94. Troubleshooting microarray experiments.



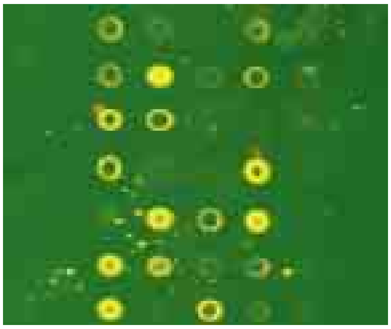
j)



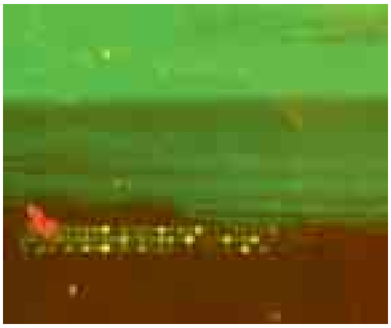
k)



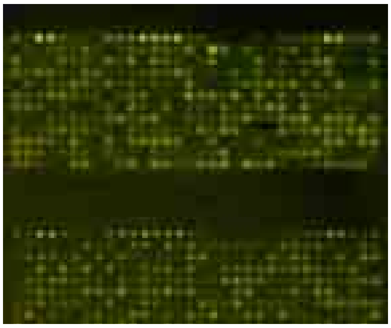
l)



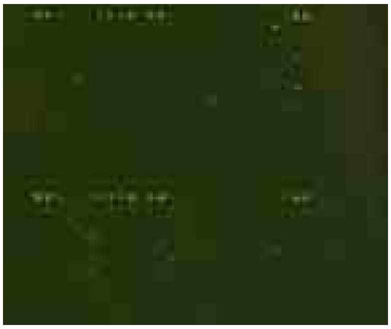
m)



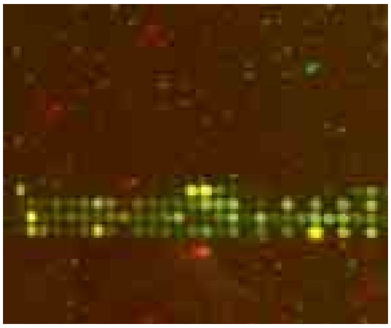
n)



o)



p)



References

References cited in text

1. Baldwin, D. *et al*. A comparison of gel-based, nylon filter and microarray techniques to detect differential RNA expression in plants. *Curr Opinions in Biol.* **2**, 96–103 (1999).

2. Watson, A. *et al*. Technology for microarray analysis of gene expression. *Current Opinions in Biotech*. **9** 609–614 (1998).

3. Schena, M. *et al*. Microarrays: Biotechnology's discovery platform for functional genomics. *Trends in Biotech*. **16**, 301–306 (1998).

4. Kozian, D. H. and Kirschbaum, B. J. Comparative gene expression analysis. *Trends in Biotech*. **17**, 73–78 (1999).

5. Braxton, S. and Bedilion, T. The integration of microarray information in the drug development process. *Current Opinions in Biotech*. **9**, 643–649 (1998).

6. Mirnics, K. *et al*. Analysis of complex brain disorders with gene expression microarrays: Schizophrenia as a disease of the synapse. *Trends in Neuroscience* **24**, 479–486 (2001).

7. Schulze, A. and Downward, J. Navigating gene expression using microarrays — a technology review. *Nature Cell Biology* **3**, e190–e195 (2001).

8. Van Berkum, N. L. and Holstege, F. C. P. DNA microarrays: raising the profile. *Current Opinions in Biotech*. **12**, 48–52 (2001).

9. Alizadeh, A. A. *et al*. Towards a novel classification of human malignancies based on gene expression patterns. *J Pathol*. **195(1)**, 41–52 (2001).

10. DeRisi, J. *et al*. Use of cDNA microarray to analyze gene expression patterns in human cancer. *Nature Genetics* **14**, 457–460 (1996).

11. Rew, D. A. DNA microarray technology in cancer research. *European Journal of Surgical Oncology* **27**, 504–508 (2001).

12. Shoemaker, D. D. *et al*. Experimental annotation of the human genome using microarray technology. *Nature* **409**, 922–927 (2001).

13. Lieb, J. D. *et al*. Promoter-specific binding of Rap1 revealed by genome-wide maps of protein-DNA association. *Nature Genetics* **28**, 327–334 (2001).

14. Hu, G. K. *et al*. Predicting splice variant from DNA chip expression data. *Genome Research* **11**, 1237–1245 (2001).

15. Meltzer, P. S. Spotting the target: microarrays for disease gene discovery. *Current Opin in Genetics and Dev*. **11**, 258–263 (2001).

16. Sapolsky, R. J. *et al*. High-throughput polymorphism screening and genotyping with high-density oligonucleotide arrays. *Genet Anal*. **14(5-6)**, 187–92 (1999).

17. Larsen, L. A. *et al*. Recent developments in high-throughput mutation screening. *Pharmacogenomics* **2(4)**, 38799 (2001).

18. Drobyshev, A. *et al*. Sequence analysis by hybridization with oligonucleotide microchip: identification of beta-thalassemia mutations. *Gene* **188(1)**, 45–52 (1997).

19. Lockhardt, D. J. and Winzeler, E. A. Genomics, gene expression and DNA arrays. *Nature* **405**, 827–836 (2000).

20. Jain, K. K. Applications of biochip and microarray systems in pharmacogenomics. *Pharmacogenomics* **1**, 289–307 (2000).

21. Gray, N. S. *et al*. Exploiting chemical libraries, structure and genomics in the search for kinase inhibitors. *Science* **281**, 533–538 (1998).

22. Kane, M. D. *et al*. Assessment of the sensitivity and specificity of oligonucleotide (50mer) microarrays. *Nucleic Acids Res*. **28**, 4552–4557 (2000).

23. Li, F. and Stormo, G. D. Selection of optimal DNA oligos for gene expression arrays. *Bioinformatics* **17**, 1067–1076 (2001).

24. Lockhart, D. J. *et al*. Expression monitoring by hybridization to high-density oligonucleotide arrays. *Nat Biotech*. **14(13)**, 1675–1680 (1996).

25. Schena, M. *et al*. Parallel human genome analysis: Microarray-based expression monitoring of 1000 genes. *Proc. Natl. Acad. Sci.* **93**, 10614–10619 (1996).

26. Schepinov, M. S. *et al*. Steric factors influencing hybridization of nucleic acid to oligonucleotide arrays. *Nucl Acids Res*. **25**, 1155–1161 (1997).

27. Schena, M. *et al*. Microarrays: biotechnology's discovery platform for functional genomics. *Trends Biotechnol*. **17**, 217–218 (1997).

28. Welford, S. M. *et al*. Detection of differentially expressed genes in primary tumor tissues using representational differences analysis coupled to microarray hybridization. *Nucleic Acids Res*. **26**, 3059–65 (1998).

29. Penn, S. G. *et al*. Mining the human genome using microarrays of open reading frames. *Nat Genet 2000* **26**, 315–318 (2000).

30. Hedge, P. *et al*. A concise guide to cDNA microarray analysis. *BioTechniques* **29**, 548–562 (2000).

31. Knight, J. When the chips are down. *Nature* **410**, 860–861 (2001).

32. Taylor, E. *et al*. Sequence verification as quality control. *Biotechniques* **31**, 62–65 (2001).

33. Schuchhardt, J. *et al*. Normalization strategies for cDNA microarrays. *Nucl. Acids Res*. **28**, e47 (2000).

34. Yamanaka, Y. *et al*. Gene expression profiles of human small airway epithelial cells treated with low doses of 14- and 16-membered macrolides. *Biochem. Biophys. Res. Comm*. **287**, 198–203 (2001).

35. Lipschutz, R. J. *et al*. High density synthetic oligonucleotide arrays. *Nature genetics* **21(supplement)**, 20–24 (1999).

36. Zammatteo, N. *et al*. Comparison between different strategies of covalent attachment of DNA to glass surfaces to build DNA microarrays. *Anal Biochem*. **280(1)**, 143–50 (2000).

37. Worley, J. *et al*. *Microarray Biochip Technology* (Schena, M., ed.), Eaton Publishing/BioTechniques Books, Natick, MA, pp. 65–85 (2000).

38. DeRisi, J. L. *et al*. Exploring the metabolic and genetic control of gene expression on genomic scale. *Science* **278**, 680–686 (1997).

39. Fodor, S. P. *et al*. Light-directed, spatially addressable parallel chemical synthesis. *Science* **251**, 767–73 (1991).

40. Mujumdar, R. B. *et al*. Cyanine dye labeling reagents: sulfoindocyanine succinimidyl esters. *Bioconjug Chem*. **4(2)**, 105–11 (1993).

41. Yu, H. *et al*. Cyanine dye dUTP analogs for enzymatic labeling of DNA probes. *Nucleic Acids Res*. **22**, 3226–32 (1994).

42. Sambrook, J. *et al. Molecular Cloning: A Laboratory Manual*, Cold Spring Harbor Laboratory Press, Cold Spring Harbor, New York (1989).

43. Ausubel, F. M., *Current Protocols in Molecular Biology*, Greene Publishing Associates and Wiley-Interscience, New York (2000).

44. Gruffat, D. *et al*. Comparison of four methods for isolating large mRNA: Apolipoprotein B mRNA in bovine and rat livers. *Anal Biochem*. **249**, 77–83 (1996).

45. Rosenow, C. *et al*. Prokaryotic RNA preparation methods useful for high density array analysis: comparison of two approaches. *Nucl. Acid Res*. **29**, e112 (2001).

46. (Farrell, R. E., ed.), *RNA Methodologies, A laboratory guide for Isolation and Characterization*, Academic Press, Inc., New York, pp. 125–157 (1997).

47. Randolp, J. B. and Waggoner, A. S. Stability, specificity and fluorescence brightness of multiply-labeled fluorescent DNA probes. *Nucl Acid Res*. **25**, 2923–2929 (1997).

48. Smoot L. M. *et al*. Global differential gene expression in response to growth temperature alteration in group A Streptococcus. *Proc. Natl. Acad. Sci*. **98**, 10416–421 (2001).

49. Talaat A. M. *et al*. Genome-directed primers for selective labeling of bacterial transcripts for DNA microarray analysis. *Nature Biotech*. **18**, 679–682 (2000).

50. Van Gelder, R. N. *et al*. Amplified RNA synthesized from limited quantities of heterogeneous cDNA. *Proc Natl Acad Sci*. **87**, 1663–1667 (1990).

51. Eberwine, J. H. *et al*. Analysis of gene expression in single live neurons. *Proc Natl Acad Sci*. **89**, 3010–3014 (1992).

52. Luo, L. *et al*. Gene expression profiles of laser-captured adjacent neuronal subtypes. *Nature Medicine* **5**, 117–122 (1999).

53. Herrler, M. Use of SMART-generated cDNA for differential gene expression. *J Molecular Medicine* **78**, B23 (2000).

54. Southern, E. M. Detection of specific sequences among DNA fragments separated by gel electrophoresis. *J. Mol. Biol*. **98**, 503 (1975).

55. Casey, J. and Davidson, N. Rates of formation and thermal stabilities of RNA:DNA and DNA:DNA duplexes at high concentrations of formamide. *Nucleic Acids Res*. **4**, 1539–1551 (1977).

56. Southern, E. *et al*. Molecular interactions on microarrays. *Nat Genet*. **21(1 Suppl)**, 5–9 (1999).

57. O'Shea, D., in *Introduction to Lasers and their applications*, Addison-Wesley, Reading, MA, pp. 51–78 (1978).

58. Smith, W. J., in *Modern Optical Engineering*, McGraw Hill, Boston, MA, pp. 142–145 (1990).

59. Skoog, D. A. *et al*., in *Principles of Instrumental analysis*, Harcourt Brace, Philadelphia, p. 108 (1998).

60. Gonzalez, R. C. and Woods, R. E., in *Digital Image Processing*, Addison-Wesley, Reading, MA, pp. 31–37 (1978).

61. Brazma A. *et al*. One-stop shop for microarray data. *Nature*. **403**, 699–700 (2000).

62. Brazma, A. and Vilo, J. Gene expression data analysis. *FEBS Letters* **480**, 17–24 (2000).

63. Brazma, A. *et al*. Minimum information about a microarray experiment (MIAME) — toward standards for microarray data. *Nature genetics*. **29**, 365–371 (2001).

64. Lee, M. *et al*. Importance of replication in microarray gene expression studies: Statistical methods and evidence from repetitive cDNA hybridizations. *Proc Natl Acad Sci*. **97**, 9834–9839 (2000).

65. Bernard, K. *et al*. Multiplex messenger assays: simultaneous, quantitative measurement of expression of many genes in the context of T cell activation. *Nucl Acids Res*. **24**, 1435–1442 (1996).

66. Yue, H. *et al*. An evaluation of the performance of cDNA microarrays for detecting changes in global mRNA expression. *Nucleic Acids Res*. **29**, e41 (2001).

67. Tseng, G. C. *et al*. Issues in cDNA microarray analysis: quality filtering, channel normalization, models of variations and assessment of gene effects. *Nucl. Acids Res*. **29**, 2549–2557 (2001).

68. Hemmilä, I.A., *Applications of Fluorescence in Immunoassays*, John Wiely and Sons, Inc. New York (1991).

	a	b
Microarray probe preparation		
CyScribe First-Strand cDNA Labelling Kit	25 reactions	RPN6200
CyScribe First-Strand cDNA Labelling Kit with CyScribe GFX Purification Kit	25 reactions	RPN6200X
CyScribe First-Strand cDNA Labelling System-dUTP	50 reactions	RPN6201
CyScribe First-Strand cDNA Labelling System-dUTP with CyScribe GFX Purification Kit	50 reactions	RPN6201X
CyScribe First-Strand cDNA Labelling System-dCTP	50 reactions	RPN6202
CyScribe First-Strand cDNA Labelling System-dCTP with CyScribe GFX Purification Kit	50 reactions	RPN6202X
CyScribe Post-Labelling Kit	24 reactions	RPN5660
CyScribe Post-Labelling Kit with CyScribe GFX Purification Kit	24 reactions	RPN5660X
CyScribe Direct mRNA Labelling Kit	24 reactions	RPN5665
Cy3-dCTP	25 nmol	PA53021
Cy5-dCTP	25 nmol	PA55021
Cy3-dUTP	25 nmol	PA53022
Cy5-dUTP	25 nmol	PA55022
Cy3-UTP	100 nmol	PA53026
Cy5-UTP	100 nmol	PA55026
CyDye Post-Labelling Reactive Dye Pack	24 reactions	RPN5661
a b a		
Lucidea Array Spotter	1	63-0040-09
Lucidea SlidePro Module 1	1	18-1162-01
Lucidea SlidePro Module 2	1	18-1162-02
Lucidea SlidePro Module 3	1	18-1162-03
Lucidea SlidePro Module 4	1	18-1162-04
Lucidea SlidePro Module 5	1	18-1162-05
Lucidea Automated Spotfinder	1	63-0038-18
Lucidea Universal ScoreCard	200 hybridizations	63-0042-85
Lucidea Reflective Slides		Available soon
a a a		
Typhoon 9610 Variable Mode Imager	1	63-0038-55

	a	b
a a a a		
ArrayVision for Scanners	1	ARV-100
Spotfire DecisionSite for Functional Genomics	1	63-0036-56
ImageQuant Solutions for Windows 2000 for Scanners	1	63-0035-16
a		
Scierra Microarray Laboratory Workflow System		
Ready-to-Run Separations Unit	1	80-6460-95
a		
RNase-free water	500 ml	US70783
QuickPrep Total RNA Extraction Kit	1	27-9271-01
RNA Extraction Kit	1	27-9270-01
QuickPrep Micro mRNA Purification Kit	1	27-9255-01
Ultraspec 3300 pro UV/Visible Spectrophotometer	1	80-2112-33
b a a a		
Microarray Hybridization Solutions version 2	1	RPK0325
Humid Hybridization Cabinet for microarrays	1	RPK0176
SSC 20 ×	100 ml	US19629
SDS 20%	500 ml	US75832
Hybond-N+ Membrane	50	RPN82B
Vistra Green Nucleic Acid Stain	500 ml	RPN5786
RapidGel-XL - 6%	100 ml	US75861
RapidGel-XL - 8%	100 ml	US75862
ALFexpress Sizer 50-500	50	27-4539-01
TBE Buffer Pre-mixed Powder 10 ×	6 bottles	US70454

Index

Microarrays

A ***microarray*** is a pattern of ssDNA probes which are immobilized on a surface (called a chip or a slide). The probe sequences are designed and placed on an array in a regular pattern of spots. The chip or slide is usually made of glass or nylon and is manufactured using technologies developed for silicon computer chips. Each microarray chip is arranged as a checkerboard of 10^5 or 10^6 spots or ***features***, each spot containing millions of copies of a unique DNA probe (often 25 nt long).

Like Southern & northern blots, microarrays use ***hybridization*** to detect a specific DNA or RNA in a sample. But whereas a Southern blot uses a *single* probe to search a complex DNA mixture, a DNA microarray uses *a **million** different* probes, fixed on a solid surface, to probe such a mixture. The exact sequence of the probes at each feature/location on the chip is known. Wherever some of the sample DNA hybridizes to the probe in a particular spot, the hybridization can be detected because the target DNA is labeled (and unbound target is washed away). Therefore one can determine which of the million different probe sequences are present in the target.

{NOTE: In a Southern, the target DNA is immobilized on a membrane; in a microarray, the probes are fixed to the slide or chip. In a Southern, the probe is labeled; in a microarray, the DNA being studied is labeled.}

Additionally, the amount of signal directly ***depends on the quantity of labeled target DNA***. Thus microarrays can give a **quantitative** description of how much of a particular sequence is present in the target DNA. This is particularly useful for studying gene expression, one common application of microarray technology.

Obviously, microarrays must be read mechanically, using a laser and detector. Good software for interpreting the raw data is crucial (as one can imagine a long list of sources of error in reading the individual spots, including nonspecific hybridization and background fluorescence).

To study gene expression, mRNA is isolated from the cells of interest and converted into labeled cDNA. This cDNA is then washed over a microarray carrying features representing all the genes that could possibly be expressed in those cells. If hybridization occurs to a certain feature, it means the gene is expressed. Signal intensity at that feature/spot indicates how *strongly* the gene is expressed (as it is a sign of how much mRNA was present in the original sample). One can therefore study gene expression in an entire cell (not just for one or two genes) under various conditions, over time, or in normal vs. diseased cells.

Microarrays are **sensitive enough to detect single base differences**, mutations, or SNPs (single nucleotide polymorphisms). This makes them useful for a wide range of applications, for example: identifying strains of viruses; identifying contamination of food products with cells from other plants or animals; detecting a panel of mutations in a patient's cancer cells that may influence the disease's response to treatment.

Protein microarrays are also being developed to allow massive screening for interactions between proteins on the microarray, and other proteins, substrates, or ligands.

From Affymetrix, makers of the GeneChip brand DNA microarrays: “Monitoring gene expression lies at the heart of a wide variety of medical and biological research projects, including classifying diseases, understanding basic biological processes, and identifying new drug targets. Until recently, comparing expression levels across different tissues or cells was limited to tracking one or a few genes at a time. Using microarrays, it is possible to simultaneously monitor the activities of thousands of genes (see Figure 1).

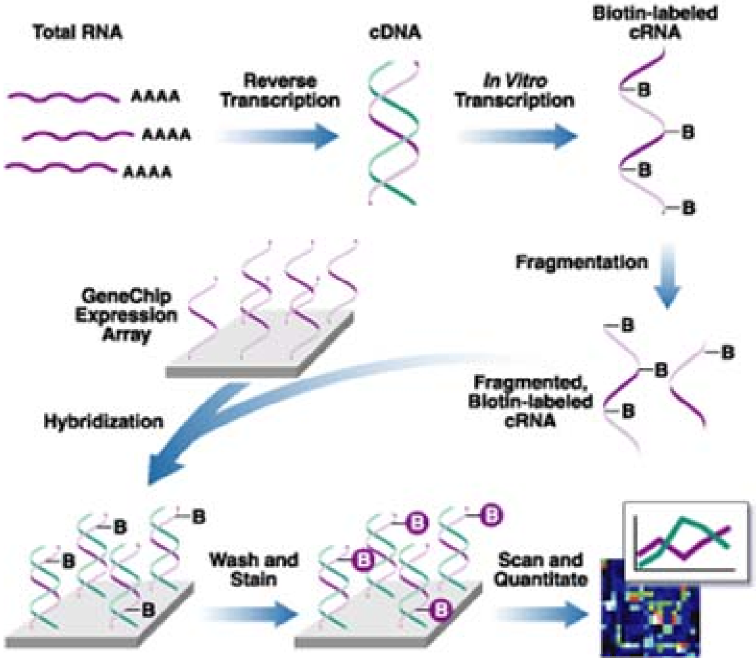
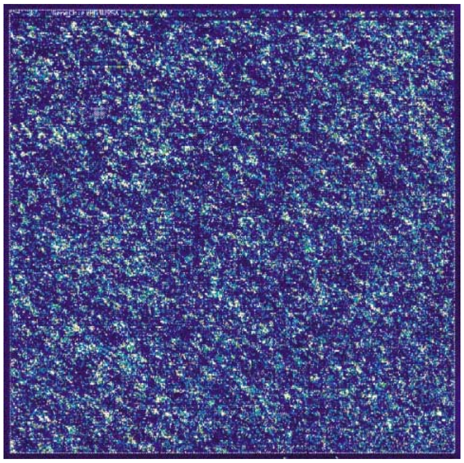


Figure 1. Standard eukaryotic gene expression assay. The basic concept behind the use of GeneChip microarrays for gene expression is simple: labeled cDNA or cRNA targets derived from the mRNA of an experimental sample are hybridized to nucleic acid probes attached to the solid support. By monitoring the amount of label associated with each DNA location, it is possible to infer the abundance of each mRNA species represented. Although hybridization has been used for decades to detect and quantify nucleic acids, the combination of the miniaturization of the technology and the large and growing amounts of sequence information, have enormously expanded the scale at which gene expression can be studied.

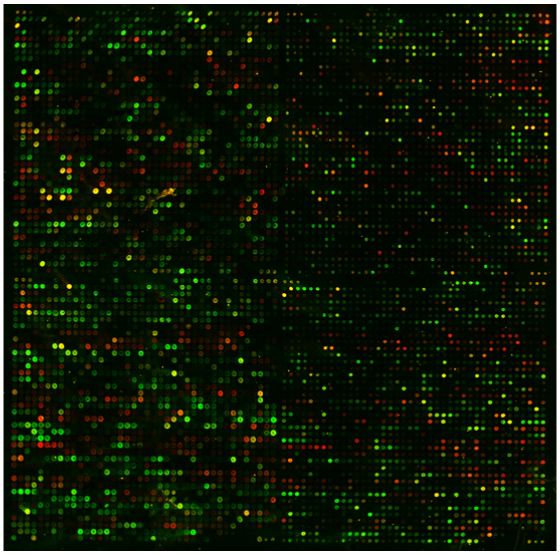
Global views of gene expression are often essential for obtaining comprehensive pictures of cell function. For example, it is estimated that between 0.2 to 10% of the 10,000 to 20,000 mRNA species in a typical mammalian cell are differentially expressed between cancer and normal tissues. Understanding the critical relative changes among all the genes in this set would be impossible without the use of whole-genome analysis. Whole-genome analyses also benefit studies where the end goal is to focus on small numbers of genes, by providing an efficient tool to sort through the activities of thousands of genes, and to recognize the key players. In addition, monitoring multiple genes in parallel allows the identification of robust classifiers, called "signatures", of disease. Often, these signatures are impossible to obtain from tracking changes in the expression of individual genes, which can be subtle or variable. Global analyses frequently provide insights into multiple facets of a project. A study designed to identify new disease classes, for example, may also reveal clues about the basic biology of disorders, and may suggest novel drug targets.”

http://www.bio.davidson.edu/courses/genomics/chip/chip.html



LEFT: Affymetrix GeneChip raw data

RIGHT: Actual data for a yeast gene expression microarray



IMPORTANT TO UNDERSTAND:

The yeast gene expression microarray above (with yellow, green & red spots) is an example of a comparison of gene expression between two conditions (in this case, yeast grown in the presence and absence of oxygen). This microarray would tell you about changes in gene expression during fermentation vs. oxidative respiration.

- Isolate mRNA from yeast grown aerobically; make cDNA and label RED
- Isolate mRNA from yeast grown anaerobically; make cDNA and label GREEN
- Wash BOTH cDNAs onto appropriate yeast microarray
- Analyze data

***Red spot** = this gene was expressed **ONLY** under aerobic conditions
***Green spot** = this gene was expressed **ONLY** under anaerobic conditions
***Yellow spot** = this gene was expressed under **BOTH** conditions
***Black spot** = **no** gene expression under either condition

---

If you are interested in how Affymetrix makes their GeneChips (proprietary name for Affymetrix product) using **photolithography**, you'll find an easy to read students' description at:

http://www.affymetrix.com/corporate/outreach/lesson_plan/downloads/student_manual_activities/activity3/activity3_manufacturing_background.pdf

Note that there is a competing method for microarray synthesis, pioneered by Stanford University. You can probably find information at their Stanford Microarray Database.

Nature Protocols

Systematic and Integrative Analysis of Large Gene Lists Using DAVID Bioinformatics Resources

Da Wei Huang[1], Brad T. Sherman[1], Richard A. Lempicki[*]

Laboratory of Immunopathogenesis and Bioinformatics, Clinical Services Program,
SAIC-Frederick, Inc., National Cancer Institute at Frederick, Frederick, MD 21702.

[1] These authors contributed equally to this study.

* Correspondence: Dr. Richard A Lempicki.

E-mail: rlempicki@mail.nih.gov
Ph. (301) 846-5093
Lab Web Site: http://david.niaid.nih.gov or http://david.abcc.ncifcrf.gov

Lab Fax: (301) 846-6762

E-mail Addresses for Authors:

Da Wei Huang huangdawei@mail.nih.gov
Brad T. Sherman bsherman@mail.nih.gov
Richard A Lempicki rlempicki@niaid.nih.gov

ABSTRACT

DAVID Bioinformatics Resources (DAVID) at http://david.abcc.ncifcrf.gov consists of an integrated biological knowledgebase and analytic tools aiming at systematically extracting biological meaning from large gene/protein lists. This protocol explains how to use DAVID, a high throughput and integrated data mining environment, to analyze gene lists derived from high throughput genomic experiments. The procedure first requires uploading a gene list containing any number of common gene identifiers followed by analysis using one or more text and pathway mining tools such as Gene Functional Classification, Functional Annotation Chart or Clustering, and Functional Annotation Table. By following this protocol, investigators are able to gain an in-depth understanding of the biological themes in lists of genes that are enriched in genome-scale studies.

SEARCH TERMS

Computational Biology; Bioinformatics; Genomics; Microarray data analysis; Bio-knowledge databases; Gene functional annotation; High throughput gene functional analysis**;** Gene functional classification.

INTRODUCTION

High-throughput genomic, proteomic and bioinformatics scanning approaches, such as, expression microarray, promoter microarray, proteomic data, and ChIP-on-CHIPs, provide significant capabilities to study a large variety of biological mechanisms including associations with diseases. These technologies usually result in a large 'interesting' gene list (ranging in size from hundreds to thousands of genes) involved in studied biological conditions. Data analysis of the large gene lists is a very important downstream task following the above example high throughput technologies in order to understand the biological meaning of the output gene lists. The data analysis of such high complex and large volume datasets is a challenging task, which requires support from special bioinformatics software packages. In this protocol, we introduce DAVID (the Database for Annotation, Visualization and Integrated Discovery) Bioinformatics Resources[1,2], which is able to extract biological features/meaning associated with large gene lists. DAVID is able to handle any type of gene list, no matter which genomic platform or software package generated them.

DAVID, released in 2003[2,3], as well as a number of other similar publicly available tools, including, but not limited to, GoMiner[4], GOstat[5], Onto-express[6], GoToolBox[7], FatiGO[8], GFINDer[9], GOBar[10], and GSEA[11], (See Supplementary Data 1 for a complete list), address various aspects of the challenge of functionally analyzing large gene lists. Although each tool has distinct features and strengths, as reviewed by Khatri et al.[12], they all adopt a common core strategy to systematically map a large number of interesting genes in a list to the associated biological annotation (e.g. Gene Ontology Terms), and then statistically highlight the most over-represented (enriched) biological annotation out of thousands of linked terms and contents. Enrichment analysis is a promising strategy that increases the likelihood for investigators to identify biological processes most pertinent to the biological phenomena under study.

The analysis of large gene lists is indeed more of an exploratory, computational procedure rather than a purely statistical solution. As compared to other similar services, DAVID provides some unique features and capabilities, such as, an integrated and expanded backend annotation database[13], advanced modular enrichment algorithms[14], and powerful exploratory ability in an integrated data mining environment[1]**.** Even though users can learn more in-depth information about DAVID algorithms in our original publications[1-3,13-15], we now briefly summarize the rationale regarding the key DAVID modules, as well as the analytic limitations (also see Table 1 for across comparisons), so that readers may be able to quickly follow the protocol.

Large Gene Lists Ready for Functional Analysis by DAVID

In this protocol, we use a previously published gene list[16] (Supplementary Data 2) as an example to illustrate the results obtained from the various DAVID analytic modules. To obtain this list, freshly isolated peripheral blood mononuclear cells (PBMCs) were treated with an HIV envelope protein (gp120) and genome-wide gene expression changes were observed using Affymetrix U95A microarray chips[16]. The aim of the experiment was to investigate cellular responses to viral envelope protein infection, which may help in understanding the mechanisms for HIV replication in resting or sub-optimally activated PBMCs.

The quality of large gene lists derived from high-throughput biological studies is one of the most important foundations that directly influences the success of the following functional analysis in DAVID. Due to the complexity of the data mining situations involved in biological studies, there is no good systematic way, at the present time, to quantitatively estimate the quality of the gene list ahead of time (i.e. before the gene functional analysis). However, based on real-life data analysis experiences during the past several years, a 'good' gene list may exhibit most, if not all of following characteristics:

1) Contain many important genes (marker genes) as expected for given study (e.g. IL8, CCL4, and TNFSF8 from the example gene list)

2) Reasonable number of genes ranging from hundreds to thousands (e.g. 100 to 2000 genes), not extremely low or high.

3) Most of the genes significantly pass the statistical threshold (e.g. selecting genes by comparing gene expression between control and experimental cells with t-test statistics: fold changes >=2 and P-values <=0.05) for selection. Importantly, statistical thresholds do not have to be sacrificed (e.g. fold changes >=1.1 and p-value<=0.2) in order to reach a comfortable gene size.

4) Notable portion of up/down-regulated genes are involved in certain interesting biological processes, rather than randomly spread throughout all possible biological processes.

5) A 'good' gene list should consistently contain more enriched biology than that of a random list in the same size range during analysis in DAVID (Supplementary Data 3 for detailed discussions).

6) High reproducibility (e.g. by independent experiments under the same conditions or by leave-one-out statistical test) to generate a similar gene list under the same conditions.

7) The high quality of the high-throughput data can be confirmed by other independent wet lab tests/experiments.

Some of the estimating points (2, 3, 6, & 7) come from upstream analysis while DAVID may help in examining others (1, 4 & 5).

Moreover, for enrichment analysis, in general, a larger gene list can have higher statistical power resulting in a higher sensitivity (more significant p-values) to slightly enriched terms, as well as to more specific terms. Otherwise, the sensitivity is decreased toward largely enriched terms and broader/general terms. Although the size of the gene list influences (in a non-linear way) the absolute enrichment p-values, which makes it difficult to directly compare the absolute enrichment p-values across gene lists, the enrichment p-values are fairly comparable within the same or same size of gene list. In addition, when different sizes of gene lists are generated from the same dataset with different threshold stringencies (within a reasonable range), the absolute enrichment p-values may vary from list to list. However, the relative rank/order of the enriched terms may remain fairly stable, which will lead to consistent global conclusions of functional annotations across the different sizes of gene lists derived from the same dataset (data not shown). This kind of reproducibility and consistency should be expected using DAVID tools if the underlying high-throughput biological studies are robust.

Interestingly, we found that many gene lists input to DAVID are in the size range of 1 to 10 genes. The enrichment statistic’s power will be very limited in such extreme cases. However, the unique exploratory capability of DAVID could still be very powerful for analyzing such small gene lists. Since the analysis is most likely in a very focused and small scope, analysts may take advantage of the unique exploratory capability of DAVID to navigate through all of the well organized heterogeneous annotation contents around the focused genes regardless of the statistics.

Submission of User's Gene Identifiers to DAVID

Comprehensively mapping a user's gene identifiers (gene IDs) to the relevant biological annotation in the DAVID database is an essential foundation for the success of any high-throughput gene functional analysis. Gene IDs and biological annotations are highly redundant within the vast array of public databases. The DAVID Knowledgebase[13], was designed to collect and integrate diverse gene identifiers as well as more than 40 well-known publicly available annotation categories (Supplementary Data 4), which are then centralized by internal DAVID identifiers in a non-redundant manner. The wide range of biological annotation coverage and the non-redundant integration of gene IDs in the DAVID Knowledgebase enables a user's gene ID to be mapped across the entire database, thus providing comprehensive coverage of gene-associated annotation. If a significant portion (>=20%) of input gene IDs fail to be mapped to an internal DAVID ID, a specially designed module, the DAVID Gene ID Conversion Tool[15], will start up in order to help map such IDs.

Principle of 'Gene Population Background' in enrichment analysis

The principle foundation of enrichment analysis is that if a biological process is abnormal in a given study, the co-functioning genes should have a higher potential (enriched) to be selected as a relevant group by the high-throughput screening technologies. To decide the degree of enrichment, a certain background must be set up in order to perform the comparison (also see Step 1 in Table 2). For example, 10% of the user's genes are kinases vs. 1% of the genes in the human genome (this is the gene population background) that are kinases. The enrichment can therefore be quantitatively measured by some common and well-known statistical methods including Chi-square, Fisher's exact test, Binomial probability, and Hypergeometric distribution. Thus, a conclusion may be obtained for the particular example, that is, kinases are enriched in the user's study, and therefore play important roles in the study. However, 10% alone cannot make such a conclusion without comparing it to the background information (i.e. 1%).

In this sense, the background is one of the critical factors that impact the conclusion to a certain degree, particularly when two ratios are close. There are many ways to set the backgrounds, e.g. all genome genes; genes on an Affymetrix chip; and a sub-set of genome genes that the user used in their study. In general, larger backgrounds, e.g. the total genes in the genome as a population background, intends to give more significant p-values, as compared to a narrowed-down set of genes as a population background, such as genes only existing on a microarray. Even though there is no gold standard for the population background, a general guideline is to set up the population background as the pool of genes which have a chance to be selected for the studied annotation category in the scope of the users' particular study.

One of the advantages of DAVID is its flexibility of setting different population backgrounds to meet different situations. DAVID has an automatic procedure to ‘guess’ the background as the global set of genes in the genome based on user's uploaded gene list. Thus, in a regular situation, users do not have to set up a population background by themselves. We found that it works generally well simply because most of the studies analyzed by DAVID are genome-wide or close to genome-wide studies. Moreover, other options are also available for user's choices including all genes in the studied genome, genes in various microarray chips, and most importantly any gene set that users define and upload. The latter feature requires significant computational power so that it is rarely found in similar web-based applications. In summary, various settings and options for population backgrounds can meet the range of needs of general users to those of power users.

DAVID Gene Name Batch Viewer

Gene IDs, such as Entrez Gene 3558, typically do not convey biological meaning in and of itself. The Gene Name Batch Viewer[1] is able to quickly attach meaning to a list of gene IDs by rapidly translating them into their corresponding gene names (Figure 3, Slide 4 of Supplementary Data 5 for more detail). Thus, before proceeding to analysis with other more comprehensive analytic tools, investigators can quickly glance at the gene names to further gain insight about their study and to answer questions such as, "Does my gene list contain important genes relevant to the study?". In addition, a set of hyperlinks are provided for each gene entry, allowing users to further explore additional functional information about each gene.

DAVID Gene Functional Classification

As the analysis proceeds, Gene Functional Classification[14] provides the distinct ability for investigators to explore and view functionally related genes together, as a unit, in order to concentrate on the larger biological network rather than at the level of an individual gene. In fact, the majority of co-functioning genes may have diversified names so that genes cannot be simply classified into functional groups according to their names. However, Gene Functional Classification, accomplished with a set of novel fuzzy clustering techniques, is able to classify input genes into functionally related gene groups (or classes) based on their annotation term co-occurrence rather than on gene names. Condensing large gene lists into biologically meaningful modules greatly improves one's ability to assimilate large amounts of information and thus switches functional annotation analysis from a gene-centric analysis to a biological module-centric analysis (Figure 4, Slide 5 and 6 of Supplementary Data 5 for more detail). Taken together with the 'drill-down' function associated with each biological module and visualizations to view the relationships between the many-genes-to-many-terms associations, investigators are able to more comprehensively understand how genes are associated with each other and with the functional annotation.

DAVID Functional Annotation Chart

Functional Annotation Chart[1-3] provides typical gene-term enrichment (over-represented) analysis, that is also provided by other similar tools, to identify the most relevant (over-represented) biological terms associated with a given gene list (Figure 5, Slide 8 of Supplementary Data 5 for more detail). Compared to other similar enrichment analysis tools, the notable difference of this function provided by DAVID is its extended annotation coverage[13], increasing from only GO in the original version of DAVID to currently over 40 annotation categories, including GO terms, protein–protein interactions, protein functional domains, disease associations, bio-pathways, sequence features, homology, gene functional summaries, gene tissue expression, and literature. (Supplementary Data 4). The annotation categories can be flexibly included or excluded from the analysis based upon a user’s choices (Slide 7 of Supplementary Data 5 for more detail). The enhanced annotation coverage alone increases the analytic power by allowing investigators to analyze their genes from many different biological aspects in a single space. In addition, to take full advantage of the well-known KEGG and BioCarta pathways, DAVID Pathway Viewer, which is accessed by clicking on pathway links within the chart report, can display genes from a user's list on pathway maps to facilitate biological interpretation in a network context (Figure 4). Finally, the choice of pre-built or user-defined gene population backgrounds provides the user with the ability to tailor the enrichment analysis to meet the user’s specific analytic situation.

DAVID Functional Annotation Clustering

Functional Annotation Clustering[14] uses a similar fuzzy clustering concept as Functional Classification by measuring relationships among the annotation terms based on the degree of their co-association with genes within the user's list in order to cluster somewhat heterogeneous, yet highly similar annotation into functional annotation groups (Figure 6, Slide 10 of Supplementary Data 5 for more detail). This reduces the burden of associating different terms associated with the similar biological process, thus allowing the biological interpretation to be more focused at the "biological module" level. The 2-D view tool is also provided for examining the internal relationships among the clustered terms and genes (Slide 6 of Supplementary Data 5). This type of grouping of functional annotation is able to give a more insightful view of the relationships between annotation categories and terms compared to the traditional linear list of enriched terms since highly related/redundant annotation terms may be dispersed among hundreds, if not thousands, of other terms.

DAVID Functional Annotation Table

Functional Annotation Table[1,2] is a query engine for the DAVID Knowledgebase, without statistical calculations (Figure 7, Slide 11 of Supplementary Data 5). For a given gene list, the tool can quickly query corresponding annotation for each gene and present them in a table format. Thus, users are able to explore annotation in a gene by gene manner. This is a useful analytic module particularly when users want to closely look at the annotation of highly interesting genes.

Purpose

This paper will mainly describe the protocol of how to use each DAVID analytic module in a logical, sequential order, as well as how to switch among the analytic modules (Figure 1). The example gene list used in this protocol (also available as demo list 2 on DAVID web site) allows new users to quickly test and experience various functions provided by DAVID. The protocol provides a routine analytic flow for new users to begin, as well as the flexibility for experienced users to use the modules in different combinations in order to balance the different focuses and strengths of each module to better meet specific analytical questions (Figure 1). Moreover, table 2 lists major statistical methods and filtering parameters that may influence the DAVID analysis and result interpretation in certain ways, for users to quickly look up specific statistical topics according to their interests.

MATERIALS

EQUIPMENT

A computer with high speed internet access and a web browser.

EQUIPMENT SETUP

Hardware requirements and computer configurations

DAVID is a web-based tool designed so that a computer with a standard web browser using default settings should work well. There is no need for special configuration and installation. Although DAVID was tested with several combinations of internet browsers and operating systems, MS Internet Explorer or Firefox in a Window XP operating system is recommended in order to obtain the most satisfactory usability.

Input Data

A list of gene identifiers is the only required input for all DAVID analytic modules/tools. The gene list may be derived from any type of high-throughput genomic, computational or proteomic study, such as DNA expression microarray, proteomics, CHIP-on-CHIP, SNP array, CHIP-sequence, etc. The format of the gene list to be uploaded is described throughout the web site, and is either one gene ID per line or a list of comma delimited gene IDs in one line (Supplementary Data 6). DAVID supports most common public gene identifiers[13] (see Supplementary Data 4). In addition, after the gene list is submitted to DAVID, all DAVID analytic modules can access the current list from the gene list manager so that there is no need to re-submit the gene list for each DAVID tool.

An example gene list derived from an HIV microarray study[16] is used in this protocol, as well as available as demo_list2 on DAVID web site. The HIV microarray study is briefly described in the introduction section. More detail can be found in the original publication[16].

Result Download

All results derived from DAVID may be explored and visualized on the web browser. Moreover, all results generated by DAVID can be downloaded in simple flat text formats, thereafter to be edited or plotted by other graphic tools, e.g. MS Excel, for publication purposes, as well as for archive purposes.

TIME TAKEN

The total analysis time varies, ranging from several minutes to hours, and is dependent on the analytical questions being addressed, the number of genes in the list being analyzed and the familiarity with the tools. It is not uncommon to make several visits to focus on different questions regarding a gene list of interest. Indeed computational time is only a small portion of the total time whereas exploring, interpreting and re-exploring both within DAVID and external to DAIVD tends to dominate most of the time. We used a PC computer with the Windows XP operating system, 2 G memory, 2.0 GHz CPU and 1Mbps internet connection for the data analysis of a gene list consisting of ~400 Affymetrix IDs (Supplementary Data 2) derived from an HIV study[16](presented in the Anticipated Results section). During the analysis course, for regular functional calls, each result was typically returned in ~10 seconds. For the most computationally intensive functions, such as Gene Functional Classification, results were typically returned within ~30 seconds, otherwise, never longer than 1 minute.

PROCEDURE

Submission of User's Gene IDs to DAVID

1| Submit a gene list to DAVID (Figure 2 & Slide 2 of Supplementary Data 5). Go to http://david.abcc.ncifcrf.gov or http://david.niaid.nih.gov and click on "Start Analysis" on the header. To do this, use the gene list manager panel that appears on the left side of the page (Figure 2) and perform the following steps: **(i)** Copy and paste a list of gene IDs into the box A or load a text file containing gene IDs to box B. See more details regarding format requirements in the Materials Section and also see Supplementary Data 6. **(ii)** Select the appropriate gene identifier type for your input gene IDs. See more details for supported ID types in the Materials Section**. (iii**) Indicate the list to be submitted as a gene list (i.e. genes to be analyzed) or as background genes (i.e. gene population background). **(iv)** Click the "Submit List" button.

CAUTION It takes ~30 seconds for a typical submission of ~1000 gene IDs; the progress bar, below the header, will disappear after a successful submission and a gene list name should appear in the list manager box; If >=20% input gene IDs cannot be recognized, the submission will be redirected to the DAVID Gene ID Conversion Tool[15] for further diagnosis. By default, the background is automatically set up as the genome-wide set of genes for the species that is found to have the majority of genes in the user's input list. However, it is always a good practice to double check the default, or select a more appropriate pre-built background through the "background" tab on top of the list manager.

2| Access DAVID analytic modules (Figure 2 & Slide 3 of Supplementary Data 5) via the tool menu page. The tool main menu is the central page which lists a set of hyperlinks leading to all available analytic modules. Clicking on each link will lead to the corresponding analytic module for analysis of your current gene list, highlighted in the gene list manager.

CRITICAL SETP: By clicking on "Start Analysis" on the header menu, users can always go back to this page at any time, no matter where they are, for choosing or switching to other analysis modules for current gene list.

Gene Name Batch Viewer

3| Run "Gene Name Batch Viewer" and explore results (Figure 3 & Slide 4 of Supplementary Data 5). Click on the "Gene Name Batch Viewer" link on the tool menu page. All the gene names will be listed by the Gene Name Batch Viewer. For a gene of interest, one or all of following options may be conducted:

(A) Click on the gene name to link to more detailed information.

(B) Click on "RG" (related genes) beside the gene name to search for other functionally related genes.

(C) Use the browser's "Find" function to search for particular items.

Gene Functional Classification

4| Run "Gene Functional Classification" and explore results (Figure 4 & Slide 5 of Supplementary Data 5). Get back to the tool menu page by clicking on "Start analysis" on the header. Click on "Gene Functional Classification Tool" to classify the input gene list into gene groups. For any gene groups of interest, one or all of following options may be conducted:

(A) Click on the gene name which leads to individual gene reports for in-depth information about the gene.

(B) Click on the red “T” (term reports) to list associated biology of the gene group.

(C) Click on “RG” (related genes) to list all genes functionally related to the particular gene group.

(D) Click on the “green icon” to invoke 2-D (gene-to-term) view.

(E) Create a new sub-gene list for further analysis on a subset of the genes.

TROUBLESHOOTING 2-D view is a Java Applet application that may take awhile to load for the first time; the 2-D view Java Applet may require you to accept the online security certificate.

CAUTION The input genes are classified using the default clustering stringency. Users may re-run the classification function leading to optimal results for the particular case by re-setting the stringency (high, medium or low) in the options on top of the result page.

Functional Annotation Chart

5| Run “Functional Annotation Chart” (Slide 7 of Supplementary Data 5). Go back to the tool menu page by clicking on “Start Analysis” on the header. **(i)** Click on “Functional Annotation Chart” to go the “Summary Page” of the tool suite. **(ii)** Choose functional annotation categories of your interest (Slide 7 of Supplementary Data 5): Accept 7 default functional annotation categories; Or expand the tree beside each main category (i.e. Main Accessions, Gene Ontology, etc) to select or deselect functional annotation categories of your interest. **(iii)** Click on the “Functional Annotation Chart” button on the bottom of the page leading to a chart report.

6| Explore the results of the "Functional Annotation Chart" (Figure 5 & Slide 8 of Supplementary Data 5). For an annotation term of interest, one or all of following options may be conducted:

(A) Click on the term name linking to a more detailed description.

(B) Click on “RT” (related terms) to list other related terms.

(C) Click on the “blue bar” to list all associated genes.

(D) Click on a pathway name to view genes on the pathway picture.

CAUTION By default, the order of the annotation terms is based on the EASE(enrichment) score. However, results can also be sorted by different values in the columns; The annotation terms with EASE score <= 0.1 are displayed in the results by default. The stringency of this filter (EASE score cutoff) may be set higher or lower through the options provided at the top of the report page in order to include more or less of the annotation terms.

Functional Annotation Clustering

7| Run “Functional Annotation Clustering” (Slide 10 of Supplementary Data 5). Go back to the tool menu page by clicking on “Start Analysis” on the header. **(i)** Click on “Functional Annotation Clustering” to go to the “Summary Page” of the tool suite. **(ii)** Select annotation categories as described in step 5. **(iii)** Click on the “Functional Annotation Clustering” button on the bottom of the page.

8| Explore the results of "Functional Annotation Clustering" (Figure 6 & Slide 10 of Supplementary Data 5). For an annotation term cluster of interest, one or all of following options may be conducted:

(A) Click on the term name linking to a more detailed description.

(B) Click on “RT” (related terms) to list other related terms.

(C) Click on the “blue bar” to list all associated genes of corresponding individual term.

(D) Click on the red “G” to list all associated genes of all terms within the cluster.

(E) Click on the “green icon” to display the 2-D (gene-to-term) view for all genes and terms within the cluster.

CAUTION The annotation terms are clustered using the default clustering stringency. Users may re-run the classification function leading to optimal results for the particular case by resetting the stringency (high, medium or low) in the options on top of the result page.

Functional Annotation Table

9| Run “Functional Annotation Table” (Slide 11 of Supplementary Data 5). Go back to the tool menu page by clicking on “Start Analysis” on the header and perform the following steps**:** **(i)** Click on “Functional Annotation Table” to go the “Summary Page” of the tool suite. **(ii)** Select annotation categories as described in step 5. **(iii)** Click on “Functional Annotation Table” button on the bottom of the page.

10| Explore the results of “Functional Annotation Table” (Figure 7 & Slide 11 of Supplementary Data 5). For a gene of your interest:

(A) Click on annotation terms for a detailed description.

(B) Click on “Related Genes” to search functionally related genes.

CAUTION When the output is too large to be displayed by internet browsers, only top 500 records are shown on the result page. However, full results are available to be downloaded as a tab delimited text file through the download link on the top of the result page.

TROUBLESHOOTING

Step	Problem	Possible Reason	Solution
1	Gene ID submission is stuck and I got the message "*You are either not sure which identifier type your list contains, or less than 80% of your list has mapped to your chosen identifier type. Please use the Gene Conversion Tool to determine the identifier type.*"	User knows the correct gene ID types, but selected wrong one that did not match the actual input IDs.	Go back to re-submit with correct selection of gene ID type; Or move forward with DAVID Gene ID Conversion tool to determine the potential gene ID type.
		User does not know the correct gene ID type corresponding to their gene list.	Submission panel in DAVID offers a special ID type, called "Not sure". Gene ID submission will be redirected to the DAVID Gene ID Conversion tool which has a mechanism to scan the entire ID system in DAVID to help you to determine the potential ID type(s) of your genes.
		There is more than one type of gene ID in the user's list	DAVID Gene ID Conversion Tool can help you determine the gene ID types and translate them to one single type.
		User's gene ids may contain a version number	Remove the version number since DAVID will not recognize them.
		>=20% of your gene IDs belong to low quality or retired IDs.	DAVID Gene ID Conversion Tool may help to identify the problem IDs. User should consider removing them from the gene list, or move forward to analysis ignoring the problem.
1, 3	The gene number that DAVID recognizes does not match the number in my gene list	Repeated IDs in user's list	DAVID ID submission will automatically remove redundancy.
		Particular ID(s) mapped to many different genes	DAVID ID Conversion Tool could help to identify the problem IDs. User should consider removing the 'bad' ID(s) from the gene list
		User's input gene IDs are gene symbol	Gene symbol is not species-specific so that one symbol may be mapped to many homologous genes across different species. You can define particular species matching your study, after the gene symbols are submitted, in the gene list manager.
3, 4, 5, 6,7, 8, 9, 10	Result page is blank or empty	30 minute timeout	If your web browser is inactive longer than 30 minutes, DAVID will clean up all information (your gene list, etc.) on the server side. Thus, you have to restart your analysis by resubmitting your gene list.
		The size of the gene list is too small	Enrichment or clustering algorithms are based on the survey between input genes against background genes. Thus, a reasonable size (e.g. >30) of input genes is required. Otherwise, certain algorithms will not work properly.
		The cutoff or stringency options are too high	Lower down the thresholds accordingly

		Wrong background selected	Background is automatically set up as the genome-wide set of genes corresponding to the species for the majority of genes in the gene list. Sometimes, the system may not choose the appropriate species. User may check and correct the appropriate background through the Gene List Manager.
		Small or minor species	Some small species may have very little annotation for the genes. There is nothing that can be done about this situation. Alternatively, you could map the genes to the homologous genes in a better annotated species.
4, 8	2-D view not displayed	Network certificate	Please accept it by clicking on "Accept". Basically, you are telling your browser to trust the DAVID application.
		Java plug-in not enabled	By default, most browsers should have the Java plug-in enabled. In case yours is not, please turn it on through Internet Options.
N/A	Service too slow	Slow computer and/or internet speed	Make sure that you have a reasonably good computer and internet speed. See recommendations in Material section.
		Gene list too large (>3,000)	Please be patient.
		DAVID server overwhelmed	The DAVID service may sometimes be slow due to too many large, simultaneous requests. We have monitoring programs to auto detect and fix the situation in a short time period. If the situation is not resolved in a reasonable amount of time, please report the problem to the DAVID team through the contact provided on the DAVID website.

ANTICIPATED RESULTS

We now submit the example gene list (~400 genes), derived from an HIV microarray study[16], to DAVID in order to illustrate the results obtained from various DAVID analytic modules. More detailed information and the availability regarding the gene list can be found in the Materials and Introduction sections of this protocol. Moreover, Supplementary Data 5 provides screen shots of each major step of the following analysis.

Submission of User's Gene IDs to DAVID

A successful submission of the gene list to DAVID is shown in Figure 2 (also Slide 2 & 3 of Supplementary Data 5). Users should see the progress bar under the header move through submission and disappear upon completion of the submission. The gene list manager panel on the left side, thereafter displays the list name (e.g. Upload_list_1) and corresponding species information (Home Sapiens [391]). The number (i.e. 391) appended after the species information is the number of genes that are recognized by DAVID. A set of hyperlinks on right side page lists the analytic modules available in the DAVID analytic pipeline. Users may follow the order of the pipeline to conduct analysis or jointly use analytic modules in varying combinations in order to meet the user's specific needs (Figure 1). Most importantly, the page serves as a central page (Figure 2) for users to choose analytic modules. Users may go back to this page at any time by clicking the "Start Analysis" button on the header in order to switch back and forth among analytic modules as needed.

Gene Name Batch Viewer

The corresponding gene names of input gene IDs were displayed as shown in Figure 3 and Slide 4 of Supplementary Data 5. Users may explore the gene names to examine whether there are any interesting study or marker genes in the list. Many immune related genes, containing names like "*interleukin*", "*chemokine*", "*kinase*" and "*tumor necrosis factor*" can be found in the list, which are consistent with that reported in the publicaton[16]. A set of hyperlinks provided for each gene can further lead to more detail information about a given gene. In addition, by clicking on the "RG" (related genes) search function beside a gene name, for example, "interleukin 8", all other functionally related chemokine genes (e.g. *cxcl 1, 2, 3, 4, 20*) will be listed so that users will be able to see other functionally similar genes in the list based on the bait gene.

Gene Functional Classification

The tool classified the example gene list(~400 genes) into 10 functional groups in an easily readable tabular format. An example output is illustrated in Figure 4 as well as in Slide 5 of Supplementary Data 5. Gene groups (with significant enrichment scores >=1), such as cytokines/chemokines (Group 1: 3.39), kinases (group 2: 2.21), clathrin membrane fusion genes (Group 3: 1.86), transcription factors (Group 6: 1.39), etc., can be easily identified. All of these gene groups are highly relevant to an HIV study and are therefore expected biological results[16]. Organizing the large gene list into gene groups allows investigators to quickly focus on the overall major common biology associated with a gene group rather than one gene at a time, thereby avoiding dilution of focus during the analysis due to too many single genes. Furthermore, the “2-D View” function associated with each group is able to display all related terms and genes in detail in one picture, in order to examine their inter-relationships. For example, for the kinase group (Group 2), a user who is not familiar with kinases may explore the terms of kinase activity, transferase activity, ATP-binding, nucleotide binding, protein metabolism, tyrosine specificity, serine/threonine specificity, regulation of G protein signaling, signal transduction, and so on in one view at the same time (Slide 6 of Supplementary Data 5). Therefore, we can quickly learn the biology for the kinase group, with the above related terms in a single view and also identify the fine differences among them. For example, there are two G-protein coupled receptor kinases, three protein tyrosine kinases and six kinases involved in cell surface receptor-linked signal transduction among the 23 kinases within the group. The fine details may be very important for pinpointing the key biology associated with a study.

Functional Annotation Chart

Over five hundred enriched (over-represented) biological terms were reported (Figure 5 and Slide 8 of Supplementary Data 5). Many of them are highly immune related, such as response to pathogenic bacteria, chemokine activity, cell migration, clathrin coated vesicle membrane, kinase activity, RNA polymerase II transcription factor activity, cell communication. This is consistent with observations previously identified by the other analytic modules, as well as meeting the expectation for the HIV study[16].

The report offers a lot of redundant details regarding the enriched biology associated with the gene list, which certainly helps the interpretation of the biology, but also may dilute the focus. Moreover, a set of hyperlinks provided for each term will lead to more details about each term, such as in-depth description, associated genes, other related terms, directed acyclic graph (DAG) of GO, etc. Notably, the pathway viewer module offers visualization of users' genes on enriched pathways. For example, "IL-10 Anti-inflammatory Signaling Pathway" was reported in the output. We can observe that IL10 was activated as an upstream immune regulator and was then further regulated by HO-1. As result, the IL1/TNFa/IL6 complex was activated leading to further downstream inflammatory responses (Figure 8). Thus, the inter-relationship of input genes was examined on the pathway in a network context.

Functional Annotation Clustering

The tool condensed the input gene list into smaller, much more organized biological annotation modules in a similar format (Figure 6 & Slide 10 of Supplementary Data 5) as that of Gene Annotation Clustering, but in a term-centric manner. Similarly, it allows investigators to focus on the annotation group level by quickly organizing many redundant/similar/hierarchical terms within the group. Annotation clusters, such as immune response, transcriptional regulation, chemokine activity, cytokine activity, kinase acitivity, signaling transduction, cell death, etc., could be found on the top of output as expected for this study[16]. The highly organized and simplified annotation results allow users to quickly focus on the major biology at an annotation cluster level instead of trying to derive the same conclusions by putting together pieces that are scattered throughout a list of hundreds of terms in a typical term enrichment analysis. In addition, the ‘G’ (genes) link provided for each cluster can comprehensively pool all related genes from different terms within the cluster. For example, each of the 7 terms within cluster 2 (inflammatory response cluster) associates with both overlapping as well as differing genes. Therefore, a pooled gene list brought together by cluster 2 regarding inflammatory response may be much more comprehensive, compared to the genes selected from one or a few individual terms.

Summary

Collectively, all of the DAVID analytic modules aim to extract biological meaning from the given gene list from different biological angles with highly consistent and expected results for a given study. Integration of the results from the different analytic modules (Figure 1) will take advantage of the different focus and strength of each module, in order to make the overall biological picture, assembled based on the gene list, more comprehensive and detailed. For a given gene list, DAVID Bioinformatics Resources is able to help users to (Figure 1 b):

- Convert gene IDs from one type to another
- Diagnose and fix problems with gene IDs
- Explore gene names in batch
- Discover enriched functionally-related gene groups
- Display relationship of many-genes-to-many-terms on 2-D view.
- Initial glance of major biological functions associated with my gene list
- Identify enriched (over-represented) annotation terms
- Visualize genes on BioCarta & KEGG pathway maps

- Link gene-disease associations
- Highlight protein functional domains and motifs
- Redirect to related literature
- List interacting proteins
- Cluster redundant and heterozygous annotation terms
- Search other functionally similar genes in genome, but not in my list
- Search other annotations functionally similar to one of my interest
- Read all annotation contents associated with a gene
- And more

ACKNOWLEDGMENTS

The authors are grateful to the referees for their constructive comments and thank Robert Stephens, David Bryant and David Liu in the ABCC group for web server support. Thanks also go to Xin Zheng and Jun Yang in the Laboratory of Immunopathogenesis and Bioinformatics (LIB) group for discussion. We also thank Bill Wilton and Mike Tartakovsky for information technology and network support. The project has been funded with federal funds from the National Institute of Allergy and Infectious Diseases (NIAID), National Institutes of Health (NIH), under Contract No. NO1-CO-56000. The annotation of this tool and publication do not necessarily reflect the views or policies of the Department of Health and Human Services, nor does mention of trade names, commercial products, or organizations imply endorsement by the United States Government.

REFERENCES

1. **Huang da, W. et al. DAVID Bioinformatics Resources: expanded annotation database and novel algorithms to better extract biology from large gene lists. *Nucleic Acids Res* 35, W169-75 (2007).**
2. **Dennis, G., Jr. et al. DAVID: Database for Annotation, Visualization, and Integrated Discovery. *Genome Biol* 4, P3 (2003).**
3. **Hosack, D.A., Dennis, G., Jr., Sherman, B.T., Lane, H.C. & Lempicki, R.A. Identifying biological themes within lists of genes with EASE. *Genome Biol* 4, R70 (2003).**
4. **Zeeberg, B.R. et al. High-Throughput GoMiner, an 'industrial-strength' integrative gene ontology tool for interpretation of multiple-microarray experiments, with application to studies of Common Variable Immune Deficiency (CVID). *BMC Bioinformatics* 6, 168 (2005).**
5. **Beissbarth, T. & Speed, T.P. GOstat: find statistically overrepresented Gene Ontologies within a group of genes. *Bioinformatics* 20, 1464-5 (2004).**
6. **Khatri, P., Bhavsar, P., Bawa, G. & Draghici, S. Onto-Tools: an ensemble of web-accessible, ontology-based tools for the functional design and interpretation of high-throughput gene expression experiments. *Nucleic Acids Res* 32, W449-56 (2004).**
7. **Martin, D. et al. GOToolBox: functional analysis of gene datasets based on Gene Ontology. *Genome Biol* 5, R101 (2004).**
8. **Al-Shahrour, F., Diaz-Uriarte, R. & Dopazo, J. FatiGO: a web tool for finding significant associations of Gene Ontology terms with groups of genes. *Bioinformatics* 20, 578-80 (2004).**
9. **Masseroli, M., Galati, O. & Pinciroli, F. GFINDer: genetic disease and phenotype location statistical analysis and mining of dynamically annotated gene lists. *Nucleic Acids Res* 33, W717-23 (2005).**
10. **Lee, J.S., Katari, G. & Sachidanandam, R. GObar: a gene ontology based analysis and visualization tool for gene sets. *BMC Bioinformatics* 6, 189 (2005).**
11. **Subramanian, A. et al. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proc Natl Acad Sci U S A* 102, 15545-50 (2005).**
12. **Khatri, P. & Draghici, S. Ontological analysis of gene expression data: current tools, limitations, and open problems. *Bioinformatics* 21, 3587-95 (2005).**
13. **Sherman, B.T. et al. DAVID Knowledgebase: a gene-centered database integrating heterogeneous gene annotation resources to facilitate high-throughput gene functional analysis. *BMC Bioinformatics* 8, 426 (2007).**
14. **Huang da, W. et al. The DAVID Gene Functional Classification Tool: a novel biological module-centric algorithm to functionally analyze large gene lists. *Genome Biol* 8, R183 (2007).**
15. **Huang, D.W., Sherman, B.T. & Lempicki, R.A. DAVID gene ID conversion tool. *Bioinformation* 2, 428-430 (2008).**

16. Cicala, C. et al. HIV envelope induces a cascade of cell signals in non-proliferating target cells that favor virus replication. *Proc Natl Acad Sci U S A* 99, 9380-5 (2002).

TABLE

Table 1: Side-by-side comparisons of DAVID's major analytic modules

	Input user's large gene lists				
DAVID Analytic Module/Tool	**Gene Name Batch Viewer**[1]	**Gene Functional Classification**[14]	**Functional Annotation Chart**[1,2]	**Functional Annotation Clustering**[14]	**Functional Annotation Table**[1,2]
Brief definition /explanation	List names of user's genes	Classify user's genes into gene groups	Identify enriched annotation terms associated with user's gene list	Cluster functionally similar terms associated with user's gene list into groups	Query associated terms for all user's genes
Key Points	Gene-centric singular exploration	Gene-centric modular analysis	Term-centric singular enrichment analysis (typical enrichment analysis)	Term-centric modular enrichment analysis	Large scale query
Example question to ask	What are the genes in my list?	What major gene family in my list?	What are enriched annotation terms for my gene list?	What are enriched annotation groups for my gene list?	What are all selected annotations for my genes?
Main Functions	• Display all gene names in a linear tabular text format • Deep links to more information around given gene • Search other functionally related genes	• Classify functionally related genes into groups • 2-D view for related gene-term relationship • Rank importance of gene groups with enrichment score • Highlight annotation terms for gene groups	• Identify enriched annotation terms in a linear tabular text format • Deep links to more information around terms and associated genes • Search other functionally related terms • View genes on pathway maps	• Cluster functionally related annotations into groups • 2-D view for related gene-term relationship • Rank importance of annotation groups with enrichment score • Pool genes for annotation groups	Query selected annotations for given genes.
Advantages	• Roughly explore genes one-by-one • Quickly check if the expected/important genes are in the list • Quickly learn annotation about genes of interests • All genes included in the analysis	• Explore genes group-by-group rather than singular gene one-by-one • Highlight important gene groups by enrichment scores • Study functionally related genes and their relationship	Simple format to explore all singular enriched terms	• Explore annotations group-by-group rather than singular term one-by-one. • Highlight important annotation groups by enrichment scores • Study	• Quickly Explore all annotations (both enrich and not enriched ones) for given genes • Good for analysis of small # of focused genes • Save entire annotation profile of given gene list in text file ready for other external

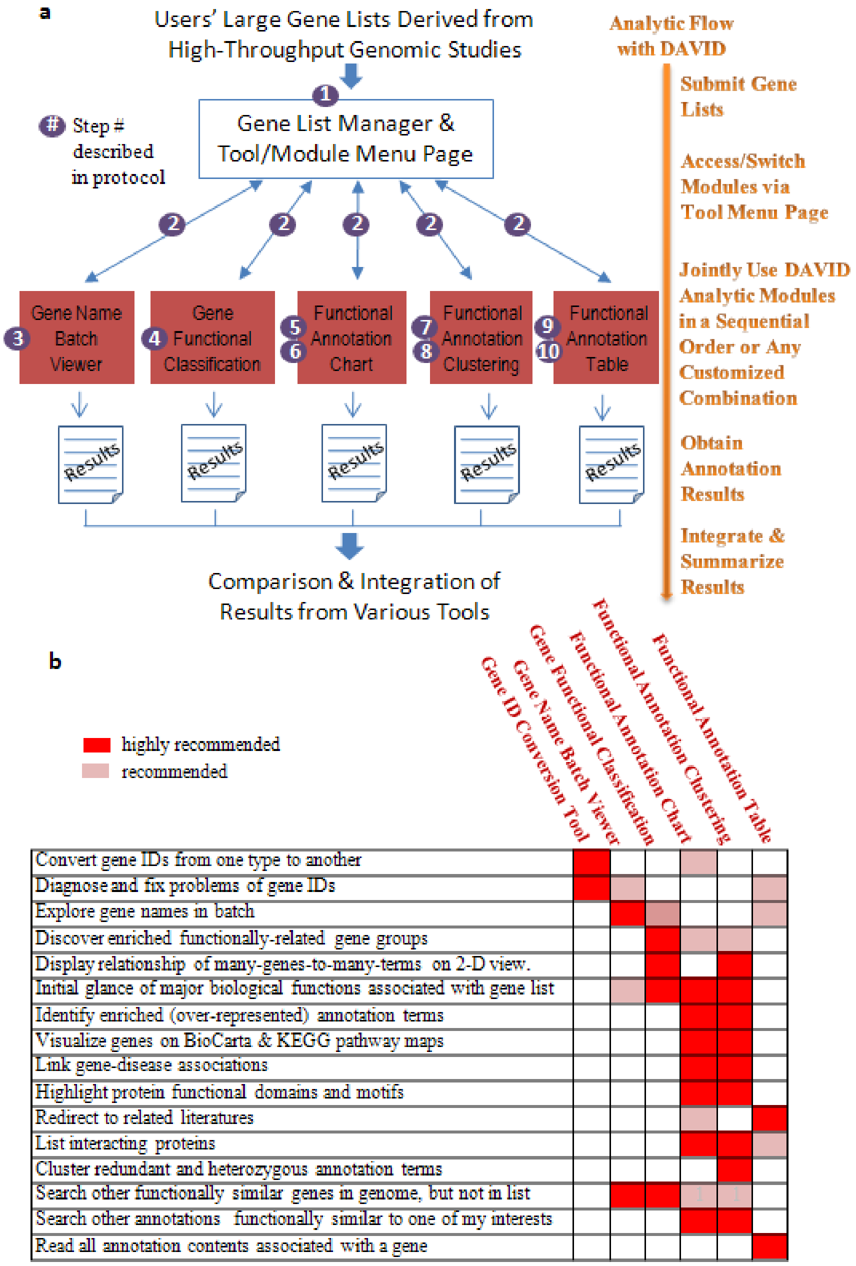
		in a network format • Good to catch major biology		functionally related genes and their relationship in a network format • Good to focus on major and fine biology	analysis
Drawbacks	• Related genes scattered in the results losing inter-relationships of genes during exploration • Difficult to judge enriched genes or noisy genes without enrichment calculation	Some genes without strong neighbors will be left out from the analysis	• Related/redundant terms scattered in the results • Some fine biology could be diluted by the redundancy • Lack of term-term relationships during analysis	Some enriched terms without strong neighbors will be left out from the analysis	• Difficult to explore large gene list. • No enrichment analysis

Table 2: Major statistical methods and associated parameters used in DAVID.

Step #	Module/Page	Statistics /Parameters	Explanation/Definition	How to Understand the Value
1	Submission of User's Gene IDs	Background genes (or called population genes)	To decide the degree of enrichment, a certain background must be set up in order to be compared to the user's gene list. For example, 10% of user's genes are kinases vs. 1% of genes in human genome (this is population background) are kinases. Thus, the conclusion is obvious in the particular example that the user's study is highly related to kinase. However, 10% itself alone cannot provide such a conclusion without comparing it to the background information.	• A general guideline is to set up the reference background as the pool of genes which have a chance to be selected for the studied annotation category under the scope of users' particular study. • Default background is the entire genome-wide genes of the species matching the user's input IDs. Pre-built backgrounds such as genes in Affy chips, etc. are available for the user's choice. • In principle, a larger gene background tends to give smaller p-values. Since most of the high-throughput studies are, or at least are close to, genome-wide scope, the default background is good for regular cases in general.
4	Gene Functional Classification [1,14]	Classification Stringency	To control the behavior of DAVID Fuzzy clustering.	• A general guideline is to choose higher stringency settings for tight, clean and smaller numbers of clusters; Otherwise lower for loose , broader and larger numbers of clusters. • Default setting is medium. • Five pre-defined levels from lowest to highest for user's choices. • Users may want to play with different stringency for more satisfactory results.
		Enrichment Score (for each group)	To rank overall importance (enrichment) of gene groups. It is the geometric mean of all the enrichment p-values (EASE scores) for each annotation term associated with the gene members in the group. To emphasize that the geometric mean is a relative score instead of an absolute *p* value, minus log transformation is applied on the average p-values	• A higher score for a group indicates that the gene members in the group are involved in more important (enriched) terms in a given study, therefore more attention should go to them. • Enrichment score of 1.3 is equivalent to non-log scale 0.05. thus, more attention should go to groups with scores >= 1.3 • However, the gene groups with lower scores could be potentially interesting, and should be explored as well if possible.
6	Functional Annotation Chart[1,2]	P-value (or called EASE Score)	To examine the significance of gene-term enrichment with a modified Fisher's Exact Test (EASE Score). For example, 10% of user's genes are kinases vs. 1% of genes in human genome (this is population background) are kinases. Thus, the EASE score is <0.05 which suggests that kinases are significantly more enriched than random chance in the study for this particular example.	• The smaller the p-values, the more significant they are. • Default cutoff is 0.1 • Users could set different levels of cutoff through option panel on the top of result page. • Due to the complexity of biological data mining of this type, p-values are suggested to be treated as score systems, i.e. suggesting roles, rather than decision making roles. Users themselves should play critical roles in judging "are the results making sense or not for expected biology".
		Benjamini	To globally correct enrichment p-values in order to control family-wide false discovery rate under certain rate (e.g. <= 0.05). It is one of the multiple testing correction techniques (Bonferroni, Benjamini and FDR) provided by DAVID.	• More terms examined, more conservative the corrections are. As a result, all the p-values get larger. • It is great if the interesting terms have significant p-values after the corrections. But since the multiple testing correction techniques are known as conservative approaches, it could hurt the sensitivity of discovery if over emphasizing them. Users' judgment could be critical as discussed in EASE Score in Functional Annotation Chart section.
		Fold Enrichment	To measure the magnitude of enrichment. For example, 10% of user's genes are kinases vs. 1% of genes in human genome (this is population background) are kinases. Thus, the fold enrichment is 10 fold. Fold enrichment	• Fold enrichment 1.5 and above are suggested to be considered as interesting. • Fold enrichment and EASE Score should be always examined side-by-side. Terms with larger fold enrichments and smaller may be interesting. • Caution should be taken when big fold enrichment are

			along with EASE Score could rank the enriched terms in a more comprehensive way.	obtained from a small number of genes (e.g. <=3). This situation often happens to the terms with a few genes (more specific terms) orfsmaller size (e.g. <100) of user's input gene list. In this case, the reliability is not as much as those fold enrichment scores obtained from larger numbers of genes.
		%	# of genes involved in given term is divided by the total # of user's input genes, i.e. Percentage of user's input gene hitting a given term. For example, 10% of user's genes hit "kinase activity".	- It gives overall idea of gene distributions among the terms. - The higher percentage does not necessarily have a good EASE Score because it also depends on the percentage of background genes as discussed in the EASE Score in Functional Annotation Chart section.
8	Functional Annotation Clustering[1,14]	Classification Stringency	To control the behalvior of DAVID Fuzzy clustering.	- A general guideline is to choose higher stringency setting for tight, clean and smaller numbers of clusters; Otherwise lower for looser , broader and larger numbers of clusters. - Default setting is medium. - Five pre-defined levels from lowest to highest for user's choices. - Users may want to play with different stringency to obtain more satisfactory results.
		Enrichment Score (for each group)	To rank overall importance (enrichment) of annotation term groups. It is the geometric mean of all the enrichment p-values (EASE scores) of each annotation term in the group. To emphasize that the geometric mean is a relative score instead of an absolute *p* value, minus log transformation is applied on the average p-values	- A higher score for a group indicates that annotation term members in the group are playing more important (enriched) roles in given study, therefore pay more attention toward them. - Enrichment score 1.3 is equivalent to non-log scale 0.05. thus, more attention should go to groups with scores >= 1.3 - However, the annotation groups with lower scores could be potentially interesting, and should be explored as well if possible.
		P-value (or called EASE Score) (for individual term members)	To examine the significance of gene-term enrichment with a modified Fisher's Exact Test (EASE Score). This p-value is calculated in exactly the same way as in the Functional Annotation Chart section.	The explanation is the same as that in Functional Annotation Chart section.
		Benjamini	To globally correct enrichment p-values of individual term members. The idea and calculations are exactly the same as that in Functional Annotation Chart section.	The explanation is the same as that in the Functional Annotation Chart section.

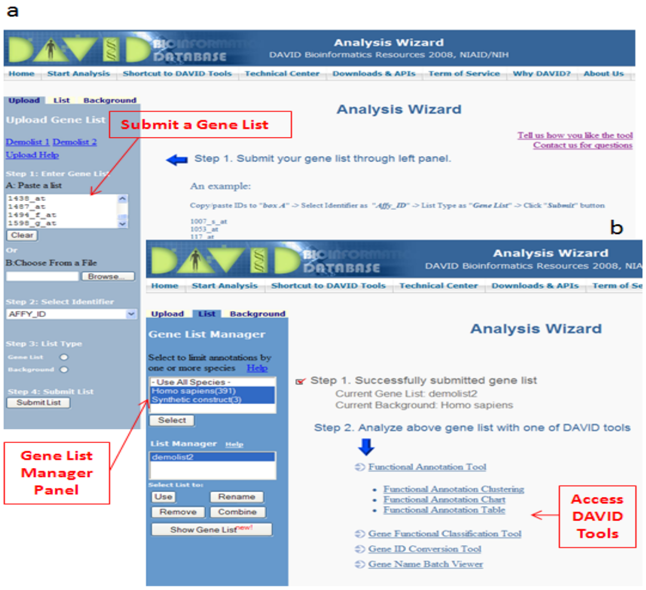
FIGURES


a
Users' Large Gene Lists Derived from High-Throughput Genomic Studies
1
Gene List Manager & Tool/Module Menu Page
# Step # described in protocol
2
2
2
2
2
3 Gene Name Batch Viewer
4 Gene Functional Classification
5 6 Functional Annotation Chart
7 8 Functional Annotation Clustering
9 10 Functional Annotation Table
Results
Results
Results
Results
Results
Comparison & Integration of Results from Various Tools
Analytic Flow with DAVID
Submit Gene Lists
Access/Switch Modules via Tool Menu Page
Jointly Use DAVID Analytic Modules in a Sequential Order or Any Customized Combination
Obtain Annotation Results
Integrate & Summarize Results
b

🟥 highly recommended
▨ recommended

	Gene ID Conversion Tool	Gene Name Batch Viewer	Gene Functional Classification	Functional Annotation Chart	Functional Annotation Clustering	Functional Annotation Table
Convert gene IDs from one type to another	🟥			▨		
Diagnose and fix problems of gene IDs	🟥	▨				▨
Explore gene names in batch		🟥	▨			▨
Discover enriched functionally-related gene groups			🟥	▨	▨	
Display relationship of many-genes-to-many-terms on 2-D view.			🟥		🟥	
Initial glance of major biological functions associated with gene list		▨	🟥	🟥	🟥	
Identify enriched (over-represented) annotation terms				🟥	🟥	
Visualize genes on BioCarta & KEGG pathway maps				🟥	🟥	
Link gene-disease associations				🟥	🟥	
Highlight protein functional domains and motifs				🟥	🟥	
Redirect to related literatures				▨		🟥
List interacting proteins				🟥	🟥	▨
Cluster redundant and heterozygous annotation terms					🟥	
Search other functionally similar genes in genome, but not in list		🟥	🟥	▨	▨	
Search other annotations functionally similar to one of my interests				🟥	🟥	
Read all annotation contents associated with a gene						🟥

Figure 1. Analytic tools/modules in DAVID. a. After the user's gene list is submitted to DAVID, the gene list manager may be accessed by all DAVID analytic modules (red boxes) at any time. The circled numbers indicate step numbers described in the procedure section to facilitate reading. **b.** DAVID analytic modules, each having different strengths and focus, can be used independently or jointly. A roadmap to help users to choose some or all DAVID analytic modules for the analysis of large gene lists.



```
1438_at
1487_at
1494_f_at
1598_g_at
```

Step 1. Submit your gene list through left panel.

An example:

Copy/paste IDs to "*box A*" -> Select Identifier as "*Affy_ID*" -> List Type as "*Gene List*" -> Click "*Submit*" button

1007_s_at
1053_at
117_at

- Use All Species -
Homo sapiens(391)
Synthetic construct(3)

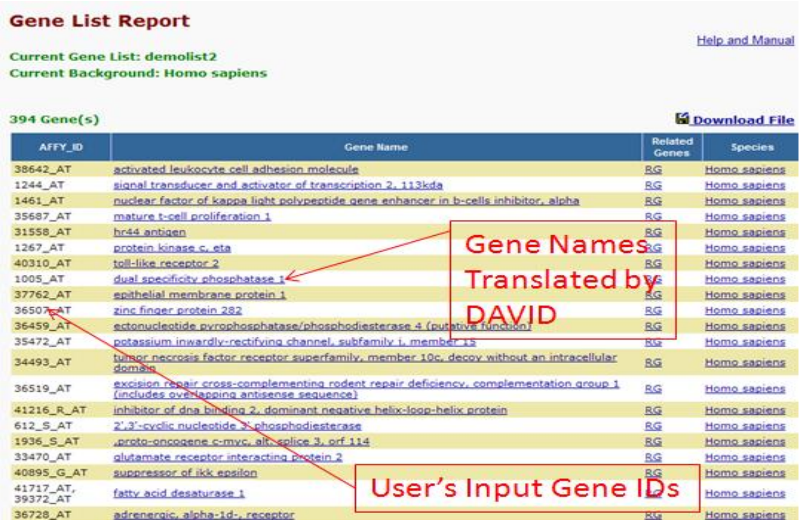
demolist2

Step 1. Successfully submitted gene list
Current Gene List: demolist2
Current Background: Homo sapiens

Step 2. Analyze above gene list with one of DAVID tools

- Functional Annotation Tool
  - Functional Annotation Clustering
  - Functional Annotation Chart
  - Functional Annotation Table
- Gene Functional Classification Tool
- Gene ID Conversion Tool
- Gene Name Batch Viewer

Figure 2. Submit a gene list to DAVID and access various analytic tools/modules. **a.** Following the example input format and steps on the left side uploading panel, a list of genes may be uploaded into DAVID. **b.** After successfully uploading a gene list(s), a set of analytic modules are available for the analysis of the current gene list highlighted in the gene list manager on the left side. Importantly, users may go to this page at any time by clicking "Start Analysis" on the header in order to access any analytic tool of interest.



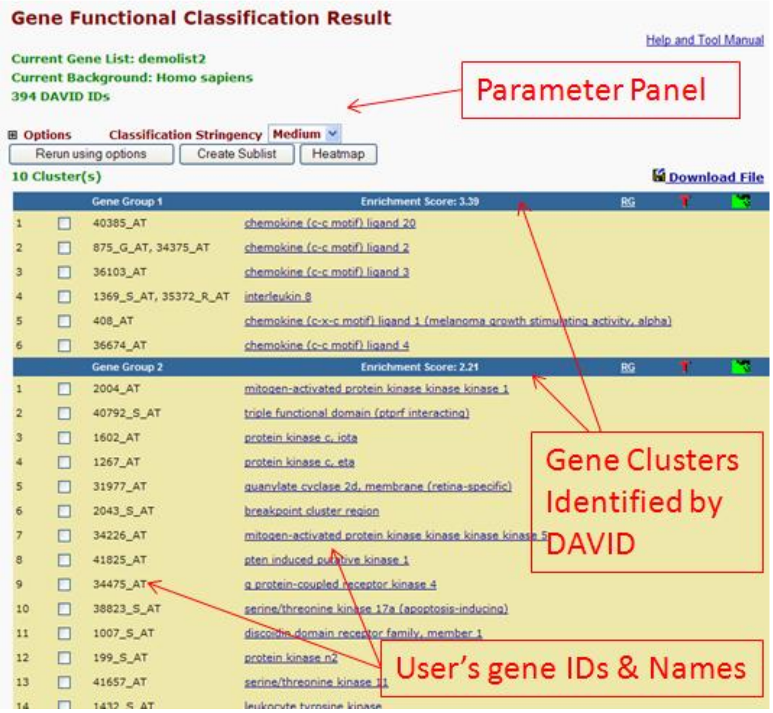
Gene List Report

Current Gene List: demolist2
Current Background: Homo sapiens

394 Gene(s)

AFFY_ID	Gene Name	Related Genes	Species
38642_AT	activated leukocyte cell adhesion molecule	RG	Homo sapiens
1244_AT	signal transducer and activator of transcription 2, 113kda	RG	Homo sapiens
1461_AT	nuclear factor of kappa light polypeptide gene enhancer in b-cells inhibitor, alpha	RG	Homo sapiens
35687_AT	mature t-cell proliferation 1	RG	Homo sapiens
31558_AT	hr44 antigen	RG	Homo sapiens
1267_AT	protein kinase c, eta	RG	Homo sapiens
40310_AT	toll-like receptor 2	RG	Homo sapiens
1005_AT	dual specificity phosphatase 1	RG	Homo sapiens
37762_AT	epithelial membrane protein 1	RG	Homo sapiens
36507_AT	zinc finger protein 282	RG	Homo sapiens
36459_AT	ectonucleotide pyrophosphatase/phosphodiesterase 4 (putative function)	RG	Homo sapiens
35472_AT	potassium inwardly-rectifying channel, subfamily j, member 15	RG	Homo sapiens
34493_AT	tumor necrosis factor receptor superfamily, member 10c, decoy without an intracellular domain	RG	Homo sapiens
36519_AT	excision repair cross-complementing rodent repair deficiency, complementation group 1 (includes overlapping antisense sequence)	RG	Homo sapiens
41216_R_AT	inhibitor of dna binding 2, dominant negative helix-loop-helix protein	RG	Homo sapiens
612_S_AT	2',3'-cyclic nucleotide 3' phosphodiesterase	RG	Homo sapiens
1936_S_AT	,proto-oncogene c-myc, alt. splice 3, orf 114	RG	Homo sapiens
33470_AT	glutamate receptor interacting protein 2	RG	Homo sapiens
40895_G_AT	suppressor of ikk epsilon	RG	Homo sapiens
41717_AT, 39372_AT	fatty acid desaturase 1	RG	Homo sapiens
36728_AT	adrenergic, alpha-1d-, receptor	RG	Homo sapiens

Figure 3. **An example layout of DAVID Gene Name Batch Viewer.** User's input gene IDs are translated into meaningful and readable gene names. The link on each gene name can lead to more in-depth information.



Gene Functional Classification Result

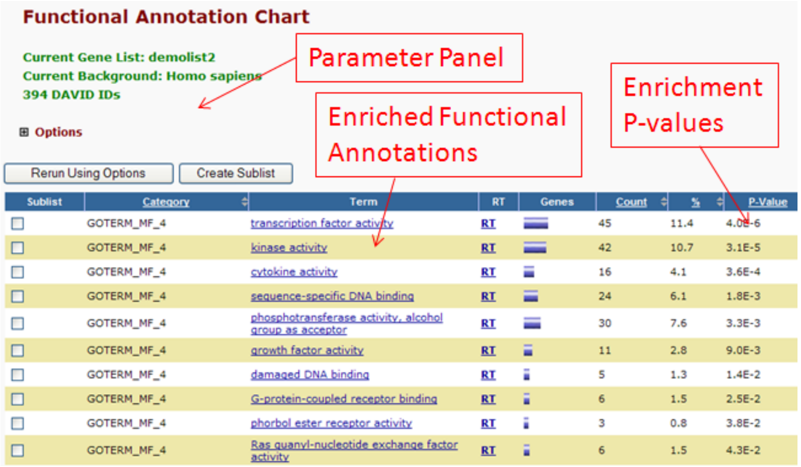
Current Gene List: demolist2
Current Background: Homo sapiens
394 DAVID IDs

⊞ Options Classification Stringency Medium

10 Cluster(s)

		Gene Group 1	Enrichment Score: 3.39	RG
1	□	40385_AT	chemokine (c-c motif) ligand 20	
2	□	875_G_AT, 34375_AT	chemokine (c-c motif) ligand 2	
3	□	36103_AT	chemokine (c-c motif) ligand 3	
4	□	1369_S_AT, 35372_R_AT	interleukin 8	
5	□	408_AT	chemokine (c-x-c motif) ligand 1 (melanoma growth stimulating activity, alpha)	
6	□	36674_AT	chemokine (c-c motif) ligand 4	
		Gene Group 2	Enrichment Score: 2.21	RG
1	□	2004_AT	mitogen-activated protein kinase kinase kinase 1	
2	□	40792_S_AT	triple functional domain (ptprf interacting)	
3	□	1602_AT	protein kinase c, iota	
4	□	1267_AT	protein kinase c, eta	
5	□	31977_AT	guanylate cyclase 2d, membrane (retina-specific)	
6	□	2043_S_AT	breakpoint cluster region	
7	□	34226_AT	mitogen-activated protein kinase kinase kinase kinase 5	
8	□	41825_AT	pten induced putative kinase 1	
9	□	34475_AT	g protein-coupled receptor kinase 4	
10	□	38823_S_AT	serine/threonine kinase 17a (apoptosis-inducing)	
11	□	1007_S_AT	discoidin domain receptor family, member 1	
12	□	199_S_AT	protein kinase n2	
13	□	41657_AT	serine/threonine kinase 11	
14	□	1432_S_AT	leukocyte tyrosine kinase	

Figure 4. An example layout of DAVID Gene Functional Classification. User's genes were organized and condensed into several functional groups. The gene members in each group share common biological functions. A set of accessory tools provided for each group will further facilitate the 'drill-down' analysis of biological inter-relationships among the gene members within the same group.



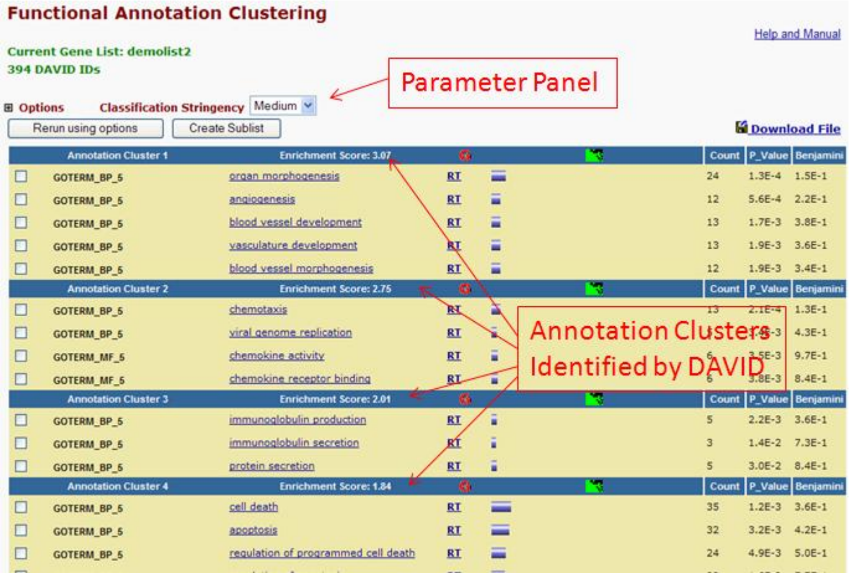
Functional Annotation Chart

Current Gene List: demolist2
Current Background: Homo sapiens
394 DAVID IDs

⊞ Options

Sublist	Category	Term	RT	Genes	Count	%	P-Value
□	GOTERM_MF_4	transcription factor activity	RT		45	11.4	4.0E-6
□	GOTERM_MF_4	kinase activity	RT		42	10.7	3.1E-5
□	GOTERM_MF_4	cytokine activity	RT		16	4.1	3.6E-4
□	GOTERM_MF_4	sequence-specific DNA binding	RT		24	6.1	1.8E-3
□	GOTERM_MF_4	phosphotransferase activity, alcohol group as acceptor	RT		30	7.6	3.3E-3
□	GOTERM_MF_4	growth factor activity	RT		11	2.8	9.0E-3
□	GOTERM_MF_4	damaged DNA binding	RT		5	1.3	1.4E-2
□	GOTERM_MF_4	G-protein-coupled receptor binding	RT		6	1.5	2.5E-2
□	GOTERM_MF_4	phorbol ester receptor activity	RT		3	0.8	3.8E-2
□	GOTERM_MF_4	Ras guanyl-nucleotide exchange factor activity	RT		6	1.5	4.3E-2

Figure 5. An example layout of DAVID Annotation Chart. The enriched functional annotation terms associated with user's gene list are identified and listed according to their enrichment p-value by DAVID. The links on the page can lead to various detail information regarding corresponding items.



Functional Annotation Clustering

Help and Manual

Current Gene List: demolist2
394 DAVID IDs

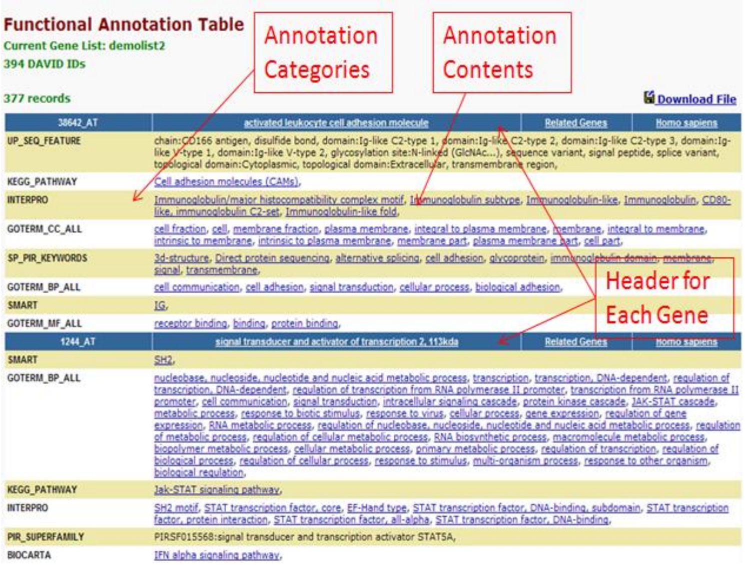
⊞ Options   Classification Stringency: Medium

Rerun using options   Create Sublist

Download File

	Annotation Cluster 1	Enrichment Score: 3.07			Count	P_Value	Benjamini
□	GOTERM_BP_5	organ morphogenesis	RT	▬	24	1.3E-4	1.5E-1
□	GOTERM_BP_5	angiogenesis	RT	▬	12	5.6E-4	2.2E-1
□	GOTERM_BP_5	blood vessel development	RT	▬	13	1.7E-3	3.8E-1
□	GOTERM_BP_5	vasculature development	RT	▬	13	1.9E-3	3.6E-1
□	GOTERM_BP_5	blood vessel morphogenesis	RT	▬	12	1.9E-3	3.4E-1
	Annotation Cluster 2	Enrichment Score: 2.75			Count	P_Value	Benjamini
□	GOTERM_BP_5	chemotaxis	RT	▬	13	2.1E-4	1.3E-1
□	GOTERM_BP_5	viral genome replication	RT	▬	[illegible]	[illegible]	4.3E-1
□	GOTERM_MF_5	chemokine activity	RT	▬	6	3.5E-3	9.7E-1
□	GOTERM_MF_5	chemokine receptor binding	RT	▬	6	3.8E-3	8.4E-1
	Annotation Cluster 3	Enrichment Score: 2.01			Count	P_Value	Benjamini
□	GOTERM_BP_5	immunoglobulin production	RT	▬	5	2.2E-3	3.6E-1
□	GOTERM_BP_5	immunoglobulin secretion	RT	▬	3	1.4E-2	7.3E-1
□	GOTERM_BP_5	protein secretion	RT	▬	5	3.0E-2	8.4E-1
	Annotation Cluster 4	Enrichment Score: 1.84			Count	P_Value	Benjamini
□	GOTERM_BP_5	cell death	RT	▬	35	1.2E-3	3.6E-1
□	GOTERM_BP_5	apoptosis	RT	▬	32	3.2E-3	4.2E-1
□	GOTERM_BP_5	regulation of programmed cell death	RT	▬	24	4.9E-3	5.0E-1

Figure 6. An example layout of DAVID Functional Annotation Clustering. The similar annotation terms are grouped into clusters so that user can read through the important terms in a way of block-by-block instead of individual-by-individual.



Functional Annotation Table

Current Gene List: demolist2
394 DAVID IDs

377 records

Download File

38642_AT	activated leukocyte cell adhesion molecule	Related Genes	Homo sapiens
UP_SEQ_FEATURE	chain:CD166 antigen, disulfide bond, domain:Ig-like C2-type 1, domain:Ig-like C2-type 2, domain:Ig-like C2-type 3, domain:Ig-like V-type 1, domain:Ig-like V-type 2, glycosylation site:N-linked (GlcNAc...), sequence variant, signal peptide, splice variant, topological domain:Cytoplasmic, topological domain:Extracellular, transmembrane region,		
KEGG_PATHWAY	Cell adhesion molecules (CAMs),		
INTERPRO	Immunoglobulin/major histocompatibility complex motif, Immunoglobulin subtype, Immunoglobulin-like, Immunoglobulin, CD80-like, immunoglobulin C2-set, Immunoglobulin-like fold,		
GOTERM_CC_ALL	cell fraction, cell, membrane fraction, plasma membrane, integral to plasma membrane, membrane, integral to membrane, intrinsic to membrane, intrinsic to plasma membrane, membrane part, plasma membrane part, cell part,		
SP_PIR_KEYWORDS	3d-structure, Direct protein sequencing, alternative splicing, cell adhesion, glycoprotein, immunoglobulin domain, membrane, signal, transmembrane,		
GOTERM_BP_ALL	cell communication, cell adhesion, signal transduction, cellular process, biological adhesion,		
SMART	IG,		
GOTERM_MF_ALL	receptor binding, binding, protein binding,		
1244_AT	signal transducer and activator of transcription 2, 113kda	Related Genes	Homo sapiens
SMART	SH2,		
GOTERM_BP_ALL	nucleobase, nucleoside, nucleotide and nucleic acid metabolic process, transcription, transcription, DNA-dependent, regulation of transcription, DNA-dependent, regulation of transcription from RNA polymerase II promoter, transcription from RNA polymerase II promoter, cell communication, signal transduction, intracellular signaling cascade, protein kinase cascade, JAK-STAT cascade, metabolic process, response to biotic stimulus, response to virus, cellular process, gene expression, regulation of gene expression, RNA metabolic process, regulation of nucleobase, nucleoside, nucleotide and nucleic acid metabolic process, regulation of metabolic process, regulation of cellular metabolic process, RNA biosynthetic process, macromolecule metabolic process, biopolymer metabolic process, cellular metabolic process, primary metabolic process, regulation of transcription, regulation of biological process, regulation of cellular process, response to stimulus, multi-organism process, response to other organism, biological regulation,		
KEGG_PATHWAY	Jak-STAT signaling pathway,		
INTERPRO	SH2 motif, STAT transcription factor, core, EF-Hand type, STAT transcription factor, DNA-binding, subdomain, STAT transcription factor, protein interaction, STAT transcription factor, all-alpha, STAT transcription factor, DNA-binding,		
PIR_SUPERFAMILY	PIRSF015568:signal transducer and transcription activator STAT5A,		
BIOCARTA	IFN alpha signaling pathway,		

Figure 7. An example layout of DAVID Annotation Table. Various annotation contents for given gene are list in a tabular format. The contents for each gene are separated by the header rows in blue color.

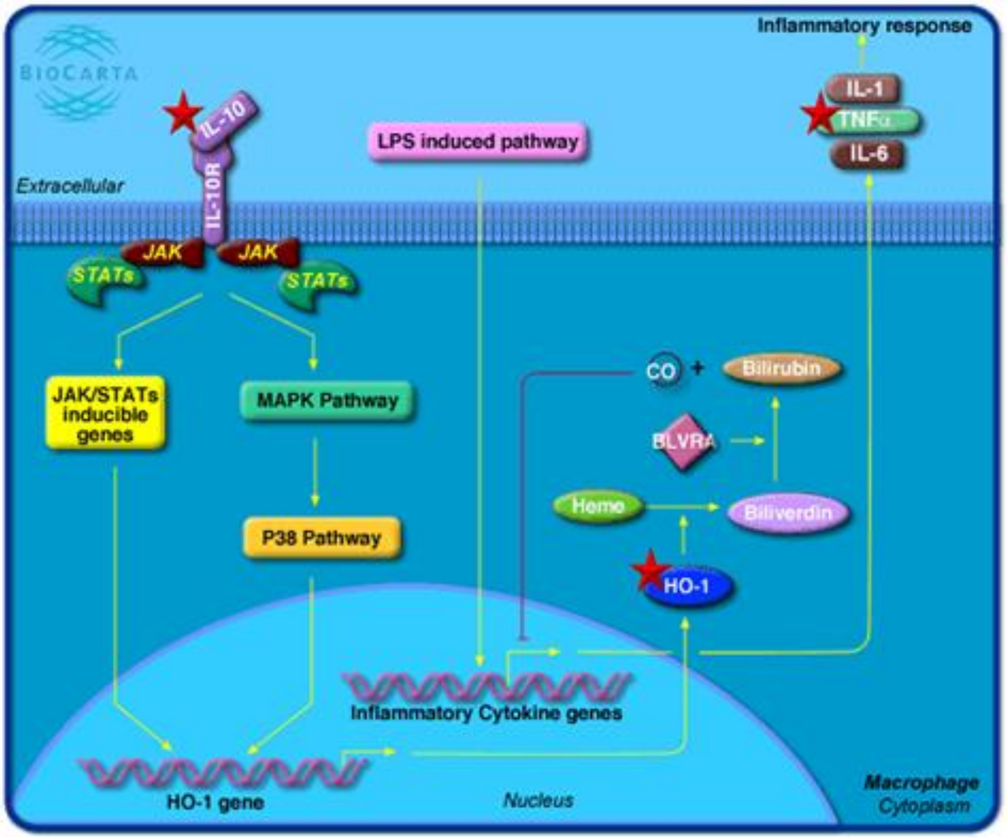


Figure 8. Pathway map viewer. The red star indicates the associations between pathway genes and the user's input genes. Following the pathway flow, IL10 was activated as an upstream immune stimulator. Then the middle stream gene, HO-1, was involved. IL-1/INFa/IL-6, as downstream regulator, was finally activated. Thus, the user's genes may be analyzed in a network context.

SUPPLEMENTARY DATA

Supplementary Data 1. Collection of ~68 similar enrichment analysis tools. The tools are roughly categorized into three classes according to their backend algorithms. Reference links are provided for more information of each tool.

Supplementary Data 2. ~400 Affymetrix IDs[16] used in this paper.

Supplementary Data 3. Comparisons of the enrichment p-values between gene lists derived from microarray study vs. same size of gene lists generated randomly. A 'good' gene lists should consistently contain more enriched biology than that of random list in the same sizes.

Supplementary Data 4. Summaries of gene identifier types and annotation categories supported in the DAVID system.

Supplementary Data 5. Screen shots of each major analysis step according to the description in the manuscript.

Supplementary Data 6. Examples for the input formats of a gene list.

Systematic and integrative analysis of large gene lists using DAVID bioinformatics resources

Da Wei Huang[1,2], Brad T Sherman[1,2] & Richard A Lempicki[1]

[1]Laboratory of Immunopathogenesis and Bioinformatics, Clinical Services Program, SAIC-Frederick Inc., National Cancer Institute at Frederick, Frederick, Maryland 21702, USA. [2]These authors contributed equally to this work. Correspondence should be addressed to R.A.L. (rlempicki@mail.nih.gov) or D.W.H. (huangdawei@mail.nih.gov)

Published online 18 December 2008; doi:10.1038/nprot.2008.211

DAVID bioinformatics resources consists of an integrated biological knowledgebase and analytic tools aimed at systematically extracting biological meaning from large gene/protein lists. This protocol explains how to use DAVID, a high-throughput and integrated data-mining environment, to analyze gene lists derived from high-throughput genomic experiments. The procedure first requires uploading a gene list containing any number of common gene identifiers followed by analysis using one or more text and pathway-mining tools such as gene functional classification, functional annotation chart or clustering and functional annotation table. By following this protocol, investigators are able to gain an in-depth understanding of the biological themes in lists of genes that are enriched in genome-scale studies.

INTRODUCTION

High-throughput genomic, proteomic and bioinformatics scanning approaches, such as expression microarray, promoter microarray, proteomic data and ChIP-on-CHIPs, provide significant capabilities to study a large variety of biological mechanisms, including associations with diseases. These technologies usually result in a large 'interesting' gene list (ranging in size from hundreds to thousands of genes) involved in studied biological conditions. Data analysis of the large gene lists is a very important downstream task following the above example of high-throughput technologies to understand the biological meaning of the output gene lists. The data analysis of such highly complex and large volume data sets is a challenging task, which requires support from special bioinformatics software packages. In this protocol, we introduce DAVID (the database for annotation, visualization and integrated discovery) bioinformatics resources[1,2], which is able to extract biological features/meaning associated with large gene lists. DAVID is able to handle any type of gene list, no matter which genomic platform or software package generated them.

DAVID, released in 2003 (refs. 2,3), as well as a number of other similar publicly available tools, including, but not limited to, GoMiner[4], GOstat[5], Onto-express[6], GoToolBox[7], FatiGO[8], GFINDer[9], GOBar[10] and GSEA[11] (see **Supplementary Data 1** for a complete list), address various aspects of the challenge of functionally analyzing large gene lists. Although each tool has distinct features and strengths, as reviewed by Khatri *et al.*[12], they all adopt a common core strategy to systematically map a large number of interesting genes in a list to the associated biological annotation (e.g., gene ontology terms), and then statistically highlight the most overrepresented (enriched) biological annotation out of thousands of linked terms and contents. Enrichment analysis is a promising strategy that increases the likelihood for investigators to identify biological processes most pertinent to the biological phenomena under study.

The analysis of large gene lists is indeed more of an exploratory, computational procedure rather than a purely statistical solution. As compared with other similar services, DAVID provides some unique features and capabilities, such as an integrated and expanded back-end annotation database[13], advanced modular enrichment algorithms[14] and powerful exploratory ability in an integrated data-mining environment[1]. Even though users can learn more in-depth information about DAVID algorithms in our original publications[1–3,13–15], we now briefly summarize the rationale regarding the key DAVID modules, as well as the analytic limitations (see **Table 1** for comparisons of DAVID's analytical modules), so that readers may be able to quickly follow the protocol.

Large gene lists ready for functional analysis by DAVID

In this protocol, we use a previously published gene list[16] (**Supplementary Data 2**) as an example to illustrate the results obtained from the various DAVID analytic modules. To obtain this list, freshly isolated peripheral blood mononuclear cells were treated with an HIV envelope protein (gp120) and genome-wide gene expression changes were observed using Affymetrix U95A microarray chips[16]. The aim of the experiment was to investigate cellular responses to viral envelope protein infection, which may help in understanding the mechanisms for HIV replication in resting or suboptimally activated peripheral blood mononuclear cells.

The quality of large gene lists derived from high-throughput biological studies is one of the most important foundations that directly influence the success of the following functional analysis in DAVID. Owing to the complexity of the data-mining situations involved in biological studies, there is no good systematic way, at present, to quantitatively estimate the quality of the gene list ahead of time (i.e., before the gene functional analysis). However, on the basis of real-life data analysis experiences during the past several years, a 'good' gene list may exhibit most, if not all, of the following characteristics:

(1) Contains many important genes (marker genes) as expected for given study (e.g., IL8, CCL4 and TNFSF8 from the example gene list in **Supplementary Data 2**).
(2) Reasonable number of genes ranging from hundreds to thousands (e.g., 100–2,000 genes), not extremely low or high.
(3) Most of the genes significantly pass the statistical threshold for selection (e.g., selecting genes by comparing gene expression between control and experimental cells with *t*-test statistics:

TABLE 1 | Side-by-side comparisons of DAVID's major analytic modules.

	Input user's large gene lists				
DAVID analytic module/tool	**Gene name batch viewer**[1]	**Gene functional classification**[14]	**Functional annotation chart**[1,2]	**Functional annotation clustering**[14]	**Functional annotation table**[1,2]
Brief definition/ explanation	List names of user's genes	Classify user's genes into gene groups	Identify enriched annotation terms associated with user's gene list	Cluster functionally similar terms associated with user's gene list into groups	Query associated terms for all user's genes
Key points	Gene-centric singular exploration	Gene-centric modular analysis	Term-centric singular enrichment analysis (typical enrichment analysis)	Term-centric modular enrichment analysis	Large-scale query
Example question to ask	What are the genes in my list?	What are the major gene families in my list?	Which annotation terms are enriched for my gene list?	Which annotation groups are enriched for my gene list?	What are the associated annotations for each of my genes?
Main functions	Display all gene names in a linear tabular text format Deep links to more information around given gene Search other functionally related genes	Classify functionally related genes into groups 2D view for related gene–term relationship Rank importance of gene groups with enrichment score Highlight annotation terms for gene groups	Identify enriched annotation terms in a linear tabular text format Deep links to more information around terms and associated genes Search other functionally related terms View genes on pathway maps	Cluster functionally related annotations into groups 2D view for related gene–term relationship Rank importance of annotation groups with enrichment score Pool genes for annotation groups	Query selected annotations for given genes
Advantages	Roughly explore genes one by one Quickly check if the expected/important genes are in the list Quickly learn annotation about genes of interests All genes are included in the analysis	Explore genes group by group rather than singular genes one by one Highlight important gene groups by enrichment scores Study functionally related genes and their relationship in a network format Good to catch major biology	Simple format to explore all singular enriched terms	Explore annotations group by group rather than singular terms one by one Highlight important annotation groups by enrichment scores Study functionally related genes and their relationship in a network format Good to focus on major and fine-level biology	Quickly explore all annotations (both enriched and non-enriched ones) for given genes Good for analysis of small number of focused genes Save entire annotation profile of a given gene list in text file ready for other external analysis
Drawbacks	Related genes scattered in the results lose interrelationships during exploration Difficult to judge important genes or nonspecific genes without enrichment calculation	Some genes without strong neighbors will be left out from the analysis	Related/redundant terms scattered in the results Some fine-level biology could be diluted by the redundancy Lack of term–term relationships during analysis	Some enriched terms without strong neighbors will be left out from the analysis	Difficult to explore large gene list No enrichment analysis

fold changes ≥ 2 and P-values ≤ 0.05). Importantly, statistical thresholds do not have to be sacrificed (e.g., fold changes ≥ 1.1 and P-value ≤ 0.2) to reach a comfortable gene size.

(4) Notable portion of up- or downregulated genes are involved in certain interesting biological processes, rather than being randomly spread throughout all possible biological processes.

(5) A 'good' gene list should consistently contain more enriched biology than that of a random list in the same size range during analysis in DAVID (see **Supplementary Data 3** for detailed discussions).
(6) High reproducibility (e.g., by independent experiments under the same conditions or by leave-one-out statistical test) to generate a similar gene list under the same conditions.
(7) The high quality of the high-throughput data can be confirmed by other independent wet lab tests or experiments.

Some of these points (2, 3, 6 and 7) come from upstream analysis, whereas DAVID may help in examining others (1, 4 and 5).

Moreover, for enrichment analysis, in general, a larger gene list can have higher statistical power resulting in a higher sensitivity (more significant *P*-values) to slightly enriched terms, as well as to more specific terms. Otherwise, the sensitivity is decreased toward largely enriched terms and broader/general terms. Although the size of the gene list influences (in a nonlinear way) the absolute enrichment *P*-values, which makes it difficult to directly compare the absolute enrichment *P*-values across gene lists, the enrichment *P*-values are fairly comparable within the same or same size of gene list. In addition, when different sizes of gene lists are generated from the same data set with different threshold stringencies (within a reasonable range), the absolute enrichment *P*-values may vary from list to list. However, the relative rank/order of the enriched terms may remain fairly stable, which will lead to consistent global conclusions of functional annotations across the different sizes of gene lists derived from the same data set (data not shown). This kind of reproducibility and consistency should be expected using DAVID tools if the underlying high-throughput biological studies are robust.

Interestingly, we found that many gene lists input to DAVID are in the size range of 1–10 genes. The enrichment statistic's power will be very limited in such extreme cases. However, the unique exploratory capability of DAVID could still be very powerful for analyzing such small gene lists. As the analysis is most likely in a very focused and small scope, analysts may take advantage of the unique exploratory capability of DAVID to navigate through all of the well-organized heterogeneous annotation contents around the focused genes regardless of the statistics.

Submission of user's gene identifiers to DAVID

Comprehensively mapping of a user's gene identifiers (gene IDs) to the relevant biological annotation in the DAVID database is an essential foundation for the success of any high-throughput gene functional analysis. Gene IDs and biological annotations are highly redundant within the vast array of public databases. The DAVID knowledgebase[13] was designed to collect and integrate diverse gene identifiers as well as more than 40 well-known publicly available annotation categories (**Supplementary Data 4**), which are then centralized by internal DAVID identifiers in a nonredundant manner. The wide range of biological annotation coverage and the nonredundant integration of gene IDs in the DAVID knowledgebase enables a user's gene ID to be mapped across the entire database, thus providing comprehensive coverage of gene-associated annotation. If a significant portion ($\geq$20%) of input gene IDs fail to be mapped to an internal DAVID ID, a specially designed module, the DAVID Gene ID Conversion Tool[15], will start up to help map such IDs.

Principle of 'gene population background' in enrichment analysis

The principle foundation of enrichment analysis is that if a biological process is abnormal in a given study, the co-functioning genes should have a higher potential (enriched) to be selected as a relevant group by the high-throughput screening technologies. To decide the degree of enrichment, a certain background must be set up to perform the comparison (also see Step 1 in **Table 2**). For example, 10% of the user's genes are kinases versus 1% of the genes in the human genome (this is the gene population background) that are kinases. The enrichment can therefore be quantitatively measured by some common and well-known statistical methods, including χ^2, Fisher's exact test, Binomial probability and Hypergeometric distribution. Thus, a conclusion may be obtained for the particular example, that is, kinases are enriched in the user's study, and therefore have important functions in the study. However, we cannot make such a conclusion with 10% alone, without comparing it with the background information (i.e., 1%).

In this sense, the background is one of the critical factors that impact the conclusion to a certain degree, particularly when two ratios are close. There are many ways to set the backgrounds, e.g., all genome genes; genes on an Affymetrix chip; and a subset of genome genes that the user used in their study. In general, larger backgrounds, e.g., the total genes in the genome as a population background, intend to give more significant *P*-values, as compared with a narrowed-down set of genes as a population background, such as genes existing only on a microarray. Even though there is no gold standard for the population background, a general guideline is to set up the population background as the pool of genes that have a chance to be selected for the studied annotation category in the scope of the users' particular study.

One of the advantages of DAVID is its flexibility of setting different population backgrounds to meet different situations. DAVID has an automatic procedure to 'guess' the background as the global set of genes in the genome on the basis of the user's uploaded gene list. Thus, in a regular situation, users do not have to set up a population background by themselves. We found that it works generally well just because most of the studies analyzed by DAVID are genome-wide or close to genome-wide studies. Moreover, other options are also available for user's choices, including all genes in the studied genome, genes in various microarray chips and most importantly any gene set that users define and upload. The last feature requires significant computational power so that it is rarely found in similar Web-based applications. In summary, various settings and options for population backgrounds can meet the range of needs of general users to those of power users.

DAVID gene name batch viewer

Gene IDs, such as Entrez Gene 3558, typically do not convey biological meaning in and of itself. The gene name batch viewer[1] is able to quickly attach meaning to a list of gene IDs by rapidly translating them into their corresponding gene names (**Fig. 1**, and see slide 4 of **Supplementary Data 5** for more detail). Thus, before proceeding to analysis with other more comprehensive analytic tools, investigators can quickly glance at the gene names to further gain insight about their study and to answer questions such as, 'Does my gene list contain important genes relevant to the study?'. In addition, a set of hyperlinks

TABLE 2 | Major statistical methods and associated parameters used in DAVID.

Step no.	Module/page	Statistics/parameters	Explanation/definition	How to understand the value
1	Submission of User's Gene IDs	Background genes (or called population genes)	To decide the degree of enrichment, a certain background must be set up to be compared with the user's gene list. For example, 10% of user's genes are kinases versus 1% of genes in human genome (this is population background) are kinases. Thus, the conclusion is obvious in the particular example that the user's study is highly related to kinase. However, 10% itself alone cannot provide such a conclusion without comparing it with the background information	A general guideline is to set up the reference background as the pool of genes that have a chance to be selected for the studied annotation category under the scope of users' particular study Default background is the entire genome-wide genes of the species matching the user's input IDs. Prebuilt backgrounds, such as genes in Affymetrix chips and so on, are available for the user's choice In principle, a larger gene background tends to give smaller *P*-values. As most of the high-throughput studies are, or at least are close to, genome-wide scope, the default background is good for regular cases in general
4	Gene Functional Classification[1,14]	Classification stringency	To control the behavior of DAVID Fuzzy clustering	A general guideline is to choose higher stringency settings for tight, clean and smaller numbers of clusters; otherwise, lower for loose, broader and larger numbers of clusters Default setting is medium Five predefined levels from lowest to highest for user's choices Users may want to play with different stringency for more satisfactory results
		Enrichment score (for each group)	To rank overall importance (enrichment) of gene groups. It is the geometric mean of all the enrichment *P*-values (EASE scores) for each annotation term associated with the gene members in the group. To emphasize that the geometric mean is a relative score instead of an absolute *P*-value, minus log transformation is applied on the average *P*-values	A higher score for a group indicates that the gene members in the group are involved in more important (enriched) terms in a given study; therefore, more attention should go to them Enrichment score of 1.3 is equivalent to non-log scale 0.05. Thus, more attention should be given to groups with scores ≥ 1.3 However, the gene groups with lower scores could be potentially interesting and should be explored as well, if possible
6	Functional Annotation Chart[1,2]	*P*-value (or called EASE score)	To examine the significance of gene–term enrichment with a modified Fisher's exact test (EASE score). For example, 10% of user's genes are kinases versus 1% of genes in human genome (this is population background) are kinases. Thus, the EASE score is < 0.05, which suggests that kinases are significantly more enriched than random chance in the study for this particular example	The smaller the *P*-values, the more significant they are Default cutoff is 0.1 Users could set different levels of cutoff through option panel on the top of result page. Owing to the complexity of biological data mining of this type, *P*-values are suggested to be treated as score systems, i.e., suggesting roles rather than decision-making roles. Users themselves should play critical roles in judging 'are the results making sense or not for expected biology'
		Benjamini	To globally correct enrichment *P*-values to control family-wide false discovery rate under certain rate (e.g., ≤ 0.05). It is one of the multiple testing correction techniques (Bonferroni, Benjamini and FDR) provided by DAVID	More terms examined, more conservative the corrections are. As a result, all the *P*-values get larger It is great if the interesting terms have significant *P*-values after the corrections. But as the multiple testing correction techniques are known as conservative approaches, it could hurt the sensitivity of discovery if overemphasizing them. Users' judgment could be critical as discussed in EASE score in Functional Annotation Chart section

(continued)

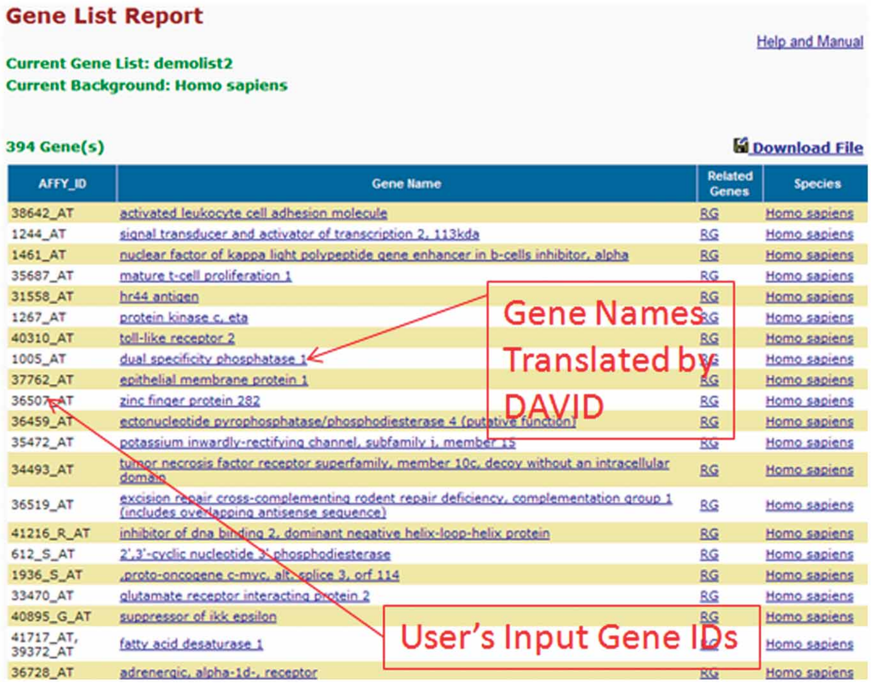
TABLE 2 | Major statistical methods and associated parameters used in DAVID (continued).

Step no.	Module/page	Statistics/parameters	Explanation/definition	How to understand the value
		Fold enrichment	To measure the magnitude of enrichment. For example, 10% of user's genes are kinases versus 1% of genes in human genome (this is population background) are kinases. Thus, the fold enrichment is tenfold. Fold enrichment along with EASE score could rank the enriched terms in a more comprehensive way	Fold enrichment 1.5 and above are suggested to be considered as interesting Fold enrichment and EASE score should always be examined side by side. Terms with larger fold enrichments and smaller may be interesting Caution should be taken when big fold enrichments are obtained from a small number of genes (e.g., ≤ 3). This situation often happens to the terms with a few genes (more specific terms) or of smaller size (e.g., <100) of user's input gene list. In this case, the reliability is not as much as those fold enrichment scores obtained from larger numbers of genes
		%	Number of genes involved in given term is divided by the total number of user's input genes, i.e., percentage of user's input gene hitting a given term. For example, 10% of user's genes hit 'kinase activity'	It gives overall idea of gene distributions among the terms The higher percentage does not necessarily have a good EASE score because it also depends on the percentage of background genes as discussed in the EASE score in Functional Annotation Chart section
8	Functional Annotation Clustering[1,14]	Classification stringency	To control the behavior of DAVID Fuzzy clustering	A general guideline is to choose higher stringency setting for tight, clean and smaller numbers of clusters; otherwise, lower for looser, broader and larger numbers of clusters Default setting is medium Five predefined levels from lowest to highest for user's choices Users may want to play with different stringency to obtain more satisfactory results
		Enrichment score (for each group)	To rank overall importance (enrichment) of annotation term groups. It is the geometric mean of all the enrichment *P*-values (EASE scores) of each annotation term in the group. To emphasize that the geometric mean is a relative score instead of an absolute *P*-value, minus log transformation is applied on the average *P*-values	A higher score for a group indicates that annotation term members in the group are playing more important (enriched) roles in given study; therefore, pay more attention toward them Enrichment score 1.3 is equivalent to non-log scale 0.05. Thus, more attention should be given to groups with scores ≥ 1.3 However, the annotation groups with lower scores could be potentially interesting, and should be explored as well if possible
		P-value (or called EASE score) (for individual term members)	To examine the significance of gene–term enrichment with a modified Fisher's exact test (EASE score). This *P*-value is calculated in exactly the same way as in the Functional Annotation Chart section	The explanation is the same as that in Functional Annotation Chart section
		Benjamini	To globally correct enrichment *P*-values of individual term members. The idea and calculations are exactly the same as that in Functional Annotation Chart section	The explanation is the same as that in the Functional Annotation Chart section

are provided for each gene entry, allowing users to further explore additional functional information about each gene.

DAVID gene functional classification

As the analysis proceeds, gene functional classification[14] provides the distinct ability for investigators to explore and view functionally related genes together, as a unit, to concentrate on the larger biological network rather than at the level of an individual gene. In fact, the majority of cofunctioning genes may have diversified names so that genes cannot be simply classified into functional groups according to their names. However, gene functional classification, accomplished with a set of novel fuzzy clustering techniques, is able to classify input genes into functionally related gene groups (or classes) on the basis of their annotation term co-occurrence rather than on gene names. Condensing large gene lists into biologically meaningful modules greatly improves one's ability to assimilate large amounts of information and thus switches functional annotation analysis from a gene-centric analysis to a biological module-centric analysis (**Fig. 2**, and see slides 5 and 6 of **Supplementary Data 5** for more details). Taken together with the 'drill-down' function associated with each biological module and visualizations to view the relationships between the many-genes-to-many-terms associations, investigators are able to more comprehensively understand how genes are associated with each other and with the functional annotation.



Gene List Report

Help and Manual

Current Gene List: demolist2
Current Background: Homo sapiens

394 Gene(s)

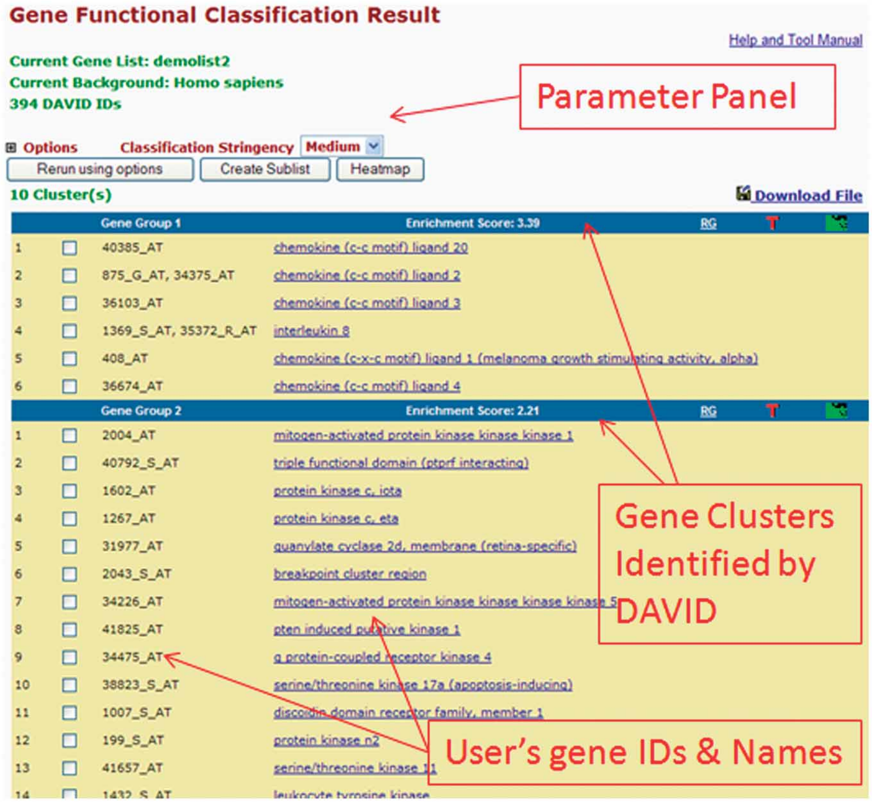
Download File

AFFY_ID	Gene Name	Related Genes	Species
38642_AT	activated leukocyte cell adhesion molecule	RG	Homo sapiens
1244_AT	signal transducer and activator of transcription 2, 113kda	RG	Homo sapiens
1461_AT	nuclear factor of kappa light polypeptide gene enhancer in b-cells inhibitor, alpha	RG	Homo sapiens
35687_AT	mature t-cell proliferation 1	RG	Homo sapiens
31558_AT	hr44 antigen	RG	Homo sapiens
1267_AT	protein kinase c, eta	RG	Homo sapiens
40310_AT	toll-like receptor 2	RG	Homo sapiens
1005_AT	dual specificity phosphatase 1	RG	Homo sapiens
37762_AT	epithelial membrane protein 1	RG	Homo sapiens
36507_AT	zinc finger protein 282	RG	Homo sapiens
36459_AT	ectonucleotide pyrophosphatase/phosphodiesterase 4 (putative function)	RG	Homo sapiens
35472_AT	potassium inwardly-rectifying channel, subfamily j, member 15	RG	Homo sapiens
34493_AT	tumor necrosis factor receptor superfamily, member 10c, decoy without an intracellular domain	RG	Homo sapiens
36519_AT	excision repair cross-complementing rodent repair deficiency, complementation group 1 (includes overlapping antisense sequence)	RG	Homo sapiens
41216_R_AT	inhibitor of dna binding 2, dominant negative helix-loop-helix protein	RG	Homo sapiens
612_S_AT	2',3'-cyclic nucleotide 3' phosphodiesterase	RG	Homo sapiens
1936_S_AT	,proto-oncogene c-myc, alt. splice 3, orf 114	RG	Homo sapiens
33470_AT	glutamate receptor interacting protein 2	RG	Homo sapiens
40895_G_AT	suppressor of ikk epsilon	RG	Homo sapiens
41717_AT, 39372_AT	fatty acid desaturase 1	RG	Homo sapiens
36728_AT	adrenergic, alpha-1d-, receptor	RG	Homo sapiens

Figure 1 | An example layout of DAVID gene name batch viewer. User's input gene IDs are translated into meaningful and readable gene names. The link on each gene name can lead to more in-depth information.

DAVID functional annotation chart

Functional annotation chart[1–3] provides typical gene–term enrichment (overrepresented) analysis, which is also provided by other similar tools, to identify the most relevant (overrepresented) biological terms associated with a given gene list (**Fig. 3**, and see slide 8 of **Supplementary Data 5** for more detail). Compared with other similar enrichment analysis tools, the notable difference of this function provided by DAVID is its extended annotation coverage[13], increasing from only GO in the original version of DAVID to presently over 40 annotation categories, including GO terms, protein–protein interactions, protein functional domains, disease associations, bio-pathways, sequence features, homology, gene functional summaries, gene tissue expression and literature (**Supplementary Data 4**). The annotation categories can be flexibly included or excluded from the analysis on the basis of a user's choices (see slide 7 of **Supplementary Data 5**). The enhanced annotation coverage alone increases the analytic power by allowing investigators to analyze their genes from many different biological aspects in a single space. In addition, to take full advantage of the well-known KEGG and BioCarta pathways, DAVID pathway viewer, which is accessed by clicking on pathway links within the chart report, can display genes from a user's list on pathway maps to facilitate biological interpretation in a network context (see slide 9 of **Supplementary Data 5**). Finally, the choice of prebuilt or



Gene Functional Classification Result

Help and Tool Manual

Current Gene List: demolist2
Current Background: Homo sapiens
394 DAVID IDs

⊞ Options Classification Stringency Medium

Rerun using options | Create Sublist | Heatmap

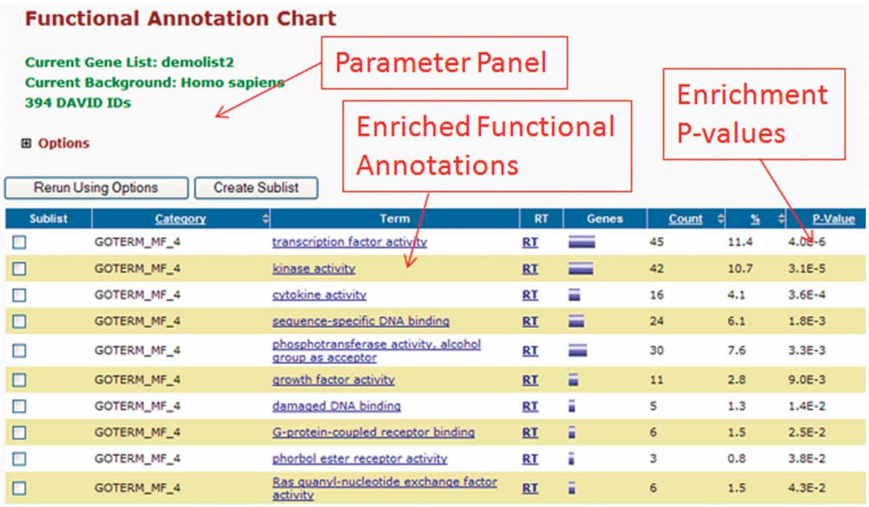
10 Cluster(s)

Download File

		Gene Group 1	Enrichment Score: 3.39	RG
1	□	40385_AT	chemokine (c-c motif) ligand 20	
2	□	875_G_AT, 34375_AT	chemokine (c-c motif) ligand 2	
3	□	36103_AT	chemokine (c-c motif) ligand 3	
4	□	1369_S_AT, 35372_R_AT	interleukin 8	
5	□	408_AT	chemokine (c-x-c motif) ligand 1 (melanoma growth stimulating activity, alpha)	
6	□	36674_AT	chemokine (c-c motif) ligand 4	
		Gene Group 2	Enrichment Score: 2.21	RG
1	□	2004_AT	mitogen-activated protein kinase kinase kinase 1	
2	□	40792_S_AT	triple functional domain (ptprf interacting)	
3	□	1602_AT	protein kinase c, iota	
4	□	1267_AT	protein kinase c, eta	
5	□	31977_AT	guanylate cyclase 2d, membrane (retina-specific)	
6	□	2043_S_AT	breakpoint cluster region	
7	□	34226_AT	mitogen-activated protein kinase kinase kinase kinase 5	
8	□	41825_AT	pten induced putative kinase 1	
9	□	34475_AT	g protein-coupled receptor kinase 4	
10	□	38823_S_AT	serine/threonine kinase 17a (apoptosis-inducing)	
11	□	1007_S_AT	discoidin domain receptor family, member 1	
12	□	199_S_AT	protein kinase n2	
13	□	41657_AT	serine/threonine kinase 11	
14	□	1432_S_AT	leukocyte tyrosine kinase	

Figure 2 | An example layout of DAVID gene functional classification. User's genes were organized and condensed into several functional groups. The gene members in each group share common biological functions. A set of accessory tools provided for each group will further facilitate the 'drill-down' analysis of biological interrelationships among the gene members within the same group.

user-defined gene population backgrounds provides the user with the ability to tailor the enrichment analysis to meet the user's specific analytic situation.



Functional Annotation Chart

Current Gene List: demolist2
Current Background: Homo sapiens
394 DAVID IDs

Sublist	Category	Term	RT	Genes	Count	%	P-Value
☐	GOTERM_MF_4	transcription factor activity	RT	▬	45	11.4	4.0E-6
☐	GOTERM_MF_4	kinase activity	RT	▬	42	10.7	3.1E-5
☐	GOTERM_MF_4	cytokine activity	RT	▬	16	4.1	3.6E-4
☐	GOTERM_MF_4	sequence-specific DNA binding	RT	▬	24	6.1	1.8E-3
☐	GOTERM_MF_4	phosphotransferase activity, alcohol group as acceptor	RT	▬	30	7.6	3.3E-3
☐	GOTERM_MF_4	growth factor activity	RT	▬	11	2.8	9.0E-3
☐	GOTERM_MF_4	damaged DNA binding	RT	▬	5	1.3	1.4E-2
☐	GOTERM_MF_4	G-protein-coupled receptor binding	RT	▬	6	1.5	2.5E-2
☐	GOTERM_MF_4	phorbol ester receptor activity	RT	▬	3	0.8	3.8E-2
☐	GOTERM_MF_4	Ras guanyl-nucleotide exchange factor activity	RT	▬	6	1.5	4.3E-2

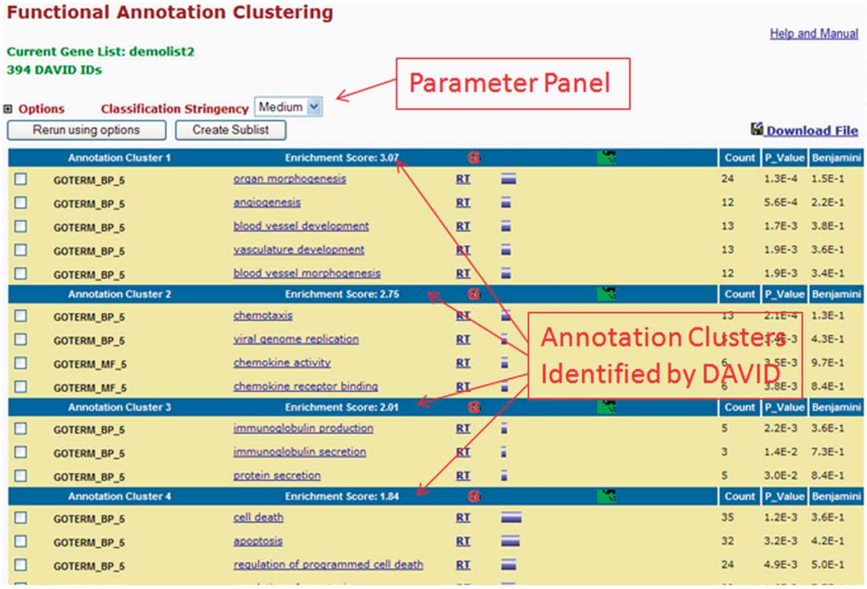
Figure 3 | An example layout of DAVID annotation chart. The enriched functional annotation terms associated with user's gene list are identified and listed according to their enrichment *P*-value by DAVID. The links on the page can lead to various detailed information regarding corresponding items.

DAVID functional annotation clustering

Functional annotation clustering[14] uses a similar fuzzy clustering concept as functional classification by measuring relationships among the annotation terms on the basis of the degree of their coassociation with genes within the user's list to cluster somewhat heterogeneous, yet highly similar annotation into functional annotation groups (**Fig. 4**, see slide 10 of **Supplementary Data 5** for more detail). This reduces the burden of associating different terms associated with the similar biological process, thus allowing the biological interpretation to be more focused at the 'biological module' level. The 2D view tool is also provided for examining the internal relationships among the clustered terms and genes (see slide 6 of **Supplementary Data 5**). This type of grouping of functional annotation is able to give a more insightful view of the relationships between annotation categories and terms compared with the traditional linear list of enriched terms, as highly related/redundant annotation terms may be dispersed among hundreds, if not thousands, of other terms.

DAVID functional annotation table

Functional annotation table[1,2] is a query engine for the DAVID knowledgebase, without statistical calculations (**Fig. 5**, see slide 11 of **Supplementary Data 5** for further details). For a given gene list, the tool can quickly query corresponding annotation for each gene and present them in a table format. Thus, users are able to explore annotation in a gene-by-gene manner. This is a useful analytic module particularly when users want to closely look at the annotation of highly interesting genes.



Functional Annotation Clustering

Current Gene List: demolist2
394 DAVID IDs

Annotation Cluster 1		Enrichment Score: 3.07			Count	P_Value	Benjamini
☐	GOTERM_BP_5	organ morphogenesis	RT	▬	24	1.3E-4	1.5E-1
☐	GOTERM_BP_5	angiogenesis	RT	▬	12	5.6E-4	2.2E-1
☐	GOTERM_BP_5	blood vessel development	RT	▬	13	1.7E-3	3.8E-1
☐	GOTERM_BP_5	vasculature development	RT	▬	13	1.9E-3	3.6E-1
☐	GOTERM_BP_5	blood vessel morphogenesis	RT	▬	12	1.9E-3	3.4E-1
Annotation Cluster 2		Enrichment Score: 2.75			Count	P_Value	Benjamini
☐	GOTERM_BP_5	chemotaxis	RT	▬	13	2.1E-4	1.3E-1
☐	GOTERM_BP_5	viral genome replication	RT	▬	5	3.4E-3	4.3E-1
☐	GOTERM_MF_5	chemokine activity	RT	▬	6	3.5E-3	9.7E-1
☐	GOTERM_MF_5	chemokine receptor binding	RT	▬	6	3.8E-3	8.4E-1
Annotation Cluster 3		Enrichment Score: 2.01			Count	P_Value	Benjamini
☐	GOTERM_BP_5	immunoglobulin production	RT	▬	5	2.2E-3	3.6E-1
☐	GOTERM_BP_5	immunoglobulin secretion	RT	▬	3	1.4E-2	7.3E-1
☐	GOTERM_BP_5	protein secretion	RT	▬	5	3.0E-2	8.4E-1
Annotation Cluster 4		Enrichment Score: 1.84			Count	P_Value	Benjamini
☐	GOTERM_BP_5	cell death	RT	▬	35	1.2E-3	3.6E-1
☐	GOTERM_BP_5	apoptosis	RT	▬	32	3.2E-3	4.2E-1
☐	GOTERM_BP_5	regulation of programmed cell death	RT	▬	24	4.9E-3	5.0E-1

Figure 4 | An example layout of DAVID functional annotation clustering. The similar annotation terms are grouped into clusters so that the user can read through the important terms in the way of block by block instead of individual by individual.

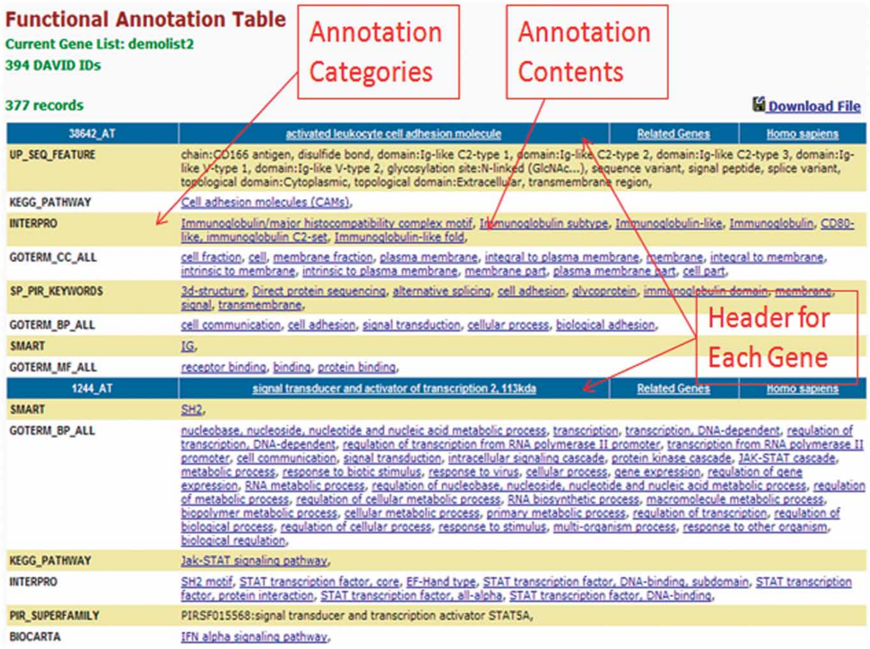
Summary

Collectively, all of the DAVID analytic modules aim to extract biological meaning from the given gene list from different biological angles with highly consistent and expected results for a given study. Integration of the results from the different analytic modules (**Fig. 6**) will take advantage of the different focus and strength of each module, to make the overall biological picture assembled on the basis of the gene list, more comprehensive and detailed. For a given gene list, DAVID bioinformatics resources is able to help users to (**Fig. 6b**):

- Convert gene IDs from one type to another
- Diagnose and fix problems with gene IDs
- Explore gene names in batch
- Discover enriched functionally related gene groups
- Display relationship of many-genes-to-many-terms on 2D view
- Provide an initial glance of major biological functions associated with gene list
- Identify enriched (overrepresented) annotation terms
- Visualize genes on BioCarta and KEGG pathway maps
- Link gene–disease associations
- Highlight protein functional domains and motifs
- Redirect to related literature
- List interacting proteins
- Cluster redundant and heterozygous annotation terms
- Search other functionally similar genes in genome, but not in list

- Search other annotations functionally similar to one of interest
- Read all annotation contents associated with a gene.

This article will mainly describe the protocol of how to use each DAVID analytic module in a logical, sequential order, as well as how to switch among the analytic modules (**Fig. 6**). The example gene list used in this protocol (also available as demo list 2 on DAVID website) allows new users to quickly test and experience various functions provided by DAVID. The protocol provides a routine analytic flow for new users to begin, as well as the flexibility for experienced users to use, the modules in different combinations to balance the different focuses and strengths of each module to better meet specific analytical questions (**Fig. 6**). Moreover, **Table 2** lists major statistical methods and filtering parameters that may influence the DAVID analysis and result interpretation in certain ways, for users to quickly look up specific statistical topics according to their interests.



Functional Annotation Table
Current Gene List: demolist2
394 DAVID IDs

377 records

Download File

38642_AT	activated leukocyte cell adhesion molecule	Related Genes	Homo sapiens
UP_SEQ_FEATURE	chain:CD166 antigen, disulfide bond, domain:Ig-like C2-type 1, domain:Ig-like C2-type 2, domain:Ig-like C2-type 3, domain:Ig-like V-type 1, domain:Ig-like V-type 2, glycosylation site:N-linked (GlcNAc...), sequence variant, signal peptide, splice variant, topological domain:Cytoplasmic, topological domain:Extracellular, transmembrane region,		
KEGG_PATHWAY	Cell adhesion molecules (CAMs),		
INTERPRO	Immunoglobulin/major histocompatibility complex motif, Immunoglobulin subtype, Immunoglobulin-like, Immunoglobulin, CD80-like, immunoglobulin C2-set, Immunoglobulin-like fold,		
GOTERM_CC_ALL	cell fraction, cell, membrane fraction, plasma membrane, integral to plasma membrane, membrane, integral to membrane, intrinsic to membrane, intrinsic to plasma membrane, membrane part, plasma membrane part, cell part,		
SP_PIR_KEYWORDS	3d-structure, Direct protein sequencing, alternative splicing, cell adhesion, glycoprotein, immunoglobulin domain, membrane, signal, transmembrane,		
GOTERM_BP_ALL	cell communication, cell adhesion, signal transduction, cellular process, biological adhesion,		
SMART	IG,		
GOTERM_MF_ALL	receptor binding, binding, protein binding,		
1244_AT	signal transducer and activator of transcription 2, 113kda	Related Genes	Homo sapiens
SMART	SH2,		
GOTERM_BP_ALL	nucleobase, nucleoside, nucleotide and nucleic acid metabolic process, transcription, transcription, DNA-dependent, regulation of transcription, DNA-dependent, regulation of transcription from RNA polymerase II promoter, transcription from RNA polymerase II promoter, cell communication, signal transduction, intracellular signaling cascade, protein kinase cascade, JAK-STAT cascade, metabolic process, response to biotic stimulus, response to virus, cellular process, gene expression, regulation of gene expression, RNA metabolic process, regulation of nucleobase, nucleoside, nucleotide and nucleic acid metabolic process, regulation of metabolic process, regulation of cellular metabolic process, RNA biosynthetic process, macromolecule metabolic process, biopolymer metabolic process, cellular metabolic process, primary metabolic process, regulation of transcription, regulation of biological process, regulation of cellular process, response to stimulus, multi-organism process, response to other organism, biological regulation,		
KEGG_PATHWAY	Jak-STAT signaling pathway,		
INTERPRO	SH2 motif, STAT transcription factor, core, EF-Hand type, STAT transcription factor, DNA-binding, subdomain, STAT transcription factor, protein interaction, STAT transcription factor, all-alpha, STAT transcription factor, DNA-binding,		
PIR_SUPERFAMILY	PIRSF015568:signal transducer and transcription activator STAT5A,		
BIOCARTA	IFN alpha signaling pathway,		

Figure 5 | An example layout of DAVID Annotation Table. Various annotation contents for a given gene are listed in a tabular format. The contents for each gene are separated by the header rows in blue color.

MATERIALS

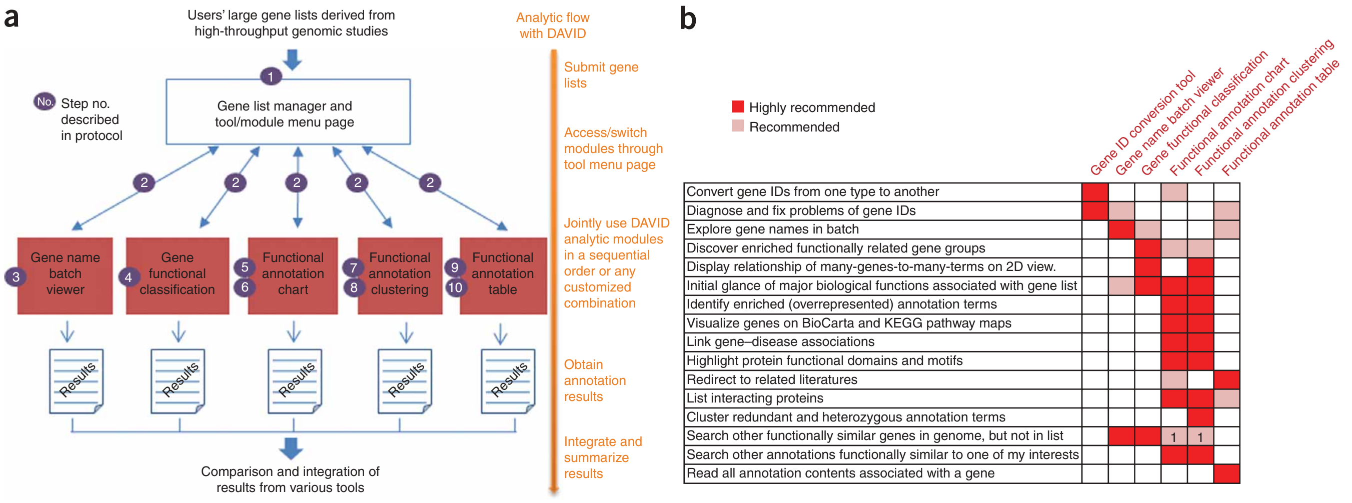
EQUIPMENT

A computer with high-speed Internet access and a Web browser.

EQUIPMENT SETUP

Hardware requirements and computer configurations DAVID is a Web-based tool designed so that a computer with a standard Web browser using default settings should work well. There is no need for special configuration and installation. Although DAVID was tested with several combinations of Internet browsers and operating systems, MS Internet Explorer or Firefox in a Window XP operating system is recommended to obtain the most satisfactory usability.

Input data A list of gene identifiers is the only required input for all DAVID analytic modules or tools. The gene list may be derived from any type of high-throughput genomic, computational or proteomic study, such as DNA expression microarray, proteomics, CHIP-on-CHIP, SNP array, CHIP-sequence and so on. The format of the gene list to be uploaded is described throughout the website and is either one gene ID per line or a list of comma-delimited gene IDs in one line (**Supplementary Data 6**). DAVID supports most common public gene identifiers[13] (see **Supplementary Data 4**). In addition, after the gene list is submitted to DAVID, all DAVID analytic modules can access the present list from the gene list manager so that there is no need to resubmit the gene list for each DAVID tool.


a
Users' large gene lists derived from high-throughput genomic studies
1
Gene list manager and tool/module menu page
No Step no. described in protocol
2
2
2
2
2
3 Gene name batch viewer
4 Gene functional classification
5 6 Functional annotation chart
7 8 Functional annotation clustering
9 10 Functional annotation table
Results
Results
Results
Results
Results
Comparison and integration of results from various tools
Analytic flow with DAVID
Submit gene lists
Access/switch modules through tool menu page
Jointly use DAVID analytic modules in a sequential order or any customized combination
Obtain annotation results
Integrate and summarize results
b

🟥 Highly recommended
▨ Recommended

	Gene ID conversion tool	Gene name batch viewer	Gene functional classification	Functional annotation chart	Functional annotation clustering	Functional annotation table
Convert gene IDs from one type to another	🟥			▨		
Diagnose and fix problems of gene IDs	🟥	▨				▨
Explore gene names in batch		🟥	▨			▨
Discover enriched functionally related gene groups			🟥	▨	▨	
Display relationship of many-genes-to-many-terms on 2D view.			🟥		🟥	
Initial glance of major biological functions associated with gene list		▨	🟥	🟥	🟥	
Identify enriched (overrepresented) annotation terms				🟥	🟥	
Visualize genes on BioCarta and KEGG pathway maps				🟥	🟥	
Link gene–disease associations				🟥	🟥	
Highlight protein functional domains and motifs				🟥	🟥	
Redirect to related literatures				▨		🟥
List interacting proteins				🟥	🟥	▨
Cluster redundant and heterozygous annotation terms					🟥	
Search other functionally similar genes in genome, but not in list		🟥	🟥	▨ 1	▨ 1	
Search other annotations functionally similar to one of my interests				🟥	🟥	
Read all annotation contents associated with a gene						🟥

Figure 6 | Analytic tools/modules in DAVID. (**a**) After the user's gene list is submitted to DAVID, the gene list manager may be accessed by all DAVID analytic modules (red boxes) at any time. The circled numbers indicate step numbers described in PROCEDURE to facilitate reading. (**b**) DAVID analytic modules, each having different strengths and focus, can be used independently or jointly. A roadmap to help users to choose some or all DAVID analytic modules for the analysis of large gene lists.

An example gene list derived from an HIV microarray study[16] is used in this protocol, as well as available as demo_list2 on DAVID website. The HIV microarray study is briefly described in INTRODUCTION. More detail can be found in the original publication[16].

Result download All results derived from DAVID may be explored and visualized on the Web browser. Moreover, all results generated by DAVID can be downloaded in simple flat text formats, thereafter to be edited or plotted by other graphic tools, e.g., MS Excel, for publication purposes as well as for archive purposes.

PROCEDURE

Submission of user's gene IDs to DAVID

1| Submit a gene list to DAVID (**Fig. 7**, and see slide 2 of **Supplementary Data 5** for further details). To do this, go to http://david.abcc.ncifcrf.gov or http://david.niaid.nih.gov and click on 'Start Analysis' on the header. Use the gene list manager panel that appears on the left side of the page (**Fig. 7**) and perform the following steps:

(A) Copy and paste a list of gene IDs into box A or load a text file containing gene IDs to box B (see more details regarding format requirements in EQUIPMENT SETUP, and also see **Supplementary Data 6**).

(B) Select the appropriate gene identifier type for your input gene IDs (see more details of supported ID types in EQUIPMENT SETUP).

(C) Indicate the list to be submitted as a gene list (i.e., genes to be analyzed) or as background genes (i.e., gene population background).

(D) Click the 'Submit List' button.

! CAUTION It takes ~30 s for a typical submission of ~1,000 gene IDs; the progress bar, below the header, will disappear after a successful submission and a gene list name should appear in the list manager box; if ≥20% input gene IDs cannot be recognized, the submission will be redirected to the DAVID Gene ID Conversion Tool[15] for further diagnosis. By default, the background is automatically set up as the genome-wide set of genes for the species that is found to have the majority of genes in the user's input list. However, it is always a good practice to double-check the default, or select a more appropriate prebuilt background through the 'background' tab on top of the list manager.

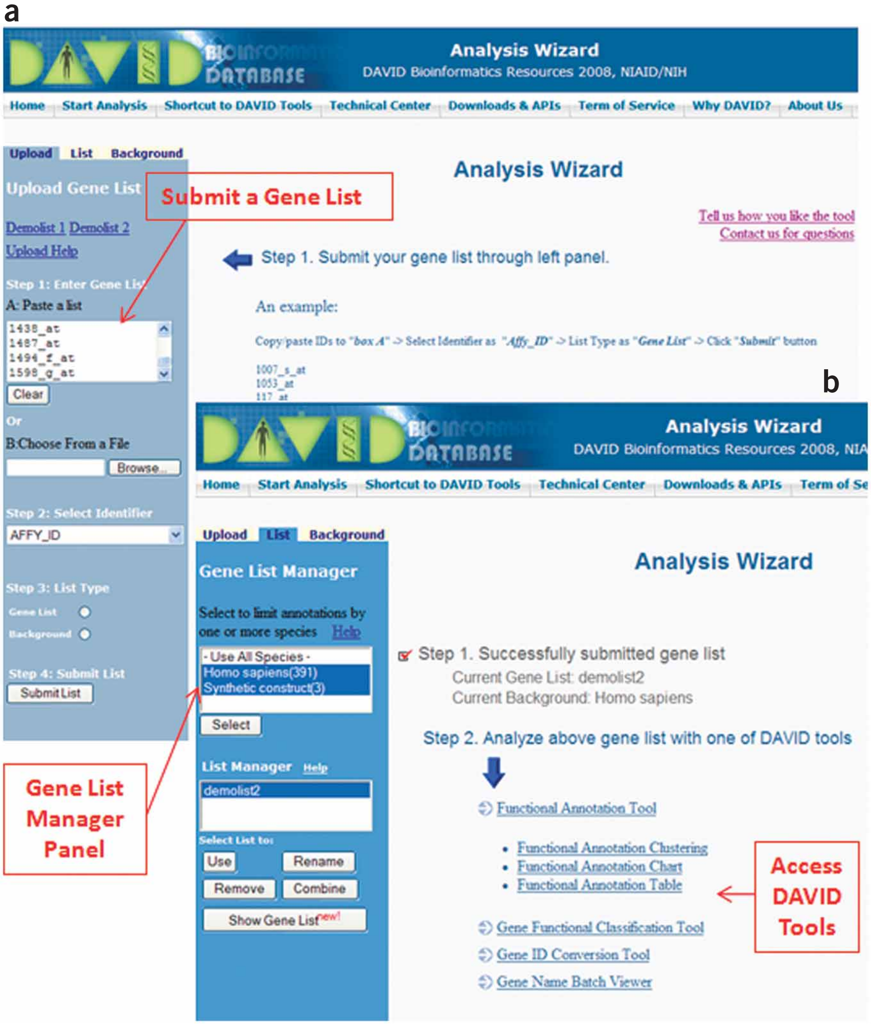
? TROUBLESHOOTING

2| Access DAVID analytic modules (**Fig. 7**, and see slide 3 of **Supplementary Data 5**) through the tools menu page. The tools main menu is the central page that lists a set of hyperlinks leading to all available analytic modules. Clicking on each link will lead to the corresponding analytic module for analysis of your present gene list, highlighted in the gene list manager.

▲ CRITICAL STEP By clicking on 'Start Analysis' on the header menu, users can always go back to this page at any time, no matter where they are, for choosing or switching to other analysis modules for the present gene list.

Gene name batch viewer

3| Run 'Gene Name Batch Viewer' and explore results (**Fig. 1**, and see slide 4 of **Supplementary Data 5**). To do this, click on the 'Gene Name Batch Viewer'



Analysis Wizard

Step 1. Submit your gene list through left panel.

An example:

Copy/paste IDs to "*box A*" -> Select Identifier as "*Affy_ID*" -> List Type as "*Gene List*" -> Click "*Submit*" button

```
1438_at
1487_at
1494_f_at
1598_g_at
```

```
1007_s_at
1053_at
117_at
```

Select to limit annotations by one or more species

Step 1. Successfully submitted gene list
Current Gene List: demolist2
Current Background: Homo sapiens

Step 2. Analyze above gene list with one of DAVID tools

- Functional Annotation Tool
  - Functional Annotation Clustering
  - Functional Annotation Chart
  - Functional Annotation Table
- Gene Functional Classification Tool
- Gene ID Conversion Tool
- Gene Name Batch Viewer

Figure 7 | Submit a gene list to DAVID and access various analytic tools/modules. (**a**) Following the example input format and steps on the left-side uploading panel, a list of genes may be uploaded into DAVID. (**b**) After successfully uploading a gene list(s), a set of analytic modules are available for the analysis of the present gene list highlighted in the gene list manager on the left side. Importantly, users may go to this page at any time by clicking 'Start Analysis' on the header to access any analytic tool of interest.

link on the tools menu page. All the gene names will be listed by the gene name batch viewer. For a gene of interest, one or all of following options may be conducted

(A) Click on the gene name to link to more detailed information.
(B) Click on 'RG' (related genes) beside the gene name to search for other functionally related genes.
(C) Use the browser's 'Find' function to search for particular items.

? TROUBLESHOOTING

Gene functional classification

4| Run 'Gene Functional Classification' and explore results (**Fig. 2**, and see slide 5 of **Supplementary Data 5**). To do this, return to the tools menu page by clicking on 'Start analysis' on the header. Click on 'Gene Functional Classification Tool' to classify the input gene list into gene groups. For any gene groups of interest, one or all of the following options may be conducted:

(A) Click on the gene name that leads to individual gene reports for in-depth information about the gene.
(B) Click on the red 'T' (term reports) to list associated biology of the gene group.
(C) Click on 'RG' (related genes) to list all genes functionally related to the particular gene group.
(D) Click on the 'green icon' to invoke 2D (gene-to-term) view.
(E) Create a new subgene list for further analysis on a subset of the genes.

! CAUTION The input genes are classified using the default clustering stringency. Users may rerun the classification function leading to optimal results for the particular case by resetting the stringency (high, medium or low) in the options on top of the result page.

▲ CRITICAL STEP 2D view is a Java Applet application that may take awhile to load for the first time; the 2D view Java Applet may require you to accept the online security certificate.

? TROUBLESHOOTING

Functional annotation chart

5| Run 'Functional Annotation Chart' (see slide 7 of **Supplementary Data 5**). To do this, return to the tools menu page by clicking on 'Start Analysis' on the header. Click on 'Functional Annotation Chart' to go the 'Summary Page' of the tool suite. Choose functional annotation categories of your interest (see slide 7 of **Supplementary Data 5**) either by accepting seven default functional annotation categories or by expanding the tree beside each main category (i.e., main accessions, gene ontology and so on) to select or deselect functional annotation categories of your interest. Then click on the 'Functional Annotation Chart' button on the bottom of the page leading to a chart report.

6| Explore the results of the 'Functional Annotation Chart' (**Fig. 3**, and see slide 8 of **Supplementary Data 5**). For an annotation term of interest, one or all of following options may be conducted:

(A) Click on the term name linking to a more detailed description.
(B) Click on 'RT' (related terms) to list other related terms.
(C) Click on the 'blue bar' to list all associated genes.
(D) Click on a pathway name to view genes on the pathway picture.

! CAUTION By default, the order of the annotation terms is based on the EASE (enrichment) score. However, results can also be sorted by different values in the columns. The annotation terms with EASE score ≤ 0.1 are displayed in the results by default. The stringency of this filter (EASE score cutoff) may be set higher or lower through the options provided at the top of the report page to include more or less of the annotation terms.

? TROUBLESHOOTING

Functional annotation clustering

7| Run 'Functional Annotation Clustering' (see slide 10 of **Supplementary Data 5**). To do this, return to the tools menu page by clicking on 'Start Analysis' on the header. Click on 'Functional Annotation Clustering' to go to the 'Summary Page' of the tool suite. Select annotation categories as described in Step 5, then click on the 'Functional Annotation Clustering' button on the bottom of the page.

8| Explore the results of 'Functional Annotation Clustering' (**Fig. 4**, and see slide 10 of **Supplementary Data 5**). For an annotation term cluster of interest, one or all of following options may be conducted:

(A) Click on the term name linking to a more detailed description.
(B) Click on 'RT' (related terms) to list other related terms.
(C) Click on the 'blue bar' to list all associated genes of corresponding individual term.
(D) Click on the red 'G' to list all associated genes of all terms within the cluster.
(E) Click on the 'green icon' to display the 2D (gene-to-term) view for all genes and terms within the cluster.

! CAUTION The annotation terms are clustered using the default clustering stringency. Users may rerun the classification

function leading to optimal results for the particular case by resetting the stringency (high, medium or low) in the options on top of the result page.
? TROUBLESHOOTING

Functional annotation table

9| Run 'Functional Annotation Table' (see slide 11 of **Supplementary Data 5**). To do this, return to the tools menu page by clicking on 'Start Analysis' on the header. Click on 'Functional Annotation Table' to go the 'Summary Page' of the tool suite. Select annotation categories as described in Step 5, then click on 'Functional Annotation Table' button on the bottom of the page.

10| Explore the results of 'Functional Annotation Table' (**Fig. 5,** and see slide 11 of **Supplementary Data 5**). For a gene of your interest, the following options may be conducted:
(A) Click on annotation terms for a detailed description.
(B) Click on 'Related Genes' to search functionally related genes.
! CAUTION As the output is too large to be displayed by Internet browsers, only top 500 records are shown on the result page. However, full results are available to be downloaded as a tab-delimited text file through the download link on top of the result page.
? TROUBLESHOOTING

● TIMING

The total analysis time varies, ranging from several minutes to hours, and is dependent on the analytical questions being addressed, the number of genes in the list being analyzed and the familiarity with the tools. It is not uncommon to make several visits to focus on different questions regarding a gene list of interest. Indeed, computational time is only a small portion of the total time, whereas exploring, interpreting and re-exploring both within DAVID and external to DAVID tends to dominate most of the time. We used a PC computer with the Windows XP operating system, 2-GB memory, 2.0 GHz CPU and 1-Mbps Internet connection for the data analysis of a gene list consisting of ~400 Affymetrix IDs (**Supplementary Data 2**) derived from an HIV study[16] (presented in ANTICIPATED RESULTS). During the analysis course, for regular functional calls, each result was typically returned in ~10 s. For the most computationally intensive functions, such as gene functional classification, results were typically returned within ~30 s; otherwise, never longer than 1 min.

? TROUBLESHOOTING

Troubleshooting advice can be found in **Table 3**.

TABLE 3 | Troubleshooting table.

Step	Problem	Possible reason	Solution
1	Gene ID submission is stuck and I got the message, "You are either not sure which identifier type your list contains, or less than 80% of your list has mapped to your chosen identifier type. Please use the Gene Conversion Tool to determine the identifier type"	User knows the correct gene ID types, but selected wrong one that did not match the actual input IDs	Go back to resubmit with correct selection of gene ID type or move forward with DAVID Gene ID Conversion tool to determine the potential gene ID type
		User does not know the correct gene ID type corresponding to their gene list	Submission panel in DAVID offers a special ID type, called 'Not sure'. Gene ID submission will be redirected to the DAVID Gene ID Conversion tool, which has a mechanism to scan the entire ID system in DAVID to help you to determine the potential ID type(s) of your genes
		There is more than one type of gene ID in the user's list	DAVID Gene ID Conversion Tool can help you determine the gene ID types and translate them to one single type
		User's gene IDs may contain a version number	Remove the version number, as DAVID will not recognize them
		≥20% of your gene IDs belong to low quality or retired IDs	DAVID Gene ID Conversion Tool may help to identify the problem IDs. The user should consider removing them from the gene list or move forward to analysis, ignoring the problem

(continued)

TABLE 3 | Troubleshooting table (continued)

Step	Problem	Possible reason	Solution
1 and 3	The gene number that DAVID recognizes does not match the number in my gene list	Repeated IDs in user's list	DAVID ID submission will automatically remove redundancy
		Particular ID(s) mapped to many different genes	DAVID ID Conversion Tool could help to identify the problem IDs. User should consider removing the 'bad' ID(s) from the gene list
		User's input gene IDs are gene symbols	Gene symbol is not species specific, so one symbol may be mapped to many homologous genes across different species. You can define particular species matching your study, after the gene symbols are submitted, in the gene list manager
3, 4, 6, 8 and 10	Result page is blank or empty	30-min timeout	If your Web browser is inactive for more than 30 min, DAVID will clean up all information (your gene list and so on) on the server side. Thus, you have to restart your analysis by resubmitting your gene list
		The size of the gene list is too small	Enrichment or clustering algorithms are based on the survey between input genes against background genes. Thus, a reasonable size (e.g., > 30) of input genes is required. Otherwise, certain algorithms will not work properly
		The cutoff or stringency options are too high	Lower down the thresholds accordingly
		Wrong background selected	Background is automatically set up as the genome-wide set of genes corresponding to the species for the majority of genes in the gene list. Sometimes, the system may not choose the appropriate species. User may check and correct the appropriate background through the Gene List Manager
		Small or minor species	Some small species may have very little annotation for the genes. There is nothing that can be done about this situation. Alternatively, you could map the genes to the homologous genes in a better-annotated species
4, 8	2D view is not displayed	Network certificate	Please accept it by clicking on 'Accept'. Basically, you are telling your browser to trust the DAVID application
		Java plug-in is not enabled	By default, most browsers should have the Java plug-in enabled. In case yours is not, please turn it on through Internet Options
NA	Service is too slow	Slow computer and/or Internet speed	Make sure that you have a reasonably good computer and Internet speed. See recommendations in MATERIALS
		Gene list is too large (> 3,000)	Please be patient
		DAVID server is overwhelmed	The DAVID service may sometimes be slow due to too many large, simultaneous requests. We have monitoring programs to autodetect and fix the situation in a short time period. If the situation is not resolved in a reasonable amount of time, please report the problem to the DAVID team through the contact provided on the DAVID website

NA, not applicable.

ANTICIPATED RESULTS

We use submission of the example gene list (~400 genes derived from an HIV microarray study[16]) to DAVID to illustrate the results obtained from various DAVID analytic modules. More detailed information and the availability regarding the gene list can be found in the Materials and Introduction sections of this protocol. Moreover, **Supplementary Data 5** provides screen shots of each major step of the following analysis.

Submission of user's gene IDs to DAVID

A successful submission of the gene list to DAVID is shown in **Figure 7** (also see slides 2 and 3 of **Supplementary Data 5**). Users should see the progress bar under the header move through submission and disappear upon completion of the submission. The gene list manager panel on the left side thereafter displays the list name (e.g., Upload_list_1) and corresponding species information (*Homo sapiens* (391)). The number (i.e., 391) appended after the species information is the number of genes that are recognized by DAVID. A set of hyperlinks on the right-side page lists the analytic modules available in the DAVID analytic pipeline. Users may follow the order of the pipeline to conduct analysis or jointly use analytic modules in varying combinations to meet the user's specific needs (**Fig. 6**). Most importantly, the page serves as a central page (**Fig. 7**) for users to choose analytic modules. Users may go back to this page at any time by clicking the 'Start Analysis' button on the header to switch back and forth among analytic modules as needed.

Gene name batch viewer

The corresponding gene names of input gene IDs are displayed as shown in **Figure 1** and slide 4 of **Supplementary Data 5**. Users may explore the gene names to examine whether there are any interesting study or marker genes in the list. Many immune-related genes, containing names like 'interleukin', 'chemokine', 'kinase' and 'tumor necrosis factor' can be found in the example list, which are consistent with that reported in the publicaton[16]. A set of hyperlinks provided for each gene can further lead to more detailed information about a given gene. In addition, by clicking on the 'RG' (related genes) search function beside a gene name, e.g., 'interleukin 8', all other functionally related chemokine genes (e.g., *cxcl 1*, *2*, *3*, *4*, *20*) will be listed so that users will be able to see other functionally similar genes in the list based on the bait gene.

Gene functional classification

The tool classified the example gene list (~400 genes) into ten functional groups in an easily readable tabular format. An example output is illustrated in **Figure 2** as well as in slide 5 of **Supplementary Data 5**. Gene groups (with significant enrichment scores ≥ 1), such as cytokines/chemokines (group 1: 3.39), kinases (group 2: 2.21), clathrin membrane fusion genes (group 3: 1.86), transcription factors (group 6: 1.39) and so on, can easily be identified. All of these gene groups are highly relevant to an HIV study and are therefore expected biological results[16]. Organizing the large gene list into gene groups allows investigators to quickly focus on the overall major common biology associated with a gene group rather than one gene at a time, thereby avoiding dilution of focus during the analysis due to too many single genes. Furthermore, the '2D View' function associated with each group is able to display all related terms and genes in detail in one picture, to examine their interrelationships. For example, for the kinase group (group 2), a user who is not familiar with kinases may explore the terms of kinase activity, transferase activity, ATP-binding, nucleotide binding, protein metabolism, tyrosine specificity, serine/threonine specificity, regulation of G protein signaling, signal transduction and so on in one view at the same time (slide 6 of **Supplementary Data 5**). Therefore, we can quickly learn the biology for the kinase group, with the above-mentioned related terms in a single view and also identify the fine differences among them. For example, there are two G-protein-coupled receptor kinases, three protein tyrosine kinases and six kinases involved in cell surface receptor-linked signal transduction among the 23 kinases within the group. The fine details may be very important for pinpointing the key biology associated with a study.

Functional annotation chart

Over 500 enriched (overrepresented) biological terms were reported (**Fig. 3**, and slide 8 of **Supplementary Data 5**). Many of them are highly immune related, such as response to pathogenic bacteria, chemokine activity, cell migration, clathrin-coated vesicle membrane, kinase activity, RNA polymerase II transcription factor activity, cell communication. This is consistent with observations identified earlier by the other analytic modules, as well as meeting the expectation for the HIV study[16]. The report offers a lot of redundant details regarding the enriched biology associated with the gene list, which certainly helps the interpretation of the biology, but also may dilute the focus. Moreover, a set of hyperlinks provided for each term will lead to more details about each term, such as in-depth description, associated genes, other related terms, directed acyclic graph (DAG) of GO and so on. Notably, the pathway viewer module offers visualization of users' genes on enriched pathways. For example, 'IL-10 Anti-inflammatory Signaling Pathway' was reported in the output. We can observe that IL10 was activated as an upstream immune regulator and was then further regulated by HO-1. As a result, the IL1/TNFa/IL6 complex was activated leading to further downstream inflammatory responses (**Fig. 8**, and slide 9 of **Supplementary Data 5**). Thus, the interrelationship of input genes was examined on the pathway in a network context.

Functional annotation clustering

The tool condensed the input gene list into smaller, much more organized biological annotation modules in a similar format (**Fig. 4**, and see slide 10 of **Supplementary Data 5**) as that of gene annotation clustering, but in a term-centric manner. Similarly, it allows investigators to focus on the annotation group level by quickly organizing many redundant/similar/hierarchical terms within the group. Annotation clusters, such as immune response, transcriptional regulation, chemokine activity, cytokine activity, kinase acitivity, signaling transduction, cell death and so on, could be found on the top of the output as expected for this study[16]. The highly organized and simplified annotation results allow users to quickly focus on the major biology at an annotation cluster level instead of trying to derive the same conclusions by putting together pieces that are scattered throughout a list of hundreds of terms in a typical term-enrichment analysis. In addition, the 'G' (genes) link provided for each cluster can comprehensively pool all related genes from different terms within the cluster. For example, each of the seven terms within cluster 2 (inflammatory response cluster) associates with both overlapping as well as differing genes. Therefore, a pooled gene list brought together by cluster 2 regarding inflammatory response may be much more comprehensive, compared with the genes selected from one or a few individual terms.

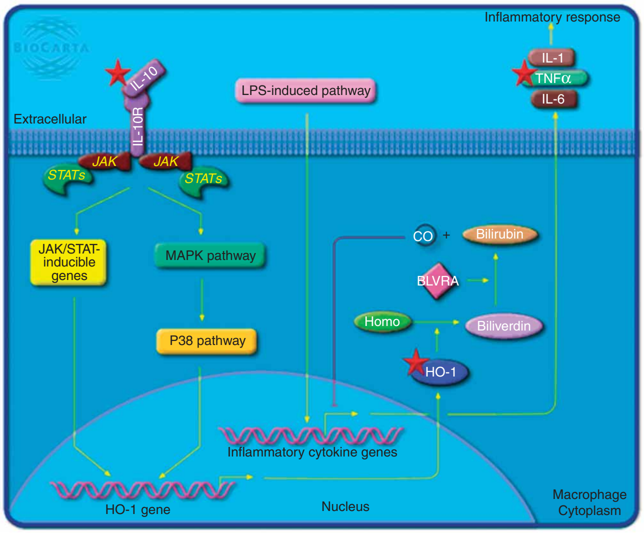


Figure 8 | Pathway map viewer. The red star indicates the associations between pathway genes and the user's input genes. Following the pathway flow, IL10 was activated as an upstream immune stimulator. Then, the middle stream gene, HO-1, was involved. IL-1/INFa/IL-6, as downstream regulator, was finally activated. Thus, the user's genes may be analyzed in a network context.

Note: Supplementary information is available via the HTML version of this article.

ACKNOWLEDGMENTS We are grateful to the referees for their constructive comments and thank Robert Stephens, David Bryant and David Liu in the ABCC group for Web server support. Thanks also go to Xin Zheng and Jun Yang in the Laboratory of Immunopathogenesis and Bioinformatics (LIB) group for discussion. We also thank Bill Wilton and Mike Tartakovsky for information technology and network support. The project has been funded with federal funds from the National Institute of Allergy and Infectious Diseases (NIAID) and National Institutes of Health (NIH), under Contract no. N01-CO-56000. The annotation of this tool and publication does not necessarily reflect the views or policies of the Department of Health and Human Services, nor does mention of trade names, commercial products or organizations imply endorsement by the United States Government.

Published online at http://www.natureprotocols.com/
Reprints and permissions information is available online at http://npg.nature.com/reprintsandpermissions/

1. Huang da, W. *et al.* DAVID bioinformatics resources: expanded annotation database and novel algorithms to better extract biology from large gene lists. *Nucleic Acids Res.* **35**, W169–W175 (2007).
2. Dennis, G. Jr. *et al.* DAVID: database for annotation, visualization, and integrated discovery. *Genome Biol.* **4**, P3 (2003).
3. Hosack, D.A., Dennis, G. Jr., Sherman, B.T., Lane, H.C. & Lempicki, R.A. Identifying biological themes within lists of genes with EASE. *Genome Biol.* **4**, R70 (2003).
4. Zeeberg, B.R. *et al.* High-Throughput GoMiner, an 'industrial-strength' integrative gene ontology tool for interpretation of multiple-microarray experiments, with application to studies of common variable immune deficiency (CVID). *BMC Bioinformatics* **6**, 168 (2005).
5. Beissbarth, T. & Speed, T.P. GOstat: find statistically overrepresented Gene Ontologies within a group of genes. *Bioinformatics* **20**, 1464–1465 (2004).
6. Khatri, P., Bhavsar, P., Bawa, G. & Draghici, S. Onto-Tools: an ensemble of web-accessible, ontology-based tools for the functional design and interpretation of high-throughput gene expression experiments. *Nucleic Acids Res.* **32**, W449–W456 (2004).
7. Martin, D. *et al.* GOToolBox: functional analysis of gene datasets based on Gene Ontology. *Genome Biol.* **5**, R101 (2004).
8. Al-Shahrour, F., Diaz-Uriarte, R. & Dopazo, J. FatiGO: a web tool for finding significant associations of Gene Ontology terms with groups of genes. *Bioinformatics* **20**, 578–580 (2004).
9. Masseroli, M., Galati, O. & Pinciroli, F. GFINDer: genetic disease and phenotype location statistical analysis and mining of dynamically annotated gene lists. *Nucleic Acids Res.* **33**, W717–W723 (2005).
10. Lee, J.S., Katari, G. & Sachidanandam, R. GObar: a gene ontology based analysis and visualization tool for gene sets. *BMC Bioinformatics* **6**, 189 (2005).
11. Subramanian, A. *et al.* Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proc. Natl. Acad. Sci. USA* **102**, 15545–15550 (2005).
12. Khatri, P. & Draghici, S. Ontological analysis of gene expression data: current tools, limitations, and open problems. *Bioinformatics* **21**, 3587–3595 (2005).
13. Sherman, B.T. *et al.* DAVID knowledgebase: a gene-centered database integrating heterogeneous gene annotation resources to facilitate high-throughput gene functional analysis. *BMC Bioinformatics* **8**, 426 (2007).
14. Huang da, W. *et al.* The DAVID gene functional classification tool: a novel biological module-centric algorithm to functionally analyze large gene lists. *Genome Biol.* **8**, R183 (2007).
15. Huang, D.W., Sherman, B.T. & Lempicki, R.A. DAVID gene ID conversion tool. *Bioinformation* **2**, 428–430 (2008).
16. Cicala, C. *et al.* HIV envelope induces a cascade of cell signals in non-proliferating target cells that favor virus replication. *Proc. Natl. Acad. Sci. USA* **99**, 9380–9385 (2002).

Genome Resource Facility

LONDON SCHOOL of HYGIENE &TROPICAL MEDICINE

Microarray Overview

Introduction

DNA microarray technology (also known as DNA arrays, DNA chips or biochips) represents one of the latest breakthroughs and indeed major achievements in experimental molecular biology. This novel technology, which started to appear during the second half of the 1990s, has historically evolved from the initial experimental reports published in the mid 1970s which indicated that labelled nucleic acids could be used to monitor the expression of nucleic acid molecules attached to a solid support. However, it was not until 1995 that the first article describing the application of DNA microarray technology to expression analysis was published in the scientific literature by Patrick Brown and his colleagues at Stanford University (Brown., *et al*. 1995).

Today, however, there is prominent evidence that the technology has made a dramatic advancements since its development and gained an increasing popularity among scientific researchers. It is also unquestionable that many scientific researchers are presumptive about the novelty of this technology regarding it as an indispensable as well as a needful research tool.

Such widespread adoption of DNA microarray technology in both industry and many academic research laboratories, was largely due to its aptitude to provide scientific researchers the opportunity to quickly and accurately perform simultaneous analysis of literally thousands of genes in a massively parallel manner, or even entire genome of an organism e.g. (Bacteria, Yeast, Virus, Protozoa, Mouse or Human) in a single experiment, hence providing extensive and valuable information on gene interaction and function.

Here, our aim is to give a brief overview of cDNA microarray technology, particularly explaining (Watson., *et al*. 1998) what is a cDNA microarray technology (Sinclair., 1999) the various types/platforms of cDNA microarray technology, highlighting both their advantages and disadvantages as well as how they are fabricated or manufactured (Burgess., 2001) the technology's basic fundamental principles and how it works are also outlined, as well as potential applications.

However, for detailed information of the potential and the scientific value of cDNA microarray technology in modern research, the reader is advised to look up the large number of excellent reviews of DNA array technology available (Rhodius., *et al* 2002). We particularly recommend the January 1999 supplement of Nature Genetics (Bowtell., 1999).

What is DNA microarray technology?

In it's broadest term, DNA microarray technology may be defined as a high-throughput and versatile technology used for parallel gene expression analysis for thousands of genes of known and unknown function, or DNA homology analysis for detecting polymorphisms and mutations in both prokaryotic and eukaryotic genomic DNA.

However, in its precise and accurate definition DNA microarray is an orderly arrangement of thousands of identified sequenced genes printed on an impermeable solid support, usually glass, silicon chips or nylon membrane.

Each identified sequenced gene on the glass, silicon chips or nylon membrane corresponds to a fragment of genomic DNA, cDNAs, PCR products or chemically synthesised oligonucleotides of up to 70mers and represents a single gene.

Usually a single DNA microarray slide/chip may contain thousands of spots each representing a single gene and collectively the entire genome of an organism. A schematic diagram of the two most commonly used DNA microarray formats to date are shown in Figure 1 and 2.

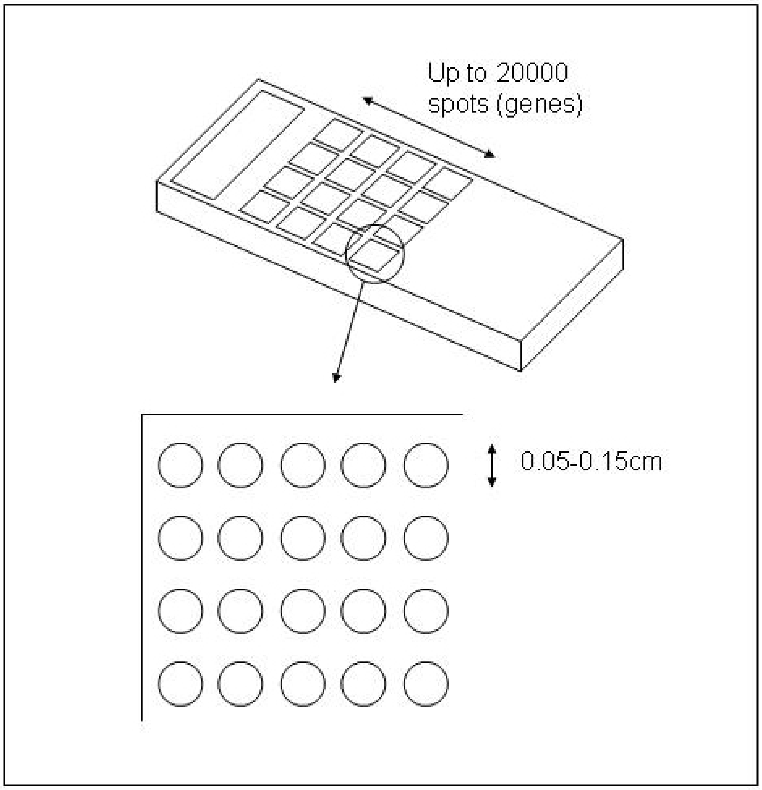


Figure 1. Glass complementary DNA (cDNA) microarray produced by using high-speed precision robot. This type of DNA microarray can bear between 10,000 - 20,000 spots (genes) on an area of 3.6 cm2. Each spot represents the product of a specific gene and is generated by depositing a few nano liters of PCR product representing that specific gene usually at concentration of 100-500 µg/ml. The diameter of each spot is also typically 50-150 µm.

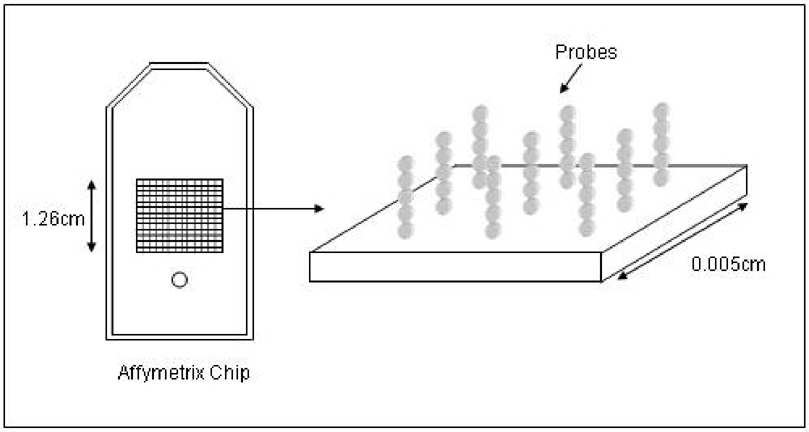


Figure 2. Illustration of a DNA GeneChip (Affymetrix).

Types of DNA microarrays

There are currently two platforms/types of DNA microarrays that are commercially available.

1. Glass DNA microarrays which involves the micro spotting of pre-fabricated cDNA fragments on a glass slide.
2. High-density oligonucleotide microarrays often referred to as a "chip" which involves *in situ* oligonucleotide synthesis.

However, from a manufacturing point of view, there are fundamental differences between the two platforms in regard to the sizes of printed DNA fragments, the methods of printing the DNA spots on the slide/chip, and also the data images generated.

Glass cDNA microarrays

Glass DNA microarrays was the first type of DNA microarray technology developed. It was pioneered by Patrick Brown and his colleagues at Stanford University and is produced by using a robotic device, which deposits (spots) a nanoliter of DNA (50-150 µm in diameter) onto a coated microscope glass slide surface in serial order with a distance of approximately 200-250 µm from each other, one spot-one gene. These moderate sized glass cDNA microarrays also bear about 10,000 spots or more on an area of 3.6 cm2.

As the name suggests, glass cDNA microarrays use specially manufactured glass slides with desired physico-chemical characteristics e.g. excellent chemical resistance against solvents, good mechanical stability (increased thermal strain point) and low intrinsic fluorescence properties.

However, to produce a complete whole genome glass DNA microarray, a series of consecutive steps are followed, ideally each step requiring an appropriate and careful approach. Here, we will not discuss in detail how each step is performed, but briefly outline these steps in the order they are followed.

The first step of manufacturing a glass cDNA microarray is selecting the material to spot onto the microscope glass surface e.g. the genes from public databases/repositories or institutional sources. This is followed by the preparation and purification of DNA sequences representing the gene of interest. In the preparation process, PCR is used to amplify the DNA from library of interest using a universal primers or gene specific primers and the purity of the DNA fragments representing genes of interest are generally checked by sequencing or using on agarose gel to concomitantly obtain an estimate of the DNA concentration. This is an important step because all the DNA fragments should be of similar concentration/molarity and size, to achieve similar reaction kinetics for all hybridisations. The third step is spotting DNA solution onto chemically modified glass slides usually with poly(L-lysine) or other cross-linking chemical coating materials such as polyethyleneimine polymer p-aminophenyl trimethoxysilane/diazotization chemistry and dendrimeric structure. It is these substrates that are coated on the surface of the glass slide that determines how the DNA solution will be immobilised on the surface e.g. covalent or non covalent. However in the course of poly(L-lysine) the negatively charged phosphate groups in the DNA molecule, form an ionic bond with the positively charged amine-derivatised surface. This spotting step is achieved via a contact printing using precisely controlled robotic pins or other equivalent delivering technology such as inkjet printing.

The last step of manufacturing glass DNA microarrays is the post-print processing step involving the drying of the DNA on the slide overnight at room temperature and the use of UV cross-linking to prevent subsequent binding of DNA, and to decrease the background signal upon hybridisation of a labelled target.

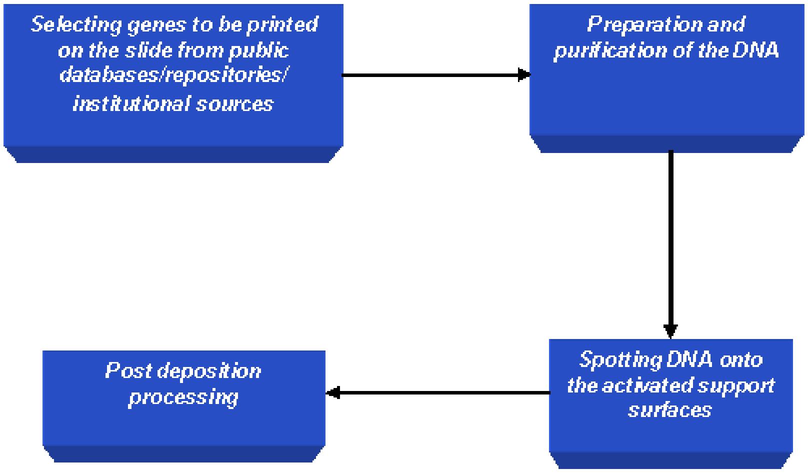

Selecting genes to be printed on the slide from public databases/repositories/ institutional sources
Preparation and purification of the DNA
Spotting DNA onto the activated support surfaces
Post deposition processing

Figure 3. Steps of manufacturing glass cDNA microarrays.

Advantages of cDNA microarrays

Advantages of Glass cDNA microarrays include their relative affordability with a lower cost. Its accessibility requiring no specific equipment for use such that hybridisation does not need specialised equipment, and data capture can be carried out using equipment that is very often already available in the laboratory and flexibility of design as necessitated by the scientific goals of the experiment. In addition to that, Glass cDNA microarrays also have increased detection sensitivity due to longer target sequences (2 kbp).

Disadvantages of cDNA microarrays

Despite their wide spread use, glass cDNA microarray have a few disadvantages such as intensive labour requirement for synthesising, purifying, and storing DNA solutions before microarray fabrication. Further, more printing devices required thus making microarrays more expensive. Also during microarray experiments in the laboratory, sequence homologies between clones representing different closely related members of the same gene family may result in a failure to specifically detect individual genes and instead may hybridise to spot(s) designed to detect transcript from a different gene. This phenomena is known as cross hybridisation.

in situIn situ oligonucleotide array format

In situ (on chip) oligonucleotide array format is a sophisticated platform of microarray technology which is manufactured by using the technology of *in situ* chemical synthesis that was first developed by Stephen Fodor *et al*. (1991). However, the industry leader in the field of *in situ* oligonucleotide microarrays (Affymetrix) has further pioneered this type of technology to manufacture so-called GeneChips which refers to its high density oligonucleotide based DNA arrays.

Presently, the commercial versions of Affymetrix GeneChips hold up to 500,000 probes/sites in a 1.28-cm2 chip area, and due to such very high information content (genes), they are finding widespread use in the hybridisation-based detection and analysis of mutations and polymorphisms, such as single nucleotide polymorphisms or disease-relevant mutations analysis ("genotyping"), as well as a wide range of other applications such gene expression studies, to mention a few.

The basic principles of manufacturing Affymetrix's GeneChips is the use of photolithography and combinatorial chemistry to manufacture short single strands of DNA onto 5-inch square quartz wafers. Unlike glass cDNA, the genes on the chip are designed based on sequence information alone, and then using an industry chip synthesiser, sequences are directly synthesised onto the surface of the 5-inch square quartz wafer at a pre-selected positions.

Detailed stepwise synthesis of *in situ* synthesis of oligonucleotides (Affymetrix GeneChips) is beyond the scope of this overview, but we will briefly outline some of the concepts related to the fabrication process.

The fabrication process of Affymetrix's GeneChips using a DNA photolithography process starts by the derivatization of the solid support, usually quartz with a covalent linker molecule terminated with a photolabile protecting group. This is firstly achieved by washing the quartz to ensure uniform hydroxylation across its surface and then placing it in a bath of silane, which reacts with the hydroxyl groups of the quartz and forms a matrix of covalently linked molecules.

The *in situ* synthesis of oligonucleotides occur in parallel, resulting in consecutive addition of A, C, G and T nucleotides to the appropriate gene sequences on the array. At each step in the synthesis process, oligonucleotide chains that for example require adenine in the next position are deprotected by light at the appropriate positions by a mask. The quartz (chip) is then flooded with a solution containing activated adenine nucleotides with a removable protection group, which are coupled to the deprotected positions. Uncoupled adenine residues are washed away and another mask is applied to further carry out the deprotection of the next nucleotide. Finally, repeating the process ~70 times, with 70 different masks, allows synthesis of the complete array of thousands of 25-mer oligonucleotides in parallel.

Advantages of *in situ* oligonucleotide array format

Advantages offered by the *in situ* oligonucleotide array format include speed, specificity and reproducibility. Speed, in terms of generating the array is prime advantage because, spotting the DNA onto the chip requires only that the DNA sequence of interest be known, therefore no time is spent in the handling of cDNA resources such as the preparation and accurate determination of handling bacterial clones, PCR products, or cDNAs, thus reducing the likelihood of contamination and mix up. However, before manufacturing the array, prior knowledge of the genome sequence is required to design the oligonucleotide sets, and when this is not available, alternative methods of printing isolated genetic material may be preferred.

Other advantages of the *in situ* oligonucleotide array format include high specificity and reproducibility. Both of these attributes are due to the way oligonucleotide sequences to be printed on the chip are designed and the use of multiple, short sequence(s) representing the unique sequence of genes. For example, when designing oligonucleotide sequences for a gene, each sequence is designed to be perfectly complementary to a target gene sequence, at the same time an additional partner sequence is designed that is identical except for a single base mismatch in its centre. This sequence mismatch strategy, along with the use of multiple sequence(s) for each gene increases specificity and helps to identify and minimise the effects of non-specific hybridisation and background signal. This strategy also allows the direct subtraction of cross-hybridisation signals and discrimination between real and non-specific signals.

Disadvantages of *in situ* oligonucleotide array format

There are several disadvantages to the *in situ* oligonucleotide array format including practical limitations in terms of affordability and flexibility. Firstly, *in situ* oligonucleotide array formats tend to have expensive specialised equipments e.g. to carry out the hybridisation, staining of label, washing, and quantitation process. Secondly, ready made in situ oligonucleotide array format (GeneChips) are still expensive, although there has been reductions in cost as the market of microarrays has expanded. Thirdly, although short-sequences used on the array confer high specificity, they may have decreased sensitivity/binding compared with glass cDNA microarrays. Such low sensitivity however is compensated for by using multiple probes.

In situ oligonucleotide array format also offers reduced flexibility although this is not the case with respect to the array design. However, there are occasions when the production of the array, hybridisation and detection equipment are restricted to centralised manufacturer facilities, thus limiting the researcher's flexibility. Similarly, the cost and time needed to manufacture the *in situ* oligonucleotide array format makes it uneconomical for an average laboratory to synthesise its own chips.

Principles of DNA Microarray experiments

The principle of DNA microarray technology is based on the fact that complementary sequences of DNA can be used to hybridise immobilised DNA molecules. This involves three major multi-stage steps;

1- Manufacturing of microarrays: This step involves the availability of a chip or a glass slide with its special surface chemistry, the robotics used for producing microarrays by spotting the DNA (targets) onto the chip or for their *in situ* synthesis.
2- Sample preparation and array hybridisation step: This step involves mRNA or DNA isolation followed by fluorescent labelling of cDNA probes and hybridisation of the sample to the immobilised target DNA.
3- Image acquisition and data analysis: Finally, this step involves microarray scanning, and image analysis using sophisticated software programs that allows us to quantify and interpret the data.

However, here, we will concentrate on how microarray experiments are performed in the laboratory, rather than the technological developments involving array construction or manufacturing using precision robotic devices.

Typically, a microarray experiment involves the comparison of a query or experimental sample representing the expression pattern of genes in a specific set of conditions, with a control sample representing all the genes that are expressed in the cells/tissue to be analysed.

An example of this is comparisons made between expression profiles of bacteria within infected cells (query) and the same bacteria cultured under standardised *in vitro* conditions of growth (control). Or similarly a comparison made between an isogenic mutant and the wildtype strain.

There are four major steps in performing a typical microarray experiment.

1. Sample preparation and labelling
2. Hybridisation
3. Washing
4. Image acquisition and Data analysis

Sample preparation and labelling

There are a number of different ways in which a DNA microarray sample is prepared and labelled. All of these different approaches however have their own advantages and disadvantages with respect to many factors such as the starting amount of RNA or DNA required, through to cost, time and data acquisition and transformation. However the choice of which one to use depends on these factors as well as the type of microarray technology used, for example the slide type and the detection equipment. Here, we used the term "sample" to refer to the free, fluorescently labelled cDNA, not to confuse with the immobilised DNA known as reporter element (s)

Initially, the sample preparation starts by isolating a total RNA containing messenger RNA that ideally represents a quantitative copy of genes expressed at the time of sample collection (experimental sample & reference sample). This step is crucial, simply because the overall success of any microarray experiment depends on the quality of the RNA.

For example purity in the sense of homogeneity or uniformity of the mRNA is a critical factor in the downstream hybridisation performance, particularly when fluorescence is used, as cellular proteins, lipids, and carbohydrates can mediate significant nonspecific binding of labeled cDNAs to matrix surfaces. The sample mRNA extracted from the biological sample of interest and the reference are then separately converted into complementary DNA (cDNA) using a reverse-transcriptase enzyme. This step also requires a short primer to initiate cDNA synthesis. Next, each cDNA (Sample and Control) are labelled with a different tracking molecule, often fluorescent cyanine dyes (i.e. Cy3 and Cy5)

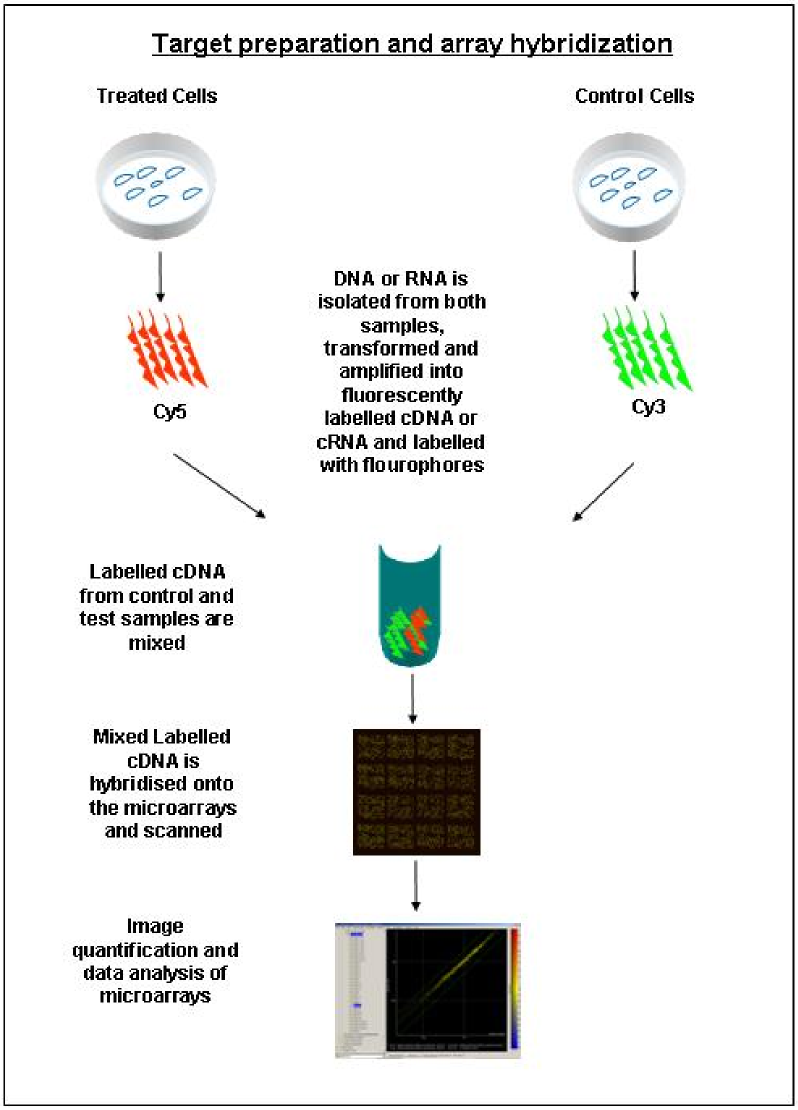
Array hybridisation

Hybridisation is the process of joining two complementary strands of DNA to form a double-stranded molecule. Here, the labelled cDNA (Sample and Control) are mixed together, and then purified to remove contaminants such as primers, unincorporated nucleotides, cellular proteins, lipids, and carbohydrates. Purification is usually carried out using filter spin columns such as Qiaquick from Qiagen. After purification, the mixed labelled cDNA is competitively hybridised against denatured PCR product or cDNA molecules spotted on a glass slide. Ideally, each molecule in the labelled cDNA will only bind to its appropriate complementary target sequence on the immobilised array.

Before hybridisation however, the microarray slides are incubated at high temperature with solutions of saline-sodium buffer (SSC), Sodium Dodecyl Sulfate (SDS) and bovine serum albumin (BSA) to reduce background due to nonspecific binding.

The slides are washed after hybridisation, first to remove any labelled cDNA that did not hybridise on the array, and secondly to increase stringency of the experiment to reduce cross hybridisation. The later is achieved by either increasing the temperature or lowering the ionic strength of the buffers.

Image acquisition and data analysis is the final step of microarray experiments. The aim is to produce an image of the surface of the hybridised array. Here the slide is dried and placed into a laser scanner to determine how much labelled cDNA (probe) is bound to each target spot. Laser excitation of the incorporated targets yield an emission with characteristic spectra, which is measured using a confocal laser microscope. Classically, microarray software often uses green spots on the microarray to represent genes upregulated compared to control, red to represent those genes that are downregulated in the experimental sample, and yellow to represent those genes of equal abundance in both experimental and control samples.



Target preparation and array hybridization

Treated Cells

Control Cells

DNA or RNA is isolated from both samples, transformed and amplified into fluorescently labelled cDNA or cRNA and labelled with flourophores

Labelled cDNA from control and test samples are mixed

Mixed Labelled cDNA is hybridised onto the microarrays and scanned

Image quantification and data analysis of microarrays

Figure 4. Microarray experimental principles.

Applications of DNA Microarray Technology

The range of applications for microarray technology is enormous. However, although in depth details of each one is beyond the scope of this overview, there are two distinctive applications of microarrays that are in wide spread use.

1. Gene expression profiling to measure the expression of genes between different cell populations.
2. Comparative genomics to analyse genomic alterations such as sequence and single nucleotide polymorphisms.

Microarray as a gene expression profiling tool

The principle aim of using microarray technology as a gene expression profiling tool is to answer some of the fundamental questions in biology such as "when, where, and to what magnitude genes of interest are expressed.'' Clearly, if a gene is not expressed in a defined time/condition at a defined cell compartment, then it is possible to imagine it can play no role in that subnetwork. This approach is based on the assumption that cellular genes are expressed in response to a particular state and that the expression profile represents the subset of gene transcripts or mRNA expressed in a cell or tissue. In addition to that, expression profiling by microarray analysis provides another approach to measure changes in the multigene patterns of expression to better understand about regulatory mechanisms and broader bioactivity functions of genes.

It is therefore appreciable that the knowledge obtained from microarray gene expression analysis will probably increase our basic understanding of the cause and consequences of diseases (pathogenesis), how drugs and drug candidates work in cells and organisms, and what gene products might have therapeutic uses or may be studied further as an appropriate targets for therapeutic intervention.

For example, in the context of microbiology, microarray gene expression is used to analyse complex cellular behaviour and to explore the complex interaction between host and microbial pathogens. This ambitious and plausible attempt to understand such molecular interaction is achieved by directly comparing gene expression profiles of host cell to the expression profile of the pathogen. Also an *ex vivo* measurement of gene expression for host cells before and after they are infected with a microbial pathogen can greatly increase our understanding of such complex molecular interplay. Furthermore by following the pattern of gene expression at different times, it is possible to elucidate which host or pathogen genes are up or downregulated over the course of infection to further identify critical target genes and drug-specific targets in both host and microbial pathogens.

Microarray as a comparative genomics tool

Another important application of microarray technology, which is also finding a widespread use is gene mutation analysis to analyse genomic alterations such as sequence and single nucleotide polymorphisms. Currently, in the context of microbiology microarray gene mutation analysis is directed to characterisation of genetic differences among microbial isolates, particularly closely related species. For example detecting the presence or absence of DNA sequences/gene(s) between pathogenic and non-pathogenic strains of the same species. Such informative approach would allow us to reveal genes exclusively present in the former that may be required for infectivity, virulence or adaptation to a particular host niche

Similarly, DNA microarray technology can be used to compare between a fully sequenced genome and an unsequenced but related genome of closely related bacteria. Interestingly, such approach can provide us a valuable information about the diversity and evolution of pathogens and symbionts. Comparisons of this kind use a microarray containing representations of all the open reading frames (ORFs) of the sequenced, reference strain and labelled DNA from the unsequenced, experimental strain. The resulting hybridised array will then reveal genes common to both strains and genes that are present in the reference strain but absent in the experimental strain. This method, however, may not detect genes present in the experimental strain, but missing in the reference strain, although the use of multi-genome (species array) may detect genes present in the experimental strain. This method may not also detect point mutations, including frame-shift mutations, small deletions and deletions in homologous repetitive elements, rearrangements of the genome that have not resulted in deletion of a gene from the experimental strain. However the use of affymetrix gene chip can identify these mutations.

An elegant and well known example of using microarray as a comparative genomics tool is the comparison made by Behr et al., 1999 between several *Mycobacterium bovis* vaccine strains e.g. the genome composition of the sequenced M. tuberculosis laboratory strain H37Rv with the closely related pathogenic species, M. bovis, and with several strains of the bacille Calmette-Guerin (BCG) vaccine variant that was produced by serial *in vitro* passage of M. bovis between 1908 and 1921.

This particular study, has revealed several chromosomal deletions in the different vaccine strains in comparison to their progenitor, these deletions are possibly thought to be responsible for the variable effectiveness of the BCG vaccine seen worldwide.

⇧

⇦

BMC Bioinformatics

MEETING ABSTRACT Open Access

Network analysis of gene fusions in human cancer

Morgan Harrell[1], Junfeng Xia[1], Zhongming Zhao[1,2,3]*

From 12th Annual UT-ORNL-KBRIN Bioinformatics Summit 2013
Buchanan, TN, USA. 22-24 March 2013

Background

Gene fusions are hybrid genes formed when two discrete genes are incorrectly joined together. Gene fusions are found to play roles in tumorigenesis. For example, the fusion gene *BCR-ABL* translates into an abnormal tyrosine kinase that accelerates development of chronic myelogenous leukemia [1]. A network is a relational representation of nodes (e.g., genes) with edges, and is a useful approach to explore biological interactions among many related nodes. Network analysis of gene fusions in cancer would aid the exploration of gene fusion occurrence and association with tumorigenesis. Hoglund et al [2] performed an initial investigation of gene fusions network after collecting 291 tumorigenesis related gene fusions from the Mitelman database in 2006. Since then, gene fusion data has exponentially increased. There is no current and comprehensive cancer-related gene fusion network to assist in targeting cancer-associated genes.

Materials and methods

We mined three public databases for cancer-related gene fusion sequences, and one database for fusions records from cancer studies, transcriptome analysis, and genetic disorders. Specifically, we processed each dataset by removing incomplete entries and then extracted gene-fusion pairs. Genes serve as the nodes in the network and each fusion pair is joined by an edge. Repeating pairs were represented once. We used Cytoscape to build five networks: one for each dataset and the fifth that encompasses all datasets. We graphed the occurrence of degree in each single dataset network to determine an empirical definition for hub genes.

Results

The comprehensive network includes 9852 genes, displays 12,791 relationships, and highlights 1248 hub genes. The network highlights genes such as *MLL* and *MALAT1*, both of which have roles in tumorigenesis. The network also highlights genes such as *WDR74* and *COL1A1*, which are not much studied.

Conclusions

This preliminary network analysis provides interesting features of tumorigenesis-related fusions. Further systematic analysis of gene fusion networks may aid researchers to better understand cancer gene fusions and test novel fusions in specific types of cancer.

Acknowledgements

We would like to thank the members in Bioinformatics and Systems Medicine Laboratory for their valuable discussion in this project. This work was partially supported by the National Library of Medicine Training Grant 2T15LM007450-11 and the Stand Up To Cancer-American Association for Cancer Research Innovative Research Grant (SU2C-AACR-IRG0109) and the VICC Cancer Center Core grant P30CA68485 from National Institutes of Health.

Authors' details

[1]Department of Biomedical Informatics, Vanderbilt University School of Medicine, Nashville, TN 37203, USA. [2]Department of Psychiatry, Vanderbilt University School of Medicine, Nashville, TN 37232, USA. [3]Department of Cancer Biology, Vanderbilt University Medical Center, Nashville, TN 37232, USA.

Published: 22 October 2013

References

1. Bartram CR, de Klein A, Hagemeijer A, van Agthoven T, Geurts van Kessel A, Bootsma D, Grosveld G, Ferguson-Smith MA, Davies T, Stone M, *et al*: **Translocation of c-ab1 oncogene correlates with the presence of a Philadelphia chromosome in chronic myelocytic leukaemia.** *Nature* 1983, **306**:277-280.
2. Hoglund M, Frigyesi A, Mitelman F: **A gene fusion network in human neoplasia.** *Oncogene* 2006, **25**:2674-2678.

doi:10.1186/1471-2105-14-S17-A13
Cite this article as: Harrell *et al.*: Network analysis of gene fusions in human cancer. *BMC Bioinformatics* 2013 **14**(Suppl 17):A13.

* Correspondence: zhongming.zhao@vanderbilt.edu
[1]Department of Biomedical Informatics, Vanderbilt University School of Medicine, Nashville, TN 37203, USA
Full list of author information is available at the end of the article

KPP: KEGG Pathway Painter

Article *in* BMC Systems Biology · April 2015

DOI: 10.1186/1752-0509-9-S2-S3 · Source: PubMed

Ganiraju Manyam
University of Texas MD Anderson Cancer Center
63 PUBLICATIONS **1,659** CITATIONS
SEE PROFILE

Aybike Birerdinc
George Mason University
88 PUBLICATIONS **943** CITATIONS
SEE PROFILE

Ancha Baranova
George Mason University
323 PUBLICATIONS **5,281** CITATIONS
SEE PROFILE

Manyam *et al. BMC Systems Biology* 2015, **9**(Suppl 2):S3
http://www.biomedcentral.com/1752-0509/9/S2/S3

BMC
Systems Biology

RESEARCH Open Access

KPP: KEGG Pathway Painter

Ganiraju Manyam[1,2], Aybike Birerdinc[1], Ancha Baranova[1,3*]

From IX International Conference on the Bioinformatics of Genome Regulation and Structure\Systems Biology (BGRS\SB-2014)
Novosibirsk, Russia. 23-28 June 2014

Abstract

Background: High-throughput technologies became common tools to decipher genome-wide changes of gene expression (GE) patterns. Functional analysis of GE patterns is a daunting task as it requires often recourse to the public repositories of biological knowledge. On the other hand, in many cases researcher's inquiry can be served by a comprehensive glimpse. The KEGG PATHWAY database is a compilation of manually verified maps of biological interactions represented by the complete set of pathways related to signal transduction and other cellular processes. Rapid mapping of the differentially expressed genes to the KEGG pathways may provide an idea about the functional relevance of the gene lists corresponding to the high-throughput expression data.

Results: Here we present a web based graphic tool KEGG Pathway Painter (KPP). KPP paints pathways from the KEGG database using large sets of the candidate genes accompanied by "overexpressed" or "underexpressed" marks, for example, those generated by microarrays or miRNA profilings.

Conclusion: KPP provides fast and comprehensive visualization of the global GE changes by consolidating a list of the color-coded candidate genes into the KEGG pathways. KPP is freely available and can be accessed at http://web.cos.gmu.edu/~gmanyam/kegg/

Background

High-throughput technologies became common tools to decipher genome-wide changes of gene expression (GE) patterns or relative protein abundance. Typical output of these large-scale studies is represented by the list comprised of hundreds of gene candidates with attached quantitative labels. Functional analysis of these gene lists is a daunting task as it requires regular recourse to the public repositories of biological knowledge or use of expensive databases of manually curated biological annotation [1,2]. On the other hand, in many cases researcher's inquiry can be successfully served by a comprehensive glimpse.

Functional analysis of markers identified from large-scale datasets can be performed using a wide variety of bioinformatics tools. As microarrays became a common tool to decipher global gene expression, centralized systems like Gene Expression Omnibus (GEO), ArrayExpress was developed to congregate the valuable profile data [3,4]. An analysis of combined datasets generated in independent microarray experiments (so-called "microarray meta-analysis"), is often being employed [5], for example, to develop biomarker panels or to extract insights into the pathogenesis of various chronic diseases [6] including human malignancies [7]. Meta-analysis lead to an increase of the complexity in microarray analysis; therefore, sophistication of subsequent functional analysis also increased. Gene Ontology (GO) and other pathway-centered types of analysis became indispensable [8].

KEGG (Kyoto Encyclopedia of Genes and Genomes) is a compendium of databases covering both annotated genomes and protein interaction networks for all sequenced organisms. Its integral part, KEGG PATHWAY, is a compilation of manually verified pathway maps displaying both the molecular interactions and the biochemical reactions [9]. The recent version of this database includes a complete set of pathways related to signal transduction and other cellular processes [10]. The extensive collection

* Correspondence: abaranov@gmu.edu
[1]School of Systems Biology, George Mason University, Fairfax, VA - 22030, USA
Full list of author information is available at the end of the article

of the pathways at KEGG can be utilized for the rapid graphical evaluation of the functional relevance of the observed changes in GE patterns. This will save the precious time of the expert biologists and bioinformatics specialists.

Pathways assembled into the KEGG database are displayed as semi-static objects that can be manipulated using tools like KGML and KEGG application programmable interface (API) [11,12]. KEGG API provides a routine that highlights specified genes within the particular metabolic pathway (http://www.genome.jp/kegg/tool/color_pathway.html). Similar task may be also executed using G-language Genome Analysis Environment [13]. Both approaches work on the pathway by pathway basis. Another tool, Pathway Express, calculates the pathway-wise impact of differentially expressed genes based on normalized fold change and depicts the pathways with differentially expressed genes [14]. However, the fold-change approach and its associated standard t-test statistics usually produce severely over-fitted models. A number of recently developed approaches generate gene rankings dissociated from the fold change estimates [15,16]. An analysis of these gene lists may benefit from the binary graphical mapping of upregulated and downregulated elements within the complete collection of pathway maps. Resulting graphical pictures may be helpful both as tool for a quick assessment of the functional relevance of a gene list and as a set of the snapshots easily convertible into the illustrative material for presentations or manuscript figures.

With this notion, here we present a web-based tool, KEGG Pathway Painter (KPP). KPP performs a batch painting of relevant pathways according to the uploaded lists of up-regulated and down-regulated genes in KEGG. KPP returns a set of images that give a holistic perspective to the functional importance of the change in the GE patterns revealed by a given high-throughput experiment and facilitate the extraction of the biological insights.

Implementation

KPP was implemented using PERL/CGI. Pathways assembled into the KEGG database are displayed as semi-static objects that can be manipulated using tools like KGML (KEGG Markup Language) and KEGG API (Application Programming Interface). The API allows access to the resources stored in KEGG system in an interactive and user-friendly way (http://www.genome.jp/kegg/rest/).

KEGG Pathway Painter (KPP) accepts the up-regulated and down-regulated gene lists as two different text files containing the gene identifiers of any sequenced organism. Permitted identifiers include GenBank id, NCBI GENE id, NCBI GI accession, Unigene ID and Uniprot ID. Conversion of the gene identifiers, extraction of the corresponding pathway and their painting is performed by specific API routines. The KPP processes data through direct interface to the KEGG database, and therefore, the KPP painted pathways are always up-to date with reference to KEGG knowledgebase. In KPP, genes of interest can be also highlighted with user-specified foreground and background colors allowing easy visual differentiating of up- and down-regulated genes.

Mapped genes are automatically consolidated within each pathway. The number of the KPP returned pathways could be filtered by either the total number of the painted genes in a given pathway or the ratio of painted genes to the total number of genes in a given pathway. The chosen pathways passing the criteria on filter are color coded according to users' preferences. Users can browse through these high-resolution pathway images along with gene information and an archive of the painted pathways can also be saved for future reference. After completion of the query, the URL to the index of resulting output images is e-mailed to the user along with the job summary.

Results and discussion

The motivation for the development of KPP came up from the idea to build a user-friendly, platform-independent and simple tool to visualize the placement of genes in their associated pathways. The simplicity of KPP is due to the acceptance of gene identifiers without reference to respective microarray platform. This isolation enhances its utility for the studies of the data from Real-Time-PCR or medium-throughput platforms or even for validation of the various hypotheses concerning an involvement of the groups of genes in one or another biological process.

This utility of KPP was demonstrated by highlighting of cell cycle related events using the publicly available prostate carcinoma dataset (GDS1439) [17] from the NCBI GEO database (see Figure 1), by aiding the selection of the mutations and epigenetic events to be tested as a companion diagnostics of treatment susceptibility and resistance in non-small lung carcinoma patients (not shown) and an analysis of the host-associated risk factors associated with lack of sustained virological response (SVR) in various cohorts of HCV patients [18,19].

In one of these examples, KPP-aided visual parsing the pathways encompassing molecular components relevant to HCV pathogenesis allowed to pinpoint the Janus kinase-signal transducers and activators of transcription signaling cascade as the major pathogenetic component responsible for not achieving SVR [18], a conclusion that was later confirmed in *in vitro* experiments with blocking antibodies, a pharmacological inhibitor, and siRNAs [20].

In another example, KPP allowed to visualize a sustained pattern of treatment-induced gene expression in patients carrying interferon/ribavirin-responding IL28B genotype C/C, while in patients with therapy-resistant

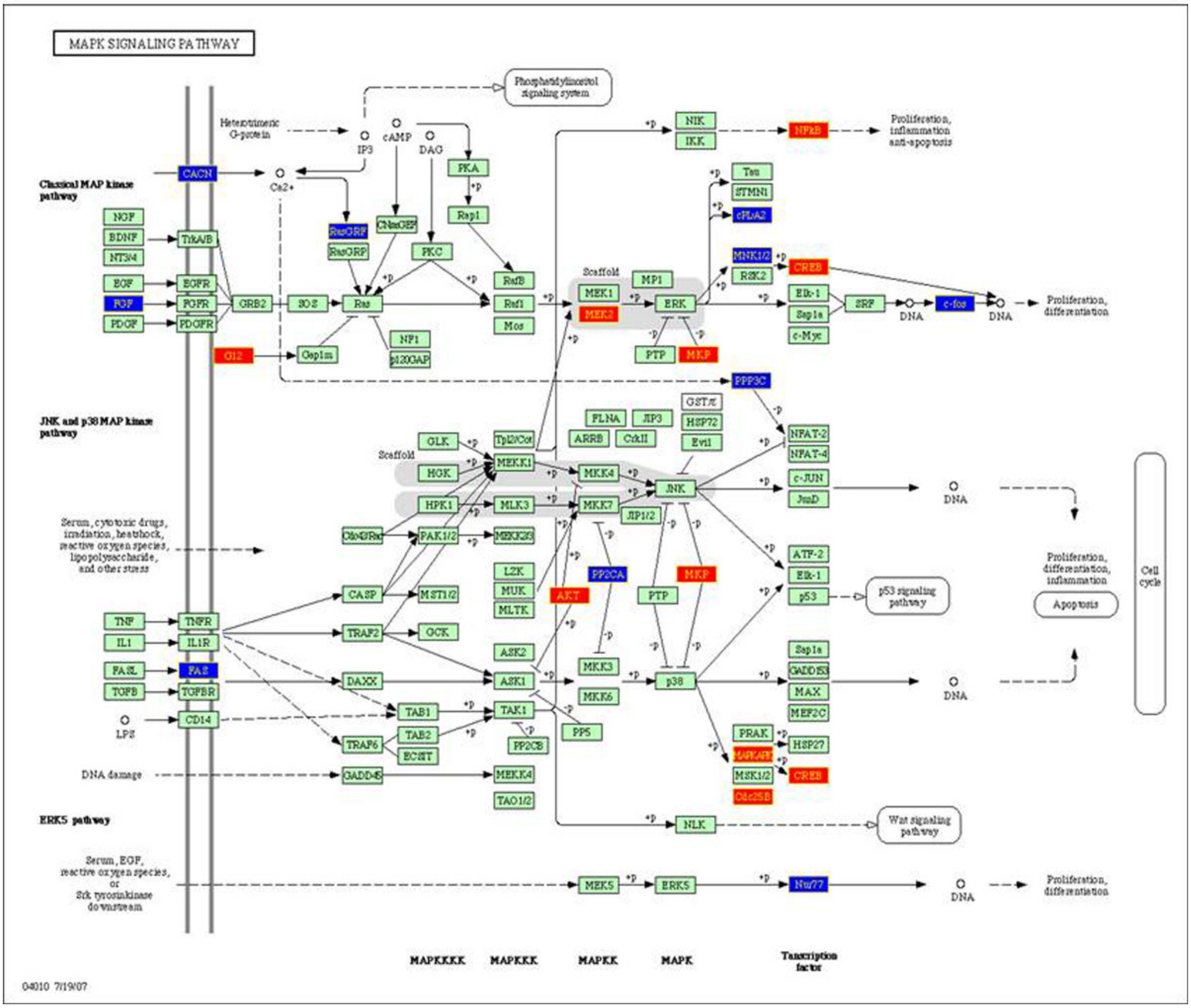


Figure 1 Image of the MAPK signaling pathway painted by KPP according to the imported list of genes differentially expressed in the prostatic carcinoma as compared to normal prostate. Red and blue boxes represent up- and down- regulated genes, respectively. The genes in green background represent the species specific genes (*Homo sapiens*, in this case).

IL28B T* genotype, the background pre-activation of interferon-dependent genes precluded further therapeutic boost [19]. Thus, KPP provided a critical insight into the lower rate of SVR observed in these patients. Furthermore, KPP analysis revealed LI28B genotype independent role of SOCS1 in therapeutic response [19]. This KPP-aided hypothesis was later investigated both in vitro experiments showing that SOCS1 acts as a suppressor of type I IFN function against HCV [21] and in serum samples interferon/ribavirin-treated Hepatitis C patients who demonstrated that methylation-based silencing of SOCS-1 is associated with better therapeutic response [22]. Thus, KPP was indispensable in acquiring mechanistic insights into the differential therapeutic response in Hepatitis C infected patients.

The major fetching point of the KPP tool lies in its tight connection with the KEGG database, as this will allow for the pathway visualization of every sequenced organism. However this flexibility comes at the cost of possible KEGG-attributed delay of the data transfer, the resultant tool is substantially more convenient for the user than the tools embed into existing pathway analysis environment, for example, Cytoskape (http://www.cytoscape.org/). Another commonly used pathway parsing tool, Reactome Skypainter (http://www.reactome.org/), is restricted to underlying knowledge base and, therefore, limits the potential set of insights to be extracted.

It is important to note that the painting of individual pathways can be performed through by the KEGG website itself (http://www.genome.jp/kegg/), however, the practicality of KPP is in its comprehensive visual representation of up- and downregulated genes in the KEGG dataset as a whole. In other words, KPP allows one to extract immediate and visual insights about cumulative change in each pathway under scrutiny. Users can browse through high-resolution pathway images and download an archive of the painted pathways that may be used as figures for upcoming manuscripts.

Conclusion

In summary, KPP provides fast and comprehensive visualization of the global GE changes by consolidating a list of the color-coded candidate genes into the KEGG pathways.

List of abbreviations:

KPP - KEGG Pathway Painter
GE - Gene Expression
KGML - KEGG Markup Language
API - Application Programming Interface
SVR - Sustainer Virological Response

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

All three authors contributed to the study design, interpretation of results and producing the manuscript. All authors read and approved the final manuscript.

Acknowledgements

The authors express gratitude to the general support provided by College of Science, George Mason University, a State Contract 14.607.21.0098 dated November 27th, 2014 (Ministry of Science and Education, Russia) and by the Human Proteome Scientific Program of the Federal Agency of Scientific Organizations, Russia.
Availability of support data: The data supporting the results of this article are included within the article and on the publicly available website http://web.cos.gmu.edu/~gmanyam/kegg/kpp.html.

Declarations

Open access fees were covered by funds of College of Science, George Mason University.
This article has been published as part of *BMC Systems Biology* Volume 9 Supplement 2, 2015: Selected articles from the IX International Conference on the Bioinformatics of Genome Regulation and Structure\Systems Biology (BGRS \SB-2014): Systems Biology. The full contents of the supplement are available online at http://www.biomedcentral.com/bmcsystbiol/supplements/9/S2.

Authors' details

[1]School of Systems Biology, George Mason University, Fairfax, VA - 22030, USA. [2]The UT MD Anderson Cancer Center, Houston, TX - 77030, USA. [3]Research Centre for Medical Genetics RAMS, Moscow, Russia.

Published: 15 April 2015

References

1. Ganter B, Giroux CN: **Emerging applications of network and pathway analysis in drug discovery and development.** *Curr Opin Drug Discov Devel* 2008, **11(1)**:86-94.
2. Chen G, Cairelli MJ, Kilicoglu H, Shin D, Rindflesch TC: **Augmenting microarray data with literature-based knowledge to enhance gene regulatory network inference.** *PLoS Comput Biol* 2014, **10(6)**:e1003666.
3. Kolesnikov N, Hastings E, Keays M, Melnichuk O, Tang YA, Williams E, Dylag M, Kurbatova N, Brandizi M, Burdett T, Megy K, Pilicheva E, Rustici G, Tikhonov A, Parkinson H, Petryszak R, Sarkans U, Brazma A: **ArrayExpress update-simplifying data submissions.** *Nucleic Acids Res* 2015, **43(Database)**:D1113-6.
4. Barrett T, Troup DB, Wilhite SE, Ledoux P, Rudnev D, Evangelista C, Kim IF, Soboleva A, Tomashevsky M, Marshall KA, Phillippy KH, Sherman PM, Muertter RN, Edgar R: **NCBI GEO: archive for high-throughput functional genomic data.** *Nucleic Acids Res* 2009, **37(Database)**:D885-90.
5. Tseng GC, Ghosh D, Feingold E: **Comprehensive literature review and statistical considerations for microarray meta-analysis.** *Nucleic Acids Res* 2012, **40(9)**:3785-99.
6. Mayburd A, Baranova A: **Knowledge-based compact disease models identify new molecular players contributing to early-stage Alzheimer's disease.** *BMC Syst Biol* 2013, **7**:121.
7. Yang Z, Chen Y, Fu Y, Yang Y, Zhang Y, Chen Y, Li D: **Meta-analysis of differentially expressed genes in osteosarcoma based on gene expression data.** *BMC Med Genet* 2014, **15**:80.
8. Carnielli CM, Winck FV, Paes Leme AF: **Functional annotation and biological interpretation of proteomics data.** *Biochim Biophys Acta* 2015, **1854(1)**:46-54.
9. Tanabe M, Kanehisa M: **Using the KEGG database resource.** *Curr Protoc Bioinformatics* 2012, **Chapter 1(Unit1.12)**.
10. Kanehisa M, Goto S, Sato Y, Furumichi M, Tanabe M: **KEGG for integration and interpretation of large-scale molecular data sets.** *Nucleic Acids Res* 2012, **40(Database)**:D109-14.
11. Kawashima S, Katayama T, Sato Y, Kanehisa M: **KEGG API: A web service using SOAP/WDSL to access the KEGG system.** *Genome Informatics* 2003, **14**:673-674.
12. Klukas C, Schreiber F: **Dynamic exploration and editing of KEGG pathway diagrams.** *Bioinformatics* 2007, **23(3)**:344-50.
13. Arakawa K, Mori K, Ikeda K, Matsuzaki T, Kobayashi Y, Tomita M: **G-language Genome Analysis Environment: a workbench for nucleotide sequence data mining.** *Bioinformatics* 2003, **19(2)**:305-6.
14. Khatri P, Sellamuthu S, Malhotra P, Amin K, Done A, Draghici S: **Recent additions and improvements to the Onto-Tools.** *Nucleic Acids Res* 2005, **33(Web Server)**:W762-5.
15. Simon R: **Microarray-based expression profiling and informatics.** *Curr Opin Biotechnol* 2008, **19(1)**:26-9.
16. Emmert-Streib F, Glazko GV, Altay G, de Matos Simoes R: **Statistical inference and reverse engineering of gene regulatory networks from observational expression data.** *Front Genet* 2012, **3**:8.
17. Varambally S, Yu J, Laxman B, Rhodes DR, Mehra R, Tomlins SA, Shah RB, Chandran U, Monzon FA, Becich MJ, Wei JT, Pienta KJ, Ghosh D, Rubin MA, Chinnaiyan AM: **Integrative genomic and proteomic analysis of prostate cancer reveals signatures of metastatic progression.** *Cancer Cell* 2005, **8(5)**:393-406.
18. Birerdinc A, Afendy A, Stepanova M, Younossi I, Manyam G, Baranova A, Younossi ZM: **Functional pathway analysis of genes associated with response to treatment for chronic hepatitis C.** *J Viral Hepat* 2010, **17(10)**:730-6.
19. Younossi ZM, Birerdinc A, Estep M, Stepanova M, Afendy A, Baranova A: **The impact of IL28B genotype on the gene expression profile of patients with chronic hepatitis C treated with pegylated interferon alpha and ribavirin.** *J Transl Med* 2012, **10**:25.
20. Zhang L, Jilg N, Shao RX, Lin W, Fusco DN, Zhao H, Goto K, Peng LF, Chen WC, Chung RT: **IL28B inhibits hepatitis C virus replication through the JAK-STAT pathway.** *J Hepatol* 2011, **55(2)**:289-98.
21. Shao RX, Zhang L, Hong Z, Goto K, Cheng D, Chen WC, Jilg N, Kumthip K, Fusco DN, Peng LF, Chung RT: **SOCS1 abrogates IFN's antiviral effect on hepatitis C virus replication.** *Antiviral Res* 2013, **97(2)**:101-7.
22. Tseng KC, Chou JL, Huang HB, Tseng CW, Wu SF, Chan MW: **SOCS-1 promoter methylation and treatment response in chronic hepatitis C patients receiving pegylated-interferon/ribavirin.** *J Clin Immunol* 2013, **33(6)**:1110-6.

doi:10.1186/1752-0509-9-S2-S3
Cite this article as: Manyam *et al.*: **KPP: KEGG Pathway Painter.** *BMC Systems Biology* 2015 **9**(Suppl 2):S3.

Published online 28 October 2014

Nucleic Acids Research, 2015, Vol. 43, Database issue ***D447–D452***
doi: 10.1093/nar/gku1003

STRING v10: protein–protein interaction networks, integrated over the tree of life

Damian Szklarczyk[1], Andrea Franceschini[1], Stefan Wyder[1], Kristoffer Forslund[2], Davide Heller[1], Jaime Huerta-Cepas[2], Milan Simonovic[1], Alexander Roth[1], Alberto Santos[3], Kalliopi P. Tsafou[3], Michael Kuhn[4,5], Peer Bork[2,*], Lars J. Jensen[3,*] and Christian von Mering[1,*]

[1]Institute of Molecular Life Sciences and Swiss Institute of Bioinformatics, University of Zurich, 8057 Zurich, Switzerland, [2]European Molecular Biology Laboratory, 69117 Heidelberg, Germany, [3]Novo Nordisk Foundation Center for Protein Research, University of Copenhagen, 2200 Copenhagen N, Denmark, [4]Biotechnology Center, Technische Universität Dresden, 01062 Dresden, Germany and [5]Max Planck Institute of Molecular Cell Biology and Genetics, 01062 Dresden, Germany

Received September 15, 2014; Accepted October 07, 2014

ABSTRACT

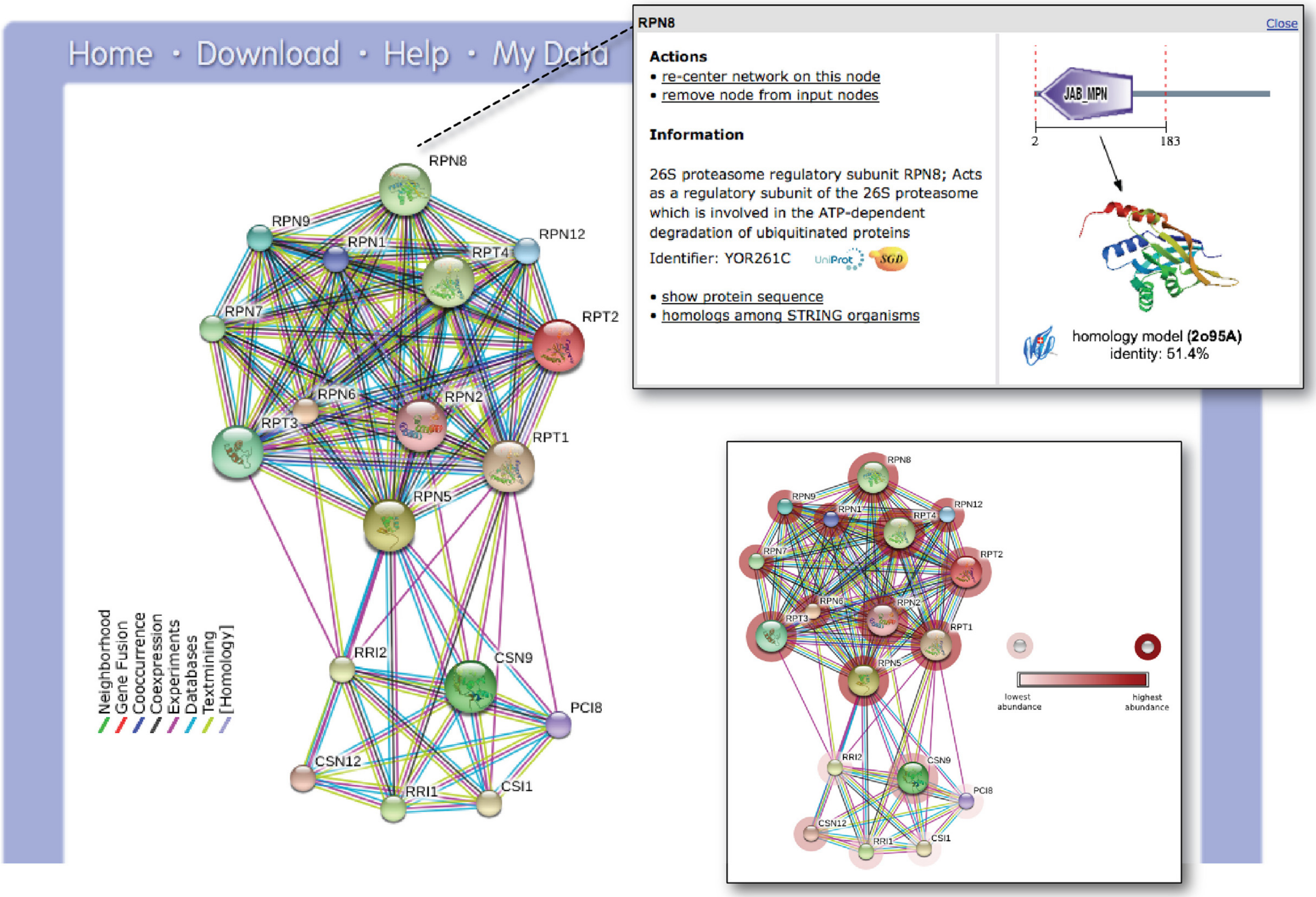
The many functional partnerships and interactions that occur between proteins are at the core of cellular processing and their systematic characterization helps to provide context in molecular systems biology. However, known and predicted interactions are scattered over multiple resources, and the available data exhibit notable differences in terms of quality and completeness. The STRING database (http://string-db.org) aims to provide a critical assessment and integration of protein–protein interactions, including direct (physical) as well as indirect (functional) associations. The new version 10.0 of STRING covers more than 2000 organisms, which has necessitated novel, scalable algorithms for transferring interaction information between organisms. For this purpose, we have introduced hierarchical and self-consistent orthology annotations for all interacting proteins, grouping the proteins into families at various levels of phylogenetic resolution. Further improvements in version 10.0 include a completely redesigned prediction pipeline for inferring protein–protein associations from co-expression data, an API interface for the *R* computing environment and improved statistical analysis for enrichment tests in user-provided networks.

INTRODUCTION

For a full description of a protein's function, knowledge about its specific interaction partners is an important prerequisite. The concept of protein 'function' is somewhat hierarchical (1–4), and at all levels in this hierarchy, interactions between proteins help to describe and narrow down a protein's function: its three-dimensional structure may become meaningful only in the context of a larger protein assembly, its molecular actions may be regulated by co-operative binding or allostery, and its cellular context may be controlled by a multitude of transport, sequestering, and signaling interactions. Given this importance of interactions, many protein annotation and classification schemes assign groups of interacting proteins into functional sets, designated either as physical complexes, signaling pathways or tightly linked 'modules' (1,5–7). However, the partitioning of interactions into distinct pathways or complexes can be somewhat arbitrary, and may not do justice to the prevalence of crosstalk and dynamic variation in the interaction landscape (8). A widely used concept that avoids partitioning of function arbitrarily is the *protein network*, i.e. the topological summary of all known or predicted protein interactions in an organism. For functional studies, arguably the most useful networks are those that integrate all types of interactions: stable physical associations, transient binding, substrate chaining, information relay and others. The STRING database (Search Tool for the Retrieval of Interacting Genes/Proteins) is dedicated to such *functional associations* between proteins, on a global scale.

Protein–protein interaction information can already be retrieved from a number of online resources. First, primary interaction databases (e.g. 9–13) which are largely collabo-

[*]To whom correspondence should be addressed. Tel: +41 44 6353147; Fax: +41 44 6356864; mering@imls.uzh.ch
Correspondence may also be addressed to Peer Bork. Tel: +49 6221 387 8526; Fax: +49 6221 387 517; bork@embl.de
Correspondence may also be addressed to Lars J. Jensen. Tel: +45 353 25025; Fax: +45 353 25001; lars.juhl.jensen@cpr.ku.dk



26S proteasome regulatory subunit RPN8; Acts as a regulatory subunit of the 26S proteasome which is involved in the ATP-dependent degradation of ubiquitinated proteins

Figure 1. The STRING network view. Combined screenshots from the STRING website, which has been queried with a subset of proteins belonging to two different protein complexes in yeast (the COP9 signalosome, as well as the proteasome). Colored lines between the proteins indicate the various types of interaction evidence. Protein nodes which are enlarged indicate the availability of 3D protein structure information. Inset top right: for each protein, accessory information is available which includes annotations, cross-links and domain structures. Inset bottom right: the same network is shown after the addition of a user-configurable 'payload'-dataset (26). In this case, the payload corresponds to color-coded protein abundance information, and reveals systematic differences in the expression strength of both complexes.

rating (14,15) provide curated experimental data originating from a variety of biochemical, biophysical and genetic techniques. Second, since protein–protein interactions can also be predicted computationally, a number of resources have their main focus on interaction prediction, using a variety of algorithms (e.g. 16–20). Lastly, a group of online resources is providing an integration of both known and predicted interactions, thus aiming for high comprehensiveness and coverage. These include STRING, as well as GeneMANIA (21), FunCoup (18), I2D (22), ConsensusPathDB (22) and others. Within this landscape of online resources, STRING places its focus on interaction confidence scoring, comprehensive coverage (in terms of number of proteins, organisms and prediction methods), intuitive user interfaces and on a commitment to maintain a long-term, stable resource (since 2000).

The basic interaction unit in STRING is the *functional association*, i.e. a specific and productive functional relationship between two proteins, likely contributing to a common biological purpose. Interactions are derived from multiple sources: (i) known experimental interactions are imported from primary databases, (ii) pathway knowledge is parsed from manually curated databases, (iii) automated text-mining is applied to uncover statistical and/or semantic links between proteins, based on Medline abstracts and a large collection of full-text articles, (iv) interactions are predicted *de novo* by a number of algorithms using genomic information (23–25) as well as by co-expression analysis and (v) interactions that are observed in one organism are systematically transferred to other organisms, via pre-computed orthology relations. STRING centers on protein-coding gene loci—alternative splice isoforms or post-translationally modified forms are not resolved, but are instead collapsed at the level of the gene locus. All sources of interaction evidence are benchmarked and calibrated against previous knowledge, using the high-level functional groupings provided by the manually curated Kyoto Encyclopedia of Genes and Genomes (KEGG) pathway maps (5).

As of the current update to version 10.0, the number of organisms covered by STRING has increased to 2031, almost doubling over the previous release. The update also

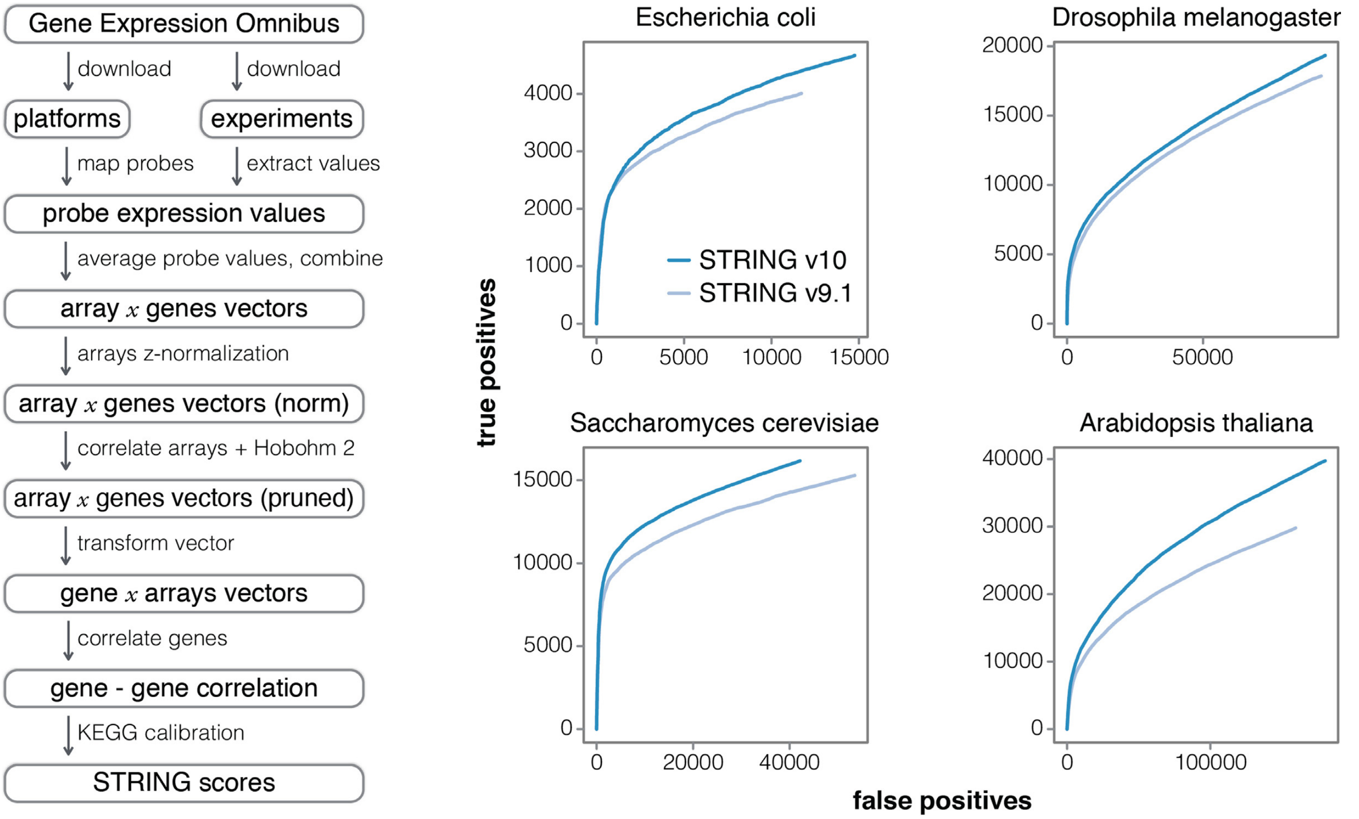


Figure 2. Improved Co-expression analysis. STRING v10 features a completely re-designed pipeline for accessing and processing gene expression information. Left: overview of the individual steps; note that redundant expression experiments are now detected and pruned automatically. Right: improved benchmark performance of the resulting co-expression links, relative to the previous version of STRING, in four model organisms (ROC curves). The benchmark is based on the KEGG pathway maps; predicted interactions are considered to be true positives when both interacting proteins are annotated to the same KEGG map.

encompassed importing and processing all primary data sources again, re-running all prediction algorithms and re-executing the entire text-mining pipeline with new dictionaries and extended text collections. Many of the features and interfaces of STRING have already been described previously (26–28). Below, we have given a short overview of the resource and describe recent additions and modifications.

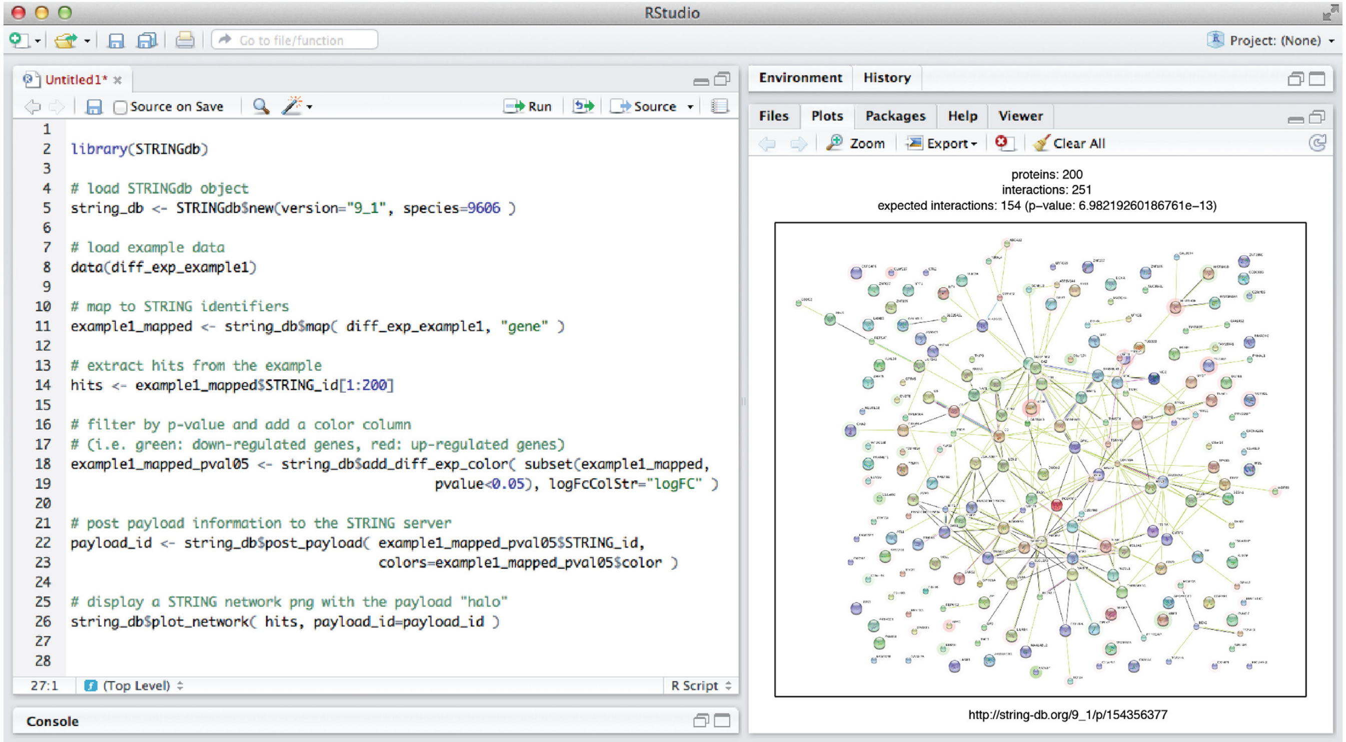
User interface

The main entry point into the STRING website is the protein search box on its start page. It supports queries for multiple proteins, can be restricted to certain organisms or clades of organisms, and uses a weighted scheme to rank annotation text matches and identifier matches. Users can also arrive via a number of external websites (29–32) that maintain cross-links with STRING, including the partner resources Search Tool for Interactions of Chemicals (STITCH; 33) and eggNOG (34)—the latter both share protein sequences, annotations and name-spaces with STRING. A third way to enter STRING is via logging on to the *My Data* section; this allows users to upload gene-lists, create identifier mappings, view their browsing history and provide additional 'payload' data to be displayed alongside the interactions.

Once a protein or set of proteins is identified, users proceed to the network view (Figure 1). From there, it is possible to inspect the interaction evidence, to re-adjust the score-cutoffs and network size limits and to view detailed information about the interacting proteins. Upon switching to the 'advanced' mode (via the tool panel below the network), users can also cluster and rearrange the network and test for statistical enrichments in the network. The latter feature has been enhanced for the current version 10.0 of STRING: enrichment detection now also covers human disease associations and tissue annotations, which might be statistically enriched in a given network. For this feature, STRING connects with the partner databases TISSUES (http://tissues.jensenlab.org) and DISEASES (http://diseases.jensenlab.org), which also share sequence and name spaces with STRING, and which annotate proteins to tissues or to disease entities based on a combination of automated text-mining and knowledge imports.

Interaction transfer between organisms

Since version 6.0 of STRING, a significant source of interactions for any given organism has been the transfer of interaction knowledge from orthologous proteins observed to be interacting in another organism. Since version 9.1, these so-called 'interolog' transfers were based on pre-computed



```

library(STRINGdb)

# load STRINGdb object
string_db <- STRINGdb$new(version="9_1", species=9606 )

# load example data
data(diff_exp_example1)

# map to STRING identifiers
example1_mapped <- string_db$map( diff_exp_example1, "gene" )

# extract hits from the example
hits <- example1_mapped$STRING_id[1:200]

# filter by p-value and add a color column
# (i.e. green: down-regulated genes, red: up-regulated genes)
example1_mapped_pval05 <- string_db$add_diff_exp_color( subset(example1_mapped,
                                                       pvalue<0.05), logFcColStr="logFC" )

# post payload information to the STRING server
payload_id <- string_db$post_payload( example1_mapped_pval05$STRING_id,
                                      colors=example1_mapped_pval05$color )

# display a STRING network png with the payload "halo"
string_db$plot_network( hits, payload_id=payload_id )


```

```
proteins: 200
interactions: 251
expected interactions: 154 (p-value: 6.98219260186761e-13)
```

http://string-db.org/9_1/p/154356377

Figure 3. Access to STRING from R/Bioconductor. Left: example session describing how to initialize a human protein network from the STRING database backend, and how to map a set of gene names against it. A subset of the proteins is then plotted as a STRING network (right), complete with auxiliary numerical payload-information highlighting some nodes of interest (red color halos).

orthology relations imported from the eggNOG database (34). Orthologs in eggNOG are provided in a hierarchical and nested fashion, allowing the transfer of interactions by traversing up and down along the hierarchy of clades in the tree of life (26). For this purpose, the nested orthology assignments should ideally be fully self-consistent: proteins assigned to an orthologous group for a given phylogenetic clade should be grouped together in all higher-level clades too. In past versions of the orthologous groups, this has not always been the case for technical reasons (orthology assignments are computed independently for each clade). However, for STRING v10, a post-processing pipeline has been devised that makes the orthology setup fully self-consistent. It implements consistency by iteratively splitting and merging orthologous groups at the various clades and levels, until a fully consistent state is achieved. As of now, this post-processed set of orthologs forms the basis for all interaction-transfers in STRING v10. In future releases, the same hierarchical and consistent set of protein families and orthologs will be used also for more intuitive navigation and search features on the user interface.

Co-expression analysis

It has long been established that co-expression is a proxy for co-regulation (35,36) and a strong indicator of functional associations. The co-expression scores in STRING v10 are computed using a revised and improved pipeline (Figure 2), making use of all microarray gene expression experiments deposited in NCBI Gene Expression Omnibus (NCBI GEO) (37). As of March 2014, GEO consisted of more than 12 000 different platforms (GPL), 45 000 experiments (GSE) and over 1 million matrices (GSM). By including the large amount of diverse arrays in the analysis we can decrease the bias of individual platforms and experiments, and reduce the impact of non-informative matrices. Prior to the analysis, 22 organisms were identified as providing sufficient data (at least 50 experiments each). The first step of the pipeline maps probe identifiers from each platform file (GPL) to STRING genes, using dictionaries from the text-mining pipeline. Samples with less than 100 map-able genes and experiments with less than three samples are excluded from further analysis. The microarray expression values (extracted from the GSE files) are then normalized (z-value normalization) and values for each probe merged into single vectors (separately for single-channel and dual-channel arrays). Additionally, single-channel array values are $\log_2$-transformed and their mean is subtracted, to make them compatible with fold-change values in the two-channel case. Expression values of genes measured by more than one probe are averaged. In order to remove the redundancy and to increase information density between the arrays, the gene expression vectors are correlated with one another (using Spearman's rank correlation) and the full set of arrays is pruned using the Hobohm-2 algorithm (38) with similarity thresholds of 0.7 and 0.95, for single-channel and dual-channel arrays, respectively. The new gene expression values are then correlated gene-by-gene (Pearson correlation) and the resulting values are calibrated against common membership in KEGG pathway maps (release 2014-07-21) in order to compute STRING scores. Lastly, the scores from single- and dual-channel arrays are combined in a probabilistic manner to get the final scores. KEGG benchmark performance clearly improves relative to STRING v9.1 (Figure 2). The improvements can be attributed to the increased size of the GEO repository (experiments added since 2011) and to changes in our pipeline, namely: (i) the additional step to prune highly correlated samples using the Hobohm-2 algorithm and (ii) several minor improvements and bug fixes.

R/Bioconductor access

Apart from directly browsing and searching the website, data access in STRING is possible also via a REST-based API (application programing interface) and via wholesale data download. With version 10.0, we have introduced a further option: direct access from the *R* programming environment, following the Bioconductor standard (39). The corresponding package is named *STRINGdb* (Figure 3), and can be downloaded from the Bioconductor repository (http://www.bioconductor.org/packages/release/bioc/html/STRINGdb.html). The package interacts with the STRING server via the REST API and via additional, dedicated web services. To optimize the speed of subsequent accesses, the entire interaction network and associated data for a given organism are downloaded from the server and cached locally in the *R* environment, whenever possible. The package is built around the iGraph framework (40), which handles the complexity of the network data structures and provides fast query/analysis functions. Once a network is loaded/cached into an iGraph object, high-level functions facilitate the most common user tasks, such as mapping protein names onto their corresponding STRING identifiers, retrieving the neighbors of a protein of interest, retrieving PubMed IDs for publications that support a given interaction, finding clusters of proteins in the network and generating stable links back to the STRING website.

The *plot_network* function can be used to display a native STRING network of proteins in *R* (Figure 3). Functions are also available to augment a given network with user-provided node colorings ('payload information', see also Figure 1), such that subsets of proteins can be tagged and visually highlighted. Statistical enrichment tests can be executed on gene lists within the STRING namespace, covering Gene Ontology and pathway annotations, as well as tissue and diseases annotations. Results can be visualized as lists of enriched terms and/or heatmaps. The R-package proves particularly valuable for users arriving with a very large set of genes, for which the web-based interface of STRING has previously been a major bottleneck.

ACKNOWLEDGEMENTS

The authors wish to thank Yan P. Yuan (EMBL Heidelberg) for excellent technical support with the STRING backend servers. Prof. Dr Thomas Rattei and his SIMAP team (University of Vienna) are gratefully acknowledged for extensive technical support during access to their systematic protein–protein similarity data.

FUNDING

Swiss Institute of Bioinformatics; Novo Nordisk Foundation Center for Protein Research (Copenhagen); European Molecular Biology Laboratory (EMBL, Heidelberg). Funding for open access charges: University of Zurich.

Conflict of interest statement. None declared.

REFERENCES

1. Ashburner,M., Ball,C.A., Blake,J.A., Botstein,D., Butler,H., Cherry,J.M., Davis,A.P., Dolinski,K., Dwight,S.S., Eppig,J.T. *et al.* (2000) Gene ontology: tool for the unification of biology. The Gene Ontology Consortium. *Nat. Genet.*, **25**, 25–29.
2. Lee,D., Redfern,O. and Orengo,C. (2007) Predicting protein function from sequence and structure. *Nat. Rev. Mol. Cell Biol.*, **8**, 995–1005.
3. Ouzounis,C.A., Coulson,R.M., Enright,A.J., Kunin,V. and Pereira-Leal,J.B. (2003) Classification schemes for protein structure and function. *Nat. Rev. Genet.*, **4**, 508–519.
4. Bairoch,A. and Boeckmann,B. (1994) The SWISS-PROT protein sequence data bank: current status. *Nucleic Acids Res.*, **22**, 3578–3580.
5. Kanehisa,M., Goto,S., Sato,Y., Kawashima,M., Furumichi,M. and Tanabe,M. (2014) Data, information, knowledge and principle: back to metabolism in KEGG. *Nucleic Acids Res.*, **42**, D199–D205.
6. Croft,D., Mundo,A.F., Haw,R., Milacic,M., Weiser,J., Wu,G., Caudy,M., Garapati,P., Gillespie,M., Kamdar,M.R. *et al.* (2014) The Reactome pathway knowledgebase. *Nucleic Acids Res.*, **42**, D472–D477.
7. Sherman,B.T., Huang da,W., Tan,Q., Guo,Y., Bour,S., Liu,D., Stephens,R., Baseler,M.W., Lane,H.C. and Lempicki,R.A. (2007) DAVID Knowledgebase: a gene-centered database integrating heterogeneous gene annotation resources to facilitate high-throughput gene functional analysis. *BMC Bioinformatics*, **8**, 426–437.
8. Gibson,T.J. (2009) Cell regulation: determined to signal discrete cooperation. *Trends Biochem. Sci.*, **34**, 471–482.
9. Kerrien,S., Aranda,B., Breuza,L., Bridge,A., Broackes-Carter,F., Chen,C., Duesbury,M., Dumousseau,M., Feuermann,M., Hinz,U. *et al.* (2012) The IntAct molecular interaction database in 2012. *Nucleic Acids Res.*, **40**, D841–D846.
10. Licata,L., Briganti,L., Peluso,D., Perfetto,L., Iannuccelli,M., Galeota,E., Sacco,F., Palma,A., Nardozza,A.P., Santonico,E. *et al.* (2012) MINT, the molecular interaction database: 2012 update. *Nucleic Acids Res.*, **40**, D857–D861.
11. Chatr-Aryamontri,A., Breitkreutz,B.J., Heinicke,S., Boucher,L., Winter,A., Stark,C., Nixon,J., Ramage,L., Kolas,N., O'Donnell,L. *et al.* (2013) The BioGRID interaction database: 2013 update. *Nucleic Acids Res.*, **41**, D816–D823.
12. Salwinski,L., Miller,C.S., Smith,A.J., Pettit,F.K., Bowie,J.U. and Eisenberg,D. (2004) The Database of Interacting Proteins: 2004 update. *Nucleic Acids Res.*, **32**, D449–D451.
13. Schaefer,M.H., Fontaine,J.F., Vinayagam,A., Porras,P., Wanker,E.E. and Andrade-Navarro,M.A. (2012) HIPPIE: Integrating protein interaction networks with experiment based quality scores. *PloS One*, **7**, e31826.
14. Orchard,S., Kerrien,S., Abbani,S., Aranda,B., Bhate,J., Bidwell,S., Bridge,A., Briganti,L., Brinkman,F.S., Cesareni,G. *et al.* (2012) Protein interaction data curation: the International Molecular Exchange (IMEx) consortium. *Nat. Methods*, **9**, 345–350.
15. Orchard,S., Ammari,M., Aranda,B., Breuza,L., Briganti,L., Broackes-Carter,F., Campbell,N.H., Chavali,G., Chen,C., del-Toro,N. *et al.* (2014) The MIntAct project–IntAct as a common curation platform for 11 molecular interaction databases. *Nucleic Acids Res.*, **42**, D358–D363.
16. Luo,Q., Pagel,P., Vilne,B. and Frishman,D. (2011) DIMA 3.0: Domain Interaction Map. *Nucleic Acids Res.*, **39**, D724–D729.
17. McDowall,M.D., Scott,M.S. and Barton,G.J. (2009) PIPs: human protein-protein interaction prediction database. *Nucleic Acids Res.*, **37**, D651–D656.
18. Schmitt,T., Ogris,C. and Sonnhammer,E.L. (2014) FunCoup 3.0: database of genome-wide functional coupling networks. *Nucleic Acids Res.*, **42**, D380–D388.

19. Zhang,Q.C., Petrey,D., Garzon,J.I., Deng,L. and Honig,B. (2013) PrePPI: a structure-informed database of protein-protein interactions. *Nucleic Acids Res.*, **41**, D828–D833.
20. Baspinar,A., Cukuroglu,E., Nussinov,R., Keskin,O. and Gursoy,A. (2014) PRISM: a web server and repository for prediction of protein-protein interactions and modeling their 3D complexes. *Nucleic Acids Res.*, **42**, W285–W289.
21. Zuberi,K., Franz,M., Rodriguez,H., Montojo,J., Lopes,C.T., Bader,G.D. and Morris,Q. (2013) GeneMANIA prediction server 2013 update. *Nucleic Acids Res.*, **41**, W115–W122.
22. Niu,Y., Otasek,D. and Jurisica,I. (2010) Evaluation of linguistic features useful in extraction of interactions from PubMed; application to annotating known, high-throughput and predicted interactions in I2D. *Bioinformatics*, **26**, 111–119.
23. Valencia,A. and Pazos,F. (2002) Computational methods for the prediction of protein interactions. *Curr. Opin. Struct. Biol.*, **12**, 368–373.
24. Huynen,M.A., Snel,B., von Mering,C. and Bork,P. (2003) Function prediction and protein networks. *Curr. Opin. Struct. Biol.*, **15**, 191–198.
25. Lewis,A.C., Saeed,R. and Deane,C.M. (2010) Predicting protein-protein interactions in the context of protein evolution. *Mol. Biosyst.*, **6**, 55–64.
26. Franceschini,A., Szklarczyk,D., Frankild,S., Kuhn,M., Simonovic,M., Roth,A., Lin,J., Minguez,P., Bork,P., von Mering,C. *et al.* (2013) STRING v9.1: protein-protein interaction networks, with increased coverage and integration. *Nucleic Acids Res.*, **41**, D808–D815.
27. Szklarczyk,D., Franceschini,A., Kuhn,M., Simonovic,M., Roth,A., Minguez,P., Doerks,T., Stark,M., Muller,J., Bork,P. *et al.* (2011) The STRING database in 2011: functional interaction networks of proteins, globally integrated and scored. *Nucleic Acids Res.*, **39**, D561–D568.
28. Jensen,L.J., Kuhn,M., Stark,M., Chaffron,S., Creevey,C., Muller,J., Doerks,T., Julien,P., Roth,A., Simonovic,M. *et al.* (2009) STRING 8–a global view on proteins and their functional interactions in 630 organisms. *Nucleic Acids Res.*, **37**, D412–D416.
29. Letunic,I., Doerks,T. and Bork,P. (2012) SMART 7: recent updates to the protein domain annotation resource. *Nucleic Acids Res.*, **40**, D302–D305.
30. Gaudet,P., Argoud-Puy,G., Cusin,I., Duek,P., Evalet,O., Gateau,A., Gleizes,A., Pereira,M., Zahn-Zabal,M., Zwahlen,C. *et al.* (2013) neXtProt: organizing protein knowledge in the context of human proteome projects. *J. Proteome Res.*, **12**, 293–298.
31. Safran,M., Dalah,I., Alexander,J., Rosen,N., Iny Stein,T., Shmoish,M., Nativ,N., Bahir,I., Doniger,T., Krug,H. *et al.* (2010) GeneCards Version 3: the human gene integrator. *Database*, **2010**, 1–16.
32. UniProt Consortium,X (2014) Activities at the Universal Protein Resource (UniProt). *Nucleic acids research*, **42**, D191–D198.
33. Kuhn,M., Szklarczyk,D., Pletscher-Frankild,S., Blicher,T.H., von Mering,C., Jensen,L.J. and Bork,P. (2014) STITCH 4: integration of protein-chemical interactions with user data. *Nucleic Acids Res.*, **42**, D401–D407.
34. Powell,S., Forslund,K., Szklarczyk,D., Trachana,K., Roth,A., Huerta-Cepas,J., Gabaldon,T., Rattei,T., Creevey,C., Kuhn,M. *et al.* (2014) eggNOG v4.0: nested orthology inference across 3686 organisms. *Nucleic Acids Res.*, **42**, D231–D239.
35. Marcotte,E.M., Pellegrini,M., Thompson,M.J., Yeates,T.O. and Eisenberg,D. (1999) A combined algorithm for genome-wide prediction of protein function. *Nature*, **402**, 83–86.
36. Eisen,M.B., Spellman,P.T., Brown,P.O. and Botstein,D. (1998) Cluster analysis and display of genome-wide expression patterns. *Proc. Natl. Acad. Sci. U.S.A.*, **95**, 14863–14868.
37. Barrett,T., Wilhite,S.E., Ledoux,P., Evangelista,C., Kim,I.F., Tomashevsky,M., Marshall,K.A., Phillippy,K.H., Sherman,P.M., Holko,M. *et al.* (2013) NCBI GEO: archive for functional genomics data sets–update. *Nucleic Acids Res.*, **41**, D991–D995.
38. Hobohm,U., Scharf,M., Schneider,R. and Sander,C. (1992) Selection of representative protein data sets. *Protein Sci.*, **1**, 409–417.
39. Gentleman,R.C., Carey,V.J., Bates,D.M., Bolstad,B., Dettling,M., Dudoit,S., Ellis,B., Gautier,L., Ge,Y., Gentry,J. *et al.* (2004) Bioconductor: open software development for computational biology and bioinformatics. *Genome Biol.*, **5**, R80.
40. Csárdi,G. and Nepusz,T. (2006) The igraph software package for complex network research. *Inter. J. Comp. Syst.*, **1695**, 1–9.

PDF Prep Tool Suite

Version 4.11

User Manual

Contact: pdfsupport@pdf-tools.com

Owner: **PDF Tools AG**

Kasernenstrasse 1
8184 Bachenbülach
Switzerland

http://www.pdf-tools.com

Table of Contents

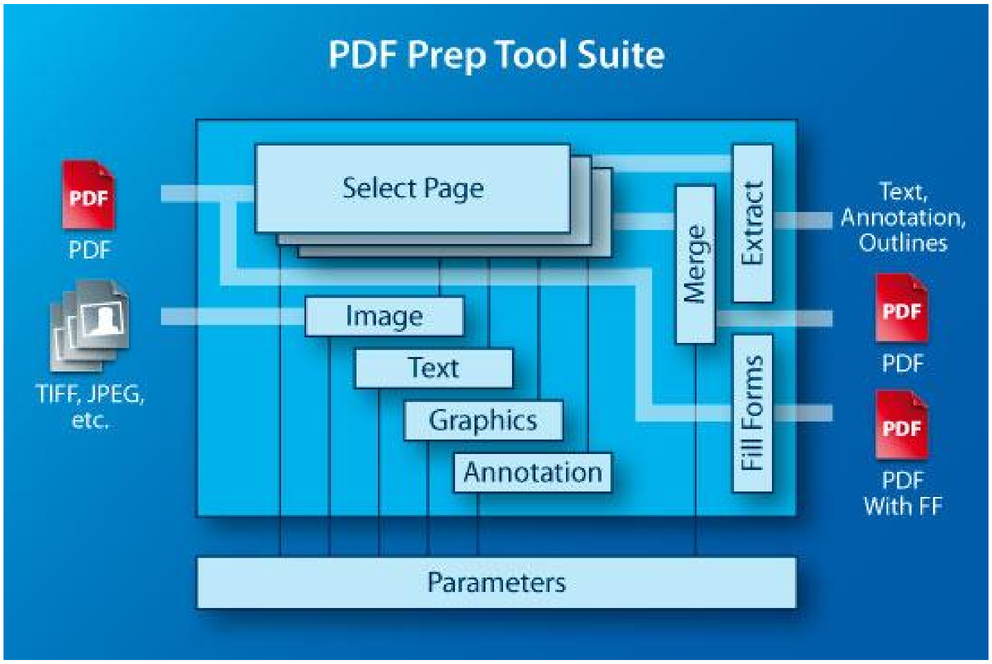
1 Introduction

The PDF Prep Tool Suite is a programming library for creating, splitting and merging PDF documents. It can be used to add content such as text, images and vector graphics. Interactive elements such as links, form fields and bookmarks can be added and processed. The component is used for the following tasks:

- Assemble PDF documents
- Personalize documents
- Fill in form fields

PDF documents can be created from scratch – for instance on the basis of a template to which data is added from a source such as a database.

Properties such as position, font, size and color are freely selectable. Once created, PDF documents can be encrypted and optimized for fast web-based viewing.



1.1 Functions

- Merge any number of pages from one or multiple PDF documents
- Apply content to the background or foreground of an existing or new page
- Text (page number, address, customer number, etc.)
- Image (company logo, scanned signature)
- Vector graphic (line, square, curve)
- Extract text including font and positioning information
- Add, fill in, delete and read out form fields

- Add internal and external links and comments
- Copy content from multiple pages to one page including positioning, scaling and rotation
- Set and get outlines (bookmarks) in PDF documents
- Define and extract document properties such as title, author, date of creation, etc.
- Read encrypted PDF documents
- Encrypt PDF documents with a password and set permission flags
- Optimize PDF files for fast web view (linearization)
- Set color as RGB or CMYK
- Set page size (media box) and visible area (crop box)
- Remove viewer access rights

1.2 SDK

The PDF Prep Tool Suite constitutes a specialized module based on the PDF Library SDK. It facilitates the generation of PDF documents based on existing PDF files or parts thereof, controlled by a simple API. It is also possible to create pages via API calls, and to add header or footer text onto pages from input files.

To facilitate the use with Microsoft Visual Basic, a COM interface is available on Windows platforms. Java applications can make use of the component via a Java interface based on JNI via a Java API package.

This document is not an introduction to PDF. You will need to refer to ISO 32000 or an Adobe PDF specification as a complementary source of information.

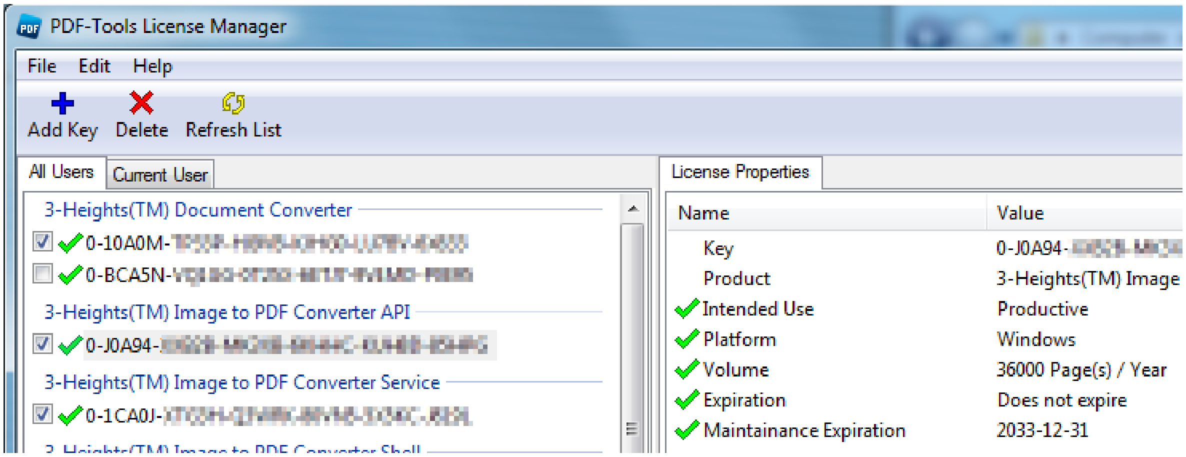
2 License Management

There are three possibilities to pass the license key to the application:

1. The license key is installed using the GUI tool (Graphical user interface). This is the easiest way if the licenses are managed manually. It is only available on Windows.
2. The license key is installed using the shell tool. This is the preferred solution for all non-Windows systems and for automated license management.
3. The license key is passed to the application at runtime via the "LicenseKey" property. This is the preferred solution for OEM scenarios.

2.1 Graphical License Manager Tool

The GUI tool *LicenseManager.exe* is located in the *bin* directory of the product kit.



3-Heights(TM) Document Converter

- ☑ ✓ 0-10A0M-[redacted]
- □ ✓ 0-BCA5N-[redacted]

3-Heights(TM) Image to PDF Converter API

- ☑ ✓ 0-J0A94-[redacted]

3-Heights(TM) Image to PDF Converter Service

- ☑ ✓ 0-1CA0J-[redacted]

3-Heights(TM) Image to PDF Converter Shell

Name	Value
Key	0-J0A94-[redacted]
Product	3-Heights(TM) Image
✓ Intended Use	Productive
✓ Platform	Windows
✓ Volume	36000 Page(s) / Year
✓ Expiration	Does not expire
✓ Maintainance Expiration	2033-12-31

List all installed license keys

The license manager always shows a list of all installed license keys in the left pane of the window. This includes licenses of other PDF Tools products.

The user can choose between:

- Licenses available for all users. Administrator rights are needed for modifications.
- Licenses available for the current user only.

Add and delete license keys

License keys can be added or deleted with the "Add Key" and "Delete" buttons in the toolbar.

- The "Add key" button installs the license key into the currently selected list.
- The "Delete" button deletes the currently selected license keys.

Display the properties of a license

If a license is selected in the license list, its properties are displayed in the right pane of the window.

Select between different license keys for a single product

More than one license key can be installed for a specific product. The checkbox on the left side in the license list marks the currently active license key.

2.2 Command Line License Manager Tool

The command line license manager tool *licmgr* is available in the *bin* directory for all platforms except Windows.

A complete description of all commands and options can be obtained by running the program without parameters:

```
licmgr
```

List all installed license keys

```
licmgr list
```

The currently active license for a specific product ist marked with a star '*' on the left side.

Add and delete license keys

Install new license key

```
licmgr store X-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX
```

Delete old license key

```
licmgr delete X-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX
```

Both commands have the optional argument `-s` that defines the scope of the action:

- `g`: For all users
- `u`: Current user

Select between different license keys for a single product

```
licmgr select X-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX
```

2.3 License Key Storage

Depending on the platform the license management system uses different stores for the license keys.

Windows

The license keys are stored in the registry:

- HKLM\Software\PDF Tools AG (for all users)
- HKCU\Software\PDF Tools AG (for the current user)

Mac OS X

The license keys are stored in the file system:

- /Library/Application Support/PDF Tools AG (for all users)
- ~/Library/Application Support/PDF Tools AG (for the current user)

Unix / Linux

The license keys are stored in the file system:

- /etc/opt/pdf-tools (for all users)
- ~/.pdf-tools (for the current user)

Note: The user, group and permissions of those directories are set explicitly by the license manager tool.

It may be necessary to change permissions to make the licenses readable for all users. Example:

```
chmod -R go+rx /etc/opt/pdf-tools
```

2.4 Setting the License Key via the API

When deploying applications that use the PrepTool API, you may prefer to pass the license key at runtime, rather than register the key on all potential target systems. The typical call sequence in the application will be as follows:

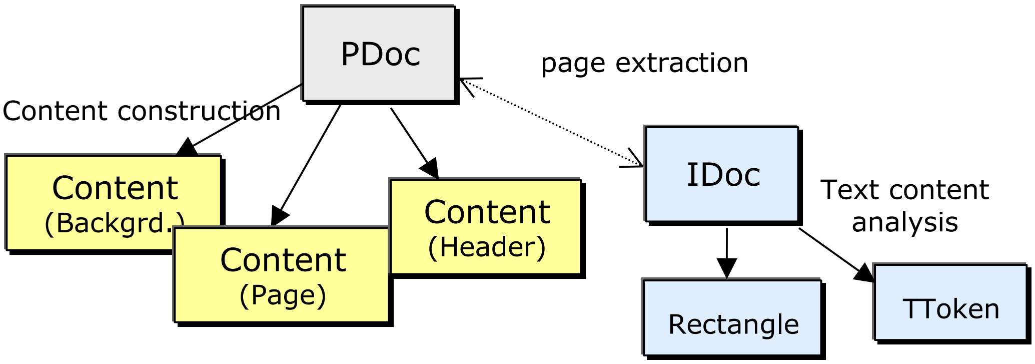
```
/* Initialize; this will load a license key stored on the computer */
PTInitialize();
/* Set the license key */
PTSetLicenseKey("0-12345-ABCDE-67890-FGHIJK-12345-ABCDE");
/* License check */
if (!PTGetLicenseIsValid())
{
    printf("no valid license found.\n");
    PTUninitialize();
    return 10;
}
```

Note: the COM and Java bindings automatically perform the “PTInitialize()” function. If a license key is installed on the computer, the application can detect that by directly calling “PTLib.getLicenseIsValid()” (Java) or querying the “LicenseIsValid” property of an IDoc, PDoc or PDFLinearizer Object (COM).

3 Object Model

The core entities in the PDF Prep Tool Suite are PDF files - either existing ones serving for input, or new ones being created. In the COM interface, these are the IDoc and PDoc types.

Another important entity for creating PDF files are content objects. A content object represents a layer of text and graphics objects that is used to construct a PDF page. The Prep Tool Suite uses content objects to construct PDF pages via API calls, and also to put a layer containing text, images etc. on top selected pages that are copied from existing PDF files.



There are also some auxiliary object types which are used to return structured information about PDF files, like text tokens on a page, or rectangle coordinates of media boxes or form field locations.

4 Processing Model

The processing model of the PDF Prep Tool Suite with regard to PDF creation is batch oriented. Pages are written sequentially without much buffering in memory. Contrary to interactive models where a document is opened, then modified randomly, and finally saved, the Prep Tool Suite works differently: Any modifications to be made to existing pages of PDF files are prepared either by reading the corresponding PDF objects into a cache where they are modified, or by posting modifications that are to be made when transferring PDF objects to output. After that, a copy operation saves the range of pages to output.

The reason for this model is resource conservation and speed.

5 Language Bindings

There are three different bindings to the Prep Tool Suite: A conventional native library interface (DLL on Win32), a COM interface for Win32, and Java wrapper classes based on JNI. The C++ classes of the implementation are not exposed directly in any API.

The native interface is most suitable for C/C++ applications on any platform (Windows or Unix), but can also be used from Visual Basic on Win32.

The COM interface is most suitable for Visual Basic applications, but can also be used by any other development environment that can make use of COM objects, such as Delphi. Unlike the other APIs, COM allows for optional and default parameters. You will get the appropriate hints in the Visual Basic development environment.

Object type/type of interface	***Native***	***COM***	***Java***
Reference to PDF output file	`Handle`	`PDoc`	`PTDoc`
Reference to PDF input file	`InputHandle`	`IDoc`	`PTInput`
Reference to current page	`ContentHandle`	`content`	`PTContent`
Reference to current header or background layer	`ContentHandle`	`content`	`PTContent`
Reference to current outline	`BookmarkHandle`	`Bookmark`	`PTBookmark`

Reference to current form field	-	`FormField`	`PTFormBox,`
			`PTFormData`

Reference to current linearized file	-	`PDFLinearizer`	`PTLinearizer`
Reference to current text token	`PTTokenInfo`	`TToken`	`PTTextToken`

The native binding uses the type VBSTR to return character strings from the Prep Tool Suite to the application. VBSTR is compatible with Visual Basic, i. e. Visual Basic will correctly free these strings again.

C applications need to explicitly free strings obtained from the Prep Tool Suite by calling PTFreeVBSTR. This function will call the Win32 function *SysFreeString()*.

On UNIX systems, PTFreeVBSTR simply calls the standard *free()* function from *stdlib.h*.

6 Getting Started

This chapter gives a brief overview of how the PDF Prep Tool Suite can be used. Important to know is:

- The PDF Prep Tool Suite never modifies an input file. Modifications are always applied and visible in the created output.
- Only one output file can be opened at a time. Several input files can be opened at once, but only one can be attached to an output file at a time.
- "Attached" means that when calling to *InputCopyPages*, or *InputCopyAll*, the pages of the attached input file are copied to the output file.

There are basically two possibilities to create a document:

6.1 Create a Document from Scratch

The PDF Prep Tool is not intended to be used as a PDF Creator, even though it provides the functionality to add text, raster graphics and vector graphics such as lines or rectangles.

When creating a document from scratch, content is to be written on the Page layer. The *Header* and *Background* layers cannot be used at this time.

In Visual Basic 6, creating a document from scratch could look like this:

```
Dim outdoc As New PREPTOOLLib.PDoc
outdoc.New "C:\temp\hello.pdf"
outdoc.Page.SetFont "Helvetica", 50
outdoc.Page.PrintText "Hello World.", 100, 300
outdoc.Close
```

6.2 Add Content to an Existing Input File

An input file can be opened by either creating an IDoc object, open a document and attach the IDoc to the PDoc object, or by a call to *InputOpen* of the PDoc object. Adding new content can be achieved by writing on the Header or Background layer. After a call to *InputCopyAll* or *InputCopyPages*, the content of the pages of the input document is merged with the Header and Background layers.

The Page layer cannot be used at this time to add content to the page. Writing on the Page layer creates a new page.

Here is a Visual Basic 6 sample:

```
Dim outdoc As New PREPTOOLLib.PDoc
outdoc.New "C:\temp\output.pdf"
outdoc.InputOpen "C:\temp\hello.pdf"
outdoc.Header.SetFont "Helvetica", 50
outdoc.Header.PrintText "Hello again.", 100, 400
outdoc.InputCopyPages 1, 1
outdoc.Close
```

7 Output PDF Creation

Creation of an PDF file for output is performed as shown in the table below. Note that the COM interface requires two steps, because a COM object cannot be created with parameters.

Native	`Handle PDocNew(const char* Filename, short PgWidth, short PgHeight, PTError* err)`

COM	`Dim Obj As New PDoc`
	`Dim Obj As Object: Set Obj = CreateObject("PrepTool.PDoc")`
	`New(Filename As String, Width As Integer, Height As Integer) As Boolean`

Java	`new PTDoc(String Filename)`
	`new PTDoc()`

The procedure *PDocNew* creates a new PDF file that is initially empty. The pages created and written to via the API will all have the specified page height and width.

Note that the PDF coordinate system has its origin at the left bottom of the page. The European format A4 has a width of 595 (points) and a height of 842.
A return value of 0 for the Handle means that the output file could not be created. In the native interface, you must use the *PTError** parameter to obtain the error code which is necessary to determine the reason why creation failed.

In the Java binding, the page format must be set by a separate method (*setPageSize*).

To create a PDF file in memory without writing it to disk, you can omit the file (i. e. specify NULL / 0 for this parameter). The Java API exhibits a constructor with no parameter for this.

The *CloseB* function will retrieve the byte array corresponding to the contents of the PDF file.

7.1 Set the PDF Version

Native	`PTError PDocSetPDFVersion(Handle h, const char* version)`
COM	`SetPDFVersion(Version As String)`
Java	`void setPDFVersion(String Version)`

This function sets the PDF version stored at the beginning of each PDF file. The default value is "1.4".

Note that the PDF version must be set before writing anything else to the output file.

7.2 Encryption

To provide a certain protection of PDF files, Adobe has specified "Standard Security" in the PDF specifications. This is based on encryption algorithms and is available in the

Prep Tool Suite.

Native	**PTError PDocSecurity(Handle h, const char* ownerPw, const char* userPw, const char* flags)**
COM	**SetSecurity(OwnerPassword As String, UserPassword As String, Flags As String)**
Java	**void SetSecurity(String Ownerpassword, String Userpassword, String Flags)**

This method will set the passwords and protection flags of the file to be created. It must be called immediately after the *New* method (before any objects are written to output).

The "Flags" parameter sets the protection attributes. It can contain a combination (or none) of the following characters:

"p": do not print the document from Acrobat

"c": changing the document is denied in Acrobat

"s": selection and copying of text and graphics is denied

"a": adding or changing annotations or form fields is denied

The following flags are defined for 128 bit encryption (PDF 1.4, Acrobat 5.0):

"i": disable editing of form fields

"e": disable extraction of text and graphics

"d": disable document assembly

"q": disable high quality printing

The flag "5" can be used in combination with one of the "old" flags to force 128 bit encryption without setting any of the i, e, d, or q flags. Note that using any of these Acrobat 5 related flags will produce a file that cannot be opened with older versions of Acrobat.

Omitting these flags will result in a PDF file that is fully usable when opened using the user password.

7.3 Disable Stream Compression

Native	**void PDocCompress(Handle h, short Yes)**
COM	not available (default: compression enabled)
Java	enabled if environment variable PDPREP_OPT_NC is defined

The function *PDocCompress* can be used to disable the compression of content streams generated by Prep Tool. By default, compression is enabled.

7.4 Font Renaming

Acrobat viewers before 4.05 had the problem of incorrectly rendering text with fonts that were multiply defined in a file. PDF Prep Tool automatically renames such fonts to work around this viewer problem. This renaming can create new problems when printing the resulting file, and when the font is not embedded in the file. In these cases, you should use *SetPreserveFontNames* method to disable the renaming feature.

Native	`PTError PDocSetPreserveFontNames(Handle h, short on)`
COM	`SetPreserveFontNames(on As Boolean)`
Java	`void setPreserveFontNames(boolean on)`

7.5 Error Handling

Error handling is implemented via a "get last error" method for PDF input and output objects.

Native	`int PDocLastError(Handle h)`
COM	`ErrCode() As ErrorType`
Java	`int getLastError()`

The COM interface defines its own error codes which are defined in the COM interface.

The native interface returns the normal "errno" codes of the operating system where appropriate, and a set of special Prep Tool errors that are defined in the include file.

NOTE: the "success" error code has been changed to conform with "errno", i.e. a value of 0 (zero) corresponds to successful operation, rather than the value 1 which previously was returned in most cases. Please refer to the file *pdptdef.h*.

The Java interface uses Java exceptions combined with the native error codes. Please refer to the Java class definitions.

7.6 Open a PDF File for Input

You can open a PDF file to retrieve information from it via the API, or to use it as a resource to copy pages to an output file, or both.

This is how to open the input file by referring to an output file object:

Native	`PTError PDocInputOpen(Handle h, const char* inputFile)`
COM	`InputOpen(Filename As String) As Boolean`

Java	`Boolean inputOpen(String Filename)`
	`Boolean inputOpen(String URL)`

A call to *PDocInputOpen* makes resources of an existing PDF file available - either to copy a non built-in font into the output file, or to copy pages to the output file.

Only one input file can be active at a time. A subsequent call to *PDocInputOpen* will automatically close the previous input file.

A return value of !=PTSuccess (Java/COM: false) means that the input file could not be opened.

In Java it is possible to provide an URL instead of a file name.

If you want to open a PDF file for input without need to create some other PDF file, you can do this as follows:

Native	`InputHandle IDocOpen(const char* Filename, PTError* errCode)`
	`InputHandle IDocMemOpen(const char* pdfBytes, int len,  PTError* errCode)`

COM	`Dim Obj1 As New IDoc`
	`Dim Obj2 As Object`
	`Set Obj2 = CreateObject("PrepTool.IDoc")`
	`Obj1.Open(Filename As String) As Boolean`
	`Obj1.OpenMem(Bytes [As Byte()]) As Boolean`

Java	`new PTInput(String Filename)`
	`new PTInput(String URL)`
	`new PTInput(byte[] pdfBytes)`

It is possible to open a PDF "file" stored in memory rather than referring to the file system using the *MemOpen* function. In Java, the PTInput constructor taking a byte array can be used for this.

When using *IDocMemOpen*, the "pdfBytes" are copied during this call and can be disposed of as needed (all language bindings).

In the COM interface the following construct can be used to ensure that the PDoc and its corresponding IDoc are running in the same appartment:

COM	`Dim Obj1 as New PDoc`
	`Dim Obj2 as IDoc`
	`Set Obj2 = Obj1.CreateIDoc`

To open a password protected (encrypted) PDF file, you need the following API calls:

Native	`InputHandle IDocOpenPw(const char* inputFile, const char* password, PTError* errCode)`
	`InputHandle IdocMemOpenPw(const char* pdfBytes, int len, const char* password, PTError* errCode)`

COM	`Open(Filename As String, Password As String) As Boolean`
	`OpenMem(Bytes, Password As String) As Boolean`

Java	`new PTInput(String Filename, String Password)`
	`new PTInput(String URL, String Password)`
	`new PTInput(byte[] pdfBytes, String Password)`

The COM API uses the same methods to open encrypted and non-encrypted files. The password parameter is optional.

7.7 Attach an Input File

When you have previously opened an existing PDF file using *IDocOpen*, you may later want to use it as a source for pages or other resources to create an output file.

Native	`PTError PDocAttach(Handle h, InputHandle hIDoc)`
COM	`Attach(Input As IDoc) As Boolean`
Java	`void attachInput(PTInput input)`

Note: Once you have attached an input file to an output PDF, you must not attach it to another output PDF file; also, you must not close it, because it will be closed automatically when the output PDF is closed.

Attaching a new input file to an output PDF will also close the previous input file (except when using the COM API).

7.8 Accessing the Current Input File

The make use of the full set of PDF file analysis features, you may want to know the input file object reference of an output file object.

Native	`InputHandle PdocGetInputHandle(Handle h)`
COM	`Input() As IDoc`
Java	`PTInput getInput()`

7.9 Set the Page Size and Orientation

Native	`PTError PDocPageSize (Handle h, short Width, short Height)`
COM	`PageSize(Width As Integer, Height As Integer)`
Java	`void setPageSize(short Width, short Height)`

Use this function to set the dimension of pages to be created. The width and height are specified in points corresponding to the standard PDF coordinate system. The MediaBox of the page will be set as [0 0 <width> <height>]. The default values are 595 by 842, i. e. A4 portrait. There are minimum and maximum values that vary between different versions of the Acrobat viewers.

If you want to create landscape pages, you can either set the width and height accordingly, or turn the coordinate system by printing from bottom to top while specifying a value of 90 for the Rotate attribute of the page. Please read the explanations about the PDF and text coordinate system in the specifications.

Native	`PTError PDocPageRotate(Handle h, short orientation)`
COM	`SetPageRotate(Orientation As Integer)`
Java	`void setPageRotate(int Orientation)`

The orientation for viewing the content of a page can be set using this function. Legal values that can be specified are 0 (default) and multiples of 90 (e. g. 270, -90, 180, etc.).

These settings do not affect pages copied from existing PDF files, see *SetInputRotate* (below) for this.

To change the format or orientation of such pages, you can create empty pages of the desired format and add the content of the existing file using the "Logo" functions. This allows you to use arbitrary coordinate transformations for positioning and scaling the page. This method will not work to copy annotations (such as form fields, links, etc.).

Native	`PTError PDocSetInputRotate(Handle h, short orientation)`
	`PTError PDocClearInputRotate(Handle h)`

COM	`SetInputRotate(Orientation As Integer)`
	`ClearInputRotate`

Java	`void setInputRotate (short Orientation)`
	`void clearInputRotate()`

SetPageRotate has the effect to replace the page rotation stored in input PDFs with the value specified when copying pages into the output PDF. To restore the behavior of keeping the value as in the input file, use *ClearInputRotate*.

7.10Set the Crop Box

The Crop Box is the displayed part of the PDF. This function allows the setting the Crop Box for pages that are created or copied. The Crop Box must never be larger than the Media Box.

Native	`PTError PDocSetCropBox(Handle h, float Left, float Bottom, float Right, float Top)`
COM	`SetCropBox(Left As Single, Bottom As Single, Right As Single, Top As Single)`
Java	`void setCropBox(float Left, float Bottom, float Right, float Top)`

The crop box can be set for a newly created page. It is also applied to the pages that are copied using *InputCopyPages* or *InputCopyAll*.

7.11Adding a New Page

A new page is automatically added to the output file when you request its handle for the first time or after a call to *PDocNewPage*.

Native	`ContentHandle PDocGetContentHandle(Handle h)`
COM	`Page() As content`
Java	`PTContent getPageContent()`

The content handle and the Java PTContent object are only valid as long as the page is in construction. Once it is written to output, it is invalid and may no longer be used.

The COM object can be reused to access the next page after a call to the *NewPage* method only. In all other cases, a new Content object reference must be obtained from the PDoc object.

7.12Accessing the Current Header or Background Content Layer

In order to construct the header content layer, you need the corresponding object reference from the output file object.

Native	`ContentHandle PDocGetHeaderHandle(Handle h)`
	`ContentHandle PDocGetBackgroundHandle(Handle h)`

COM	`Header() As content`
	`Background() As content`

Java	`PTContent getHeaderContent()`
	`PTContent getBackgroundContent()`

A header (or background) content reference is valid as long as the header is not cleared. After a call to *HeaderClear* or *BackgroundClear*, it becomes invalid and may no longer be used.

The header content layer will be placed on top of pages copied into an output PDF (using *InputCopyPages*), while the background layer will be placed behind. Note that the background content may be hidden by non-transparent pages of an input file.

8 Retrieving File Information

In the native interface, you refer to a handle of type "InputHandle".

In the COM interface, you refer to an object of type "IDoc".

In the Java binding, you refer to an object of class "PTInput".

You can obtain this kind of object reference in one of the ways described above.

8.1 Obtain the PDF Version

Native	`VBSTR IDocPDFVersion(InputHandle h)`
COM	`GetVersion() As String`
Java	`String getPDFVersion()`

Returns the PDF version of the file, as stored in the file header.

8.2 Obtain the File Name

Native	`VBSTR IDocGetFileName(InputHandle h)`
COM	`GetFileName() As String`
Java	`String getFileName()`

Retrieves the name of the PDF file. If the file was opened from memory, a unique string is returned starting with "internal: ". If the file is not open, an empty string is returned.

8.3 Obtain the Keys List

Native	`PTError IDocGetInfoKeys(InputHandle h, VBSTR* keys)`
COM	`GetInfoKeys() As String`
Java	`String getInfoKeys()`

This function returns a carriage-return separated list of the keys that are present in the /Info attribute of the PDF file. The keys are returned with the leading slash character – e. g. /Author, /Title, etc.

8.4 Obtain Document Attributes

Native	`PTError IDocGetInfoAttr(InputHandle h, const char* key, VBSTR* value)`
	`PTError IDocGetInfoAttrU(InputHandle h, const char* key, PDBSTR value)`

COM	`GetInfoAttr(ByVal Key As String) As String`
Java	`String getInfoAttr(String Key)`

This function returns the value of a document attribute stored in the /Info attribute of the PDF file.

8.5 Get Meta Data

Native	`VBSTR IDocGetMetaData(InputHandle h)`
COM	`GetMetaData() As String`
Java	`String getMetaData()`

GetMetaData returns the XML meta data stored in the PDF document.

8.6 Get the Name and Current Data of a Form Field

Native	`PTError IDocGetFormData(InputHandle h, short FieldNum, VBSTR* Name, VBSTR* Data, VBSTR* Description, int* FormFlags, int* AnnotFlags, VBSTR* FieldType)`
COM	`GetFormData(ByVal FieldNum As Integer, Name As String, Data As String, Descr As String, Multiline As Boolean) As Boolean`
Java	`PTFormData getFormData(int FieldNum)`

The parameter FieldNum (default=1) is an iterator by which you can obtain the names and current data of all text form fields. FieldNum runs from 1 to the number of form fields. A result of !=PTSuccess/False/null will be returned, if you go beyond the last form field.

PTFormData.Name	= Name as String
PTFormData.Data	= Data as String
PTFormData.Description	= Description as String

The native and the Java interface also supply type information (FieldType). This type information is composed of the field type of the field itself, followed by the export values (separated by new-line characters).

Note that there can be more than one instance of a form field. If this is the case, each instance can have different flags, and you need to use *GetFormBox* to check the individual settings.

8.7 Get the Position of a Form Field

Native	`PTError API IDocGetFormBox(InputHandle h, const char* fieldName, float box[], short inst, int* page, short* fontID DEFAULT_NULL, float* fs, short* al, int* formFlags, int* annotFlags)`
COM	`GetFormBox(ByVal FieldName As String, ByVal Instance As Short, X`

	As Single, Y As Single, W As Single, H As Single, Page As Integer, FontID As PtFormFontType, Fontsize As Single, Alignment As Short, Formflag As PTFormFlags, Annotflags As PTAnnotFlags) As Boolean
Java	**PTFormBox getFormBox(String Fieldname, int Instance)** **PTFormBox getFormBox(String Fieldname)**

There can be more than one field occurrence with a certain name. All these occurrences share the same data and also other attributes like description or multi-line. Individual form fields have their own location, display text in a different font and with different alignment. *GetFormBox* returns the latter information that belongs to individual form fields. The "instance" parameter serves to distinguish different occurrences. Instance numbers start at 1. The function will return a *PTSuccess* (True/non-null) result if the instance is found.

The parameters are:

- Fieldname: the name of the form field (IN)
- Instance: a numerator to distinguish between form fields that have the same name (IN)
- box,

  box[0-3] =X/Y/W/H: the coordinates of the rectangle occupied by the form field

  ```
  if (box.length > 4)
          Page = (int) box [4];
          if (box.length >= 10) {
                  FontID = (int) box [5];
                  FontSize = box [6];
                  Alignment = (short) box [7];
                  FormFlags = (short) box [8];
                  AnnotFlags = (short) box [9];
  ```
- Page: the number of the page on which the field is located (1..number of pages)
- FontID: the identification of the font used to display text (e. g. Helvetica = 0, see declaration of font constants)
- FontSize: the size of the text being displayed
- Alignment: the alignment for displaying the form text (0 = left, 1 = centered, 2 = right)
- Formflags: the flags set for the form ("/Ff" entry in the form field’s dictionary, see *AddTextField*)
- Annotflags: the general annotation flags ("/F") set for the field

The form specific flags being returned are described in the PDF specification:

- 1: read-only (flag & 1 != 0)
- 2: required (flag & 2 != 0)

- 3: no export (flag & 4 != 0)
- 13: multi-line (flag & 4096 != 0), etc.

The general annotation flags are

- 1: invisible
- 2: hidden
- 3: printable, etc.

8.8 Get Information about Pages

Native	`int IDocNumPages(InputHandle h)`
	`PTError IDocAcquirePage(InputHandle h, int Page)`
	`PTError IDocPageBox(InputHandle h, float* X, float* Y, float* Width, float* Height)`
	`PTError IDocMediaBox(InputHandle h, float* X, float* Y, float* Width, float* Height)`
	`short IDocPageRotate(InputHandle h)`

COM	`NumPages() As Long`
	`GoPage(PageNum As Long) As Boolean`
	`GetVisibleBox(Left, Bottom, Right, Top)`
	`GetMediaBox(Left, Bottom, Right, Top)`
	`GetRotate() As Integer`

Java	`int getNumPages()`
	`boolean acquirePage(int PageNumber)`
	`PTRectangle getPageBox()`
	`PTRectangle getMediaBox()`
	`short getPageRotate()`

This set of functions can be used to retrieve information about individual pages in a PDF file.

The "visible box" corresponds to the crop box; if none is present, the media box is returned.

The "Rotate" attribute of a page tells a viewer application that the page shall be rotated for presentation.

8.9 Retrieve Text from a PDF File

Native	`PTError IDocReadText(InputHandle h, VBSTR* text, PTTokenInfo* m, VBSTR* font)`
	`PTError IDocReadTextU(InputHandle h, PDBSTR* text, PTTokenInfo*`

	`m, VBSTR* font)`
COM	`GetToken() As TToken`
Java	`PTTextToken readTextToken()`

This function retrieves text fragments from a PDF file's pages. The metrics structure contains the coordinates, font size, width and orientation of the retrieved character string. The page number is also contained, because *ReadText* passes automatically to the next page when no more text is found on a page.

The *PTTokenInfo* structure contains a float array indicating to position of each individual character of the retrieved string (CharRightPos). This float array is dynamically allocated and must be initialized before calling *IDocReadText* and again afterwards when not used any more. This is done with the functions *PTInitToken()* and *PTFreeToken()*. For C++ programmers, the class *CPTTokenInfo* is available which takes care of initializing the structure and freeing the allocated memory again.

8.10Retrieve Bookmarks from a PDF File

Native	`BookmarkHandle IDocGetBookmarkRoot(InputHandle h)`
	`PTError PBMGoNext(BookmarkHandle h)`
	`PTError PBMGoUp(BookmarkHandle h)`
	`PTError PBMGoDown(BookmarkHandle h)`
	`PTError PBMReset(BookmarkHandle h)`
	`PTError PBMGetTitle(BookmarkHandle h, VBSTR* title)`
	`PTError PBMGetTitleU(BookmarkHandle h, PDBSTR* title)`
	`PTError PBMGetLevel(BookmarkHandle h, int* level)`
	`PTError PBMGetNumChildren(BookmarkHandle h, int* numChildren)`
	`PTError PBMKidsVisible(BookmarkHandle h, bool* kidsVisible)`
	`PTError PBMGetInfo(BookmarkHandle h, VBSTR* info)`
	`PTError PBMRelease(BookmarkHandle h)`
	`BookmarkHandle API PBMClone(BookmarkHandle h)`

COM	`GetBookmarkRoot() As Bookmark`
Java	`PTBookmark getBookmarkRoot()`

The bookmark root node can be retrieved trough the function *GetBookmarkRoot*. Java and COM uses the classes PTBookmark and Bookmark to encapsulate the appropriate native functions (*GetTitle*, *GoNext*, ..)

Navigate trough the tree by using the functions *GoNext* to go to the next bookmark, *GoDown* to go one level deeper, *GoUp* to go one level up and reset to move to the root bookmark. *GoDown*, *GoUp* and *GoNext* return false if there isn't a next bookmark or the node has no children (*GoDown*) or is no parent (*GoUp*).

To retrieve information about the current bookmark use the get functions. *GetTitle* returns the bookmark title. The native method *GetTitleU* returns the title in Unicode.

Java and COM methods always return Unicode strings. *GetLevel* returns the current level of the bookmark. The root level is –1. *GetNumChildren* returns the number of children for the current bookmark. Use the Clone function to get a copy of the current bookmark. For Java and the native API you have to release a bookmark to free the memory. Use the release function to do this.

To release the title string in the native API, use the functions PTFreeVBSTR and PTFreePDBSTR.

The *GetInfo* function returns additional information about a bookmark. This information is returned in a character string. The string content depends on the type of action or destination attached to the bookmark. The following types are supported:

GoTo:	Go to a destination in the current document (Starting at 0).
GoToR:	Go to a destination in another document.
Launch:	Launch an application.
URI:	Open an Internet link.

The action types have to be interpreted as follows:

Action type	**String**
GoTo	GoTo *Destination*
GoToR	GoToR *file Destination*
Launch	Launch *file*
URI	URI web-link

A destination can be one of the following:

page /XYZ *left top zoom*
page /Fit
page /FitH *top*
page /FitV *left*
page /FitR *left bottom right top*
page /FitB
page /FitBH *top*
page /FitBV *left*

For more information about action types and destinations, refer to the PDF-Reference.

Use a parser to split up the string into the tokens. The separation between two arguments is the blank character.

8.11Retrieve Annotations from a PDF File

Native	`PTError IDocGetAnnotation(InputHandle h, PTAnnotType* AnnotType, float rect[], int* BorderStyle, int* pIdentification)`
	`PTError IDocGetAnnotationInfo(InputHandle h, VBSTR* AnnotInfo)`
	`PTError IDocGetAnnotationInfoU(InputHandle h, PDBSTR* AnnotInfo)`

COM	`GetAnnotation(Type As PTAnnotType, Info As String, Left As`

	`Single, Bottom As Single, Right As Single, Top As Single, BorderStyle As Long, Identification As Long) As Boolean`
Java	`PTAnnotData readAnnotation()`

These functions are used to retrieve annotations from a PDF document. Two types of annotations can be retrieved, text and link annotations. The type is retrieved through the *PTAnnotType* structure.

The function *GetAnnotation* returns one annotation per call for the current page. Call the function again to retrieve the next annotation. The function will return an error if it has no next annotation.

Use the *GetAnnotationInfo* function in the native API to retrieve the info string from the current annotation. The *GetInfoAnnotationU* function, retrieves the info string in Unicode. Use PTFreeVBSTR and PTFreePDBSTR to free these strings.

The return values are interpreted as follows:

AnnotType: the type of the annotation (eText or eLink)
rect[]: the location of the annotation on the page (left, bottom, right, top)

Info: If it's a text annotation this holds the text. If it's a Link annotation this holds an action or named destination. See 'Retrieve Bookmarks from a PDF File' for more information about the structure of the info string in this case.
Java encapsulates the return values in the class *PTAnnotData*.

8.12Retrieve the Border Style from Annotations

Native	n.a.
COM	`GetBorderStyle(ID As Long) As IBorderStyle`
Java	n.a.

If the annotation has a Border Style dictionary (entry BS), this function returns an *IBorderStyle* interface, otherwise nothing is returned. ID is the identification of the annotation which is received using the method *GetAnnotation*.

IBorderStyle has the following properties:

String BS	Describes the border style. The following substrings are possible: "S" (Solid), "D" (Dashed), "B" (Beveled), "I" (Inset), "U" (Underline).
Long ColorRGB	The color as RGB value. ColorRGB = red + 256 * green + 256 * 256 * blue. Where red, green and blue are values 0-255.
String DashArray	A dash array defining a pattern of dashes and gaps to be used in drawing a dashed border. The array is returned a string, the separator is the blank. For example, a "1 2" string specifies a border drawn with 1-point dashes alternating with 2-point gaps.
Integer DashOff	The size of the gaps. See DashArray.
Integer DashOn	The size of the dash. See DashArray.
Integer Width	The border width in points. If this value is 0, no border is drawn.

8.13Get List of Fonts

Native	`PTError IDocGetFonts(InputHandle h, int PageNumber, VBSTR* Fonts)`
COM	`GetFonts(Optional ByVal PageNumber As Long) As String`
Java	`String getFonts(int Page)`

This function returns a "\r" (Chr$(13)) separated list of the fonts contained in the PDF file. If the Page parameter is specified as 0, the whole document is searched for fonts.

8.14Get Color Information

Native	`short IDocNumColorSpaces(InputHandle h)`
	`VBSTR IDocGetSeparation(InputHandle h, short index)`

COM	`NumColors() As Long`
	`GetColor(ByVal Index As Long) As String`

Java	`int getNumColors()`
	`String getColor(int Index)`

These functions return information about ColorSpace entries in the resources dictionary of the current page of the input file (*AquirePage* must previously be called).

The index to retrieve the names of the color space separation runs from 1 to the number of colors.

8.15Save File Attachment

Native	`PTError IDocSaveAttachment(InputHandle h, const char* filename)`
COM	`SaveFileAttachment(FileName As String) As Boolean`
Java	`void saveAttachment(OutputStream os)`

This function permits retrieval of the file that is embedded in a *FileAttachment* annotation.

Note that *SaveFileAttachment* depends on the *GetAnnotation* function, and will only work when the last annotation returned by *GetAnnotation* is a file attachment.

8.16Close the File

Native	`PTError IDocClose(InputHandle h)`
COM	`Close() As Boolean`
Java	`void close()`

This function closes the input file and releases all resources associated with it. When the COM object’s reference count goes to zero, an automatic close is performed.

When using the Java API, you must be careful: call "close" only, if you obtained the

PTInput object using "new". If you obtained it via *PTDoc.getInput()*, the input file will be closed when closing the PTDoc object.

8.17Get UserUnit

Native	`float IDocUserUnit(InputHandle h)`
COM	`GetUserUnit() As Single`
Java	`n.a.`

Returns the UserUnit as float if defined in the PDF document. If no UserUnit is defined, 1 is returned.

Content Construction

The following methods refer to objects of type content, and can thus be equally applied to "print" to a page or to construct the content layer of a header.

In the native interface, you refer to a handle of type "ContentHandle".

In the COM interface, you refer to an object of type "Content".

In the Java binding, you refer to an object of class "PTContent".

8.18Set the Font for Text Output

Native	`PTError PConSetFont(ContentHandle h, const char* fontName, float fontSize)`
COM	`SetFont(ByVal FontName As String, ByVal FontSize As Single) As Boolean`

Java	`boolean setFont(String Name, float Size)`
	`boolean setFont(String Name)`
	`boolean setFont(float Size)`

In the native and COM interfaces, the parameters Name and Size are optional. Once you have set the font's name, it is possible to change its size by just passing the new size. For missing arguments, you can specify 0.

The procedure *PConSetFont* must be called prior to *PConPutText* to set the font to be used and its size. Only predefined Acrobat fonts can be specified here ("Helvetica", "Helvetica-Bold", "Helvetica-Oblique", "Times-Roman", "Times-Italic", "Times-Bold", "Courier", "Courier-Oblique", "Courier-Bold", "Symbol", "ZapfDingbats").

The fonts "Helvetica-BoldOblique", "Courier-BoldOblique" and "Times-BoldItalic" are built in fonts, but cannot be used because their definition requires additional information which is not yet supported. However, if these fonts or any other non standard font is defined in the current input file, PT will copy that font to the output file.

SetFont returns FALSE if the font cannot be set (i. e. is not a standard font and is not found in the current input file).

Note the following issues about using fonts:

When there is no current input file (see *PDocInputOpen*), you must only use standard built-in fonts like Helvetica, Times-Roman, etc. (see *PConSetFont*).

When there is a current input file, *SetFont* tries to find a font with this name in the input file and copy the font data to the output file. It is then legal to use this font.

To use a non standard font, you can thus create a template file containing the font data. As Acrobat optimizes the font data to what is actually necessary, make sure you place the full variety of characters that you later need into the file. To refer to the font, you specify its name in the "fontName" parameter. It is actually sufficient to specify only a significant portion of the name (matching is case sensitive - check the spelling of the font in the template file!).

Some fonts will have the effect that the encoding of individual characters in the PDF file is different from the corresponding ASCII code. Currently, you can only use fonts that conform to certain conventions. The standard fonts "Helvetica", "Helvetica-Bold", "Helvetica-Oblique", "Times-Roman", "Times-Italic", "Times-Bold", "Courier", "Courier-Oblique", "Courier-Bold", "Symbol", "ZapfDingbats" should always work. Other fonts will be embedded into the PDF file. Their encoding depends on the tool which produced the PDF file. PDFWriter on Windows produces a standard ASCII encoding (WinAnsiEncoding) for a font for which Distiller Assistant will create an encoding which shifts codes by 29. (You will notice this also in Acrobat, when you select text, copy it to the clipboard, and try to use it in another application).

The Prep Tool DLL uses a heuristic to determine if there is a code shift by evaluating the "FirstChar" key of the font dictionary. It uses this value to shift the code, assuming that this code corresponds to the first printable character which is a blank space (ASCII 32). When you prepare a template PDF, make sure it contains a blank space (plus all other characters you want to have available).

8.19Set Text Spacing

Native	`PTError PConSetCharSpacing(ContentHandle h, float value)`
	`PTError PConSetWordSpacing(ContentHandle h, float value)`
	`PTError PConSetTz(ContentHandle h, float value)`

COM	`SetCharSpacing(ByVal Value As Single)`
	`SetWordSpacing(ByVal Value As Single)`
	`SetTz(ByVal Value As Single)`

Java	`void setCharSpacing(float value)`
	`void setWordSpacing(float value)`
	`void setTz(float value)`

The character spacing (Tc) adds some space between each character of a text string. The measure is in points. It does not scale with the text's font size. The word spacing is an additional spacing that is applied to space characters only.

The "Tz" value controls the horizontal scaling of text. The default value is 100.

8.20Set the Gray Level for Lines and Filling

Native	`PConSetGray(ContentHandle h, float line, float fill)`
COM	`SetGrayLevel(ByVal GrayLine As Single, ByVal GrayFill As Single)`

Java	`void setGray(float line, float fill)`
	`void setGrayLine(float value)`
	`void setGrayFill(float value)`

Text characters consist of a line shape (that is usually not drawn) and the fill area. Thus, you can set the gray level of text by setting the gray level for filling ("g" operator in PDF).

.. etc.

8.21Set the Color for Lines

Native	`PTError PConLineColor(ContentHandle h, float red, float green, float blue)`
	`PTError PConLineColorCMYK(ContentHandle h, float cyan, float magenta, float yellow, float black)`

COM	`SetLineColor(ByVal red As Single, ByVal green As Single, ByVal blue As Single)`
	`SetLineColorCMYK(ByVal cyan As Single, ByVal magenta As Single, ByVal yellow As Single, ByVal black As Single)`

Java	`void setLineColor(float red, float green, float blue)`
	`void setLineColorCMYK(float cyan, float magenta, float yellow, float black)`

This method sets the color of lines. The values of r, g, b must lie in the range of 0 and 1. They correspond to the contributions of red, green and blue. 0,0,0 corresponds to black, 1,0,0 to red, etc. Alternatively the color can be set using CMYK (Cyan, Magenta, Yellow, Black) parameters. The range of the CMYK color parameters lies between 0 and 1.

8.22Set the Color for Filling

Native	`PTError PConFillColor(ContentHandle h, float red, float green, float blue)`
	`PTError PConFillColorCMYK (ContentHandle h, float cyan, float magenta, float yellow, float black)`

COM	`SetFillColor(ByVal red As Single, ByVal green As Single, ByVal blue As Single)`
	`SetFillColorCMYK(ByVal cyan As Single, ByVal magenta As Single, ByVal yellow As Single, ByVal black As Single)`

Java	`void setFillColor(float red, float green, float blue)`
	`void setFillColor(float cyan, float magenta, float yellow, float black)`

This method sets the color for filling shapes. It also affects the color of text.

8.23Set the Alpha Transparency for Filling and Stroking

Native	`PTError PConSetFillAlpha(ContentHandle h, float alpha)`
	`PTError PConSetStrokeAlpha(ContentHandle, float alpha)`

COM	`SetFillAlpha(ByVal alpha b As Single) As Boolean`
	`SetStrokeAlpha(ByVal alpha b As Single) As Boolean`

Java	`n.a.`

This method sets the alpha transparency for filling shapes and stroking lines. It also affects text.

8.24Using Color Spaces

It is also possible to use color spaces to set the fill and line colors. In order to have a specific color space available for use, it must be defined in the current input file, or it must have been previously copied to the current output file from some other input file (see *PDocInputCopyColor*).

Native	`PTError PConSetFillCS(ContentHandle h, const char* color, float scn)`
	`PTError PConSetLineCS(ContentHandle h, const char* color, float scn)`

COM	`SetFillCS(ByVal Color As String, ByVal scn As Single) As Boolean`
	`SetLineCS(ByVal Color As String, ByVal scn As Single) As Boolean`

Java	`void setFillCS(String Color, float scn)`
	`void setFillCS(String Color)`
	`void setLineCS(String Color, float scn)`
	`void setLineCS(String Color)`

These functions return a boolean indicating successful setting of the color. To obtain a list of all available colors, you can use the functions *IDocNumColorSpaces* and *IDocGetSeparation*.

8.25Placement of Character Strings

Native	`PTError PConPutText(ContentHandle h, const char* text)`
	`PTError PconPutTextU(ContentHandle h, const PDNSTR text)`
	`PTError PConPutLn(ContentHandle h)`

COM	`PrintText(Text As String, x As Single, y As Single)`
	`PrintNewLine()`

Java	`void putText(String Text)`
	`void putLn()`

Print a text string using the current font. You previously need to set the location (Text Matrix).

The COM interface optionally accepts new coordinates for the text.

PutLn adds a T* operator to the stream.

8.26Placement of a Logo

Native	`PTError PDocLogo(Handle h, const char* logoFile, short backGround)`
	`PTError PDocLogoFile(Handle h, const char* logoFile, PTClipType ct)`
	`PTError PDocLogoInput(Handle h, InputHandle hIDoc, PTClipType ct)`
	`PTError PConPrintLogo(ContentHandle h, long id)`
	`InputHandle PDocGetLogoHandle(Handle h)`

COM	`{PDoc.}SetLogoFile(ByVal Filename As String, Optional Clipping As PTClipType) As Boolean`
	`{PDoc.}SetLogoInput(LogoInput As IDoc, Optional Clipping As PTClipType) As Boolean`
	`{PContent.}PrintLogo(Num As Long) As ErrorType`
	`{PDoc.}Logo() As IDoc`

Java	`boolean {PTDoc.}setLogoFile(String Filename)`
	`boolean {PTDoc.}setLogoFile(String Filename, int cliptype)`
	`boolean {PTDoc.}setLogoFile(String URL)`
	`boolean {PTDoc.}setLogoFile(PTInput input, int cliptype)`
	`void {PTContent.}putLogo(int LogoPageNum)`
	`PTInput {PTDoc.}getLogoFile()`

First, you need to define which PDF file to extract logos from. Subsequently, you can select any page of the logo file as the logo to be placed either on the page content or on the header layer.

The box, which should be applied when copying the logo page can be set to any box (*pdClipTrimBox*, *pdClipCropBox*, *pdClipMediaBox*, *pdClipBleedBox*). The default is the *TrimBox*.

The native interface works slightly different for backward compatibility reasons: *PDocLogo* implicitly also prints the logo from page one, and it is possible to put it in the background. The new function *PDocLogoFile* only opens the file and leaves it up to *PConPrintLogo* to use it.

A PTNullRef value returned by the PrintLogo function indicates that the page being used as a logo does not contain any contents and thus has no effect on the output. The Java function putLogo will not raise an exception in this case as it would when encountering some other error (such as PTFailed when passing an invalid page number).

Please note that pages merged from existing PDF files may not be transparent and thus cover the background logo. On the other hand, the logo may not be transparent and hide existing contents if placed in the foreground. The best technique is thus to make sure the logo is transparent as required and place it in the foreground. As a help to this, it is possible to apply a crop box to the logo file (see below). Unfortunately, Acrobat insists on a minimal size for cropped pages. You may need other ways to reduce the crop box further (the "pdcat" tool can do it). With Adobe Acrobat, you can remove any background rectangles with the TouchUp Object Tool. PrepTool inspects the logo's content stream and removes a white background if it is the first object in the stream.

There is no coordinate transformation when placing the logo, i. e. it will be shown at the same offsets to the coordinate system origin (0,0 - left, bottom) as in the uncropped logo file. You can use the DrawCmd function to set a coordinate system transformation (PDF "cm" operator), if you want to set the position of the logo via the API.

The bounding box (clip rectangle applied to the logo when being placed on a page) for the logo corresponds to the TrimBox if specified - otherwise the MediaBox of the logo file).

Note: the same logo can be applied to several PDF files to be merged.

Several logo files can be used to contribute to the construction of a PDF document. An output document keeps the logo files open, and you can switch back to a previously used logo file by setting it again. The *PrintLogo* (*putLogo*) method applies to the currently active logo file.

8.27Placement of an Image

An image imported via *CreateImage* can be placed into a page (or header) content using *PrintImage*.

The X/Y/W/H parameters can be omitted. In this case, no coordinate transformation to place the image in the specified rectangle is generated. If you want to rotate the image, it would be necessary that you explicitly generate the transformation matrix before placing the image.

Native	**`PTError PConPrintImage(ContentHandle h, int ident, float X, float Y, float Width, float Height)`**
COM	**`PrintImage(ByVal Ident As Long, ByVal X As Single, ByVal Y As Single, ByVal Width As Single, ByVal Height As Single) As Boolean`**
Java	**`void PintImage(int ident, float X, float Y, float Width, float Height)`**

To place an image via a coordinate transformation, you need to issue the following PDF operator sequence:

DrawCmd("q")	save the current coordinate system state
DrawCmd("1 2 3 4 5 6 cm")	set the coordinate transformation using the "cm" operator
PrintImage(1)	generate the XObject placement into the stream
DrawCmd("Q")	restore to the saved coordinate system state

Note that (1, 2, 3, 4, 5, 6) is just an example showing the syntax of the command. The actual numbers will be determined by the scaling, rotation, and positioning parameters you have.

8.28Embedding any PDF Text Operator

Native	`PTError PConTextOp(ContentHandle h, const char* command)`
COM	`TextCmd(ByVal Command As String)`
Java	`void putTextOp(String Command)`

You can pass any legal PDF text operator directly to the PDF stream. Correctness of the command is not checked. PT only makes sure that your command will be surrounded by "BT" and "ET" operators.

8.29Set the Spacing of Text Lines

Native	`PTError PConSetLineSpacing (ContentHandle h, float value)`
COM	`SetLineSpacing(TL As Single)`
Java	`void setLineSpacing(float TL)`

This function will send a "TL" operator to the PDF stream. A useful line spacing would be equal to the current font size.

8.30Set the Text Matrix

Native	`PTError PConSetTm(ContentHandle h, float a, float b, float c, float d, x float, y float)`
COM	`SetTm(a As Single, b As Single, c As Single, d As Single, x As Single, y As Single)`
Java	`void setTm(float a, float b, float c, float d, float x, float y)`

Set the text matrix. The default text matrix is [1 0 0 1 0 0]. The first 4 numbers determine the orientation of the text being written subsequently. [1 0 0 1] means text is written in increasing x direction and constant y coordinate. The last two numbers in the text matrix define the coordinates of the starting point for text.

8.31Set a Relative Starting Position for Text (Tab)

Native	`PTError PConPutTab(ContentHandle h, float a, float b)`
COM	n.a.
Java	`void putTab(float a, float b)`

Issues a “Td” operator with the specified arguments. (See PDF specification).

8.32Calculate the Width for a Character String

Native	`float PConGetTextWidth(ContentHandle h, const char* text)`
	`float PConGetTextWidthU(ContentHandle h, const PDBSTR text)`

COM	`GetTextWidth(ByVal Text As String) As Single`
Java	`float getTextWidth(String Text)`

This function calculates the length that the specified text string would need with the current font settings. You can use this to adjust the starting coordinates for center or right alignment.

Please note that the function does not take into account any character or word spacing that you might have set using the *TextOp* function.

8.33Text Tables

Native	`short PConTableHeight(ContentHandle h, short nrRows)`
	`PTError PConTableDraw(ContentHandle h, short Left, short Top, short NumRows, short NumCols, short ColumnWidths[])`
	`PTError PConTableText(ContentHandle, short Row, short Column, const char* Text, short Alignment)`
	`PTError PConTableTextU(ContentHandle, short Row, short Column, const PDBSTR Text, short Alignment)`

COM	`GridHeight(nRows As Integer) As Integer`
	`PrintGrid(x As Integer, y As Integer, nRows As Integer, col1Width As Integer, col2Width As Integer, col3Width As Integer, col4Width As Integer)`
	`GridText(row As Integer, col As Integer, Text As String, Alignment As Integer)`

Java	`short calucateGridHeight(short nRows)`
	`void drawGrid(short x, short y, short nRows, short colWidths[])`
	`void putGridText(short row, short col, String text, short Alignment)`

This set of functions lets you draw the border lines of a simple table and fill the table with text. You need to set the text font and size first. This setting will determine the vertical dimensions of the table.

PrintGrid must always be called prior to printing text. If you do not want any grid lines to be drawn, set the line width to zero (*SetLineWidth(0)*).

The column widths need to be specified explicitly. In the COM interface, you can have at most 4 columns. The parameters are optional. You will get as many columns as you specify widths.

8.34Draw a Line or Polygon

Native	`PTError PConSetLineWidth(ContentHandle h, float Width)`
	`PTError PConMoveTo(ContentHandle h, float X, float Y)`
	`PTError PConDrawTo(ContentHandle h, float X, float Y)`

COM	`SetLineWidth(Width As Single)`
	`MoveTo(x As Single, y As Single)`
	`DrawTo(x As Single, y As Single)`

Java	`void setLineWidth(float Width)`
	`void moveTo(float x, float y)`
	`void drawTo(float x, float y)`

Use these functions to draw a line or line polygon.

Note that there are different possible settings for line joins. Please refer to the PDF specifications ("j" operator).

8.35Draw a Rectangle

Native	`PTError PConRectangle(ContentHandle h, float x, float y, float width, float height, short how)`
COM	`DrawRect(x As Single, y As Single, w As Single, h As Single, how As ShapeFlags)`
Java	`void drawRectangle(float Left, float Bottom, float Width, float Height, int FillType)`

Draw a rectangle with the specified location and dimensions. The parameter "how" determines, if the rectangle is filled and if the border is drawn: 0=fill area only, 1=both, 2=border only

8.36Draw Curves

Native	`PTError PConCurveTo(ContentHandle h, float xy[], short type)`
COM	`CurveTo(x1 As Single, y1 As Single, x2 As Single, y2 As Single, Optional x3 As Single, Optional y3 As Single, type As PTCurveType)`
Java	`void curveTo(float xy[], char type)`

Draw a Bézier curve of the specified type (‘c’, ‘v’, or ‘y’; see PDF specifications).

The ‘c’ type curve requires 3 coordinate pairs, the other types only 2.

This function can be used to extend the current path – just like *drawTo*.

8.37Area Filling and Clipping

Native	`PTError PConDrawArea(ContentHandle h, short clip)`
COM	`DrawArea(Optional Clip As Boolean)`

Java	`void drawArea()`
	`void drawAndClipArea()`

Close the path constructed with *drawTo* (and/or *curveTo* calls, and fill with the current color. Optionally, the clip area is also set to this area.

8.38Embedding any PDF Non-Text Commands

Native	`PTError PConDrawOp(ContentHandle h, const char* command)`
COM	`DrawCmd(ByVal Command As String)`
Java	`void putDrawOp(String Command)`

Pass the specified PDF command string as is to the content stream. You can use this to make use of many PDF features that are not available by specific API calls.

9 Form Fields, Annotations

9.1 Set the Data

Native	`PTError PDocInputSetFormData(Handle h, const char* fieldName, const char* fieldData, short Formflags, short Annotflags)`
COM	`InputSetFormData(Fieldname As String, Data As String, Ff As PTFormFlag, Af as PTAnnotFlag) As Boolean`

Java	`void inputSetFormData(String Fieldname, String Data)`
	`void inputSetFormData(String Fieldname, String Data, boolean noRO)`
	`void inputSetFormData(String Fieldname, String Data, short Formflags, short Annotflags)`

Use this to populate the text fields of an input file with data, after opening the form template using *PDocInputOpen* and before calling *PDocInputCopyPages* to generate the output containing the new data.

This method is called in the context of the output file, because the data is not actually set in the input file first, but rather added on the fly when the pages are copied to the output file. You will not get an error when specifying an invalid field name or a field name that is not copied, because the page containing the field is not in the range of pages that you specify in *InputCopyPages*.

Note that it is possible to define multiple fields with the same name in Acrobat. All these fields have the data in common, but may differ how they appear (placement, font, alignment, etc.). *PDocInputSetFormData* will set the data in all instances, respecting their individual appearance settings.

The *NoReadOnly* parameter allows you to leave the *ReadOnly* attribute of the fields (use 1). Specifying a value of 0 will set all instances of the field to "read only".

The attributes of the form fields can not be set via the API. Set font, font size, alignment and so on using Acrobat Exchange in the template file.

Note that no text formatting is supported, and only the standard Acrobat fonts can be used (unless the field has been created using *AddTextField* – see below).

Text wrapping will be performed automatically in multi-line fields. You may also supply already formatted data (e. g. for numbers and dates). To explicitly mark a newline in multi-line text, use "\r" (Chr$(13)). With this exception, you must use only printable characters.

You can also print data on a page using output to the header layer. If you want to place a bar code or image on the page, this is the way to do it.

PT allows you to re-use a specific page from the input file as a template that is filled with data and copied to output many times. Please be aware of the fact that you have form fields with identical names but different data in the output file. Once you open this file in Acrobat, a change of the data of one field will affect all other fields with this name. As a precaution, you may thus want to set these fields to read-only.

The value of a check box is set using the export string (checked) or the constant string "Off" (unchecked).

Radio buttons are set by specifying the export string (value of the button to be "on"). "Off" can be used to set all to the "off" state.

9.2 Define a Custom Font

Native	`PTError IDocSetFormFont(InputHandle h, short fontID, const char* Basefont)`
COM	`SetFormFont(FontID As PTFormFontType, ByVal BaseFontName As String) As Boolean`
Java	`void setFormFont(short fontID, String basefont)`

This function defines a custom font that can be used for text form fields. This function only applies to fonts of form fields.

9.3 Get a Font Name

Native	`PTError IDocGetFontName(InputHandle h, short fontID, VBSTR* name)`
COM	`GetFontName(FontID As PTFormFontType) As String`
Java	`String getFontName(short fontID)`

This function returns the name of the base font that corresponds to the specified font number. This function only applies to fonts of form fields.

9.4 Delete a Form Field

Native	`PTError IDocDeleteFormField(InputHandle h, const char* fieldName)`
COM	`DeleteFormField(ByVal Fieldname As String) As Boolean`
Java	`void deleteFormField(String Fieldname)`

This function deletes all instances of a form field from a template file. Note that the enumerator of the function *IDocGetFormData* is affected. When field 1 is deleted, field 2 becomes number 1 etc.

9.5 Add a Text Form Field

Native	`PTError IDocAddTextField(InputHandle h, const char* fieldName, const char* fieldDescr, float box[], int page, short fontID, float fontSize, short alignment, int FormFlags, int AnnotFlags, int borderRGB, int backgroundRGB, int rotate, int textRGB)`
COM	`AddTextField(ByVal FieldName As String, ...) As Boolean`
Java	`void addTextField(String fieldName, ...)`

	`void addTextFieldEX(String fieldName, ...)`

This function adds a text form field to a PDF file. Note that this field is put into the transient memory cache of an input file which cannot be saved as such, but must be copied to an output file. To fill in data into a text field that is added this way, you can use the *SetFormData* method. The name of the text field may not contain a "." (period).

The colors (borderRGB, backgroundRGB) are encoded in the following way:

RGB = RED[0..255] + 256*GREEN[0..255] + 256*256*BLUE[0..255]

or as Hex numbers: RGB = 0xBBGGRR (&H00BBGGRR in Visual Basic)

For example: &H000000FF is red, &H0000FF00 is green, etc. (as in the Visual Basic color settings)

9.6 Copy a Form Field

Native	`PTError PDocAddFieldFromLogo(Handle h, const char* fieldName, int pageNumber, float X, float Y, float Width, float Height, const char* newName)`
COM	`AddFieldFromLogo(ByVal FieldName As String, PageNumber As Long Optional X As Single, Y As Single, Width As Single, Height As Single, NewName As String) As Boolean`

Java	`void addFieldFromLogo(String fieldName, int pageNumber)`
	`void addFieldFromLogo(String fieldname, int pageNumber, float X, float Y, float Width, float Height)`
	`void addFieldFromLogo(String fieldname, int pageNumber, float X, float Y, float Width, float Height, String newName)`

This function copies a form field from an existing PDF document to an output PDF document. The file where the field is taken from is the current logo file (s. *SetLogoFile*). You cannot change anything about the field except the coordinates and the name.

9.7 Form Flattening

Native	`PTError PDocSetFlatten(Handle h, short On, int mode)`
COM	`SetFlatten(ByVal On As Boolean, Mode As PTFlattenMode)`

Java	`void setFlatten(boolean on)`
	`void setFlatten(Boolean on, int mode)`

This output file setting has the effect that text fields are rendered into the page content during subsequent *InputCopyPages* calls. This means that the form field is eliminated, and its content now constitutes part of the page content.

9.8 Add a Text Annotations

This function adds text annotations to an input handle.

Native	`PTError IDocAddTextAnnotation(InputHandle h, const char* label, const char* content, float rect[X1, Y1, X2, Y2], int page, float color[R, G, B], int annotFlags)`
COM	`AddTextAnnotation(Name As String, Content As String, X As Single, Y As Single, Width As Single, Height As Single, Page As Long, R As Single, G As Single, B As Single, Flags As PTAnnotFlags) As Boolean`

Java	`void addTextAnnotation(String title, String content, PTRectangle location, int colorRGB)`
	`void addTextAnnotation(String title, String content, PTRectangle location, int colorRGB, int annotFlags)`

The "Name" is the title of the text annotation, the "Content" is the text that is visible when the annotation is opened. The three values for the colors (R: Red, G: Green, B: Blue) are in the range from 0 to 1. The text annotation is closed per default. The default for the annotation flags is *PTFlagAnnotPrintable*.

Native: The four values of the position rectangle mark the lower left corner (X1, Y1) and the upper right corner (X2, Y2), 0, 0 being in the lower left. The parameter of the page number is zero based.

COM: The four values of the position mark the lower left corner (X, Y) and the width and height of the opened text annotation, 0, 0 being in the lower left. The parameter of the page number is non-zero based.

9.9 Delete an Annotation

This function deletes a text annotation.

Native	`PTError IDocDeleteAnnotation(InputHandle h, int Identification)`
COM	`DeleteAnnotation(Identification As Long) As Boolean`
Java	`boolean deleteAnnotation(int id)`

The parameter Identification is the value that can be retrieved using the method *GetAnnotation*.

9.10Delete Viewer Extension Rights

A PDF document can have so called Viewer Extension Rights, which allow the document to be modified (do form filling) and save it with the Acrobat Reader. Modifying such a document with the Prep Tool Suite will destroy the Viewer Extension Rights. A warning message will therefore appear when the document is opened in Acrobat Reader. This function deletes the Viewer Extension Rights and therefore there will be no warning message when opened in Acrobat Reader.

Native	`int IDocDeleteViewerRights(InputHandle h)`

COM	`DeleteViewerRights() As Boolean`
Java	n.a.

9.11Add an Image Annotation

This function is only available for the COM interface. It adds an annotation containing an image. It is applied to the output handle. The type of the annotation can be either a standard stamp annotation or a custom stamp annotation. Prior to calling *AddImageAnnotation*, it is required to create an image and get its ID using *CreateImage* or *CreateImageEx*.

Native	n.a.
COM	`{PDoc.}AddImageAnnotation(Page As Long, r1 As Single, r2 As Single, r3 As Single, r4 As Single, ImageID As Long, Optional SubType As Integer)`
Java	n.a.

Parameters:

Page	The page number where the annotation is to be placed.
r1, r2, r3, r4	Positioning parameters in PDF points (left, bottom, width, height), 0/0 at lower left.
ImageID	The image ID which is returned from *CreateImage* or *CreateImageEx*.
SubTypes (optional)	The available sub types are: 0 = Standard Stamp Annotation 1 = CstmStamp Annotation other = Unkown Subtype

9.12Set the Line Spacing in a Form Field

The spacing between the text lines in a form field can be defined by using the following method.

Native	`void IDocSetFormLineSpacing(InputHandle h, float fLineSpacing)`
COM	`InputSetFormLineSpacing(float LineSpacing)`
Java	`void setFormLineSpacing(float value)`

9.13Get the Name of the Font in a Form Field

The name of the font used in a form field can be queried by using the following method. The value of the id parameter can be retrieved by invoking the `IDocGetFormBox` method.

Native	`PTError IDocGetFontName(InputHandle h, short id, VBSTR* name)`
COM	`GetFontName(PTFormFontType FontID) As String`
Java	`String getFontName(int FontID)`

10 Generate Output

10.1Create Another Page

Native	`PTError PDocNewPage(Handle h)`
COM	`NewPage()`
Java	`void newPage()`

A page is automatically create when you access the page object – either at the beginning of a file, or after you copied pages from an input file. If you need a new page (i.e. a page brake), you call the *NewPage* function. The dimensions of the page are inherited from the last setting made (*PDocNew*, *PDocPageSize*).

10.2Copy Pages from the Input File

Native	`PTError PDocInputCopyPages(Handle h, int firstPage, int lastPage)`
	`PTError PDocMerge(Handle h, const char* inputFile, int firstPage, int lastPage)`

COM	`InputCopyPages(FirstPage As Long, LastPage As Long) As Boolean`
	`Merge(FileName As String, FirstPage As Integer, LastPage As Integer) As  Boolean`

Java	`void inputCopyPages(int FirstPage, int LastPage)`
	`void merge(String Filename, int Firstpage, int Lastpage)`

InputCopyPages and *Merge* will copy the specified range of pages from the currently open input file to output. If the file contains form fields, the field data will be set according to previous *PDocInputSetFormData* calls. You can repeatedly call *CopyPages* and change the header in between.

The pages being copied can be modified not only by setting form data prior to *InputCopyPages*, but also by setting the new Rotate (page orientation) value (s. *SetInputRotate*).

Note that you should copy all pages containing forms of an input file. Leaving away a page with form fields will result in orphan entries; duplication of pages works for viewing the resulting file, but Acrobat 4.0 may behave unexpectedly if you want to modify a form field of a duplicated page.

Merge (add) pages from an existing PDF file into the output document. The range of pages to be added is specified using the parameters *firstPage* and *lastPage*. The current *Header* will be placed on all of these pages. *PDocMerge* works like *PDocInputOpen* followed by *PDocInputCopyPages*.

PDocMerge returns FALSE (0), if the input file cannot be processed. *PDocMerge* will automatically close the current input file, if one exists.

Be careful with repeated merging of PDF files. The font alias used for the header text is determined before the file to be merged is known. Repeated merging in combination with placing header text will result in a name conflict. The alias used by Prep Tool is "/FHdr". You can avoid this problem by using *PDocInputOpen* before setting the font, and then copy the pages using *PDocInputCopyPages*. A potential conflict still remains when you add further PDF files while keeping the header with its font.

10.3Copy Color Spaces from the Input File

Native	`PTError PDocInputCopyColor(Handle h, const char* Color)`
COM	`InputCopyColor(ByVal Color As String) As Boolean`
Java	`void inputCopyColor(String Color)`

Use this function to copy a color space object from the current input file to the output file. This feature is useful if you want to use Pantone colors: store the set of colors you need in a PDF file, and use this file at runtime to provide the necessary color definition objects. This function works in conjunction with the *PConSetFillCS* and *PConSetLineCS* functions.

10.4Copy Named Destinations from the Input File

Native	`PTError PDocInputCopyDestNames(Handle h)`
COM	`InputCopyDestNames()`
Java	`void inputCopyDestNames()`

This function will copy all named destination entries from the input to the output file. A situation where this makes sense is when you have bookmarks and links that are also to be copied. If you do not copy the named destinations, the bookmarks and links will work, but loose the zoom level, because resolution only works on the page level. Not copying the named destinations will save space in the resulting file.

10.5Copy Custom Objects from the Input File

Native	`PTErrpor PDocInputCopyCustObjs(Handle h)`
COM	`InputCopyCustomObjs()`
Java	`void inputCopyCustomObjs()`

The input file may contain entries in the Catalog object that are not taken care of by any of the existing copy functions. To copy these entries along with any referenced objects, you can use this method. Note that it will copy e. g. also viewer settings or open actions, if such settings are not specified explicitly for the output documents and if present in the input file. This function can be used to merge JavaScript resources, it returns the error *pdAlreadyWritten* when duplicate JavaScript names are encountered.

10.6Copy All Objects from the Input File

Native	`PTError PDocInputCopyAll(Handle h)`

COM	`InputCopyAll() As Boolean`
Java	`void inputCopyAll()`

This method is equivalent to *InputCopyPages* (for all pages), *InputCopyDestNames*, *InputCopyBookmarks*, and *InputCopyCustomObjs*. In other words, it copies the whole file content from input to output.

10.7Import Bitmap Images

Native	`short PDocCreateImage(Handle h, int Width, int Height, short bits, short  color, char* imageData, int imageSize, char* palette, short compression, char* mask)`

COM	`CreateImage(ByVal Width As Long, ByVal Height As Long, ByVal Bits As Integer, ByVal Color As Boolean, ByVal ImageData As Byte(), Optional ByVal Palette As Byte(), Optional ByVal IsJPEG as Boolean, Optional Mask As Byte(), Optional Softmask As Byte()) As Long`
	`CreateImageEx (ByVal Width As Long, ByVal Height As Long, ByVal Bits As Integer, ByVal Color As Boolean, ByVal ImageData As Byte(), Optional ByVal Palette As Byte(), Optional Mask As Byte(), Optional CompressionType As Integer) As Long`

Java	`int createImage(int Width, int Height, short bits, boolean color, byte[] image, byte[] palette)`
	`int createJPEGImage(int Width, int Height, ..)`
	`int createImage(int Width, int Height, short bits, boolean color, byte[] image, byte[] palette, boolean isJPEG )`
	`int createImage(int w, int h, short bits, boolean color, byte[] image, byte[] palette, byte[] mask, int image_type)`
	`final static image_type_standard = 0`
	`final static image_type_JPEG = 1`

The *CreateImage* function creates an image XObject according to the data provided. The format of the data must correspond to one of the PDF standards for color, grayscale or bi-level images. Color images have a palette. The palette size in PDF must be 768. If the effective palette is smaller, the unused part must be set to zero in the native interface. The COM interface will automatically handle smaller palette sizes.

A positive number value returned by *CreateImage* identifies the XObject for reference in *PrintImage* calls. A value of zero indicates failure to create the image.

10.8Add Page Numbers

Native	`PTError PDocPutPageNumbers(Handle h, float X, float Y, const char* Format, long StartPage, short Orientation)`
COM	`PutPageNumbers(Format As String, X As Single, Y As Single,`

	`Startpage As Long, Orientation As Integer)`
Java	`void putPageNumbers(String Format, float X, float Y, int StartPage, int Orientation)`

Expand the page marker "%p" in the format string to reflect the current page number and put this string on each page just like other header text. With the *firstPgNr* parameter, you specify where page numbers should start. The page numbering string will appear on each header that is displayed, starting with the next *PDocMerge*. The first time the page numbering string is displayed, it will carry the page number specified in *firstPgNr*. Note that the header or the page numbering string may be created after having copied some pages to the output file. Any previously set format string will be removed. To stop putting page numbers on a page, you can thus call this function with an empty format string.

The font used for the page number text is the same as for the header. Do not forget to specify a font. If you work with different font sizes, then the last setting will be the one used for the page number string, e. g. if *PDocHeaderFont* was called after *PDocHeaderPgInfo*. The font of page numbers is unpredictable if you do not have a header layer!

Example: PDocHeaderPgInfo(h, 10, 10, "Page %p of 10", 2);

10.9Change the Header or Background

Native	`PTError PDocHeaderClear(Handle h)`
	`PTError PdocBackgroundClear(Handle h)`

COM	`HeaderClear()`
	`BackgroundClear()`

Java	`void clearHeader()`
	`void clearBackground()`

When you want to change the text, graphics objects or logo added to merged pages, call *PDocHeaderClear* and build the new header as desired. You cannot continue to add anything (text, graphics, logos) to a header stream, once it is applied to pages (i.e. after calling *CopyPages* or *Merge*). This is because the header stream is written to the output file at this point, and any changes that you make after that are ignored.

After calling *PDocHeaderClear*, you need to set the font for header text again. It is possible to re-use the last font imported from an input file by specifying the same name again.

10.10Add Bookmarks

Native	`PTError PDocInputCopyBookmarks(Handle h, int level)`
	`PTError PDocAddWebBookmark(Handle h, int level, const char* title, const char* URL, int kidsVisible)`
	`PTError PDocAddGoToBookmark(Handle h, int level, const char* title, int page, short X, short Y, int kidsVisible, float zoom)`

	`PTError PDocAddGoToBookmarkU(Handle h, int level, const PDBSTR, int page, short X, short Y, int kidsVisible, float zoom)`
	`PTError PDocAddGoToRBookmark(Handle h, int level, const char* title, const char *destFile, int destPage, short X, short Y, int kidsVisible, float zoom)`
	`PTError PDocAddGoToRBookmarkU(Handle h, int level, const PDBSTR title, const char *destFile, int destPage, short X, short Y, int kidsVisible, float zoom)`
	`PTError PDocAddOpenFileBookmark(Handle h, int level, const char* title, const char* destFile, int kidsVisible)`
	`PTError PDocAddOpenFileBookmarkU(Handle h, int level, const PDBSTR title, const char* destFile, int kidsVisible)`
	`PTError PDocAddNullBookmark(Handle h, int level, const char* title, int kidsVisible)`
	`PTError PDocAddNullBookmarkU(Handle h, int level, const PDBSTR title, int kidsVisible)`
	`PTError PDocAddJavaScriptBookmark(Handle h, int level, const char* title, const char* script, int kidsVisible)`
	`PTError PDocAddJavaScriptBookmarkU(Handle h, int level, const PDBSTR title, const char* script, int kidsVisible)`

COM	`AddWebBookmark(Title As String, Level As Integer, URL As String, ShowKits As Boolean)` etc.
Java	`void addWebBookmark(String Title, int Level, String URL)` etc.

All these methods have optionally a further parameter for specifying that bookmarks on lower levels shall be visible.

The bookmark tree of the output file can be constructed using the above functions. The first bookmark must be placed on level zero. Subsequent bookmarks can be placed at most one level above the previous level.

An URL is something like "http://www.pdf-tools.com", but it is also possible to put relative links like "../index.html".

"GoTo" targets are pages in the same document as the one being created. “page” is the page number (starting at 1). The "x" and "y" parameters can be used to set the view window according to the /XYZ entry in link annotations (see PDF specification). Specify zero values to disable this feature. The "z" parameter describes the zoom value. 1 stands for 100%, 1.1 for 110%, etc, 0 for keep the current zoom value.

"GoToR" targets are "remote" links, i. e. links to another PDF file (you will note the extra file name parameter).

"OpenFile" targets are files that represent a document or application that is to be launched. Document files are opened with the application that is registered for the document type. On Windows systems, the file extension is used for this.

A Java Script added to a bookmark will be executed when the bookmark is selected.

If bookmarks refer to named destinations, they will be resolved to avoid conflicts between names in different files.

10.11Add Links

Native	`PTError PDocAddWebLink(Handle h, int page, const float rect[], const char* URL, int style)`
	`PTError PDocAddGoToLink(Handle h, int page, const float rect[], int destPage, short x, short y, int style, float zoom)`
	`PTError PDocAddGoToRLink(Handle h, int page, const float rect[], const char* destFile, int destPage, short X, short Y, int style, float zoom)`
	`PTError PDocAddJavaScriptLink(Handle h, int page, const float rect[],  const char* script, int style)`
	`PTError PdocAddNamedDestLink(Handle h, int page, const float rect[], int style, const char* destName)`

COM	`AddWebLink(Page As Long, Left As Single, Bottom As Single, Right As Single, ByVal URL As String, Optional Style)` etc.
Java	`void addWebLink(PTRectangle Rect, String URL)` etc.

The action behavior of links corresponds to that of bookmarks. Links are located as an annotation on a page. Therefore, you need to specify the page number and coordinate rectangle where to put the link instead of the hierarchy level in the bookmark tree.

The Prep Tool Suite supports several border styles; the value –1 will suppress the border, 0 will result in a solid black border, 1 is dotted red, 2 red solid, 3 green dashed, 4 green solid, 5 blue dashed, 6 blue solid.

10.12Add File Attachments

Native	`PTError AddFileAttachment(Handle h, int page, float* rect, const char* filepath, const char* icontype, const char* description, const char* author, const char* subject, int rgb, int opacity)`
COM	`AddFileAttachment(Page As Long, Left As Single, Bottom As Single, Right As Single, Top As Single, Filepath As String, IconType As String, Description As String, Optional Author As String, Optional Subject As String, Optional ColorRGB As Long, Optional Opacity As Long)`
Java	`void addFileAttachment(PTRectangle rect, InputStream is, String iconType, String description, String author, String subject, int rgb, int opacity)`

This function adds a file attachment annotation to a PDF file.

Parameters:

- page: the page number (first page is 1)
- Left, Bottom, Right, Top: the annotation's rectangle on the page
- Filepath: the name of the file to be attached
- IconType: the icon's type name ("PushPin","Graph","Paperclip", or "Tag")
- Description: the description field (used as default for the file name when extracting the attachment)
- Author: the author field (optional)
- Subject: the subject field(optional)
- ColorRGB: the RGB value of the color for the icon
- Opacity: the opacity value in percent (0..100); 0 means transparent; 100 means opaque (default)

Note that each standard icon type has its specific rectangle width and height in the Acrobat viewer. Setting other values has the effect that Acrobat viewers will change the appearance when clicking on the icon.

Paperclip size: 7/17	Graph size: 20/20
PushPin size: 14/20	Tag size: 20/16

Implementation restriction:

Creation of the icon appearance stream is not supported when Using opacity less than 100. Acrobat viewers will correctly display these icons, but third party viewers that depend on the appearance stream may not show the icon.

10.13Add Destination

Native	`PTError AddGotoDestination(Handle h, const char* name, int page, short X, short Y, float Z)`
COM	`AddGotoDestination(Name As String, Page As Long, X As Integer, Y As Integer, Z As Single) As Boolean`

Java	`void addGotoDestination(String name, int page)`
	`void addGotoDestination(String name, int page, int X, int Y, float zoom)`

This function adds a Named Destination to the document. A named destination points to a certain location (e.g. the beginning of a chapter) in the PDF. The location is defined by the page number and the X, Y and Z (Zoom) position.

10.14Set Document Action

Native	`PTError PDocSetDocumentAction(Handle h, PTDocumentAction documentaction, const char* Script)`
COM	`SetDocumentAction(ActionType As PTDocumentActionType, Script As String) As Boolean`

Java	`boolean setDocumentAction(int DocumentAction, String Script)`

This function adds a JavaScript to a document action. The five document actions are: 0 on close, 1 before save, 2 after save, 3 before print and 4 after print.

10.15Set Form Fontsize Range

Native	`PTError PDocSetFormFontSizeRange(Handle h, float Max, float Min)`
COM	`SetFormFontSizeRange(Max As Single, As Single)`
Java	`boolean setFormFontSizeRange(float Max, float Min)`

With SetFormFontSizeRange it is possible to limit the font sizes for auto-sized form fields. The default values are Max = 12 and Min = 5.

10.16Document Open Settings

The following group of functions facilitates the setting of the page layout and mode and how the first page shall be displayed when opening the document in a viewer (as offered in the "Document Info"-> "Open.. " dialogue of Acrobat).

Native	`PTError PDocSetPageMode(Handle h, const char* Mode)`
COM	`SetPageMode(ByVal Mode As String)`
Java	`void setPageMode(String Mode)`

The page modes currently supported by Acrobat viewers are "UseNone", "UseOutlines", "UseThumbs", and "/FullScreen". The *SetPageMode* function will override any settings from input files that would otherwise be copied during *InputCopyCustomObjs*.

Native	`PTError PDocSetPageLayout(Handle h, const char* Layout)`
COM	`SetPageLayout(ByVal Layout As String)`
Java	`void setPageLayout(String Layout)`

The following layouts can be specified: "SinglePage", "OneColumn", "TwoColumnLeft", "TwoColumnRight".

Native	`PTError PDocSetOpenAction(Handle h, int Page, const char* Magnification)`
COM	`SetOpenAction(ByVal Page As Long, ByVal Magnification As String)`
Java	`void setOpenAction(int Page, String Magnification)`

The page to be shown initially when a file is opened can be specified using this function. At the same time, the zoom factor or type of "fit" can be specified. Legal values for page numbers are 1 through the number of pages that the file contains; the magnification can be a positive integer number representing the zoom factor in percent (100 = normal 100% zoom). The minimum and the maximum is viewer dependent (currently 25 – 1600). Other legal "magnifications" are "Window" for "Fit Window", "Width" for "Fit Width", and "Visible" for "Fit Visible". Any other value will be mapped to "Default".

Native	`PTError PDocClearViewerPreferences(Handle h)`
	`PTError PDocAddViewerPreference(Handle h, const char* Key, const char* Value)`

COM	`AddViewerPreference(ByVal Key As String, ByVal Value As String, Optional ClearExisting As Boolean)`

Java	`void clearViewerPreferences()`
	`void addViewerPreference(String Key, String Value)`

The viewer preferences entries can be created (or suppressed) by these functions. For a complete listing of all possible settings, please refer to the PDF specifications.

Viewer preferences are stored in a dictionary. The *AddViewerPreferences* function adds a pair of values consisting of the dictionary key and its associated value. Examples are "/HideToolbar true", "/FitWindow true", "/CenterWindow true", "/NonFullScreenPageMode /UseThumbs".

10.17Set Document Information Attributes

Several document attribute values can be set via the following methods. Note that the value string will be re-encoded from WinAnsiEncoding to PDFEncoding (see Adobe PDF Reference Manual). This means that only characters existing in both encodings may be contained.

Native	`void PDocSetInfo(Handle h, const char* Title, const char* Subject, const char* Author, const char* Keywords)`
COM	`SetInfo(ByVal Title As String, ByVal Subject As String, ByVal Author As String, ByVal Keywords As String)`
Java	`void setInfo(String Title, string Subject, string Author, String Keywords)`

SetInfo allows you to set some of the document attributes in the information object.

Native	`void PDocSetAttr(Handle h, const char* Key, const char* Value)`
	`void PDocSetAttrU(Handle h, const char* Key, const PDBSTR Value)`

COM	`SetAttr(ByVal Key As String, ByVal Value As String)`
Java	`void setInfoAttr(String Key, String Value)`

SetAttr permits to set (and add) any value in the information object of the PDF file.

10.18Set Document Metadata

Native	`PTError PDocSetMetaData(Handle h, const char* data)`
COM	`SetMetaData(Data As String)`
Java	`void setMetaData(String data)`

SetMetaData sets the meta data in the root object of the PDF file. Note that the data string should constitute a valid XML expression.

10.19Close the Output File

Native	`PTError PDocClose(Handle h)`
	`PTError PDocRelease(Handle h)`

COM	`Close() As Boolean`
Java	`void close()`

Close the output document. This procedure writes out any pending output and closes the file.

PDocRelease releases the handle and all memory resources associated with it. No further calls are allowed with this handle.

If you want to verify that the file has been successfully closed, you first want to call *PDocClose*, and then *PDocRelease*. If *PDocClose* fails, you can still use the handle to retrieve error information.

In the Java binding, the close method also releases the associated resources. If an error occurs during the close operation, an exception is signaled carrying the error code.

In the COM binding, releasing the last object reference will automatically close the file and release all associated resources.

To retrieve the bytes of a memory resident PDF file, use the following functions:

Native	`VBSTR PDocCloseB(Handle h, int* length)`
COM	`bytes = CloseB()`

Java	`void close()`
	`byte[] getBytes()`

The *CloseB* functions perform a normal close and return the bytes of the memory resident PDF file. Note that the memory buffer of the file is disposed on close. The memory buffer returned by these functions must be freed by the application.

In Java, the byte array is remains stored with the Java wrapper object and can be multiply accessed through *getBytes()* (until the Java object is "finalized").

10.20Set the license key at runtime

Set the license key programmatically at runtime instead of installing it on the system.

Native	`int PTSetLicenseKey(const char* szLicenseKey)`
COM	`SetLicenseKey(bstrLicenseKey As String) As Boolean`
Java	`boolean setLicenseKey(String szLicenseKey)`

Parameters: The license key

Return value: True: The license key is valid.

Check whether a valid license key has been installed in the system or passed at runtime.

Native	`int PTGetLicenseIsValid()`
COM	`LicenseIsValid() As Boolean`
Java	`boolean getLicenseIsValid()`

Return value: True: A valid license was found.

11 Linearization

Linearization is the processing performed on a PDF file to optimize it for viewing in a web browser. The elements of the PDF file are regrouped, so that all information necessary to display the first page is located at the beginning of the file. Furthermore, information about file offsets is stored in the header of the file and in the so called hint tables.

Due to the nature of linearization, this process can begin only when a PDF file is created completely. The functions supporting linearization are thus separate from other PDF Prep Tool functions.

Native	`PTError PDLinearize(const char* Input, int Length, const char* InputPassword, const char* OutputFileName, const char* OwnerPassword, const char* UserPassword, const char* Permissions)`
	`PTError PDLinearizeMem(const char* Input, int Length, const char* InputPassword, VBSTR* OutputBuffer, int* OutputLength, const char* OwnerPassword, const char* UserPassword, const char* outPermissions)`

The native interface offers just these two functions. Input can be provided either as the file name of the input file when specifying a length of 0, or as the memory buffer containing the PDF "file" along with the length of that buffer.

The first function writes the linearized PDF to a file, while the second returns it in a memory buffer. This memory buffer must be freed using PTFreeVBSTR.

The return result of these function is a *PTError*.

COM	`Dim tool As New PDFLinearizer`
	`Dim tool As Object`
	`Set tool = CreateObject("PrepTool.PDFLinearizer")`
	`SetSecurity(ByVal OwnerPassword As String, ByVal UserPassword As String, ByVal Permissions As String)`
	`OpenInput(ByVal Filename As String, Optional ByVal Password As String) As ErrorType`
	`OpenMem(ByVal PDFBytes As Variant, Optional ByVal Password As String) As ErrorType`
	`SaveFile(ByVal Filename As String) As ErrorType`
	`SaveMem(Optional Result As ErrorType) As Variant`

The COM interface for linearizing PDF files is also quite straight forward. A call of *SetSecurity* is optional and will only be used if the resulting file shall be encrypted.

The COM object can be reused for several linearizations. As the input file resources will be freed on *SaveFile* or *SaveMem*, it is necessary to re-open a file before linearization can be performed again during one of the "Save.. " functions. Password and permission settings are preserved.

Java	`PTLinearizer tool = new ..`
	`  PTLinearizer(String Filename, String Password)`
	`  PTLinearizer(String Filename)`
	`  PTLinearizer(byte[] PDFBytes, String Password)`
	`  PTLinearizer(byte[] PDFBytes)`
	`void setSecurity(String Ownerpassword, String Userpassword, String Flags)`
	`byte[] getLinearizedBytes()`
	`void doLinearization(String Filename)`

The Java API is similar to the COM interface with the difference that no reuse of the *PTLinearizer* object is permitted. Once linearization has been performed, all resources are freed.

12 Return Codes C

0	Success	
1001	PTNotPDF	the file does not start with %PDF
1002	PTTrailer	the trailer of the PDF file could not be found
1003	PTXref	the XRef table could not be found as defined in trailer these two errors indicate that the PDF file has been corrupted as sometimes happens when copied in ASCII mode by FTP
1004	PTNullRef	an object reference could not be resolved (object missing in file)
1005	PTBadParamValue	an illegal parameter value was specified in a method
1006	PTObjRead	a particular PDF object could not be read from the file
1007	PTAlreadyWritten	a particular PDF object was attempted to write twice
1008	PTBadCallSequence	a particular function was called in an inappropriate context
1009	PTInternal	an unexpected situation was encountered that could not be handled
1010	PTUnexpectedVal	an unexpected value was encountered in a PDF object
1011	PTIO	an input/output error was encountered
1012	PTInvalidHandle	the handle specified is not valid
1013	PTDuplicate	an attempt to create a duplicate object is made
1014	PTIllegalFont	an invalid font name was specified
1015	PTNoSuchPage	an invalid page number was specified
1016	PTNotFound	requested information not found for specified criteria
1017	PTFailed	License key invalid or generic error
1018	PTEncrypted	input file is encrypted (password protected)
1019	PTInvalidPassword	the password supplied is not correct

Cytoscape 3.4.0 User Manual

Table of Contents

Introduction

This version of Cytoscape builds upon the new 3.x architecture, developer API and set of user controls established. If you're familiar with former versions of Cytoscape, this version will feel completely familiar and you'll be all set to go. In future releases, we will continue to tweak and improve both the software and the documentation. This manual will be updated to reflect all the latest changes.

This manual describes the installation and use of Cytoscape. For a more thorough understanding of Cytoscape and its ecosystem, we highly recommend reading the ***Welcome Letter*** *accessible on the http://cytoscape.org website.*

Launching Cytoscape

Cytoscape is a Java application verified to run on the Linux, Windows, and Mac OS X platforms. Although not officially supported, other UNIX platforms such as Solaris or FreeBSD may run Cytoscape if Java version 8 or later is available for the platform.

System Requirements

The system requirements for Cytoscape depend on the size of the networks you want to load, view and manipulate.

Note that as of Cytoscape v3.2, networks are loaded faster and in less memory than with previous versions. While this is good news, networks created on v3.2 on a given memory configuration (e.g., 1GB) may not be loadable by prior Cytoscape versions on the same memory configuration.

Required Resources

	SMALL NETWORK VISUALIZATION	LARGE NETWORK ANALYSIS/VISUALIZATION
PROCESSOR	1GHz	As fast as possible, with multiple cores
MEMORY	512MB	2GB+
GRAPHICS CARD	Integrated video	High-end graphics card
MONITOR	XGA (1024X768)	Wide or Dual Monitor

Specific system requirements, limitations, and configuration options apply to each platform, as described in the ***Release Notes*** *available on the http://cytoscape.org website.*

Getting Started

Install Java

Cytoscape requires Java 8 or later.

- While Cytoscape versions prior to v3.2 run on Java 6, Oracle and other JVM suppliers have dropped Java 6 support. Consequently, Cytoscape v3.2 and later don't support Java 6 either. With v3.3, we have also dropped support for Java 7 for the same reason.
- We recommend a 64 bit Java Runtime Environment (JRE). While Cytoscape runs with 32 bit Java versions, using a 64 bit Java allows the largest networks to be loaded and enables the fastest network processing. For Windows, the default JRE download provided at java.com is 32 bits regardless of the Windows version. While Cytoscape will run with a 32 bit JRE, it will be limited to loading only small networks. We recommend downloading and installing a 64 bit JRE.
- We currently recommend only Java 8.

For additional information, select the Release Notes button on the Cytoscape web site.

Install Cytoscape

Downloading and Installing

There are a number of options for downloading and installing Cytoscape. See the download page at the http://cytoscape.org website for all options.

- Automatic installation packages exist for Windows, Mac OS X, and Linux platforms – best for most users.
- You can install Cytoscape from a compressed archive distribution.
- You can build Cytoscape from the source code.
- You can check out the latest and greatest software from our Git repository (https://github.com/cytoscape/cytoscape).

Cytoscape installations (regardless of platform) containing the following files and directories:

Cytoscape files and directories

DIRECTORY / FILE	DESCRIPTION
P/CYTOSCAPE_V3.3.0	Cytoscape program files, startup scripts, and default location for session files
P/CYTOSCAPE_V3.3.0/CYTOSCAPE.VMOPTIONS	Cytoscape memory configuration settings
P/CYTOSCAPE_V3.3.0/SAMPLEDATA	Preset networks as described in the embedded README.txt file
P/CYTOSCAPE_V3.3.0/FRAMEWORK	Cytoscape program files
P/CYTOSCAPE_V3.3.0/APPS	Cytoscape core app program files
U/CYTOSCAPECONFIGURATION	Cytoscape properties and program cache files
U/CYTOSCAPECONFIGURATION/CYTOSCAPE3.PROPS	Cytoscape configuration settings

The `p/` directory signifies the program directory, which varies from platform to platform. For Cytoscape to work properly, all files should be left in the directory in which they were unpacked. The core Cytoscape application assumes this directory structure when looking for the various libraries needed to run the application.

The `u/` directory signifies the user's home directory, which varies from user to user and from platform to platform. To change the user home directory from the default, one can set the Java environment variable `user.home` to the desired directory – this is useful when Cytoscape is installed on a workstation, but the home directory is stored on a central file server. `user.home` can be set by adding the following option to the Cytoscape.vmoptions file or the _JAVA_OPTIONS environment variable, substituting the desired path as appropriate:

-Duser.home=/path/to/desired/home

Your operating system may have other mechanisms for setting environment variables – see your operating system documentation for further details.

A quick note on upgrading your Cytoscape installation

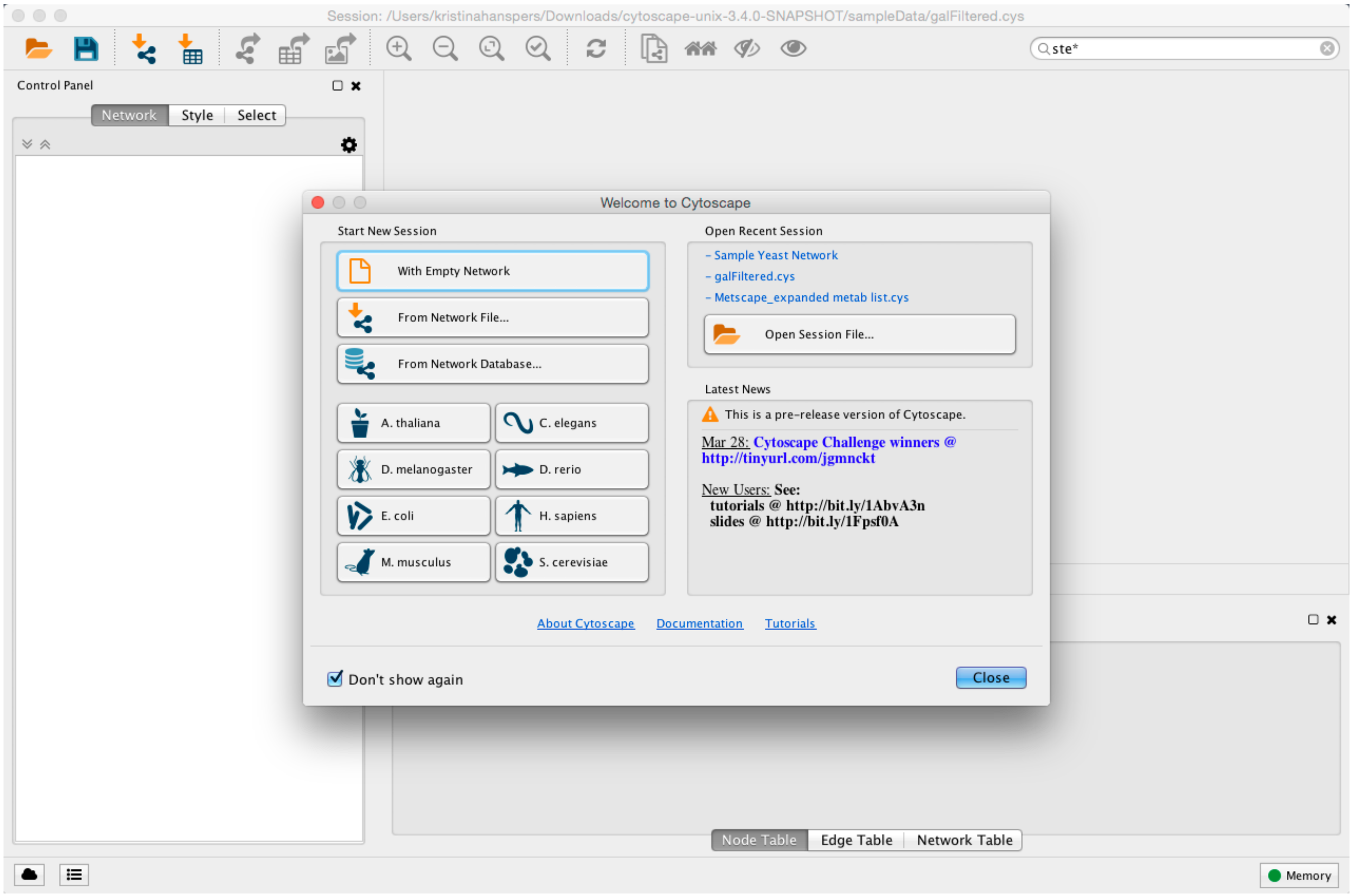
If you have a previous Cytoscape installation you have two options:

1. Starting with a clean slate. For this you should delete your previous installation directory and the `CytoscapeConfiguration` directory (see below for the location of this directory).
2. Just keep what you have and simply pick a distinct, new directory for installation. In the unlikely event that you should encounter any problem, delete the `.props` files in your `CytoscapeConfiguration` directory. If that doesn't help try deleting the `CytoscapeConfiguration` directory. This latter step will cause you to lose all of the apps that you have installed via the App Store, so only do that if you are having problems or if you don't mind reinstalling your apps. The core apps will not be affected by this step.

Launch the Application

As with any application, launch Cytoscape by double-clicking on the icon created by the installer, by running `cytoscape.sh` from the command line (Linux or Mac OS X) or by double-clickinging `cytoscape.bat` or the Program Launch icon (Windows).

After launching Cytoscape a window will appear that looks like this:



This is a pre-release version of Cytoscape.

Mar 28: Cytoscape Challenge winners @ http://tinyurl.com/jgmnckt

New Users: See:
tutorials @ http://bit.ly/1AbvA3n
slides @ http://bit.ly/1Fpsf0A

If your Cytoscape window does not resemble this, further configuration may be required. Consult the ***Release Notes*** *available on the http://cytoscape.org website.*

Note on Memory Consumption

For most regular users, Cytoscape will estimate and reserve the proper amount of memory. An incorrect estimate may result in Cytoscape hanging at startup or Cytoscape unable to load your network. Unless Cytoscape fails to start or open your network, it has likely estimated the available memory correctly, and you can continue to the Quick Tour. If Cytoscape misjudges the memory size or can't allocate enough memory, it could be that you're running with a 32 bit JRE and could get better results by installing a 64 bit JRE – see the Install Java section above.

When Cytoscape starts, it displays the current memory usage in the lower right corner of the main interface. You can click on the **Memory** button at any time to access an option to **Free Unused Memory**. While most users won't need to use this option, it can be useful for users who have multiple large networks loaded.

Overall Memory Size for Cytoscape

By default, Cytoscape uses an estimate for initial and maximum memory allocation based on your operating system, system architecture (32 or 64 bit), and installed memory. You can change Cytoscape's initial and/or maximum memory size by editing the Cytoscape.vmoptions file, which resides in the same directory as the Cytoscape executable. The file contains one option per line, with each line terminated by a linefeed, and an extra linefeed at the end of the file. Note that for the MacOS platform, the situation is slightly different – if you are launching Cytoscape by clicking on the Cytoscape icon, you must edit the .../Cytoscape.app/Contents/vmoptions.txt file instead. To access this in Finder, you will need to right-click the Cytoscape app icon and select "Show Package Contents", which will display the Contents subdirectory that contains vmoptions.txt.

For example, if you want Cytoscape to initially allocate 2GB of memory and use up to a maximum of 4GB, edit the Cytoscape.vmoptions file to contain the following lines (... do not forget the linefeed at the end of each line, and an extra linefeed at the end of the file!):

-Xms2GB

-Xmx4GB

Stack Size

There is one more option related to memory allocation. Some of the functions in Cytoscape use larger stack space (a temporary memory for some operations, such as layout). Since this value is set independently from the values above, sometimes layout algorithms fail due to an *out of memory* error. To avoid this, you can set a larger heap size for Cytoscape tasks by using the ***taskStackSize*** option in the `cytoscape3.props` file (located in the `CytoscapeConfiguration` directory). This can be edited within Cytoscape using the Preferences Editor (**Edit → Preferences → Properties...***) - look for taskStackSize. The value should be specified in bytes.

Command Line Arguments

Cytoscape recognizes a number of optional command line arguments, including run-time specification of network files, node and edge data files, and session files. This is the output generated when Cytoscape is executed with the "-h" or "–help" flag:

```
usage: cytoscape.{sh|bat} [OPTIONS]
 -h,--help             Print this message.
 -v,--version          Print the version number.
 -s,--session <file>   Load a cytoscape session (.cys) file.
 -N,--network <file>   Load a network file (any format).
 -P,--props <file>     Load cytoscape properties file (Java properties
                       format) or individual property: -P name=value.
 -V,--vizmap <file>    Load vizmap properties file (Cytoscape VizMap
                       format).
 -S,--script <file>    Execute commands from script file.
 -R,--rest <port>      Start a rest service.
```

Any file specified for an option may be specified as either a path or as a URL. For example you can specify a network as a file (assuming that myNet.sif exists in the current working directory): `cytoscape.sh -N myNet.sif`.

Note: if there are spaces in the file path, be sure to put quotes around it: `cytoscape.bat -N "C:\Program Files\Cytoscape\sampleData\galFiltered.sif"`.

Or you can specify a network as a URL: `cytoscape.sh -N http://example.com/myNet.sif`.

Command Line Arguments

ARGUMENT	DESCRIPTION
-h,--help	This flag generates the help output you see above and exits.
-v,--version	This flag prints the version number of Cytoscape and exits.
-s,--session <file>	This option specifies a session file to be loaded. Since only one session file can be loaded at a given time, this option may only specified once on a given command line. The option expects a `.cys` Cytoscape session file. It is customary, although not necessary, for session file names to contain the .cys extension.
-N,--network <file>	This option is used to load all types of network files. SIF, GML, and XGMML files can all be loaded using the -N option. You can specify as many networks as desired on a single command line.
-P,--props <file>	This option specifies Cytoscape properties. Properties can be specified either as a properties file (in Java's standard properties format), or as individual properties. To specify individual properties, you must specify the property name followed by the property value where the name and value are separated by the '=' sign. For example to specify the defaultSpeciesName: `cytoscape.sh -P defaultSpeciesName=Human`. If you would like to include spaces in your property, simply enclose the name and value in quotation marks: `cytoscape.sh -P "defaultSpeciesName=Homo Sapiens"`. The property option subsumes previous options -noCanonicalization, -species, and -

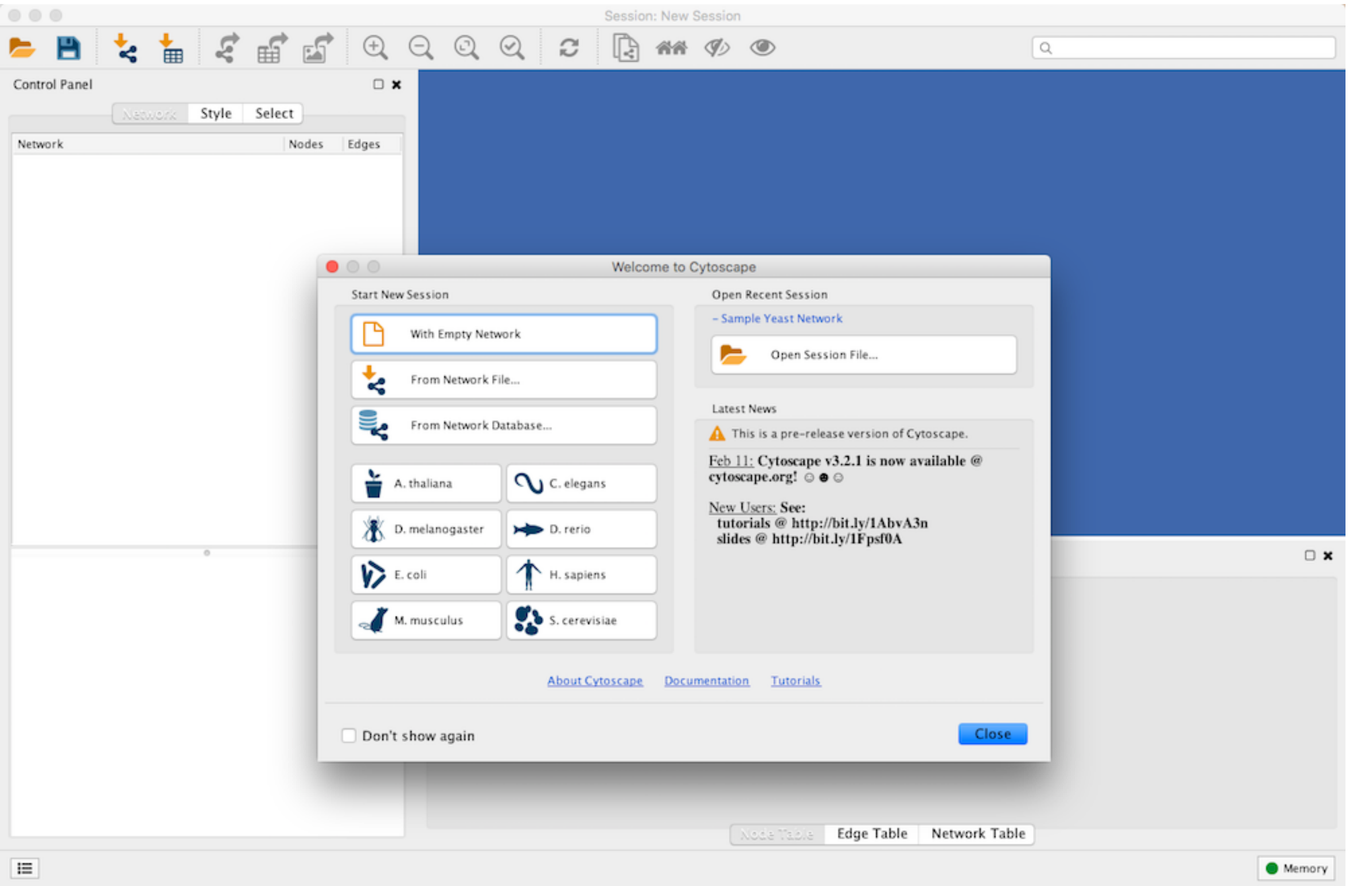
	bioDataServer. Now it would look like: `cytoscape.sh` `-P defaultSpeciesName=Human` `-P noCanonicalization=true` `-P bioDataServer=myServer`.
`-V,--vizmap <file>`	This option specifies a Style file.
`-S,--script <file>`	This option executes commands from a specifed Cytoscape script file.
`-R,--rest <port>`	This option starts a Cytoscape REST service on the specified port.

All options described above (except for starting a REST service) can be accessed from the menu once Cytoscape is running.

Quick Tour of Cytoscape

Welcome Screen

When you start Cytoscape, you can access basic functions from the **Welcome Screen**:



This is a pre-release version of Cytoscape.

Feb 11: Cytoscape v3.2.1 is now available @ cytoscape.org!

New Users: See:
tutorials @ http://bit.ly/1AbvA3n
slides @ http://bit.ly/1Fpsf0A

The **Welcome Screen** is designed to access commonly used features of Cytoscape including:

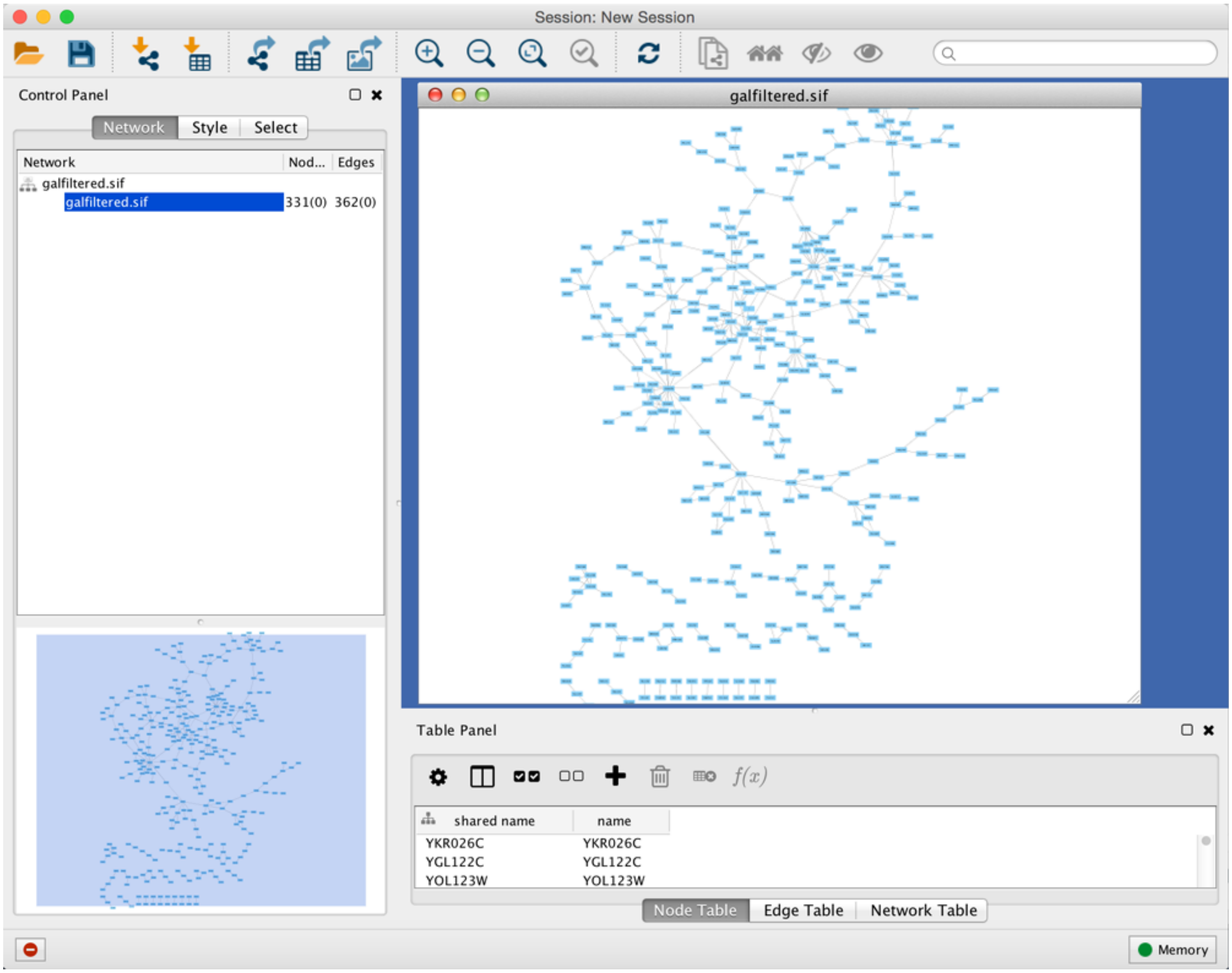
- Create new network
- Import network
  - From file

    - From public database
- Import interactome for model organisms
- Open recently used session file

Also, a news panel always display latest information for users. For information on user privacy, see the **Cytoscape Privacy Policy**.

Basic Features

When a network is loaded, Cytoscape will look similar to the image below:



Network	Nod...	Edges
galfiltered.sif		
galfiltered.sif	331(0)	362(0)

shared name	name
YKR026C	YKR026C
YGL122C	YGL122C
YOL123W	YOL123W

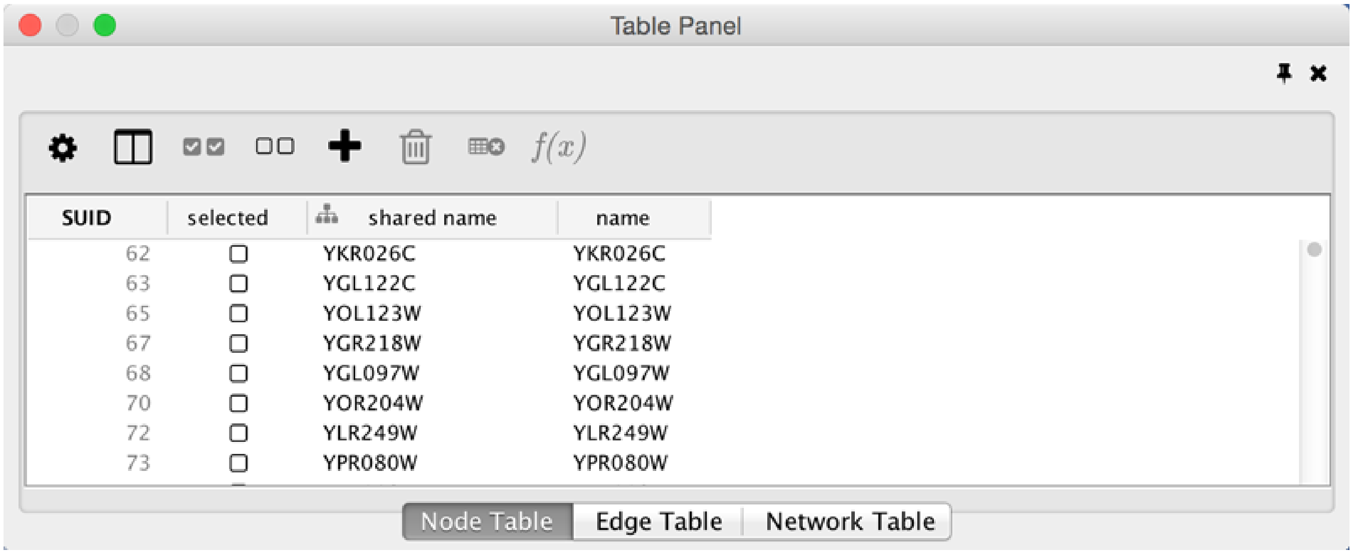
Most functionalities are self-explanatory, but we'll go through a concise explanation for clarity. The main window here has several components:

- The Menu Bar at the top (see below for more information about each menu).
- The Tool Bar, which contains icons for commonly used functions. These functions are also available via the menus. Hover the mouse pointer over an icon and wait momentarily for a description to appear as a tooltip.

- The Network Panel (Network tab of the Control Panel, top left). This contains an optional network overview pane (shown at the bottom left).
- The main Network View Window, which displays the network.
- The Table Panel (bottom right panel), which displays columns of selected nodes and edges and enables you to modify the values of column data.

The Network Panel and Table Panel are dockable tabbed Panels. You can undock any of these panels by clicking on the **Float Window** control □ in the upper-right corner of the CytoPanel. This is useful when you want assign the network panel as much screen space as possible. To dock the window again, click the **Dock Window** icon 📌. Clicking the **Hide Panel** icon ✖ will hide the panel; this can be shown again by choosing **View** → **Show** and selecting the relevant panel.

If you click this control, for example on the Table Panel, you will now have two Cytoscape windows, the main window, and a new window labeled Table Panel, similar to the one shown below. A popup will be displayed when you put the mouse pointer on a cell.

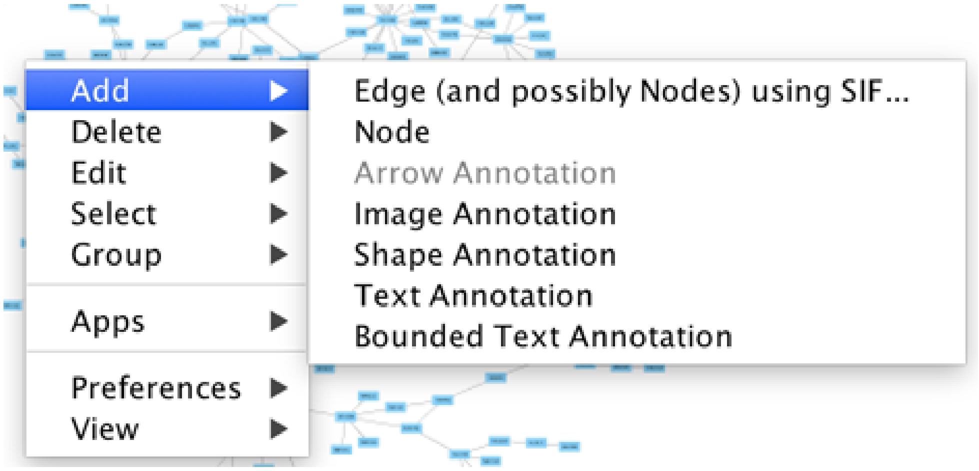


SUID	selected	shared name	name
62	□	YKR026C	YKR026C
63	□	YGL122C	YGL122C
65	□	YOL123W	YOL123W
67	□	YGR218W	YGR218W
68	□	YGL097W	YGL097W
70	□	YOR204W	YOR204W
72	□	YLR249W	YLR249W
73	□	YPR080W	YPR080W

Note that Table Panel now has a Dock Window control. If you click this control, the window will dock onto the main window. For more information on the panels in Cytoscape, see the Panels section.

Network Editing

Cytoscape also has an edit functionality that enables you to build and modify networks interactively within the network canvas. To edit a network, just right-click on the open space of network window, select the menu item **Add** → **Node**, a new node will be added to the canvas. To add an edge, right click on a node, choose the menu item **Edit** → **Add Edge**. Then select the target node, a new edge will be added between the two nodes. In a similar way annotation objects can be added; pictures, shapes or textboxes; much like in MS PowerPoint or similar software. Detailed information on network editing can be found in the Editing Networks section.

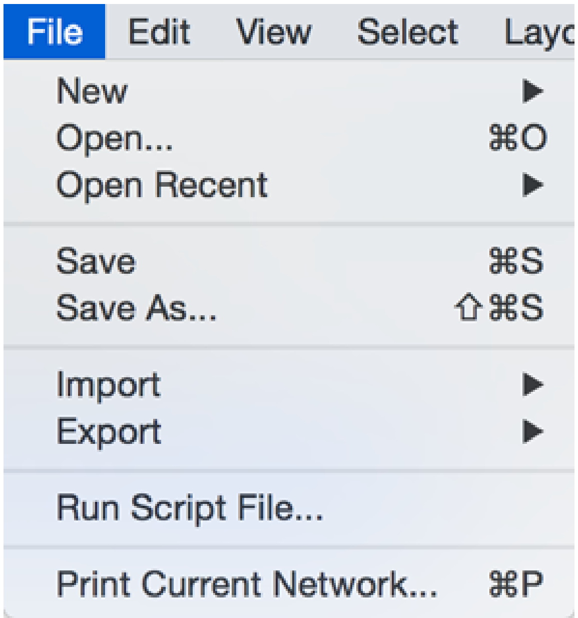


The Menus

File

The File menu contains most basic file functionality: **File** → **Open** for opening a Cytoscape session file; **File** → **New** for creating a new network, either blank for editing, or from an existing network; **File** → **Save** for saving a session file; **File** → **Import** for importing data such as networks and data; and **File** → **Export** for exporting data and images. **File** → **Export** → **Network View as Graphics** lets you export the network in either JPEG, PDF, PNG, Post Script or SVG format.

File → **Recent Session** will list recently opened session files for quick access. **File** → **Run** allows you to specify a Cytoscape script file to run, and **File** → **Print Current Network...** allows printing.



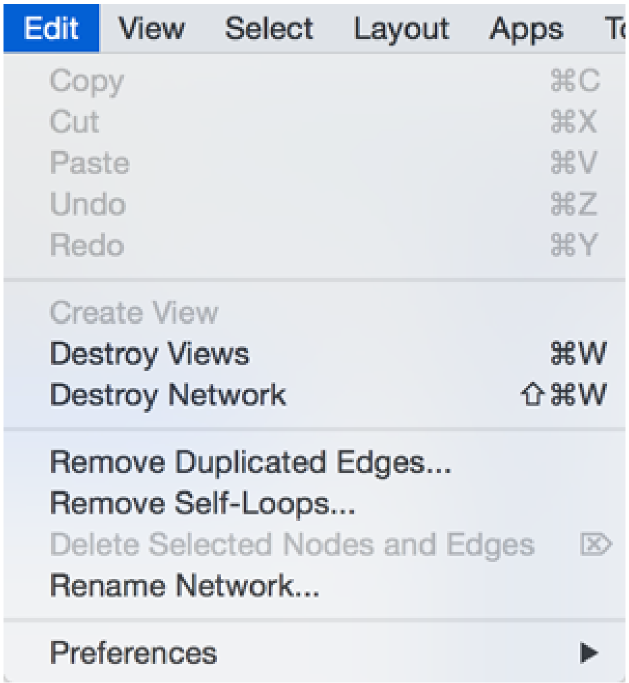
Edit

The **Edit** menu contains **Cut**, **Copy** and **Paste** functions, as well as **Undo** and **Redo** functions which undo and redo edits made in the **Table Panel**, the **Network Editor** and to layout.

There are also options for creating and destroying views (graphical representations of a network) and networks (the raw network data - not yet visualized), as well as an option for deleting selected nodes and edges from the current network. All deleted nodes and edges can be restored to the network via **Edit** → **Undo**.

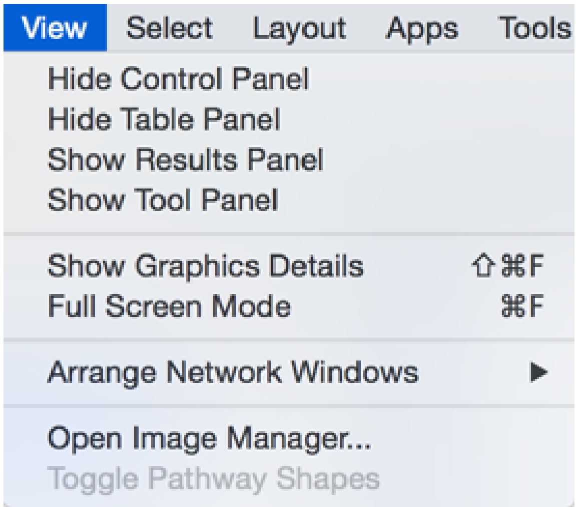
There are also other editing options; **Remove Duplicated Edges** will delete edges that are complete duplicates, keeping one edge, **Remove Self-Loops** removes edges that have the same source and target node, and **Delete Selected Nodes and Edges...** deletes a selected subset of the network. **Rename Network...** allows you to rename the currently selected network.

Editing preferences for properties and apps is found under **Edit** → **Preferences** → **Properties...**. More details on how to edit preferences can be found here.



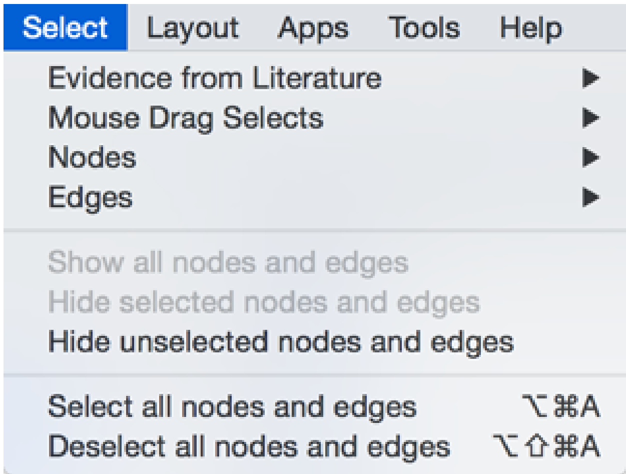
View

The **View** menu allows you to display or hide the **Control Panel**, **Table Panel**, **Tool Panel** and the **Result Panel**. It also provides the control of other view-related functionality.



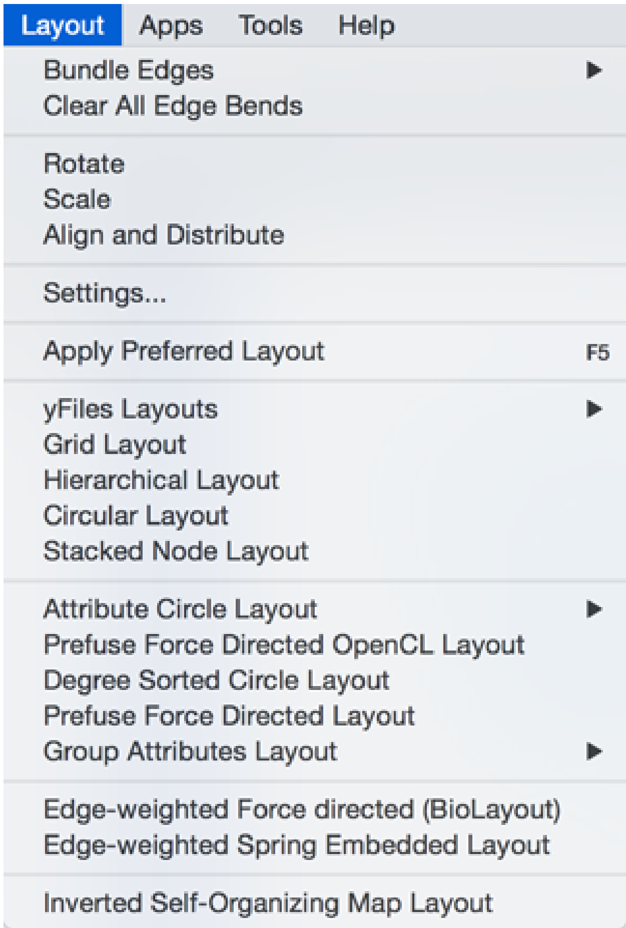
Select

The **Select** menu contains different options for selecting nodes and edges.


Select
Layout
Apps
Tools
Help
Evidence from Literature ▶
Mouse Drag Selects ▶
Nodes ▶
Edges ▶
Show all nodes and edges
Hide selected nodes and edges
Hide unselected nodes and edges
Select all nodes and edges ⌥⌘A
Deselect all nodes and edges ⌥⇧⌘A

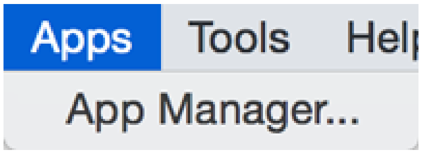
Layout

The **Layout** menu has an array of features for visually organizing the network. The features in the top portion of the network (**Bundle Edges**, **Clear Edge Bends**, **Rotate**, **Scale**, **Align and Distribute**) are tools for manipulating the network visualization. The bottom section of the menu lists a variety of layout algorithms which automatically lay a network out.


Layout
Apps
Tools
Help
Bundle Edges ▶
Clear All Edge Bends
Rotate
Scale
Align and Distribute
Settings...
Apply Preferred Layout F5
yFiles Layouts ▶
Grid Layout
Hierarchical Layout
Circular Layout
Stacked Node Layout
Attribute Circle Layout ▶
Prefuse Force Directed OpenCL Layout
Degree Sorted Circle Layout
Prefuse Force Directed Layout
Group Attributes Layout ▶
Edge-weighted Force directed (BioLayout)
Edge-weighted Spring Embedded Layout
Inverted Self-Organizing Map Layout

Apps

The **Apps** menu gives you access to the **App Manager** (**Apps** → **App Manager**) for managing (install/update/delete) your apps and may have options added by apps that have been installed. Depending on which apps are loaded, the apps that you see may be different than what appear here. The below picture shows a Cytoscape installation without installed apps.

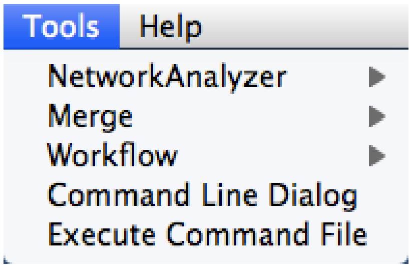


Note: A list of available Cytoscape apps with descriptions is available online at: http://apps.cytoscape.org

In previous versions of Cytoscape, apps were called plugins and served a similar function.

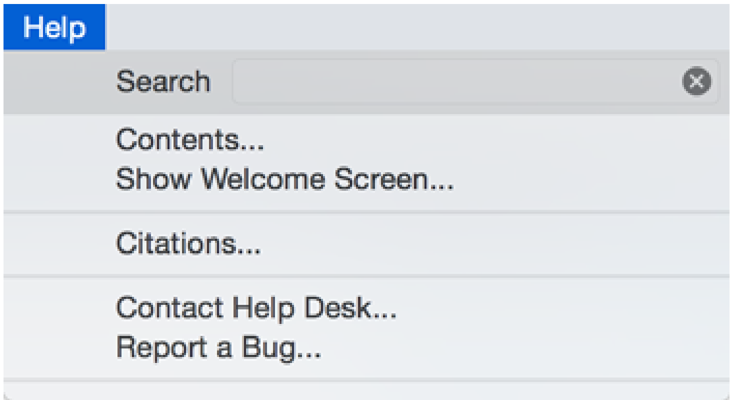
Tools

The **Tools** menu contains advanced features like the **Command Line Dialog**, **Network Analyzer**, **Network Merge** and **Workflow**.

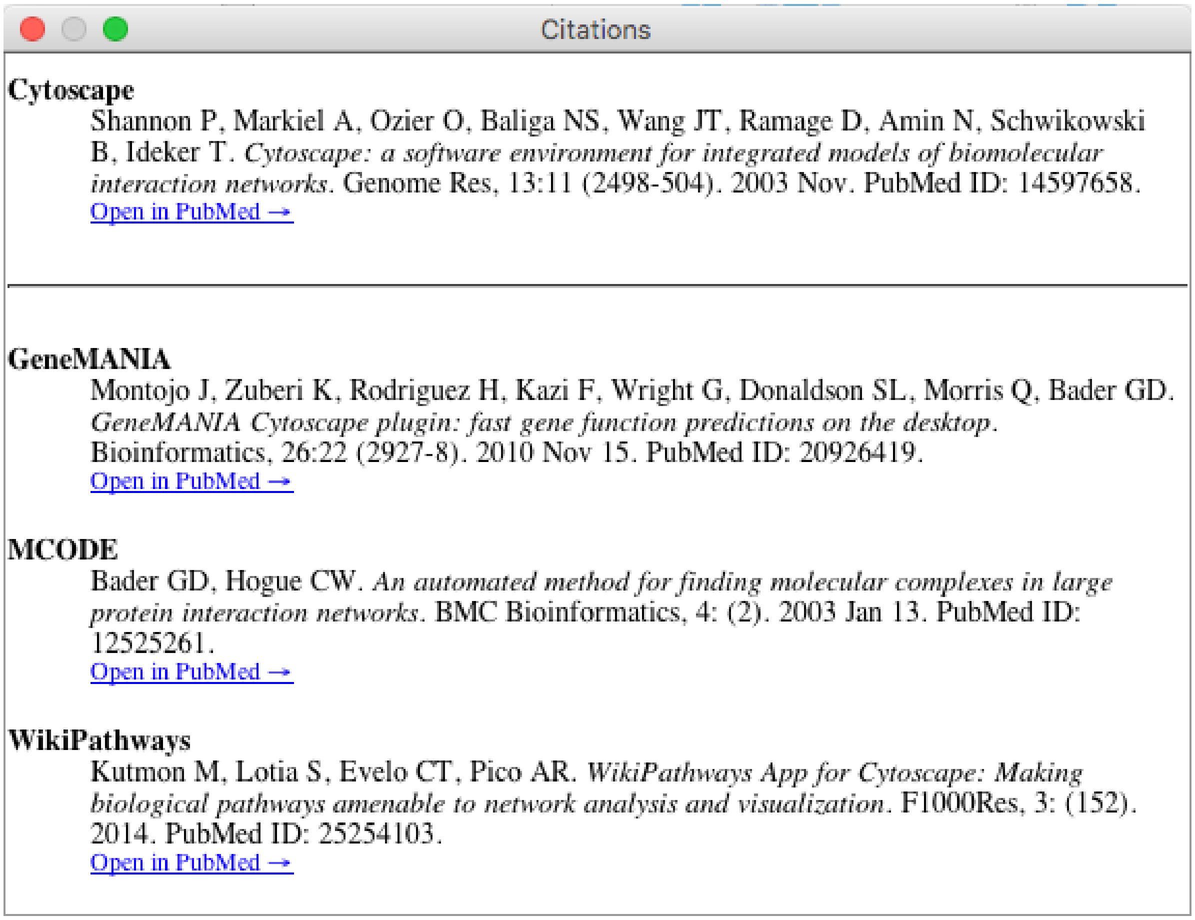


Help

The **Help** menu allows you to launch the online help viewer and browse the table of contents for this manual (**Contents**).



The **Citations** option displays the main literature citation for Cytoscape, as well as a list of literature citations for installed apps. The list will be different depending on the set of apps you have installed.



Cytoscape

Shannon P, Markiel A, Ozier O, Baliga NS, Wang JT, Ramage D, Amin N, Schwikowski B, Ideker T. *Cytoscape: a software environment for integrated models of biomolecular interaction networks*. Genome Res, 13:11 (2498-504). 2003 Nov. PubMed ID: 14597658.
Open in PubMed →

GeneMANIA

Montojo J, Zuberi K, Rodriguez H, Kazi F, Wright G, Donaldson SL, Morris Q, Bader GD. *GeneMANIA Cytoscape plugin: fast gene function predictions on the desktop*. Bioinformatics, 26:22 (2927-8). 2010 Nov 15. PubMed ID: 20926419.
Open in PubMed →

MCODE

Bader GD, Hogue CW. *An automated method for finding molecular complexes in large protein interaction networks*. BMC Bioinformatics, 4: (2). 2003 Jan 13. PubMed ID: 12525261.
Open in PubMed →

WikiPathways

Kutmon M, Lotia S, Evelo CT, Pico AR. *WikiPathways App for Cytoscape: Making biological pathways amenable to network analysis and visualization*. F1000Res, 3: (152). 2014. PubMed ID: 25254103.
Open in PubMed →

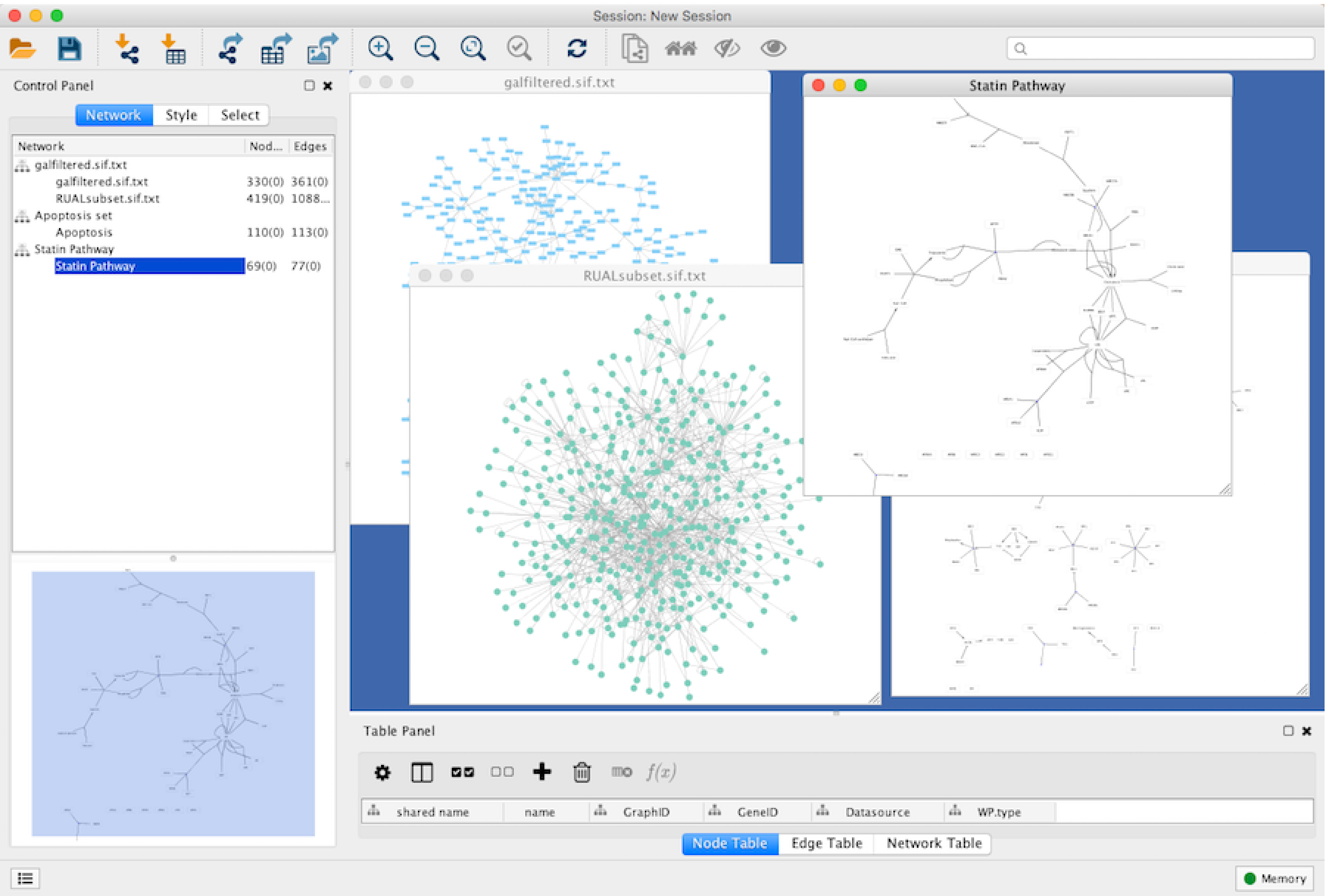
The **Update Organism Preset Networks** option updates the preset networks available on the **Welcome** screen to the latest version.

The **Help** menu also allows you to connect directly to Cytoscape Help Desk and the Bug Report interface.

Network Management

Cytoscape allows multiple networks to be loaded at a time, either with or without a view. A network stores all the nodes and edges that are loaded by the user and a view displays them.

An example where a number of networks have been loaded is shown below:



Network	Nod...	Edges
galfiltered.sif.txt		
galfiltered.sif.txt	330(0)	361(0)
RUALsubset.sif.txt	419(0)	1088...
Apoptosis set		
Apoptosis	110(0)	113(0)
Statin Pathway		
Statin Pathway	69(0)	77(0)

shared name	name	GraphID	GeneID	Datasource	WP.type

The network manager (in Control Panel) shows the networks that are loaded. Clicking on a network here will make that view active in the main window, if the view exists. Each network has a name and size (number of nodes and edges), which are shown in the network manager. If a network is loaded from a file, the network name is the name of the file.

Some networks are very large (thousands of nodes and edges) and can take a long time to display. For this reason, a network in Cytoscape may not contain a "view". Networks that have a view are in normal black font and networks that don't have a view are highlighted in red. You can create or destroy a view for a network by right-clicking the network name in the network manager or by choosing the appropriate option in the **Edit** menu. You can also destroy previously loaded networks this way.

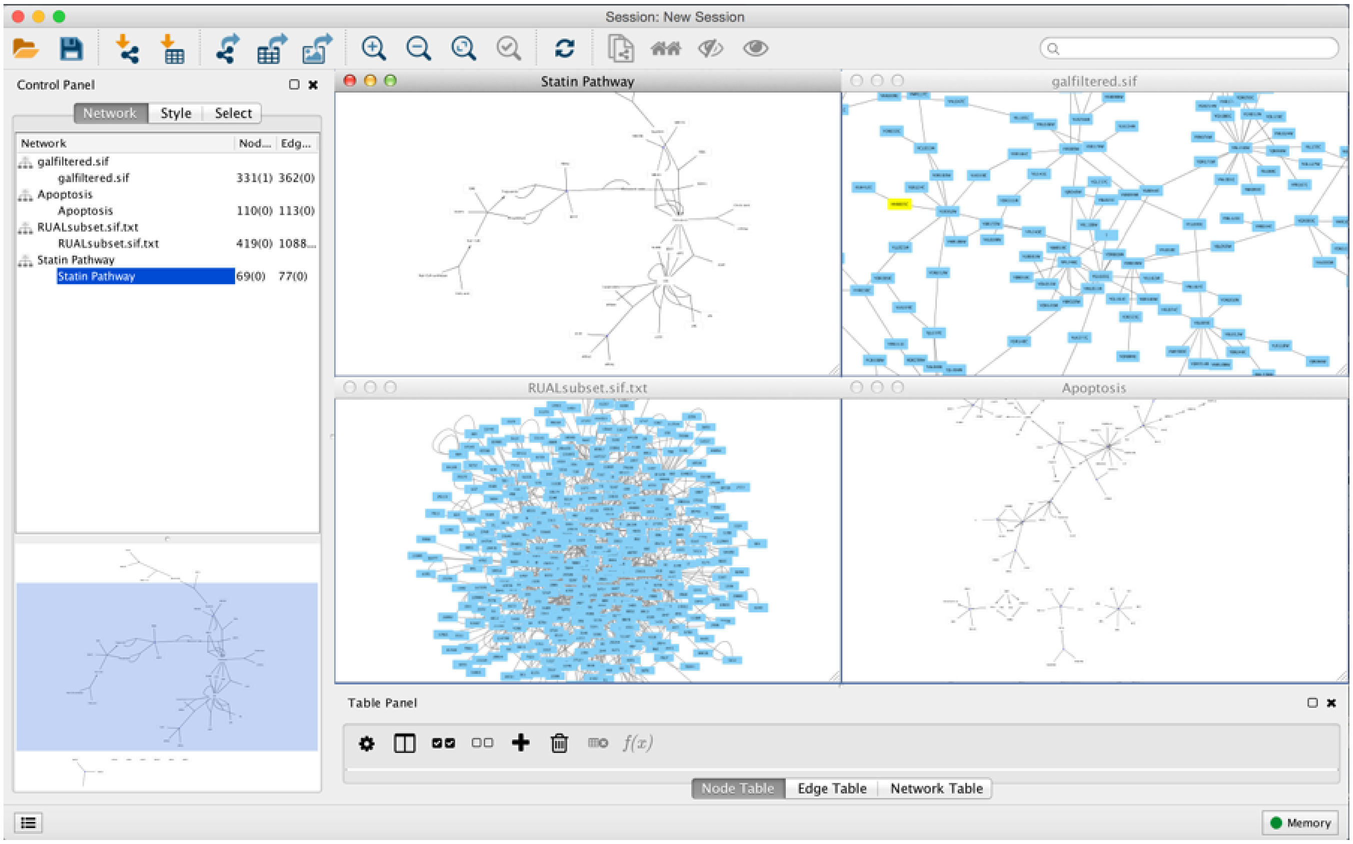
Certain operations in Cytoscape will create new networks. If a new network is created from an old network, for example by selecting a set of nodes in one network and copying these nodes to a new network (via the **File** → **New** → **Network** option), it will be shown immediately follows the network that it was derived from.

Network views can also be detached (undocked) from the main Cytoscape window. When detached, a view window can be dragged to another monitor, resized, maximized and minimized by using the normal window controls for your operating system. Notice, however, that closing a view window does not destroy it, but simply reattaches it to the Cytoscape window.

Arrange Network Windows

When you have detached network view windows, you can arrange them by selecting one of these options under **View** → **Arrange Network Windows**:

Grid

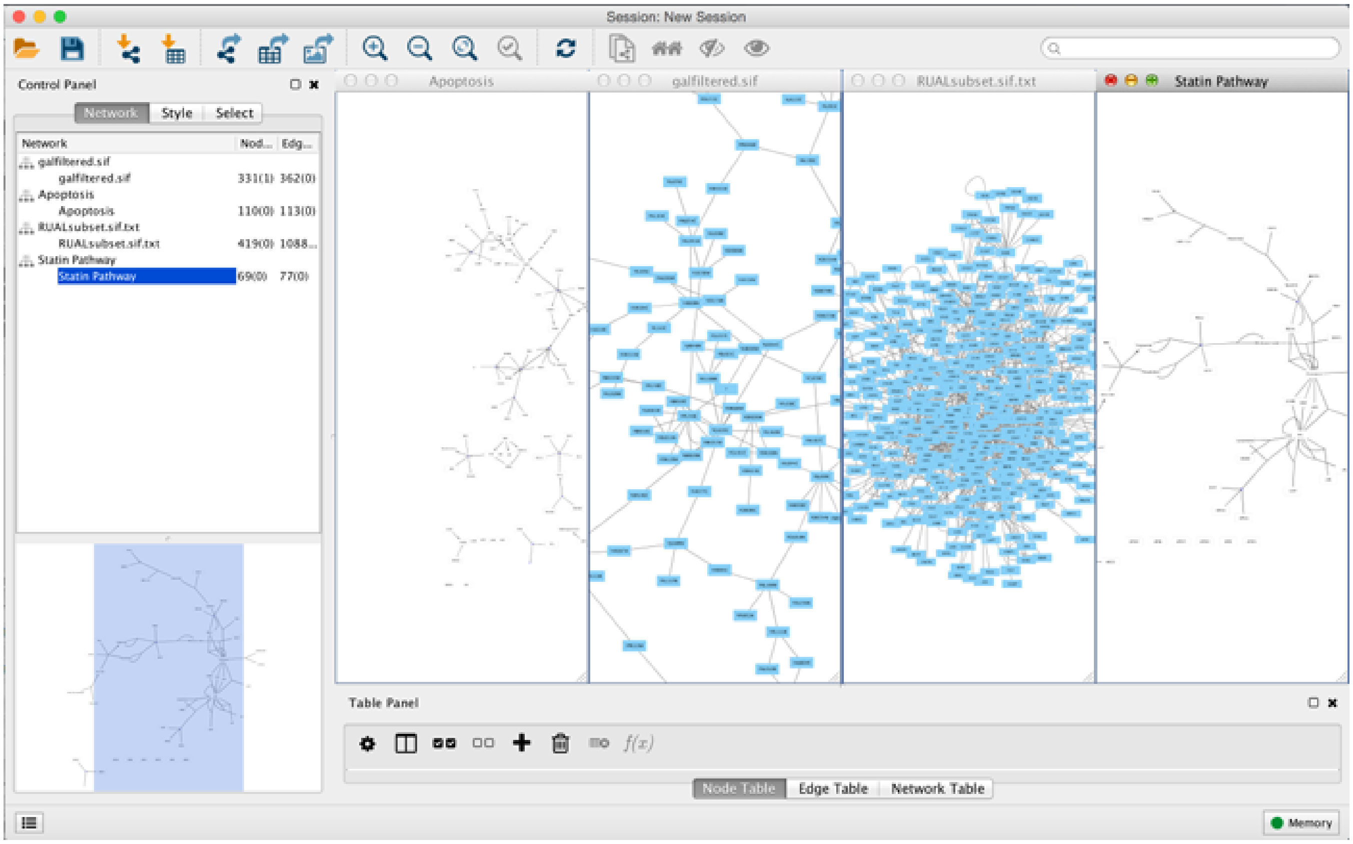


Network	Nod...	Edg...
galfiltered.sif		
galfiltered.sif	331(1)	362(0)
Apoptosis		
Apoptosis	110(0)	113(0)
RUALsubset.sif.txt		
RUALsubset.sif.txt	419(0)	1088...
Statin Pathway		
Statin Pathway	69(0)	77(0)

Cascade

Vertical Stack

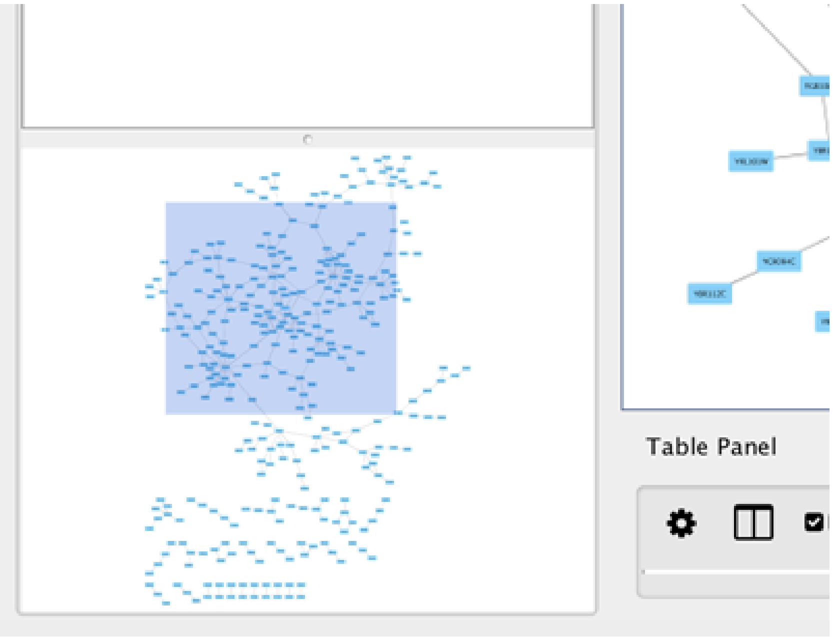
Side by Side



Network	Nod...	Edg...
galfiltered.sif		
galfiltered.sif	331(1)	362(0)
Apoptosis		
Apoptosis	110(0)	113(0)
RUALsubset.sif.txt		
RUALsubset.sif.txt	419(0)	1088...
Statin Pathway		
Statin Pathway	69(0)	77(0)

The Network Overview Window

The network overview window shows an overview (or "bird's eye view") of the network. It can be used to navigate around a large network view. The blue rectangle indicates the portion of the network currently displayed in the network view window, and it can be dragged with the mouse to view other portions of the network. Zooming in will cause the rectangle to appear smaller and vice versa.



Creating Networks

There are 4 different ways of creating networks in Cytoscape:

1. Importing pre-existing, fixed-format network files.
2. Importing pre-existing, unformatted text or Excel files.
3. Importing data from from public databases.
4. Creating an empty network and manually adding nodes and edges.

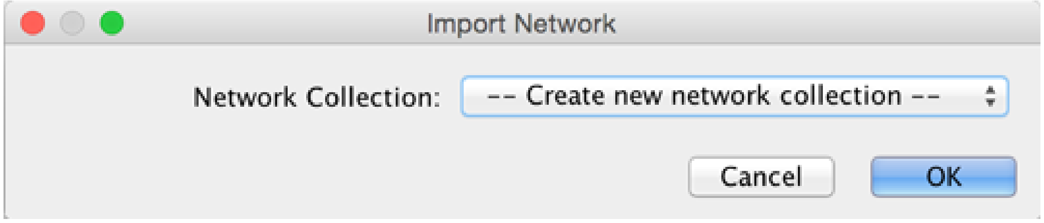
Import Fixed-Format Network Files

Network files can be specified in any of the formats described in the Supported Network Formats section. Networks are imported into Cytoscape through the **File** → **Import** → **Network menu**. The network file can either be located directly on the local computer, or found on a remote computer (in which case it will be referenced with a URL).

Load Networks from Local Computer

In order to load a network from a local file you can select **File** → **Import** → **Network** → **File...** or click on on the tool bar. Choose the correct file in the file chooser dialog and press Open. Some sample network files of different types have been included in the sampleData folder in Cytoscape.

After you choose a network file, another dialog will pop up. Here, you can choose either to create a new network collection for the new network, or load the new network into an existing network collection. When you choose the latter, make sure to choose the right mapping column to map the new network to the existing network collection.



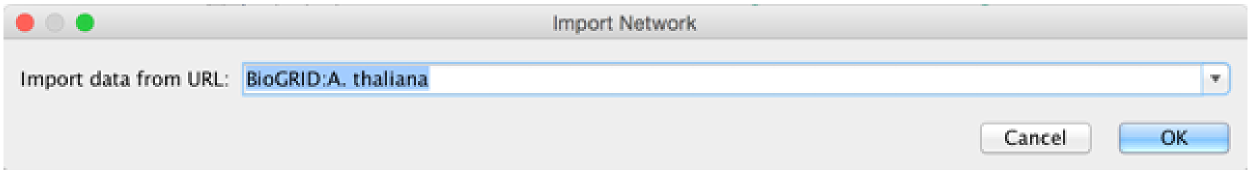
Import Network

Network Collection: -- Create new network collection --

Network files in SIF, GML, and XGMML formats may also be loaded directly from the command line using the -N option.

Load Networks from a Remote Computer (URL import)

To load a network from a remote file, you can select **File** → **Import** → **Network** → **URL...**. In the import network dialog, insert the appropriate URL, either manually or using URL bookmarks. Bookmarked URLs can be accessed by clicking on the arrow to the right of the text field (see the Bookmark Manager in Preferences for more details on bookmarks). Also, you can drag and drop links from a web browser to the URL text box. Once a URL has been specified, click on the OK button to load the network.

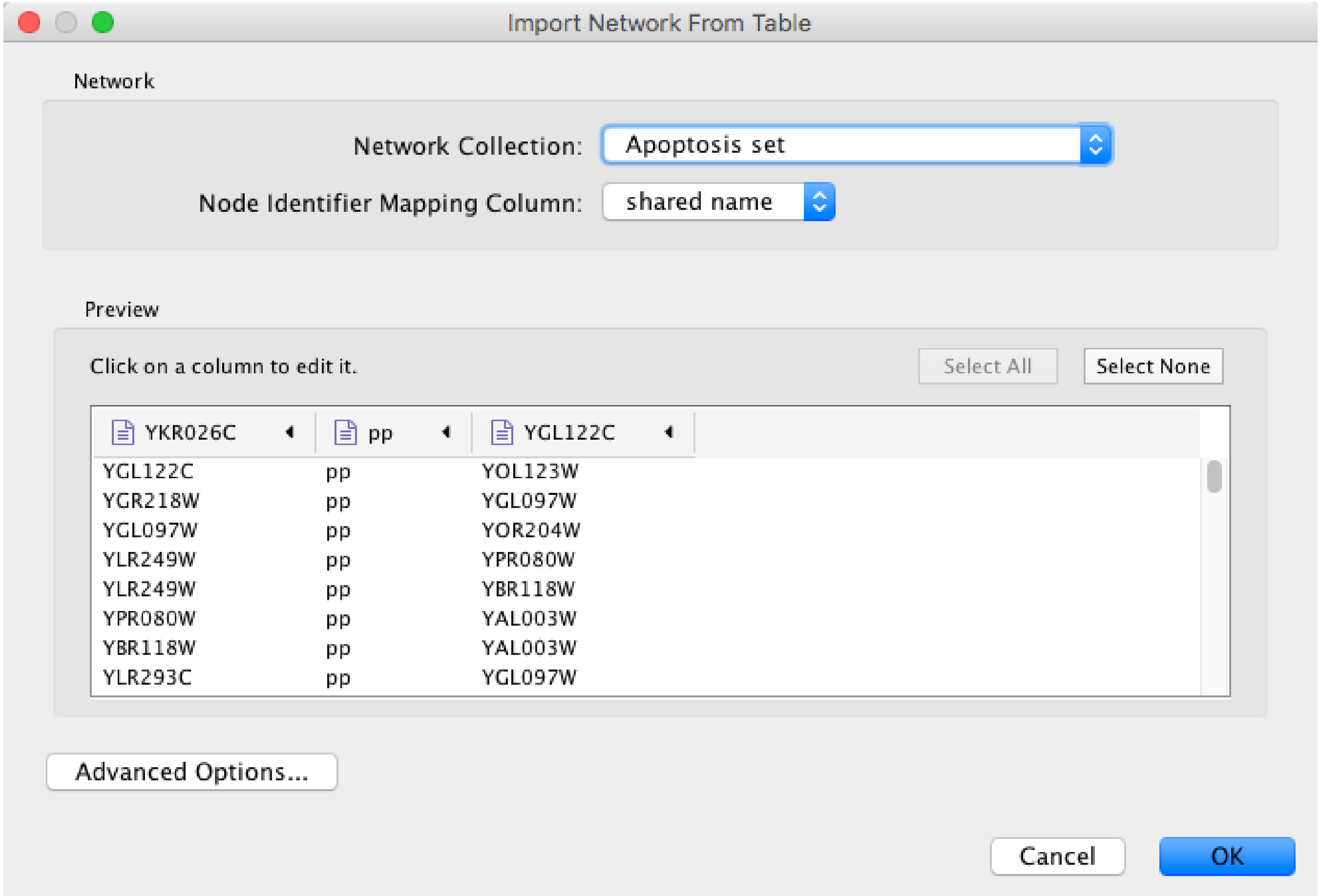


Import data from URL: BioGRID:A. thaliana

Another issue for network import is the presence of firewalls, which can affect which files are accessible to a computer. To work around this problem, Cytoscape supports the use of proxy servers. To configure a proxy server, go to **Edit** → **Preferences** → **Proxy Settings...**. This is further described in the **Preferences** section.

Import Networks from Unformatted Table Files

Cytoscape supports the import of networks from delimited text files and Excel workbooks using **File** → **Import** → **Network** → **File...**. An interactive GUI allows users to specify parsing options for specified files. The screen provides a preview that shows how the file will be parsed given the current configuration. As the configuration changes, the preview updates automatically. In addition to specifying how the file will be parsed, the user must also choose the columns that represent the source and target nodes as well as an optional edge interaction type.



Network Collection: Apoptosis set

Node Identifier Mapping Column: shared name

YKR026C	pp	YGL122C
YGL122C	pp	YOL123W
YGR218W	pp	YGL097W
YGL097W	pp	YOR204W
YLR249W	pp	YPR080W
YLR249W	pp	YBR118W
YPR080W	pp	YAL003W
YBR118W	pp	YAL003W
YLR293C	pp	YGL097W

Supported Files

The **Import Network from Table** function supports delimited text files and Microsoft Excel Workbooks. For Excel Workbooks with multiple sheets, one sheet can be selected for import at a time. The following is a sample table file:

Sample Network in Table

SOURCE	TARGET	INTERACTION	BOOLEAN DATA	STRING DATA	FLOATING POINT DATA
YJR022W	YNR053C	pp	TRUE	abcd12371	1.2344543
YER116C	YDL013W	pp	TRUE	abcd12372	1.2344543
YNL307C	YAL038W	pp	FALSE	abcd12373	1.2344543
YNL216W	YCR012W	pd	TRUE	abcd12374	1.2344543
YNL216W	YGR254W	pd	TRUE	abcd12375	1.2344543

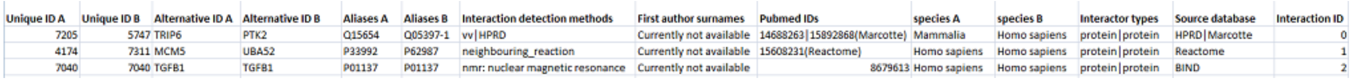
The network table files should contain at least two columns for creating network with edges. If the file has only one column, the created network will not contain any edges. The interaction type is optional in this format. Therefore, a minimal network table looks like the following:

Minimal Network Table

SOURCE	TARGET

YJR022W	YNR053C
YER116C	YDL013W
YNL307C	YAL038W
YNL216W	YCR012W
YNL216W	YGR254W

One row in a network table file represents an edge and its edge data columns. This means that a network file is considered a combination of network data and edge column data. A table may contain columns that aren't meant to be edge data. In this case, you can choose not to import those columns by clicking on the column header in the preview window. This function is useful when importing a data table like the following (1):



<table>
<tr><th>Unique ID A</th><th>Unique ID B</th><th>Alternative ID A</th><th>Alternative ID B</th><th>Aliases A</th><th>Aliases B</th><th>Interaction detection methods</th><th>First author surnames</th><th>Pubmed IDs</th><th>species A</th><th>species B</th><th>Interactor types</th><th>Source database</th><th>Interaction ID</th></tr>
<tr><td>7205</td><td>5747</td><td>TRIP6</td><td>PTK2</td><td>Q15654</td><td>Q05397-1</td><td>vv|HPRD</td><td>Currently not available</td><td>14688263|15892868(Marcotte)</td><td>Mammalia</td><td>Homo sapiens</td><td>protein|protein</td><td>HPRD|Marcotte</td><td>0</td></tr>
<tr><td>4174</td><td>7311</td><td>MCM5</td><td>UBA52</td><td>P33992</td><td>P62987</td><td>neighbouring_reaction</td><td>Currently not available</td><td>15608231(Reactome)</td><td>Homo sapiens</td><td>Homo sapiens</td><td>protein|protein</td><td>Reactome</td><td>1</td></tr>
<tr><td>7040</td><td>7040</td><td>TGFB1</td><td>TGFB1</td><td>P01137</td><td>P01137</td><td>nmr: nuclear magnetic resonance</td><td>Currently not available</td><td>8679613</td><td>Homo sapiens</td><td>Homo sapiens</td><td>protein|protein</td><td>BIND</td><td>2</td></tr>
</table>

This data file is a tab-delimited text file and contains network data (interactions), edge data, and node data. To import network and edge data from this table, choose Unique ID A as source, Unique ID B as target, and Interactor types as interaction type. Next, turn off columns used for node data (Alternative ID A, species B, etc.). Other columns can be imported as edge data.

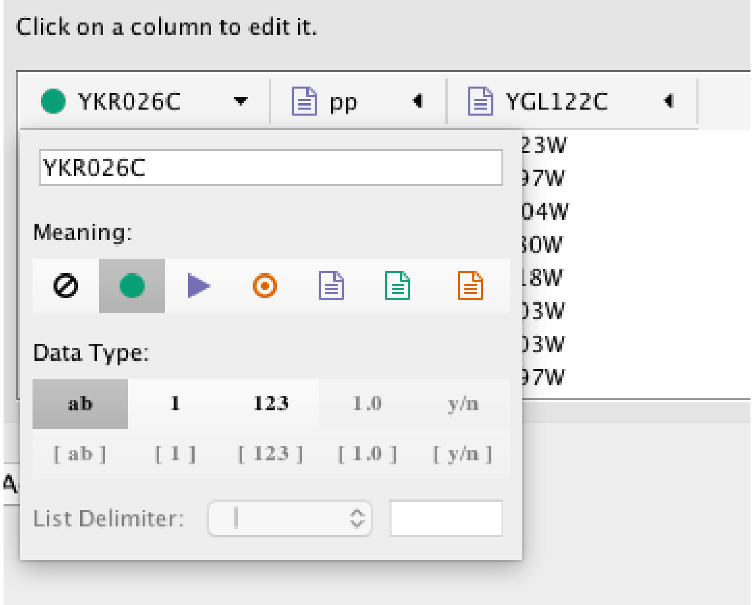
The network import function cannot import node table columns - only edge table columns. To import node table columns from this table, please see the **Node and Edge Column Data** section of this manual.

Note (1): This data is taken from the *A merged human interactome* datasets by Andrew Garrow, Yeyejide Adeleye and Guy Warner (Unilever, Safety and Environmental Assurance Center, 12 October 2006). Actual data files are available at http://wiki.cytoscape.org/Data_Sets/.

Basic Operations

To import network from text/Excel tables, please follow these steps:

1. Select **File** → **Import** → **Network** → **File...** or click on on the tool bar.
2. Select a table file in the file chooser dialog.
3. Define the interaction parameters by specifying which columns of data contain the Source Interaction, Target Interaction, and Interaction Type. Clicking on any column header will bring up the interface for selecting source, interaction and target:



Click on a column to edit it.

● YKR026C	pp	YGL122C
		23W
		97W
		04W
		30W
		18W
		03W
		03W
		97W

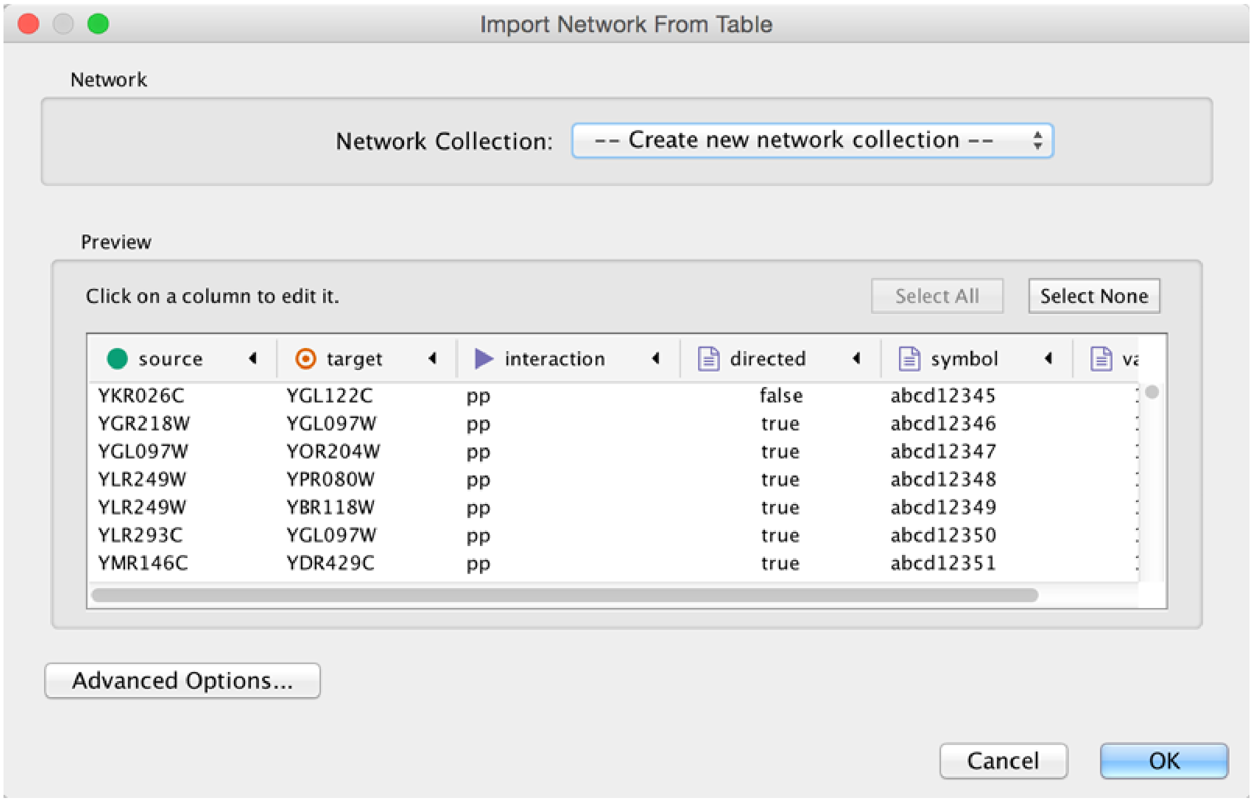
YKR026C

Meaning:

Data Type:

4. (Optional) Define edge table columns, if applicable. Network table files can have edge table columns in addition to network data.

- Enable/Disable Table Columns: You can enable/disable column data by selecting the [attachment:disablecolumn.png] symbol in the column editor.



Import Network From Table

Network

Network Collection: -- Create new network collection --

Preview

Click on a column to edit it.

● source	⊙ target	▶ interaction	directed	symbol	va
YKR026C	YGL122C	pp	false	abcd12345	
YGR218W	YGL097W	pp	true	abcd12346	
YGL097W	YOR204W	pp	true	abcd12347	
YLR249W	YPR080W	pp	true	abcd12348	
YLR249W	YBR118W	pp	true	abcd12349	
YLR293C	YGL097W	pp	true	abcd12350	
YMR146C	YDR429C	pp	true	abcd12351	

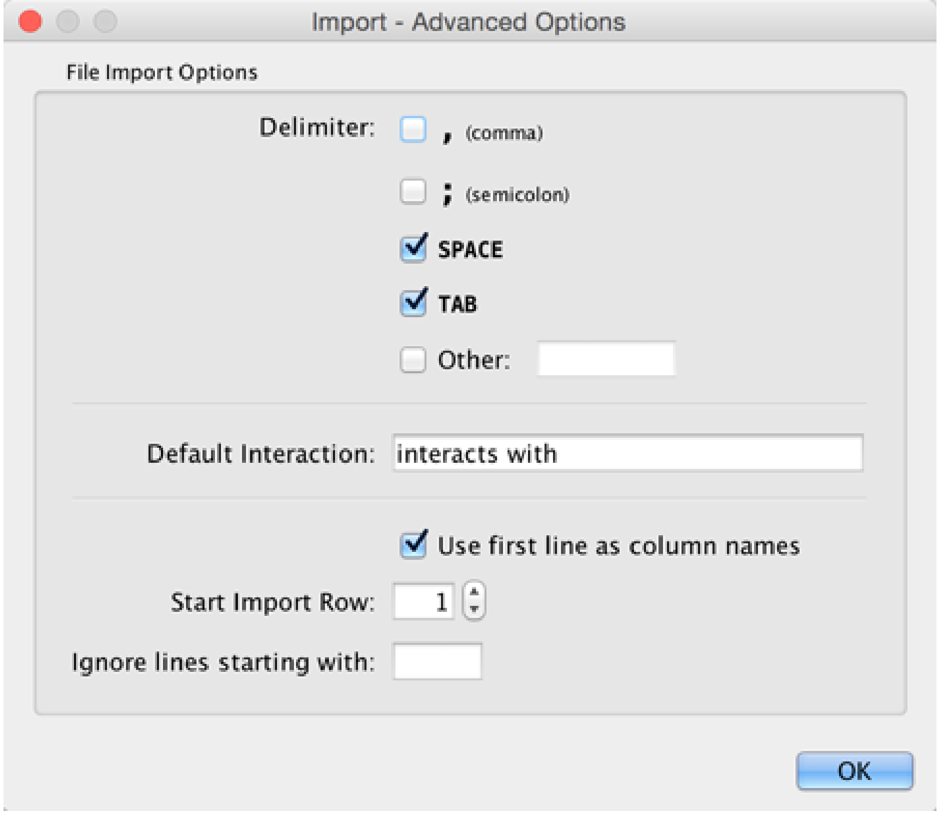
- Change Column Name and Data Types: You can also modify the column name and data type in the column editor. For more detail, see **Modify Column Name/Type** below.

5. Click the OK button.

Import List of Nodes Without Edges

The table import feature supports lists of nodes without edges. If you select only a source column, it creates a network without interactions. This feature is useful with the node expansion function available from some web service clients. Please read the section Importing Networks from External Database for more detail.

Advanced Options



Import - Advanced Options

File Import Options

Delimiter: □ , (comma)

□ ; (semicolon)

☑ SPACE

☑ TAB

□ Other:

Default Interaction: interacts with

☑ Use first line as column names

Start Import Row: 1

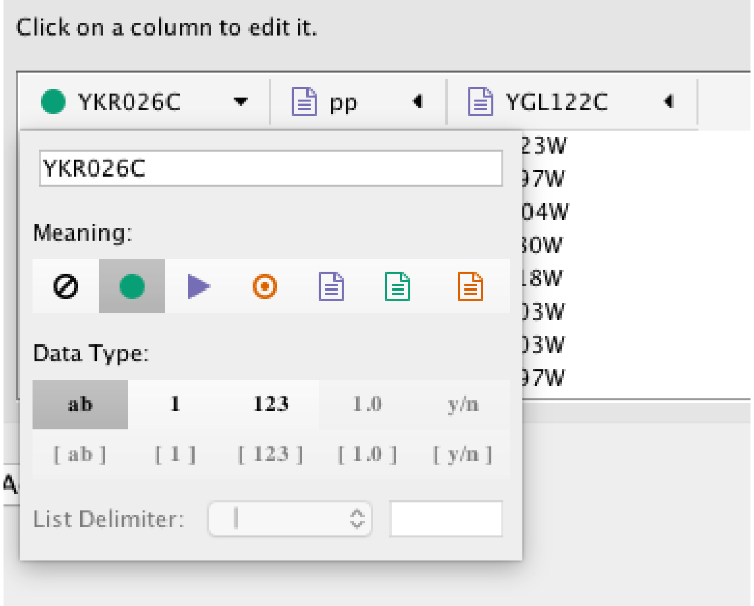
Ignore lines starting with:

You can select several options by clicking the **Advanced Options** button in the main import interface.

- Delimiter: You can select multiple delimiters for text tables. By default, Tab and Space are selected as delimiters.
- Default Interaction
- Transfer first line as column names: Selecting this option will cause all edge columns to be named according to the first data entry in that column.
- Start Import Row: Set which row of the table to begin importing data from. For example, if you want to skip the first 3 rows in the file, set 4 for this option.
- Ignore lines starting with: Rows starting with this character will not be imported. This option can be used to skip comment lines in text files.

Modify Column Name/Type

In the **Import Network from Table** interface, you can change the name and data type of column by clicking on any column header:



Click on a column to edit it.

Column names and data types can be modified here.

- Modify Column Name - just enter a new column name.
- Modify Column Data Type - The following column data types are supported:
  - String
  - Boolean (True/False)
  - Integer
  - Floating Point
  - List of (one of) String/Boolean/Integer/Floating Point

Cytoscape has a basic data type detection function that automatically suggests the column data type according to its entries. This can be overridden by selecting the appropriate data type from the radio buttons provided. For lists, a global delimiter must be specified (i.e., all cells in the table must use the same delimiter).

Import Networks from Public Databases

Cytoscape has a feature called **Import Network from Public Databases**. Users can access various kinds of databases through this function, **File** → **Import** → **Network** → **Public Databases...**.

What is a Web Service?

A web service is a standardized, platform-independent mechanism for computers to interact over the internet. These days, many major biological databases publish their data with a web service API:

- List of Biological Web Services: http://taverna.sourceforge.net/services
- Web Services at the EBI: http://www.ebi.ac.uk/Tools/webservices/

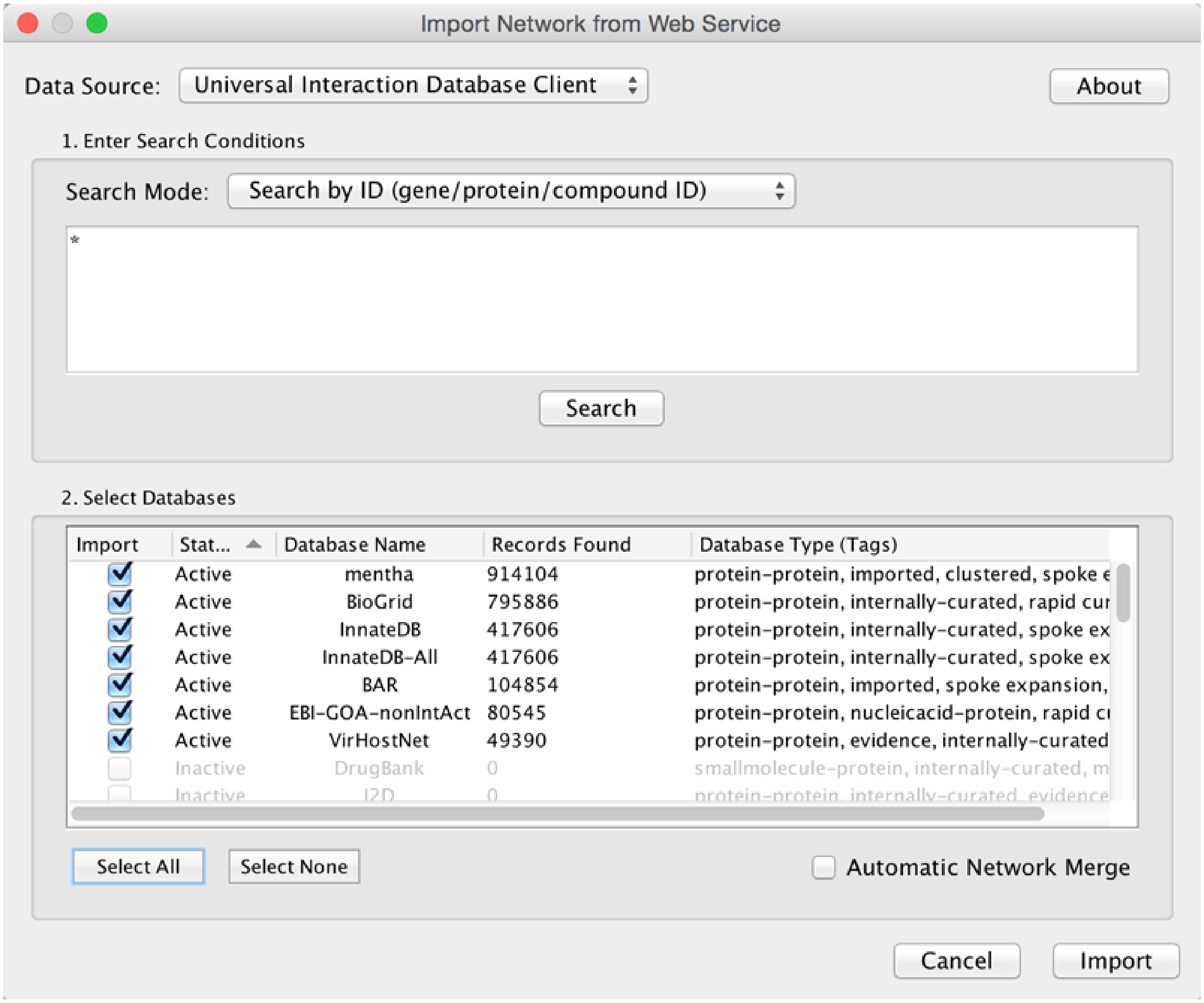
Cytoscape core developer team have developed several web service clients using this framework. Cytoscape supports many web services including:

- PSICQUIC: Standard web service for biological interaction data sets. The full list of PSICQUIC-compatible databases is available here.

The following sections describe how to import network from external databases.

Getting Started

To get started, select **File** → **Import** → **Network** → **Public Databases...**.



Data Source: Universal Interaction Database Client

1. Enter Search Conditions

Search Mode: Search by ID (gene/protein/compound ID)

*

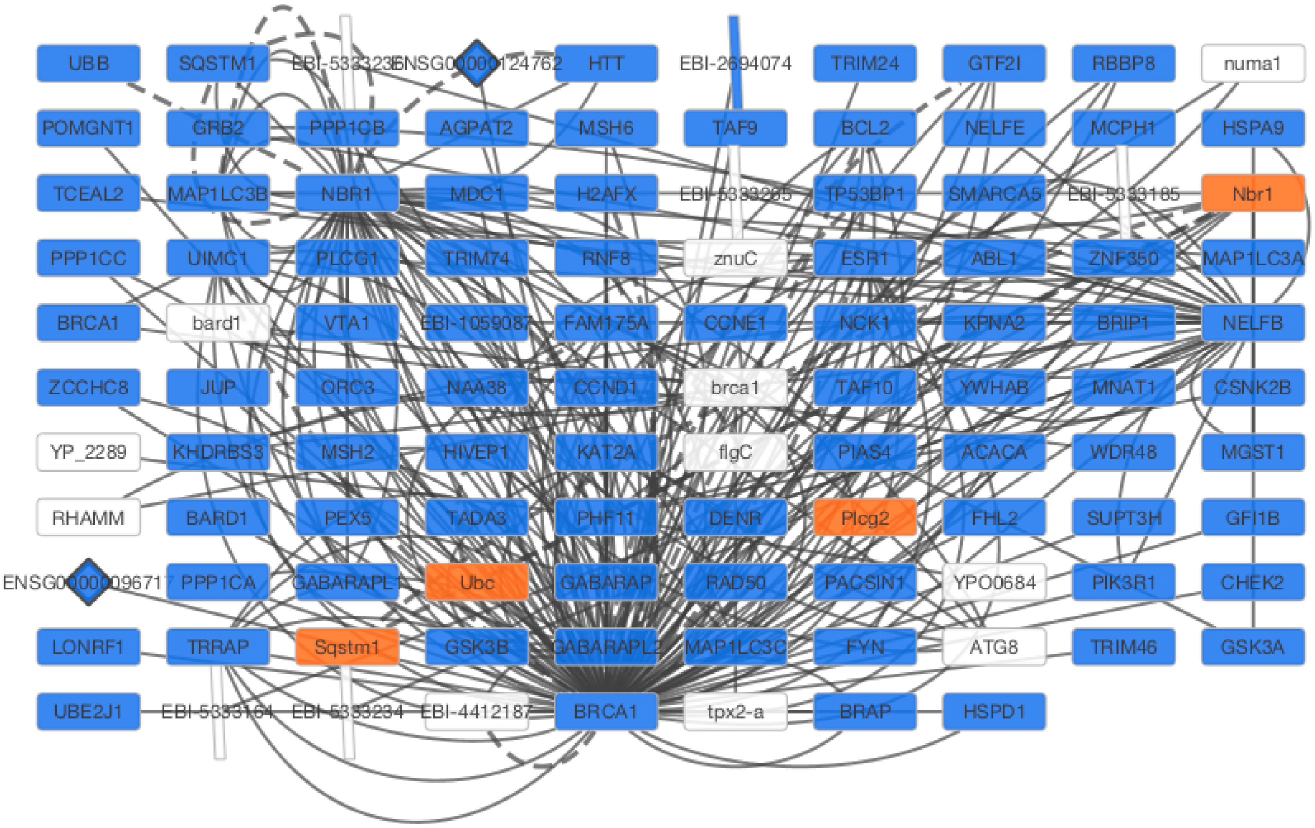
2. Select Databases

Import	Stat...	Database Name	Records Found	Database Type (Tags)
☑	Active	mentha	914104	protein-protein, imported, clustered, spoke e
☑	Active	BioGrid	795886	protein-protein, internally-curated, rapid cur
☑	Active	InnateDB	417606	protein-protein, internally-curated, spoke ex
☑	Active	InnateDB-All	417606	protein-protein, internally-curated, spoke ex
☑	Active	BAR	104854	protein-protein, imported, spoke expansion,
☑	Active	EBI-GOA-nonIntAct	80545	protein-protein, nucleicacid-protein, rapid cu
☑	Active	VirHostNet	49390	protein-protein, evidence, internally-curated
□	Inactive	DrugBank	0	smallmolecule-protein, internally-curated, m
□	Inactive	I2D	0	protein-protein, internally-curated, evidence

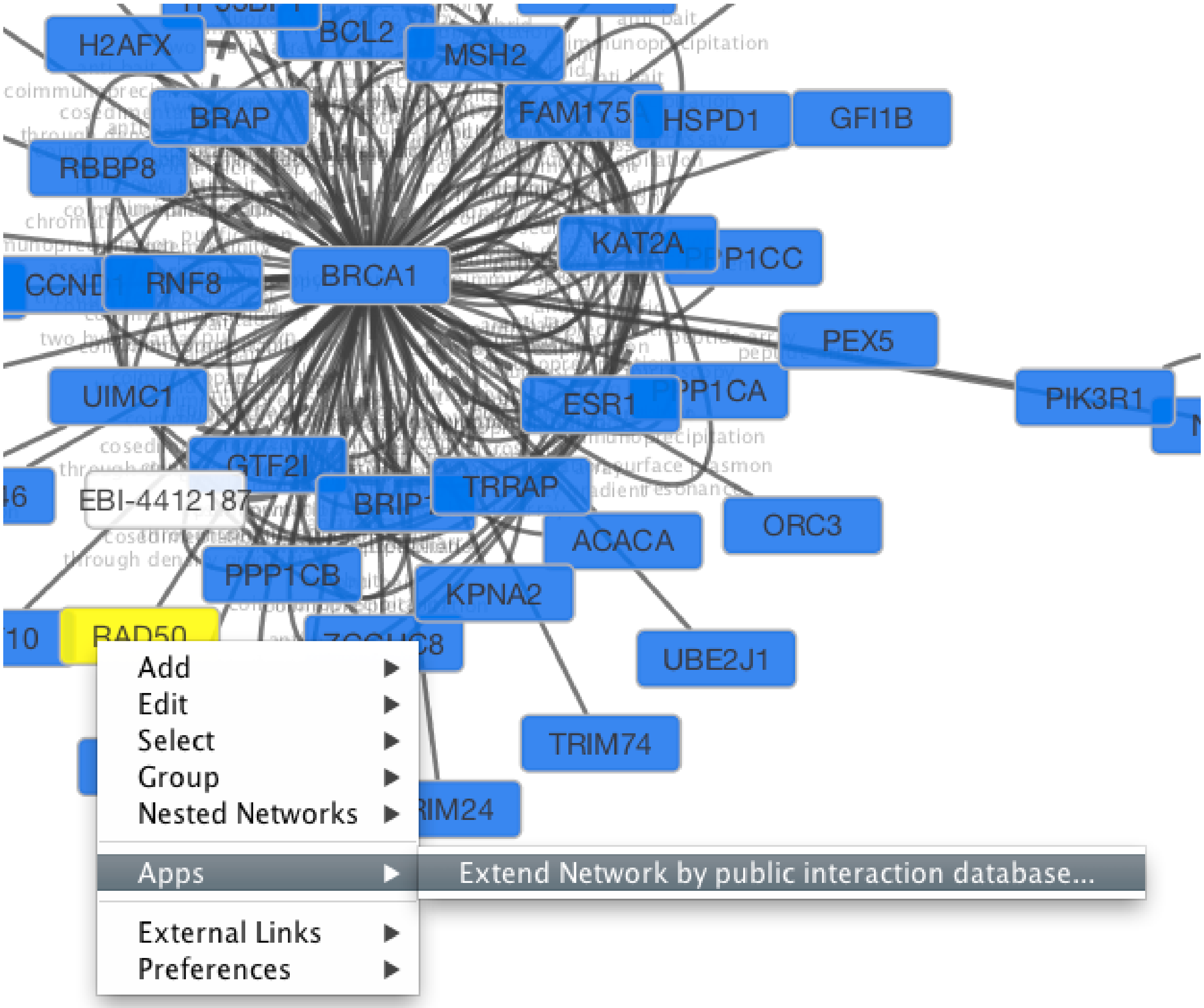
□ Automatic Network Merge

Example: Retrieving Protein-Protein Interaction Networks from Multiple Databases

- Select **File** → **Import** → **Network** → **Public Databases...**.
- From the pull-down menu, select the **Interaction databases Universal Client**.
- Enter one or more search terms, such as BRCA1.
- Click the **Search** button to start the search.
- Select databases from the hits. This selection will be saved as your default database list.
- Click the **Import** button to import selected network data.



After confirming the download of interaction data, the network of BRCA1 will be imported and visualized.



Tip: Expanding the Network: Several of the Cytoscape web services provide additional options in the node context menu. To access these options, right-click on a node and select **Apps** → **Extend Network by public interaction database...**. For example, in the screenshot, we have loaded the BRCA1 network from IntAct, and have chosen to merge this node's neighbors into the existing network.

PSICQUIC Options

PSICQUIC Web Service Client has three search modes:

- Search by ID
- Search by MIQL
- Search by Species

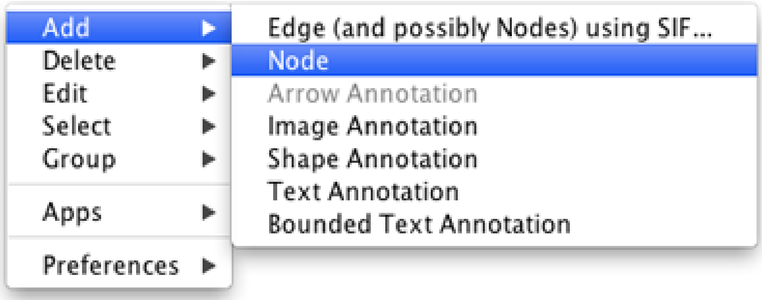
By default, search mode is set to **Search by ID**. You can search all databases by ID, such as gene symbol, Uniprot ID, or NCBI gene ID. If the search mode is set to MIQL, you can use MIQL (https://code.google.com/p/psicquic/wiki/MiqlReference27) for search. If you want to search interactions by keywords or specific functions, this is the powerful query language to filter the result. The last option is for importing all interactions for the species (i.e., interactome).

Create a New Network or Edit One Manually

A new, empty network can also be created and nodes and edges manually added. To create an empty network, go to **File** → **New** → **Network** → **Empty Network**, and then manually add network components by right clicking on the network canvas or on a node. You can edit an existing network using the same process.

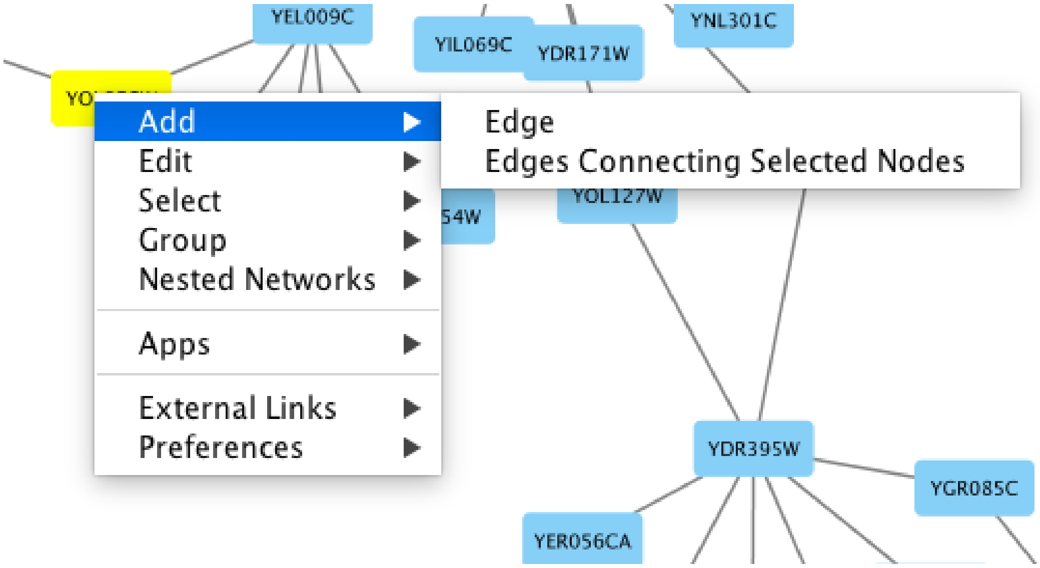
Adding a Node

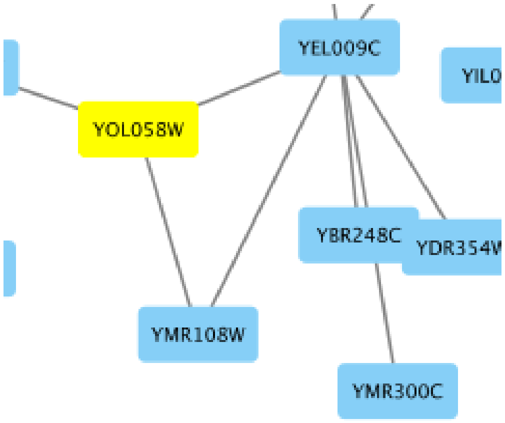
To add a new node, right-click on an empty space of the network view panel. Select **Add** → **Node** item from the pop-up menu.



Adding an Edge

To add an edge to connect nodes, right-click on the source node. Select **Edit** → **Add Edge** from the pop-up menu. Next, click on the target node. The Images below show the two steps for drawing an edge between two nodes. You can abort the drawing of the edge by pressing Esc key. You can also select two or more nodes to connect and in the right-click menu select **Add** → **Edges Connecting Selected Nodes** to create edges connecting all selected nodes.



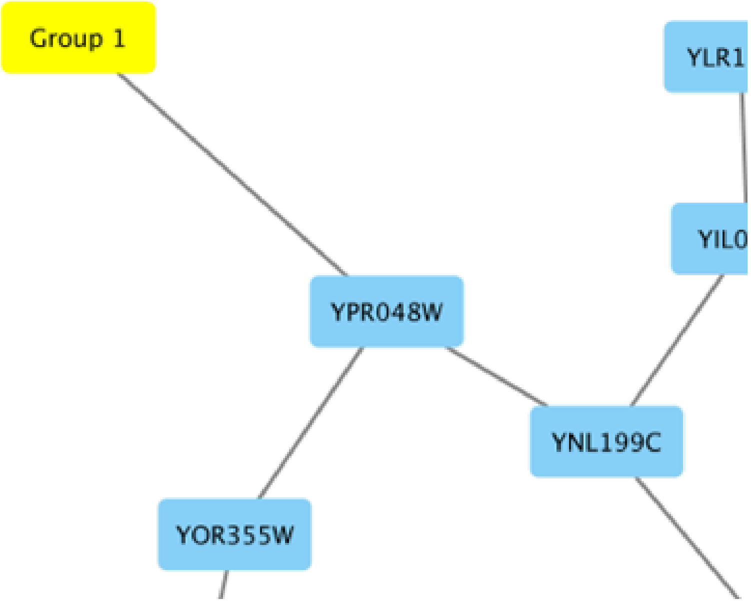


You can delete nodes and edges by selecting a number of nodes and edges, then selecting **Edit** → **Cut**. You can also delete selected nodes and edges from the **Edit** menu, under **Edit** → **Delete Selected Nodes and Edges...**. You can recover any nodes and edges deleted from a network by going to **Edit** → **Undo**.

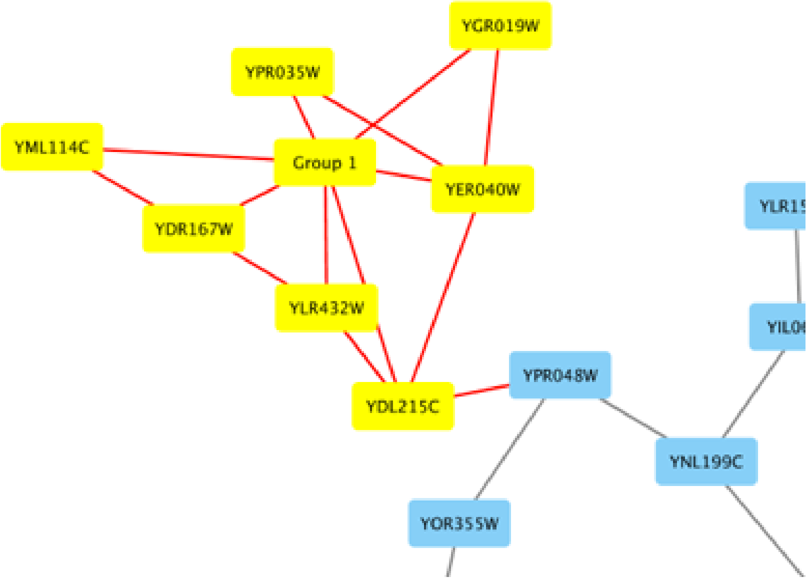
Grouping Nodes

Any number of nodes can be grouped together and displayed as either one group node or as the individual nodes. To create a group, select two or more nodes and right-click to select **Group** → **Group Selected Nodes**. You will be prompted to select a name for the group node. Once a group is created, you can use the right-click menu to collapse or expand the group. You can also quickly collapse/expand a group by double-clicking on the group node or any of its children to toggle back and forth.

Collapsed group

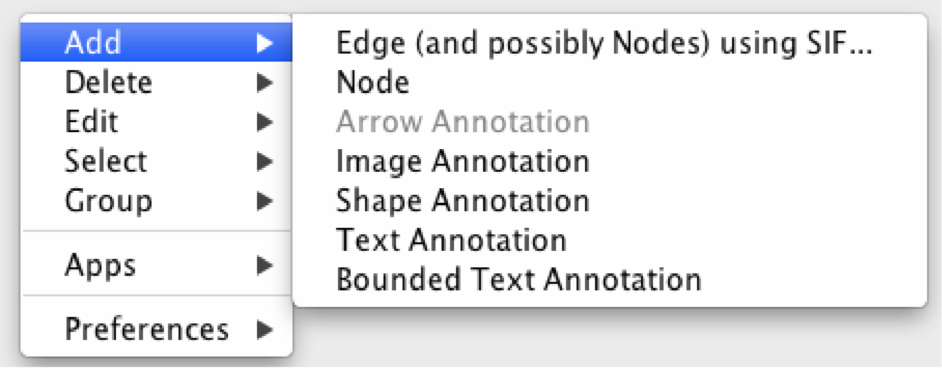


Expanded group



Adding Network Annotations

Annotations in the form of text, images or shapes can be added to the network canvas by right-clicking anywhere on the canvas and selecting one of the Annotation choices in the **Add** menu. You can add an image of your own, choose from a shapes library or add either plain or bounded text. Shapes and text are customizable and any added annotations can be edited from the right-click context menu.



Nested Networks

Cytoscape has the ability to associate a **Nested Network** with any node. A nested network is any other network currently defined in Cytoscape. This capability allows for creation of network hierarchies as well as circular relationships. For example, various module-finding plugins make use of nested networks in the overview network that they generate. There each node representing a module contains a nested network.

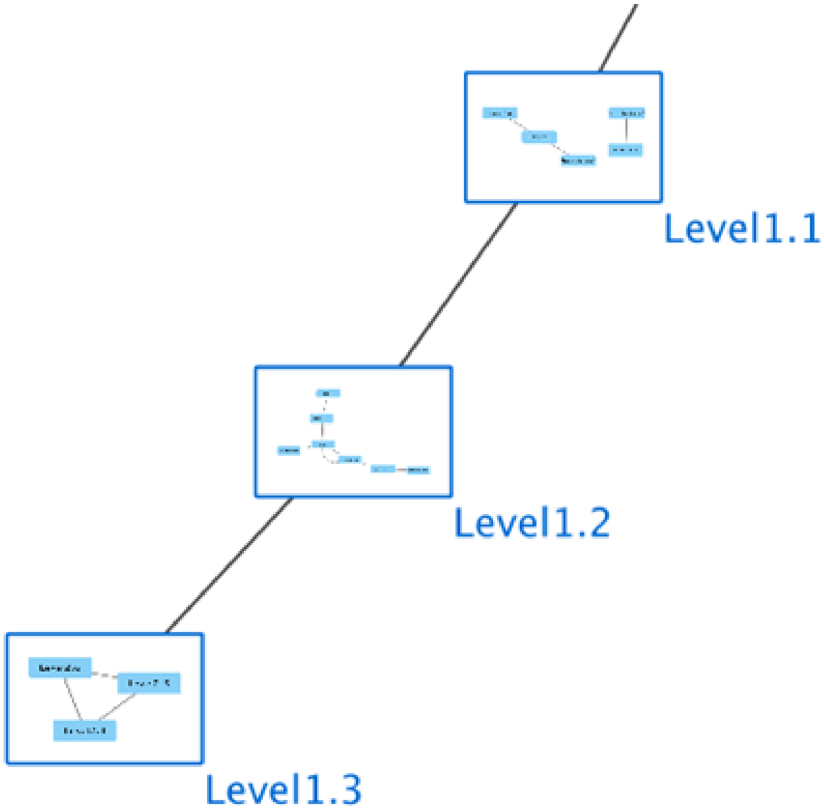
Creating Nested Networks

There are currently two ways in which a user can create nested networks.

- By importing a Nested Network Format (NNF) file. (See: NNF Network Format).
- By manually constructing networks and assigning nested networks to individual nodes through the right-click node context menu. (See Nested Network Node Context Menu).

Visualization of Nested Networks

Nodes containing nested networks that are zoomed in sufficiently display an image for the nested network. If no current network view exists for the nested network the image will be a default icon, otherwise it will be a low-resolution rendering of the nested network’s current layout.



Supported Network File Formats

Cytoscape can read network/pathway files written in the following formats:

- Simple interaction file (SIF or .sif format)
- Nested network format (NNF or .nnf format)
- Graph Markup Language (GML or .gml format)
- XGMML (extensible graph markup and modelling language).
- SBML
- BioPAX
- PSI-MI Level 1 and 2.5
- GraphML
- Delimited text
- Excel Workbook (.xls, .xlsx)
- Cytoscape.js JSON

The SIF format specifies nodes and interactions only, while other formats store additional information about network layout and allow network data exchange with a variety of other network programs and data sources. Typically, SIF files are used to import interactions when building a network for the first time, since they are easy to create in a text editor or spreadsheet. Once the interactions have been loaded and network layout has been performed, the network may be saved to GML or XGMML format for interaction with other systems. All file types listed (except Excel) are text files and you can edit and view them in a regular text editor.

SIF Format

The simple interaction format is convenient for building a graph from a list of interactions. It also makes it easy to combine different interaction sets into a larger network, or add new interactions to an existing data set. The main disadvantage is that this format does not include any layout information, forcing Cytoscape to re-compute a new layout of the network each time it is loaded.

Lines in the SIF file specify a source node, a relationship type (or edge type), and one or more target nodes:

```
nodeA <relationship type> nodeB
nodeC <relationship type> nodeA
nodeD <relationship type> nodeE nodeF nodeB
nodeG
...
nodeY <relationship type> nodeZ
```

A more specific example is:

```
node1 typeA node2
node2 typeB node3 node4 node5
node0
```

The first line identifies two nodes, called node1 and node2, and a single relationship between node1 and node2 of type typeA. The second line specifies three new nodes, node3, node4, and node5; here “node2” refers to the same node as in the first line. The second line also specifies three relationships, all of type typeB and with node2 as the source, with node3, node4, and node5 as the targets. This second form is simply shorthand for specifying multiple relationships of the same type with the same source node. The third line indicates how to specify a node that has no relationships with other nodes. This form is not needed for nodes that do have relationships, since the specification of the relationship implicitly identifies the nodes as well.

Duplicate entries are ignored. Multiple edges between the same nodes must have different edge types. For example, the following specifies two edges between the same pair of nodes, one of type xx and one of type yy:

```
node1 xx node2
node1 xx node2
node1 yy node2
```

Edges connecting a node to itself (self-edges) are also allowed:

```
node1 xx node1
```

Every node and edge in Cytoscape has a name column. For a network defined in SIF format, node names should be unique, as identically named nodes will be treated as identical nodes. The name of each node will be the name in this file by default (unless another string is mapped to display on the node using styles). This is discussed in the section on **Styles**. The name of each edge will be formed from the name of the source and target nodes plus the interaction type: for example, `sourceName (edgeType) targetName`.

The tag can be any string. Whole words or concatenated words may be used to define types of relationships, e.g. geneFusion, cogInference, pullsDown, activates, degrades, inactivates, inhibits, phosphorylates, upRegulates, etc.

Some common interaction types used in the Systems Biology community are as follows:

```
  pp .................. protein - protein interaction
  pd .................. protein -> DNA
  (e.g. transcription factor binding upstream of a regulating gene.)
```

Some less common interaction types used are:

```
  pr .................. protein -> reaction
  rc .................. reaction -> compound
  cr .................. compound -> reaction
  gl .................. genetic lethal relationship
  pm .................. protein-metabolite interaction
  mp .................. metabolite-protein interaction
```

Delimiters

Whitespace (space or tab) is used to delimit the names in the simple interaction file format. However, in some cases spaces are desired in a node name or edge type. The standard is that, if the file contains any tab characters, then tabs are used to delimit the fields and spaces are considered part of the name. If the file contains no tabs, then any spaces are delimiters that separate names (and names cannot contain spaces).

If your network unexpectedly contains no edges and node names that look like edge names, it probably means your file contains a stray tab that's fooling the parser. On the other hand, if your network has nodes whose names are half of a full name, then you probably meant to use tabs to separate node names with spaces.

Networks in simple interactions format are often stored in files with a `.sif` extension, and Cytoscape recognizes this extension when browsing a directory for files of this type.

NNF

The NNF format is a very simple format that unlike SIF allows the optional assignment of single nested network per node. No other node columns can be specified. There are only 2 possible line formats:

- A node "node" contained in a "network:"

`network node`

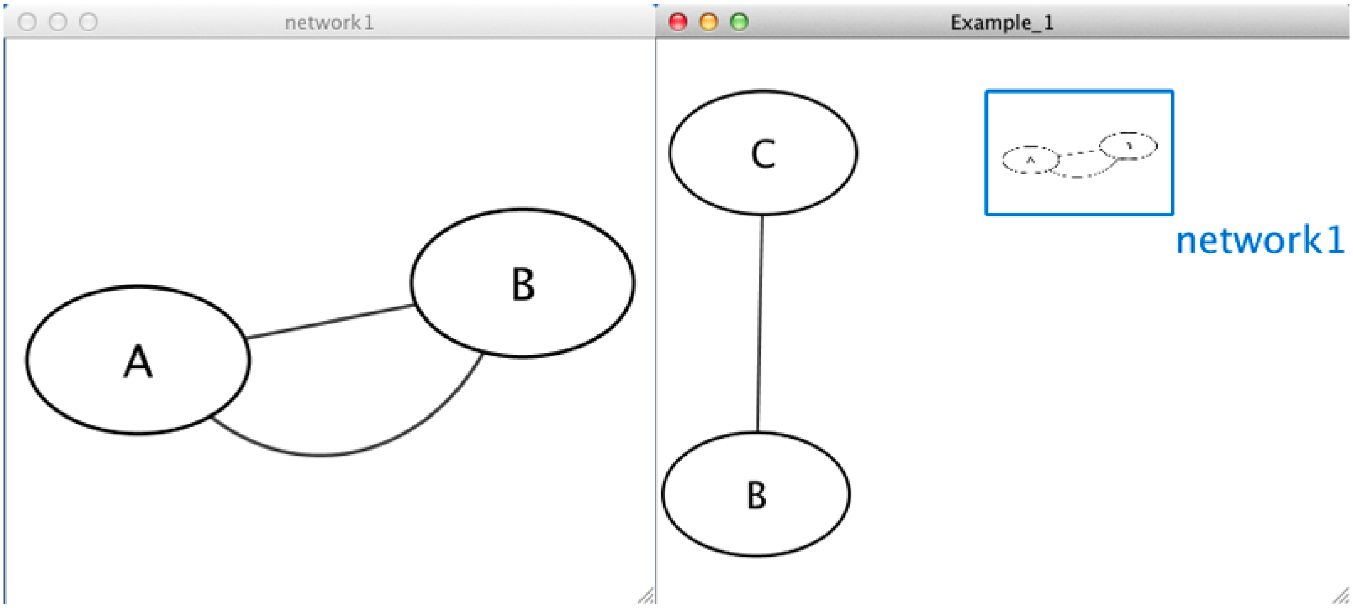
- 2 nodes linked together contained in a network:

`network node1 interaction node2`

If a network name (first entry on a line) appeared previously as a node name (in columns 2 or 4), the network will be nested in the node with the same name. Also, if a name that has been previously defined as a network (by being listed in the first column), later appears as a node name (in columns 2 or 4), the previously defined network will be nested in the node with the same name. In summary: any time a name is used as both, a network name , and a node name, this implies that the network will be nested in the node of the same name. Additionally comments may be included on all lines. Comments start with a hash mark ‘#‘ and continue to the end of a line. Trailing comments (after data lines) and entirely blank lines anywhere are also permissible. Please **note** that if you load multiple NNF files in Cytoscape they will be treated like a single, long concatenated NNF file! If you need to embed spaces, tabs or backslashes in a name, you must escape it by preceding it with a backslash, so that, e.g. an embedded backslash becomes two backslashes, an embedded space a backslash followed by a space etc.

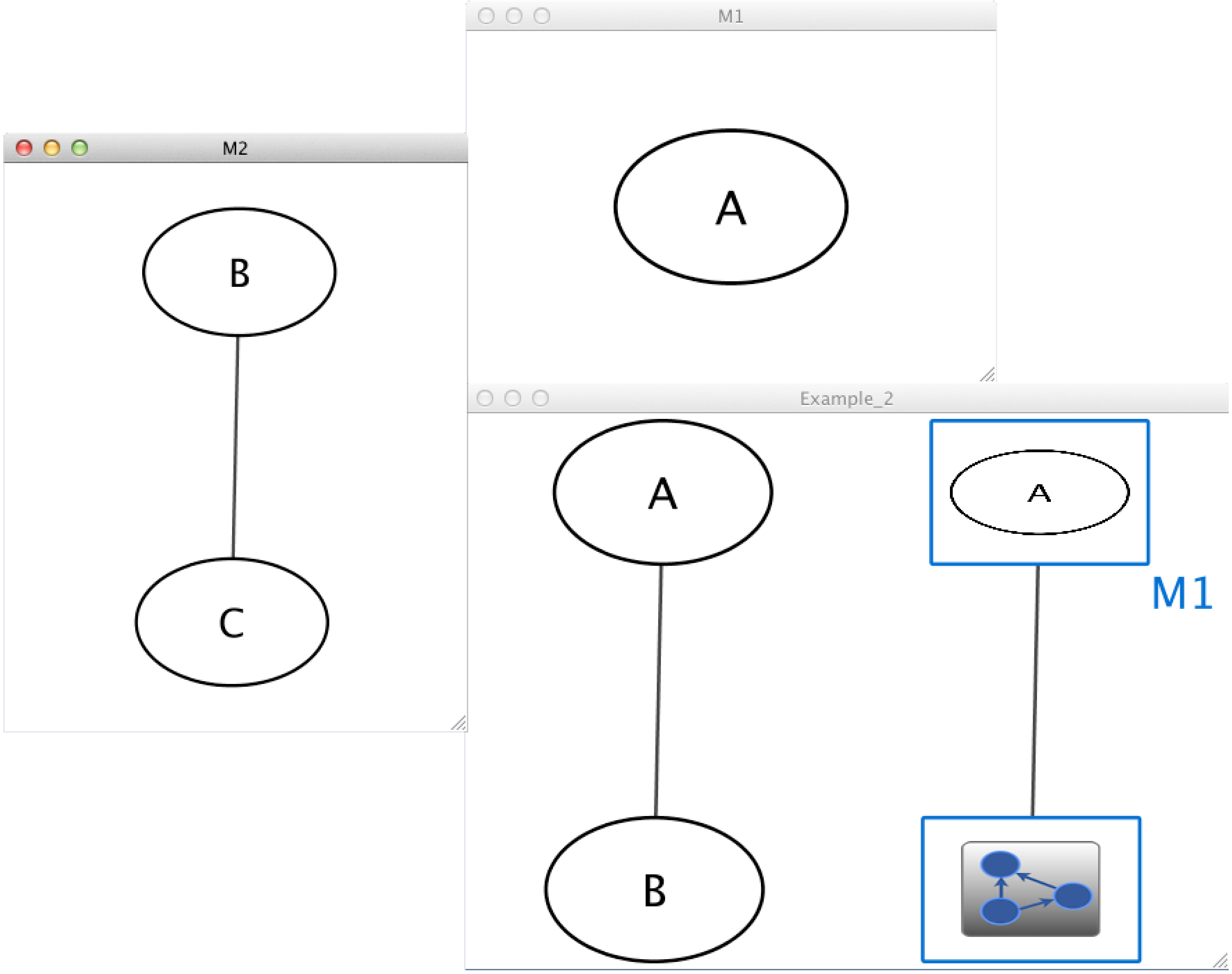
Examples

Example 1



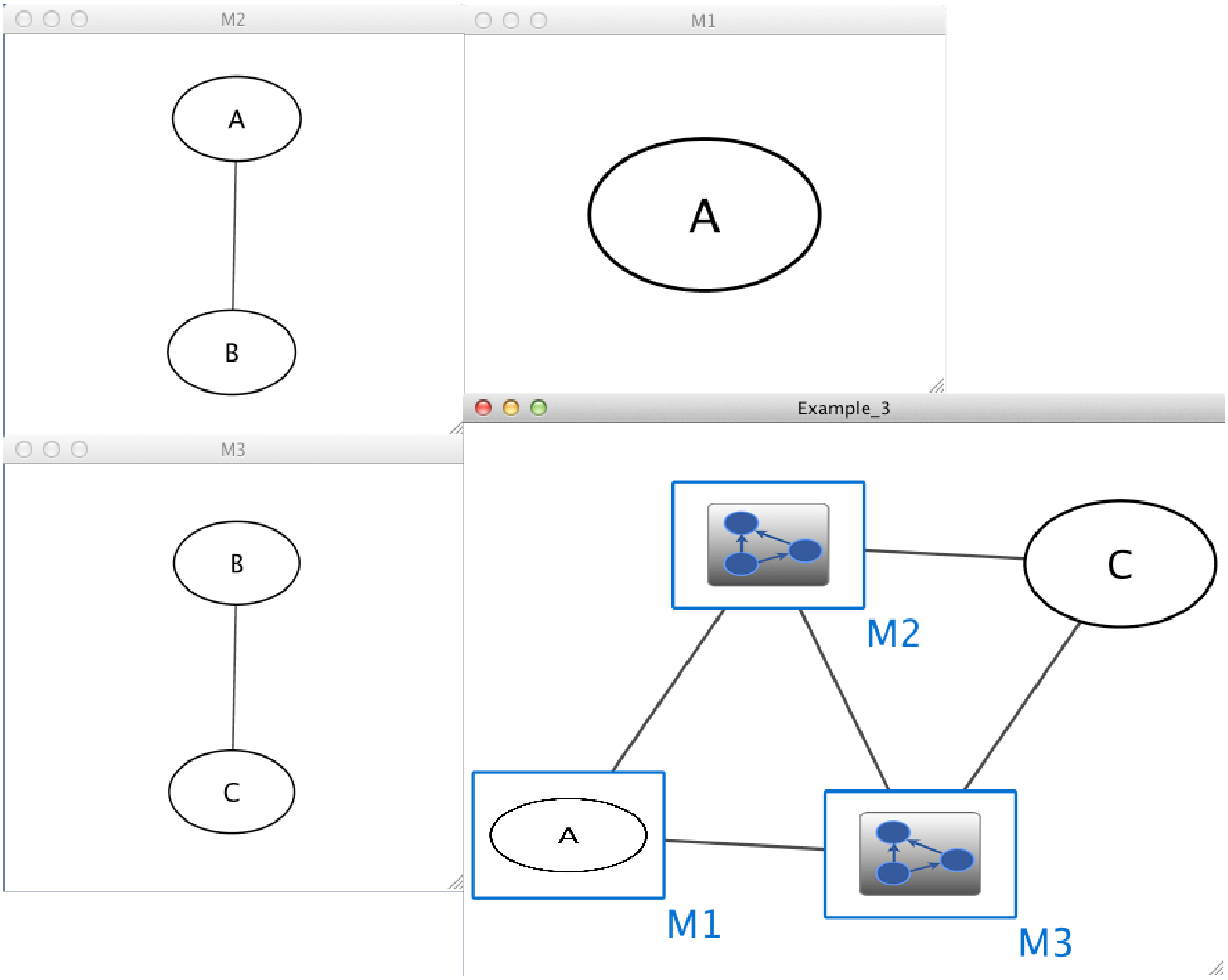
```
Example_1      C
Example_1      network1
network1       A         pp         B
network1       B         pp         A
Example_1      C         pp         B
```

Example 2



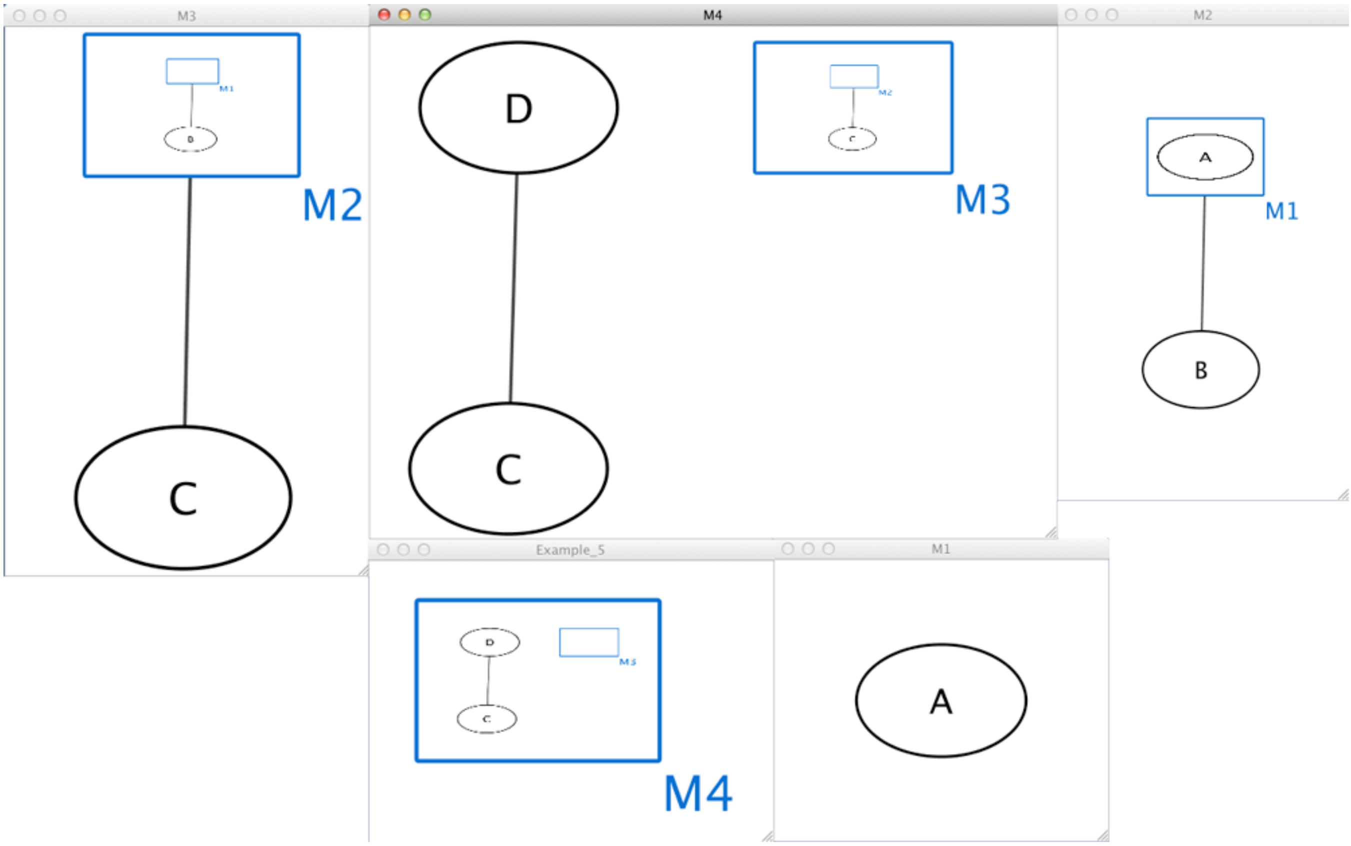
```
Example_2       M1
Example_2       M2
M1              A
M2              B         pp         C
Example_2       A         pp         B
Example_2       M1        im         M2
```

Example 3



```
Example_3       M1      im      M2
Example_3       M3      im      M1
Example_3       M2      im      M3
Example_3       C       pp      M3
Example_3       M2      pp      C
M1              A
M2              A       pp      B
M3              B       pp      C
```

Example 4



```
Example_4       M4
M4              D
M4              M3
M3              M2          pp          C
M2              M1          pp          B
M1              A
M4              C           pp          D
```

GML Format

In contrast to SIF, GML is a rich graph format language supported by many other network visualization packages. The GML file format specification is available at:

http://www.infosun.fmi.uni-passau.de/Graphlet/GML/

It is generally not necessary to modify the content of a GML file directly. Once a network is built in SIF format and then laid out, the layout is preserved by saving to and loading from GML. Properties specified in a GML file will result in a new style named `Filename.style` when that GML file is loaded.

XGMML Format

XGMML is the XML evolution of GML and is based on the GML definition. In addition to network data, XGMML contains node/edge/network column data. The XGMML file format specification is available at:

http://cgi5.cs.rpi.edu/research/groups/pb/punin/public_html/XGMML/

XGMML is now preferred to GML because it offers the flexibility associated with all XML document types. If you're unsure about which to use, choose XGMML.

There is a java system property "cytoscape.xgmml.repair.bare.ampersands" that can be set to "true" if you have experience trouble reading older files.

This should only be used when an XGMML file or session cannot be read due improperly encoded ampersands, as it slows down the reading process, but this is still preferable to attempting to fix such files using manual editing.

SBML (Systems Biology Markup Language) Format

The Systems Biology Markup Language (SBML) is an XML format to describe biochemical networks. SBML file format specification is available at:

http://sbml.org/documents/

BioPAX (Biological PAthways eXchange) Format

BioPAX is an OWL (Web Ontology Language) document designed to exchange biological pathways data. The complete set of documents for this format is available at:

http://www.biopax.org/

PSI-MI Format

The PSI-MI format is a data exchange format for protein-protein interactions. It is an XML format used to describe PPI and associated data. PSI-MI XML format specification is available at:

http://psidev.sourceforge.net/mi/xml/doc/user/

GraphML

GraphML is a comprehensive and easy-to-use file format for graphs. It is based on XML. The complete set of documents for this format is available at:

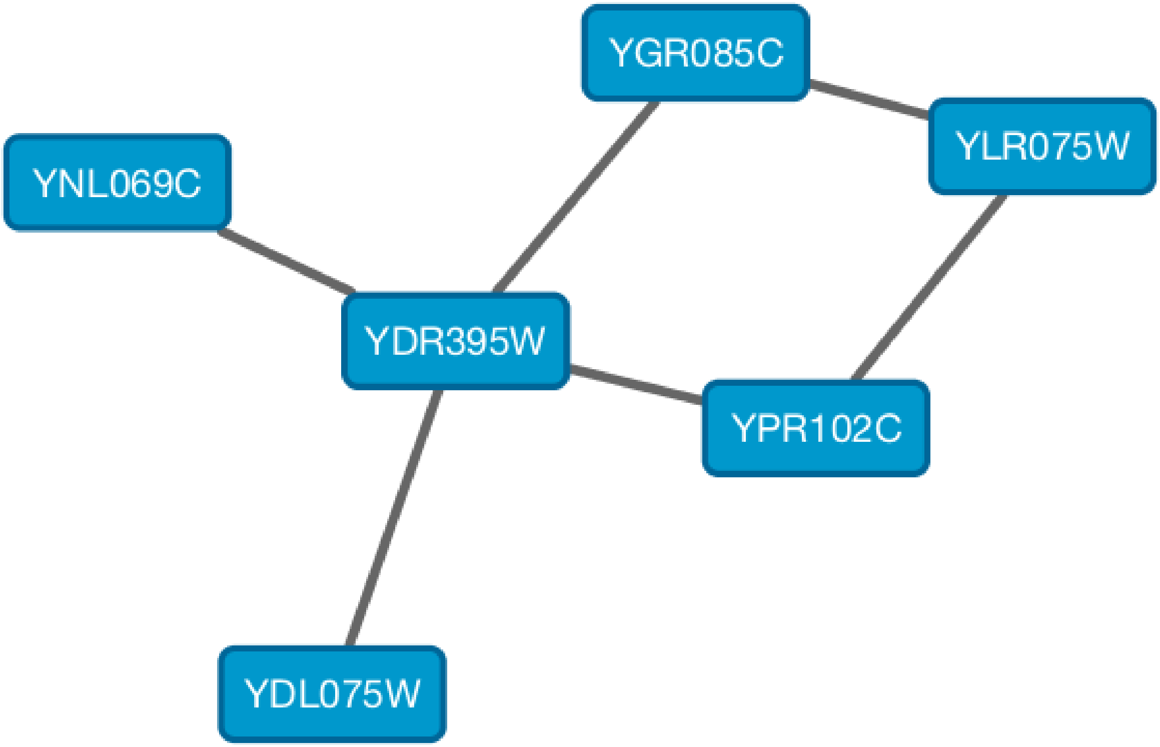
http://graphml.graphdrawing.org/

Delimited Text Table and Excel Workbook

Cytoscape has native support for Microsoft Excel files (.xls, .xlsx) and delimited text files. The tables in these files can have network data and edge columns. Users can specify columns containg source nodes, target nodes, interaction types, and edge columns during file import. Some of the other network analysis tools, such as igraph (http://cneurocvs.rmki.kfki.hu/igraph/), has feature to export graph as simple text files. Cytoscape can read these text files and build networks from them. For more detail, please read the Import Free-Format Tables section of the **Creating Networks** section.

Cytoscape.js JSON

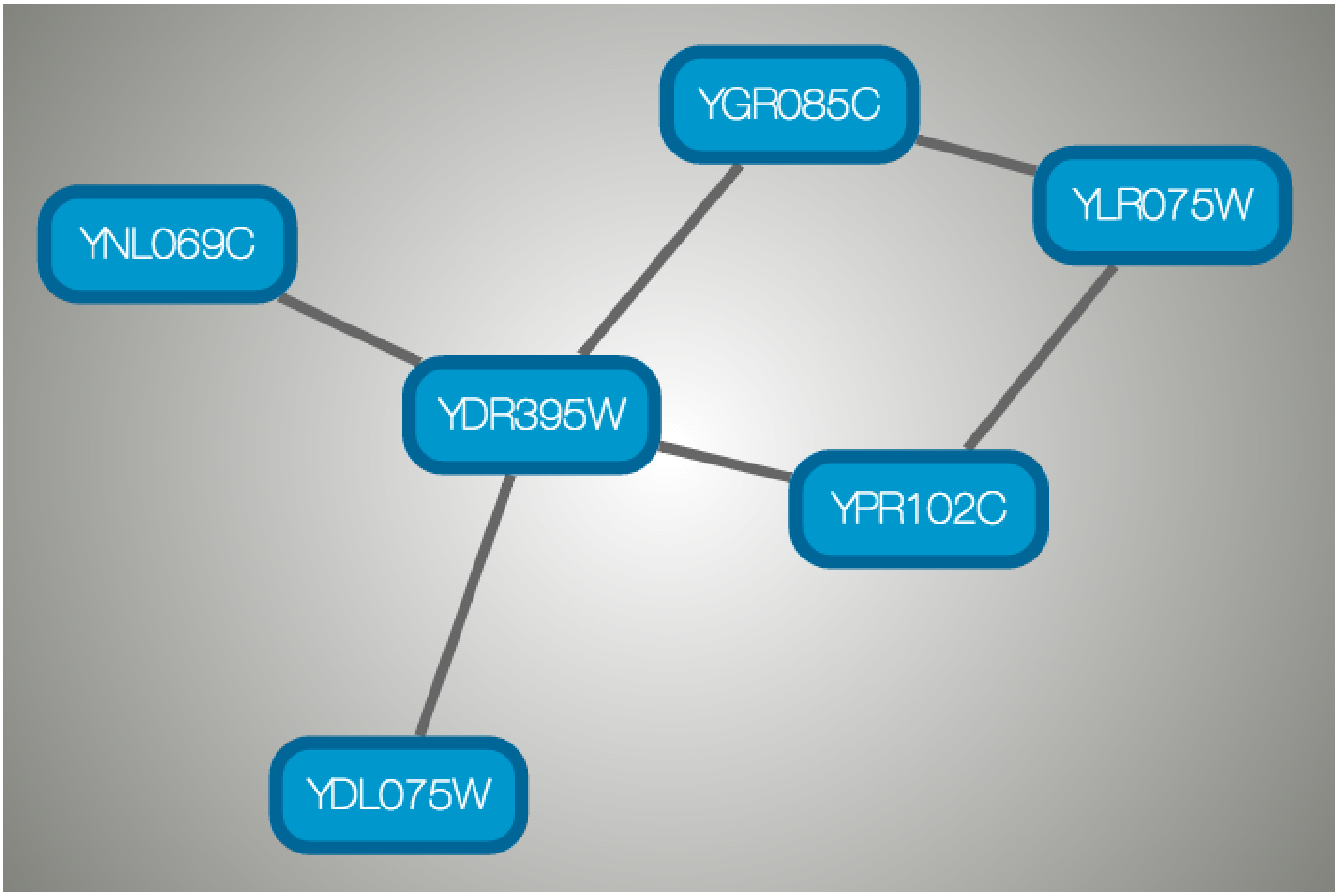
From Cytoscape 3.1.0 on, Cytoscape supports Cytoscape.js JSON files. You can use this feature to export your network visualizations to web browsers. Cytoscape.js has two ways to represent network data, and currently both reader and writer support only the array style graph notation. For example, this network in Cytoscape:



will be exported to this JSON:

```
{
  "elements" : {
    "nodes" : [ {
      "data" : {
        "id" : "723",
        "selected" : false,
        "annotation_Taxon" : "Saccharomyces cerevisiae",
        "alias" : [ "RPL31A", "RPL34", "S000002233", "ribosomal protein L31A (L34A) (YL28)" ],
        "shared_name" : "YDL075W",
        "SUID" : 723,
        "degree_layout" : 1,
        "name" : "YDL075W"
      },
      "position" : {
        "x" : 693.0518315633137,
        "y" : -49.47506554921466
      },
      "selected" : false
    }, {
      "data" : {
        "id" : "726",
        "selected" : false,
        "annotation_Taxon" : "Saccharomyces cerevisiae",
        "alias" : [ "RP23", "RPL16B", "S000005013", "ribosomal protein L16B (L21B) (rp23)
(YL15)" ],
        "shared_name" : "YNL069C",
        "SUID" : 726,
        "degree_layout" : 1,
        "name" : "YNL069C"
      },
      "position" : {
        "x" : 627.3147710164387,
        "y" : -205.99251969655353
      },
      "selected" : false
    }, {
      "data" : {
        "id" : "658",
        "selected" : false,
        "annotation_Taxon" : "Saccharomyces cerevisiae",
        "alias" : [ "RPL11B", "S000003317", "ribosomal protein L11B (L16B) (rp39B) (YL22)" ],
        "shared_name" : "YGR085C",
        "SUID" : 658,
        "degree_layout" : 2,
        "name" : "YGR085C"
      },
      "position" : {
        "x" : 804.3092778523762,
        "y" : -245.6235926946004
      },
      "selected" : false
    }, {
      "data" : {
        "id" : "660",
        "selected" : false,
        "annotation_Taxon" : "Saccharomyces cerevisiae",
        "alias" : [ "KAP108", "S000002803", "SXM1" ],
        "shared_name" : "YDR395W",
        "SUID" : 660,
        "degree_layout" : 8,
        "name" : "YDR395W"
      },
      "position" : {
        "x" : 730.8733342488606,
        "y" : -157.50702317555744
      },
      "selected" : false
    }, {
      "data" : {
        "id" : "579",
        "selected" : false,
        "annotation_Taxon" : "Saccharomyces cerevisiae",
        "alias" : [ "RPL11A", "S000006306", "ribosomal protein L11A (L16A) (rp39A) (YL22)" ],
        "shared_name" : "YPR102C",
        "SUID" : 579,
        "degree_layout" : 2,
        "name" : "YPR102C"
      },
      "position" : {
        "x" : 841.1395696004231,
        "y" : -130.77909119923908
      },
      "selected" : false
    }, {
      "data" : {
        "id" : "578",
        "selected" : false,
        "annotation_Taxon" : "Saccharomyces cerevisiae",
        "alias" : [ "GRC5", "QSR1", "RPL10", "S000004065", "ribosomal protein L10" ],
        "shared_name" : "YLR075W",
        "SUID" : 578,
        "degree_layout" : 2,
        "name" : "YLR075W"
      },
      "position" : {
        "x" : 910.3755162556965,
        "y" : -217.0562556584676
      },
      "selected" : false
    } ],
    "edges" : [ {
      "data" : {
        "id" : "659",
        "source" : "658",
        "target" : "578",
        "selected" : false,
        "interaction" : "pp",
          "shared_interaction" : "pp",
          "shared_name" : "YGR085C (pp) YLR075W",
          "SUID" : 659,
          "name" : "YGR085C (pp) YLR075W"
        },
        "selected" : false
      }, {
        "data" : {
          "id" : "661",
          "source" : "658",
          "target" : "660",
          "selected" : false,
          "interaction" : "pp",
          "shared_interaction" : "pp",
          "shared_name" : "YGR085C (pp) YDR395W",
          "SUID" : 661,
          "name" : "YGR085C (pp) YDR395W"
        },
        "selected" : false
      }, {
        "data" : {
          "id" : "724",
          "source" : "660",
          "target" : "723",
          "selected" : false,
          "interaction" : "pp",
          "shared_interaction" : "pp",
          "shared_name" : "YDR395W (pp) YDL075W",
          "SUID" : 724,
          "name" : "YDR395W (pp) YDL075W"
        },
        "selected" : false
      }, {
        "data" : {
          "id" : "733",
          "source" : "660",
          "target" : "579",
          "selected" : false,
          "interaction" : "pp",
          "shared_interaction" : "pp",
          "shared_name" : "YDR395W (pp) YPR102C",
          "SUID" : 733,
          "name" : "YDR395W (pp) YPR102C"
        },
        "selected" : false
      }, {
        "data" : {
          "id" : "727",
          "source" : "660",
          "target" : "726",
          "selected" : false,
          "interaction" : "pp",
          "shared_interaction" : "pp",
          "shared_name" : "YDR395W (pp) YNL069C",
        "SUID" : 727,
        "name" : "YDR395W (pp) YNL069C"
      },
      "selected" : false
    }, {
      "data" : {
        "id" : "580",
        "source" : "578",
        "target" : "579",
        "selected" : false,
        "interaction" : "pp",
        "shared_interaction" : "pp",
        "shared_name" : "YLR075W (pp) YPR102C",
        "SUID" : 580,
        "name" : "YLR075W (pp) YPR102C"
      },
      "selected" : false
    } ]
  }
}
```

And this is a sample visualization in Cytoscape.js:



Important Note

Export network and table to Cytoscape.js feature in Cytoscape creates a JSON file **WITHOUT** style. This means that you need to export the style in a separate JSON file if you apply style to your network. Please read the Style section for more details.

Node and Edge Column Data

Interaction networks are useful as stand-alone models. However, they are most powerful for answering scientific questions when integrated with additional information. Cytoscape allows the user to add arbitrary node, edge and network information to Cytoscape as node/edge/network data columns. This could include, for example, annotation data on a gene or confidence values in a protein-protein interaction. These column data can then be visualized in a user-defined way by setting up a mapping from columns to network properties (colors, shapes, and so on). The section on **Styles** discusses this in greater detail.

Import Data Table Files

Cytoscape offers support for importing data from delimited text and MS Excel data tables.

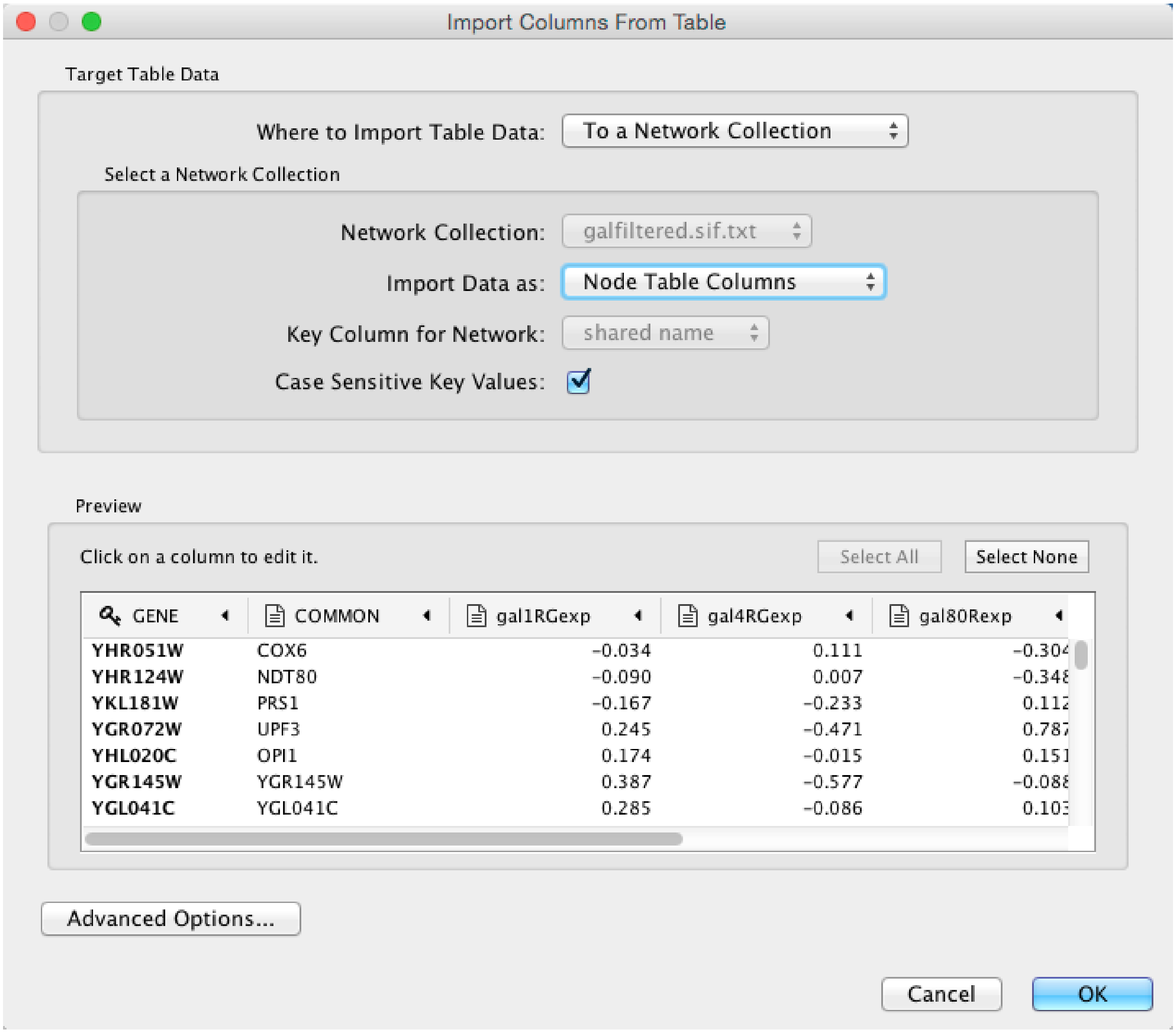
Sample Data Table 1

Sample Data

OBJECT KEY	ALIAS	SGD ID
AAC3	YBR085W|ANC3	S000000289
AAT2	YLR027C|ASP5	S000004017
BIK1	YCL029C|ARM5|PAC14	S000000534

The data table file should contain a primary key column and at least one data column. The maximum number of data columns is unlimited. The **Alias** column is an optional feature, as is using the first row of data as column names. Alternatively, you can specify each column name from the **File** → **Import** → **Table** → **File...** user interface.

Basic Operation



Target Table Data

Where to Import Table Data: To a Network Collection

Select a Network Collection

Network Collection: galfiltered.sif.txt

Import Data as: Node Table Columns

Key Column for Network: shared name

Case Sensitive Key Values: ☑

Preview

Click on a column to edit it.

GENE	COMMON	gal1RGexp	gal4RGexp	gal80Rexp
YHR051W	COX6	-0.034	0.111	-0.304
YHR124W	NDT80	-0.090	0.007	-0.348
YKL181W	PRS1	-0.167	-0.233	0.112
YGR072W	UPF3	0.245	-0.471	0.787
YHL020C	OPI1	0.174	-0.015	0.151
YGR145W	YGR145W	0.387	-0.577	-0.088
YGL041C	YGL041C	0.285	-0.086	0.103

1. Select **File** → **Import** → **Table** → **File...**.
2. Select a data file. The file can be either a text or Excel (.xls/.xlsx) file.
3. In the **Target Table Data** section, choose where to import the data to. You can choose an existing network collection, a specific network only, or you can choose to import the data to an **Unassigned Table** (described below).
4. Depending on what you choose in the **Where to import Table Data** drop-down, you will need to select a network collection or specific network. You will also need to select **Importing Type**, that is whether the data is node, edge or network table columns.
5. If the table is not properly delimited in the preview panel, change the delimiter in the **Advanced Options** panel. The default delimiter is tab. This step is not necessary for Excel Workbooks.
6. By default, the first column is designated as the primary key. Change the key column if necessary.
7. Click **OK** to import.

Unassigned Table

As of Cytoscape 3.1. it is possible to import data tables without assigning them to existing networks, meaning the data doesn't have to correspond to any nodes/edges currently loaded. If a data table is imported as unassigned and a network is later imported that maps to the data in terms of nodes or edges, the data will link automatically. This is useful when loading a large dataset (for example expression data), defining a **Style** for visualizing the data on networks and later loading individual networks to view the data, for example from an online database. This feature allows the data to be automatically linked to any network that is applicable, without having to load the data for each network.

Legacy Cytoscape Attributes Format

In addition to tabular data, the simple attribute file format used in previous versions of Cytoscape is still supported. Node and edge data files are simply formatted: a node data file begins with the name of the column on the first line (note that it cannot contain spaces). Each following line contains the name of the node, followed by an equals sign and the data value. Numbers and text strings are the most common data types. All values for a given column must have the same type. For example:

```
FunctionalCategory
YAL001C = metabolism
YAR002W = apoptosis
YBL007C = ribosome
```

An edge data file has much the same structure, except that the name of the edge is the source node name, followed by the interaction type in parentheses, followed by the target node name. Directionality counts, so switching the source and target will refer to a different (or perhaps non-existent) edge. The following is an example edge data file:

```
InteractionStrength
YAL001C (pp) YBR043W = 0.82
YMR022W (pd) YDL112C = 0.441
YDL112C (pd) YMR022W = 0.9013
```

Since Cytoscape treats edge data as directional, the second and third edge data values refer to two different edges (source and target are reversed, though the nodes involved are the same).

Each data column is stored in a separate file. Node and edge data files use the same format, and have the suffix ".attrs".

Node and edge data may be loaded via the **File** → **Import** → **Table** menu, just as data table files are.

When expression data is loaded using an expression matrix, it is automatically loaded as node data unless explicitly specified otherwise.

Node and edge data columns are attached to nodes and edges, and so are independent of networks. Data values for a given node or edge will be applied to all copies of that node or edge in all loaded network files, regardless of whether the data file or network file is imported first.

Detailed file format (Advanced users)

Every data file has one header line that gives the name of the data column, and optionally some additional meta-information about that data column. The format is as follows:

```
columnName (class=JavaClassName)
```

The first field is always the column name: it cannot contain spaces. If present, the class field defines the name of the class of the data values. For example, java.lang.String or String for Strings, java.lang.Double or Double for floating point values, java.lang.Integer or Integer for integer values, etc. If the value is actually a list of values, the class should be the type of the objects in the list. If no class is specified in the header line, Cytoscape will attempt to guess the type from the first value. If the first value contains numbers in a floating point format, Cytoscape will assume java.lang.Double; if the first value contains only numbers with no decimal point, Cytoscape will assume java.lang.Integer; otherwise Cytoscape will assume java.lang.String. Note that the first value can lead Cytoscape astray: for example,

```
floatingPointDataColumn
firstName = 1
secondName = 2.5
```

In this case, the first value will make Cytoscape think the values should be integers, when in fact they should be floating point numbers. It’s safest to explicitly specify the value type to prevent confusion. A better format would be:

```
floatingPointDataColumn (class=Double)
firstName = 1
secondName = 2.5
```

or

```
floatingPointDataColumn
firstName = 1.0
secondName = 2.5
```

Every line past the first line identifies the name of an object (a node in a node data file or an edge in a edge data file) along with the String representation of the data value. The delimiter is always an equals sign; whitespace (spaces and/or tabs) before and after the equals sign is ignored. This means that your names and values can contain whitespace, but object names cannot contain an equals sign and no guarantees are made concerning leading or trailing whitespace. Object names must be the Node ID or Edge ID as seen in the left-most column of the Table Panel if the data column is to map to anything. These names must be reproduced exactly, including case, or they will not match.

Edge names are all of the form:

```
sourceName (edgeType) targetName
```

Specifically, that is

`sourceName space openParen edgeType closeParen space targetName`

Note that tabs are not allowed in edge names. Tabs can be used to separate the edge name from the "=" delimiter, but not within the edge name itself. Also note that this format is different from the specification of interactions in the SIF file format. To be explicit: a SIF entry for the previous interaction would look like

```
sourceName edgeType targetName
```

or

```
sourceName whiteSpace edgeType whiteSpace targetName
```

To specify lists of values, use the following syntax:

```
listDataColumnName (class=java.lang.String)
firstObjectName = (firstValue::secondValue::thirdValue)
secondObjectName = (onlyOneValue)
```

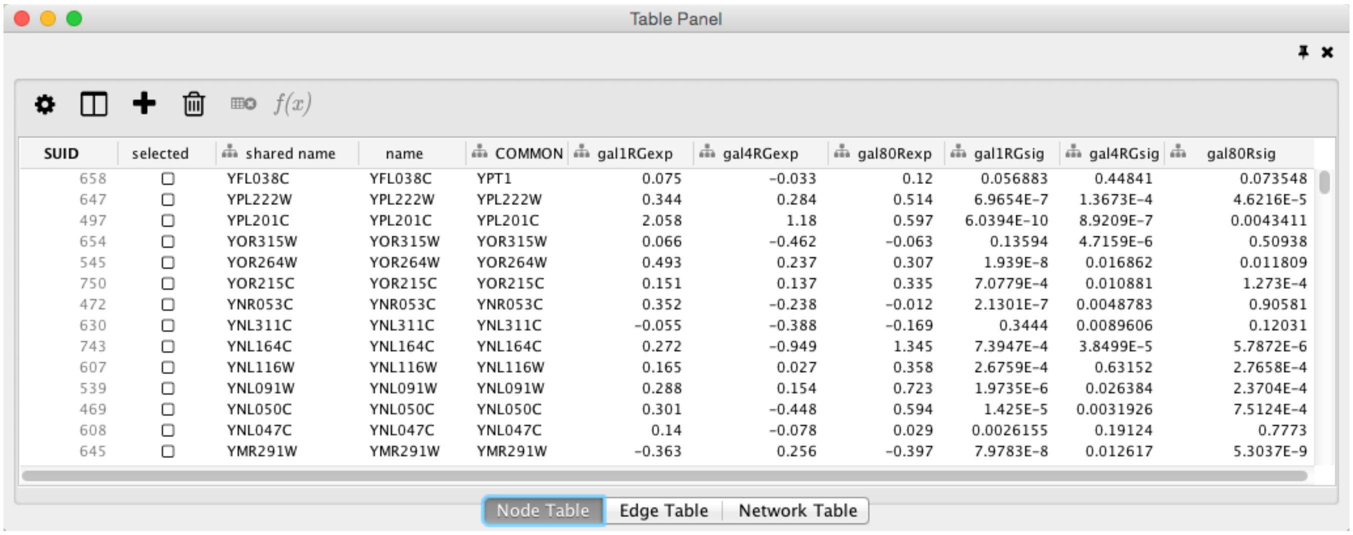
This example shows a data column whose value is defined as a list of text strings. The first object has three strings, and thus three elements in its list, while the second object has a list with only one element. In the case of a list every data value uses list syntax (i.e. parentheses), and each element is of the same class. Again, the class will be inferred if it is not specified in the header line. Lists are not supported by Styles and so can't be mapped to network properties.

Newline Feature

Sometimes it is desirable to for data values to include linebreaks, such as node labels that extend over two lines. You can accomplish by inserting into the data value. For example:

```
newlineDataColumn
YJL157C = This is a long\nline for a label.
```

Table Panel



SUID	selected	shared name	name	COMMON	gal1RGexp	gal4RGexp	gal80Rexp	gal1RGsig	gal4RGsig	gal80Rsig
658	□	YFL038C	YFL038C	YPT1	0.075	-0.033	0.12	0.056883	0.44841	0.073548
647	□	YPL222W	YPL222W	YPL222W	0.344	0.284	0.514	6.9654E-7	1.3673E-4	4.6216E-5
497	□	YPL201C	YPL201C	YPL201C	2.058	1.18	0.597	6.0394E-10	8.9209E-7	0.0043411
654	□	YOR315W	YOR315W	YOR315W	0.066	-0.462	-0.063	0.13594	4.7159E-6	0.50938
545	□	YOR264W	YOR264W	YOR264W	0.493	0.237	0.307	1.939E-8	0.016862	0.011809
750	□	YOR215C	YOR215C	YOR215C	0.151	0.137	0.335	7.0779E-4	0.010881	1.273E-4
472	□	YNR053C	YNR053C	YNR053C	0.352	-0.238	-0.012	2.1301E-7	0.0048783	0.90581
630	□	YNL311C	YNL311C	YNL311C	-0.055	-0.388	-0.169	0.3444	0.0089606	0.12031
743	□	YNL164C	YNL164C	YNL164C	0.272	-0.949	1.345	7.3947E-4	3.8499E-5	5.7872E-6
607	□	YNL116W	YNL116W	YNL116W	0.165	0.027	0.358	2.6759E-4	0.63152	2.7658E-4
539	□	YNL091W	YNL091W	YNL091W	0.288	0.154	0.723	1.9735E-6	0.026384	2.3704E-4
469	□	YNL050C	YNL050C	YNL050C	0.301	-0.448	0.594	1.425E-5	0.0031926	7.5124E-4
608	□	YNL047C	YNL047C	YNL047C	0.14	-0.078	0.029	0.0026155	0.19124	0.7773
645	□	YMR291W	YMR291W	YMR291W	-0.363	0.256	-0.397	7.9783E-8	0.012617	5.3037E-9

When Cytoscape is started, the **Table Panel** appears in the bottom right of the main Cytoscape window. This browser can be hidden and restored using the F5 key or the **View** → **Show/Hide Table Panel** menu option. Like other Panels, the browser can be undocked by pressing the little icon in the top right corner.

To swap between displaying node, edge, and network Data Tables use the tabs on the bottom of the Table Panel. By default, the Table Panel displays columns for all nodes and edges in the selected network. To display columns for only selected nodes/edges, click the **Change Table Mode** button ⚙ at the top left. To change the columns that are displayed, click the **Show Column** ◫ button and choose the columns that are to be displayed (select various columns by clicking on them, and then click elsewhere on the screen to close the column list).

Most column values can be edited by double-clicking the cell (only the ID cannot be edited). Newline characters can be inserted into String columns either by pressing **Enter** or by typing "\n". Once finished editing, click outside of the editing cell in the Table Panel or press **Shift-Enter** to save your edits. Pressing **Esc** while editing will undo any changes.

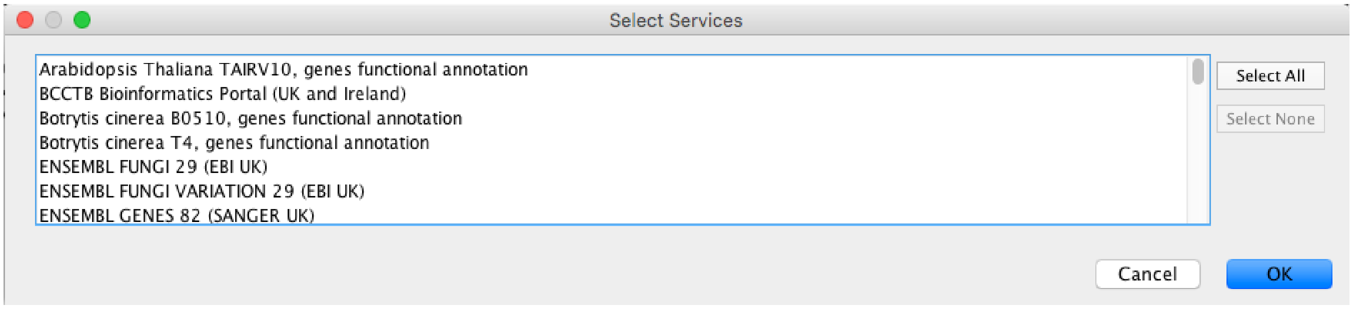
Rows in the panel can be sorted alphabetically by specific column by clicking on a column heading. A new column can be created using the **Create New column** ✚ button, and must be one of four types - integer, string, real number (floating point), or boolean. Columns can be deleted using the **Delete Columns...** 🗑 button. **NOTE: Deleting columns removes them from Cytoscape, not just the Table Panel!** To remove columns from the panel without deleting them, simply unselect the column using the **Select Columns** ◫ button.

Import Data Table from Public Databases

It is also possible to import node data columns from public databases via web services, for example from BioMart (http://www.biomart.org).

Basic Operation

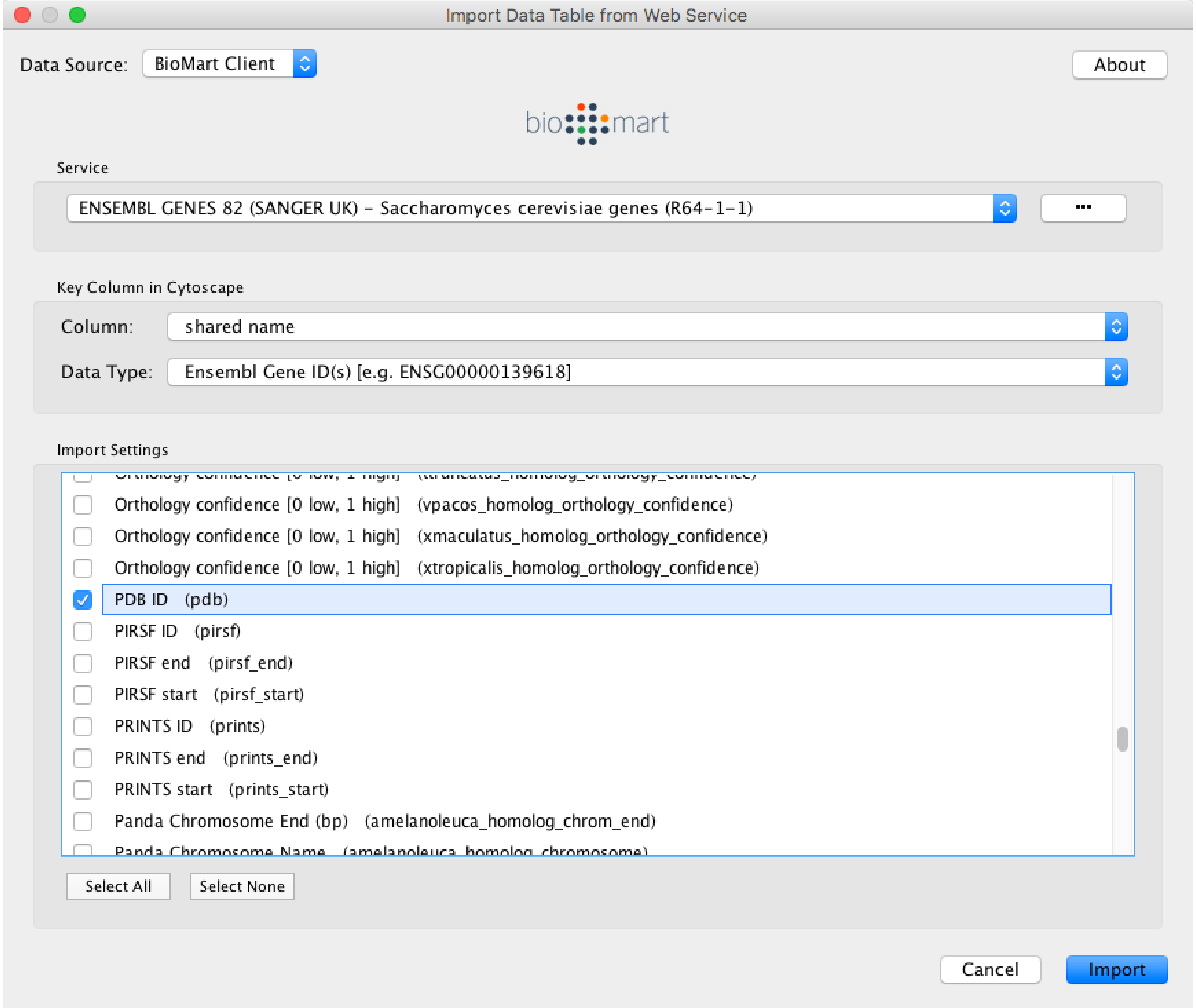
1. Load a network, for example galFiltered.sif.
2. Select **File** → **Import** → **Table** → **Public Databases...**.
3. You will first be asked to select from a set of web services. For this example, we will choose **ENSEMBL GENES 73 (SANGER UK)**.



Arabidopsis Thaliana TAIRV10, genes functional annotation
BCCTB Bioinformatics Portal (UK and Ireland)
Botrytis cinerea B0510, genes functional annotation
Botrytis cinerea T4, genes functional annotation
ENSEMBL FUNGI 29 (EBI UK)
ENSEMBL FUNGI VARIATION 29 (EBI UK)
ENSEMBL GENES 82 (SANGER UK)

1. In the **Import Data Table from Web Services** dialog, select a **Data Source**. Since galFiltered.sif is a yeast network, select **ENSEMBL GENES - SACCHAROMYCES CEREVISIAE**.
2. For **Key Column in Cytoscape**, select *shared name* for **Column** and *Ensembl Gene ID* for **Data Type**.

The type of identifier selected under **Data Type** must match what is used in the selected **Column** in the network.



Data Source: BioMart Client

ENSEMBL GENES 82 (SANGER UK) – Saccharomyces cerevisiae genes (R64-1-1)

Column: shared name

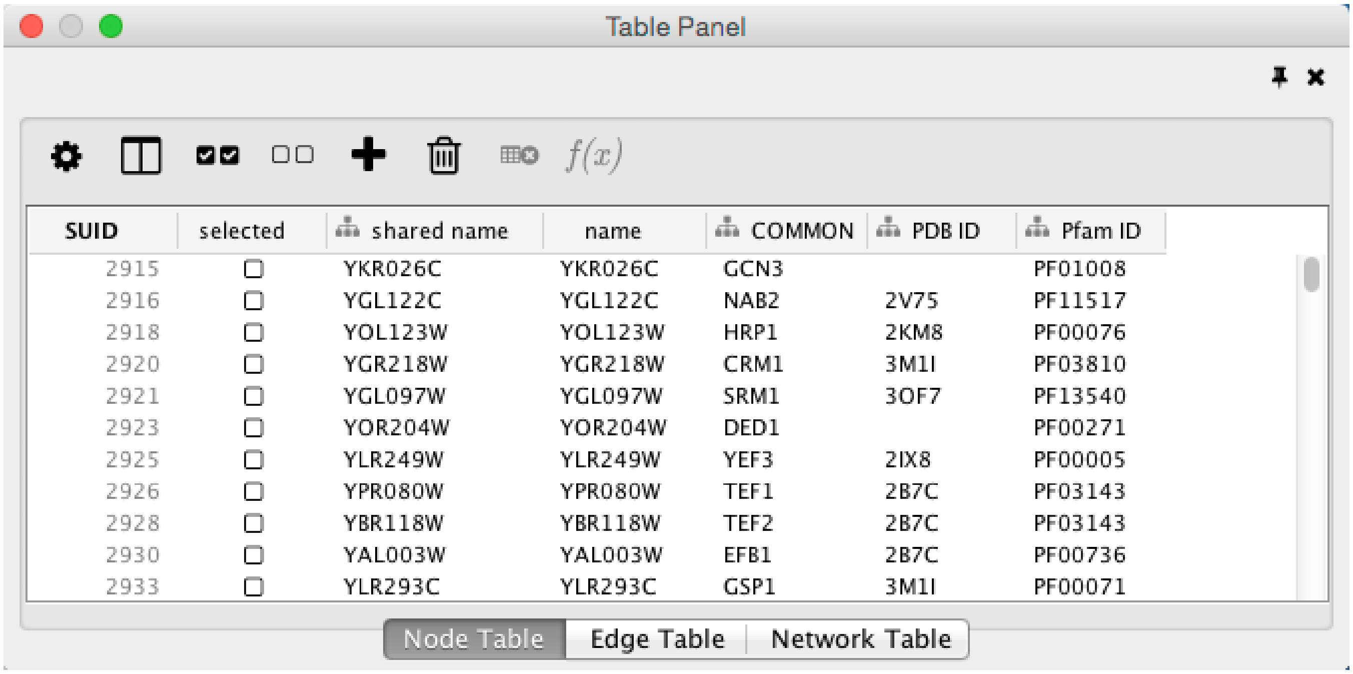
Data Type: Ensembl Gene ID(s) [e.g. ENSG00000139618]

□ Orthology confidence [0 low, 1 high] (ttruncatus_homolog_orthology_confidence)
□ Orthology confidence [0 low, 1 high] (vpacos_homolog_orthology_confidence)
□ Orthology confidence [0 low, 1 high] (xmaculatus_homolog_orthology_confidence)
□ Orthology confidence [0 low, 1 high] (xtropicalis_homolog_orthology_confidence)
☑ PDB ID (pdb)
□ PIRSF ID (pirsf)
□ PIRSF end (pirsf_end)
□ PIRSF start (pirsf_start)
□ PRINTS ID (prints)
□ PRINTS end (prints_end)
□ PRINTS start (prints_start)
□ Panda Chromosome End (bp) (amelanoleuca_homolog_chrom_end)
□ Panda Chromosome Name (amelanoleuca_homolog_chromosome)

1. Select the data columns you want to import.

2. Select **Import**.

When import is complete, you can see the newly imported data columns in the Table Panel.



SUID	selected	shared name	name	COMMON	PDB ID	Pfam ID
2915	□	YKR026C	YKR026C	GCN3		PF01008
2916	□	YGL122C	YGL122C	NAB2	2V75	PF11517
2918	□	YOL123W	YOL123W	HRP1	2KM8	PF00076
2920	□	YGR218W	YGR218W	CRM1	3M1I	PF03810
2921	□	YGL097W	YGL097W	SRM1	3OF7	PF13540
2923	□	YOR204W	YOR204W	DED1		PF00271
2925	□	YLR249W	YLR249W	YEF3	2IX8	PF00005
2926	□	YPR080W	YPR080W	TEF1	2B7C	PF03143
2928	□	YBR118W	YBR118W	TEF2	2B7C	PF03143
2930	□	YAL003W	YAL003W	EFB1	2B7C	PF00736
2933	□	YLR293C	YLR293C	GSP1	3M1I	PF00071

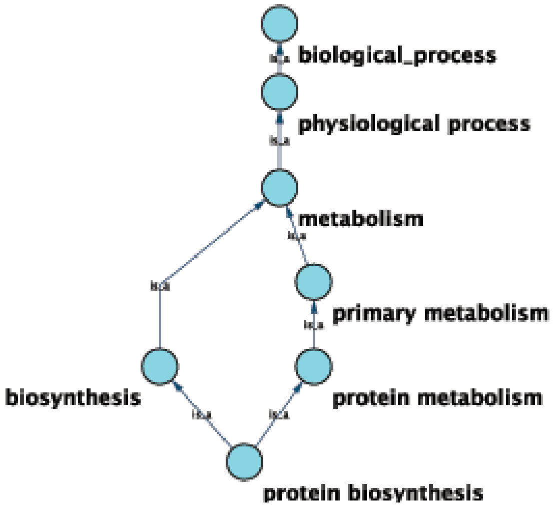
Ontology and Annotation Import

Annotations in Cytoscape are stored as a set of ontologies (e.g. the Gene Ontology, or GO). An ontology consists of a set of controlled vocabulary terms that annotate the objects. For example, using the Gene Ontology, the Saccharomyces Cerevisiae CDC55 gene has a biological process described as “protein biosynthesis”, to which GO has assigned the number 6412 (a GO ID).

GO 8150 biological_process

- GO 7582 physiological processes
  - GO 8152 metabolism
    - GO 44238 primary metabolism
      - GO 19538 protein metabolism
        - GO 6412 protein biosynthesis

Graphical View of GO Term 6412: protein biosynthesis



Cytoscape can use this ontology DAG (Directed Acyclic Graph) to annotate objects in networks. The Ontology Server (originally called “BioDataServer (http://www.ncbi.nlm.nih.gov/pubmed/12066840)”) is a Cytoscape feature which allows you to load, navigate, and assign annotation terms to nodes and edges in a network. Cytoscape 2.4 now has an enhanced GUI for loading ontology and associated annotation, enabling you to load both local and remote files.

Ontology and Annotation File Format

The standard file formats used in the **Cytoscape Ontology Server** are OBO and Gene Association. The GO website details these file formats:

Ontologies and Definitions: http://www.geneontology.org/GO.downloads.shtml#ont

Current Annotations: http://www.geneontology.org/GO.current.annotations.shtml

Default List of Ontologies

Cytoscape provides a list of ontologies available in OBO format. If an Internet connection is available, Cytoscape will import ontology and annotation files directly from the remote source. The table below lists the included ontologies.

Default List of Ontologies

ONTOLOGY NAME	DESCRIPTION
Gene Ontology Full	This data source contains a full-size GO DAG, which contains all GO terms. This OBO file is written in version 1.2 format.
Generic GO slim	A subset of general GO Terms, including higer-level terms only.
Yeast GO slim	A subset of GO Terms for annotating Yeast data sets maintained by SGD.
	A structured controlled vocabulary of concrete and abstract (generic)

Molecule role (INOH Protein name/family name ontology)	protein names. This ontology is a INOH pathway annotation ontology, one of a set of ontologies intended to be used in pathway data annotation to ease data integration. This ontology is used to annotate protein names, protein family names, and generic/concrete protein names in the INOH pathway data. INOH is part of the BioPAX working group.
Event (INOH pathway ontology)	A structured controlled vocabulary of pathway-centric biological processes. This ontology is a INOH pathway annotation ontology, one of a set of ontologies intended to be used in pathway data annotation to ease data integration. This ontology is used to annotate biological processes, pathways, and sub-pathways in the INOH pathway data. INOH is part of the BioPAX working group.
Protein-protein interaction	A structured controlled vocabulary for the annotation of experiments concerned with protein-protein interactions.
PATO	PATO is an ontology of phenotypic qualities, intended for use in a number of applications, primarily phenotype annotation. For more information, please visit the PATO wiki.
Mouse pathology	The Mouse Pathology Ontology (MPATH) is an ontology for mutant mouse pathology. This is Version 1.
Human disease	This ontology is a comprehensive hierarchical controlled vocabulary for human disease representation. For more information, please visit the Disease Ontology website.

Although Cytoscape can import all kinds of ontologies in OBO format, annotation files are associated with specific ontologies. Therefore, you need to provide the correct ontology-specific annotation file to annotate nodes/edges/networks in Cytoscape. For example, while you can annotate human network data using the GO Full ontology with human Gene Association files, you cannot use a combination of the human Disease Ontology file and human Gene Association files, because the Gene Association file is only compatible with GO.

Gene Association File

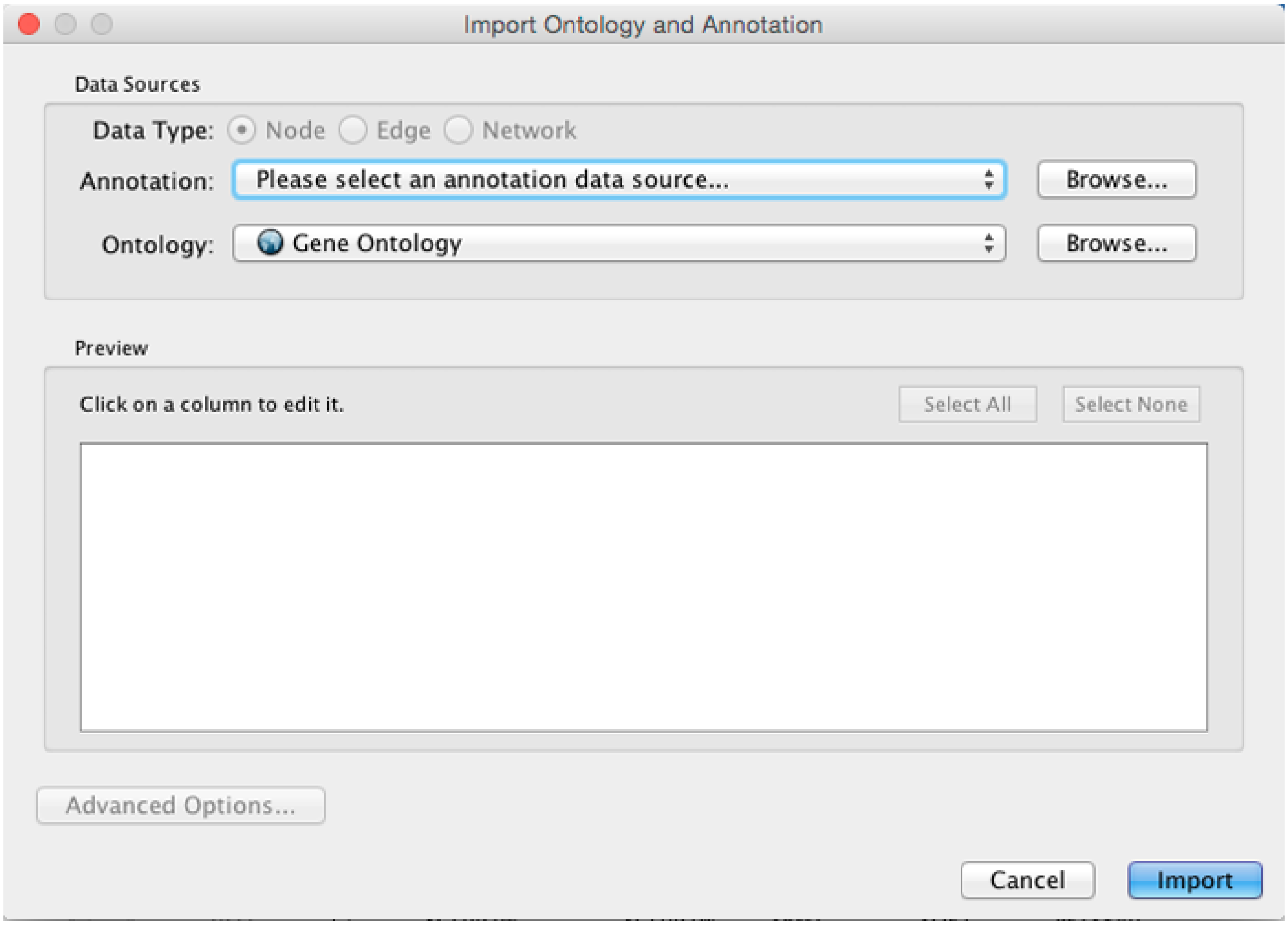
The **Gene Association** files provide annotation only for the Gene Ontology. It is a species-specific annotation file for GO terms. Gene Association files will only work with Gene Ontology annotation.

Sample Gene Association File (gene_association.sgd - annotation file for yeast):

Sample Gene Association File

SGD	S000003916	AAD10	GO:0006081	SGD_REF:S000042151|PMID:10572264	ISS	P	aryl-alcohol dehydrogenase (putative)	YJR155W gene	taxon:4932
SGD	S000005275	AAD14	GO:0008372	SGD_REF:S000069584	ND	C	aryl-alcohol dehydrogenase (putative)	YNL331C gene	taxon:4932

Import Ontology and Annotation



Import Ontology and Annotation

Data Sources

Data Type: ● Node ○ Edge ○ Network

Annotation: Please select an annotation data source...

Ontology: Gene Ontology

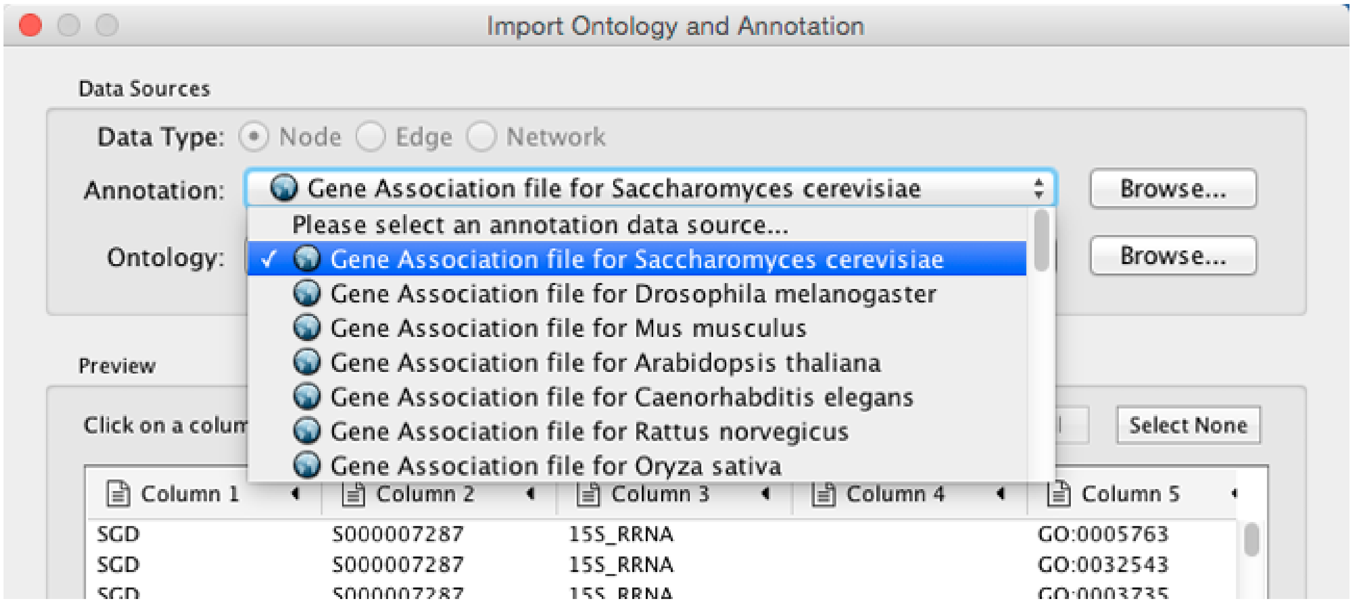
Preview

Click on a column to edit it.

Cytoscape provides a graphical user interface to import both ontology and annotation files at the same time.

Note: All data sources in the preset list are remote URLs, meaning a network connection is required.

- Select **File** → **Import** → **Ontology and Annotation...** to open the “Import Ontology and Annotation” interface. From the **Annotation** drop-down list, select a gene association file for your network. For example, if you want to annotate the yeast network, select “Gene Association file for Saccharomyces cerevisiae”.



Import Ontology and Annotation

Data Sources

Data Type: ● Node ○ Edge ○ Network

Annotation: Gene Association file for Saccharomyces cerevisiae

Ontology:

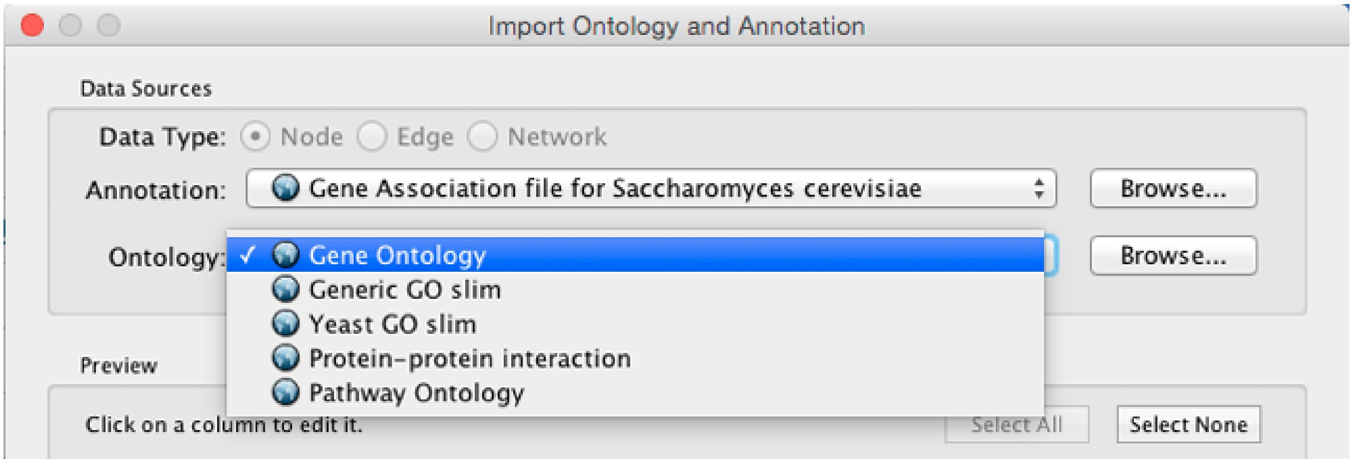
Preview

Click on a colum

Column 1	Column 2	Column 3	Column 4	Column 5
SGD	S000007287	15S_RRNA		GO:0005763
SGD	S000007287	15S_RRNA		GO:0032543
SGD	S000007287	15S_RRNA		GO:0003735

Please select an annotation data source...
✓ Gene Association file for Saccharomyces cerevisiae
Gene Association file for Drosophila melanogaster
Gene Association file for Mus musculus
Gene Association file for Arabidopsis thaliana
Gene Association file for Caenorhabditis elegans
Gene Association file for Rattus norvegicus
Gene Association file for Oryza sativa
Browse...
Browse...
Select None

* Select an Ontology data (OBO file) from the Ontology drop-down list. If the file is not loaded yet, it will be shown in red. The first three files are Gene Ontology files. You can load other ontologies, but you need your own annotation file to annotate networks.



Import Ontology and Annotation

Data Sources

Data Type: ● Node ○ Edge ○ Network

Annotation: Gene Association file for Saccharomyces cerevisiae

Ontology: Gene Ontology

Preview

Click on a column to edit it.

Once you click the **Import** button, Cytoscape will start loading OBO and Gene Association files from the remote sources. If you choose GO Full it may take a while since it is a large data file.

- When Cytoscape finishes importing files, the import window will be automatically closed. All columns mapped by this function have the prefix “annotation” and look like this: annotation.[column_name].

Note: Cytoscape supports both OBO formats: version 1.0 and 1.2.

Column Data Functions and Equations

Column Formulas

Introduction

Column data values may be formulas. A typical example is **=ABS($otherColumn + LOG(10.2))**. Formulas are modeled after Excel(tm) but only support references to other columns at the same node, edge or network. Since Cytoscape column names may contain embedded spaces, optional braces around the column name (required if the name is not simply a letter followed by one or more letters or digits) is allowed e.g. **${a name with spaces}**. Backslashes, opening braces and dollar signs in column names have to be escaped with a leading backslash. For example the column name **ex$am{p\le** would have to be written as **${ex\$am\{p\\le}**. Finally, column names are case sensitive.

String constants are written with double-quotes “. In order to embed a double-quote or a backslash in a string they have to be escaped with a leading backslash, therefore the string “\ must be written as “\“\\“. Formula results must be compatible with the type of the column that they have been assigned to. The rules are rather lax though, for example anything can be interpreted as a string and all numeric values will be accepted for a boolean (or logical) column data where non-zero will be interpreted as **true** and zero as **false**. For integer columns, floating point values will be converted using the rules of the Excel(tm) **INT** function. Parentheses can be used for grouping and to change evaluation order. The operator precedence rules follow those of standard arithmetic.

Operators

Currently supported operators are the four basic arithmetic operators and the **^** exponentiation operator. **+**, **-**, *, and \ are left-associative and **^** is right-associative. The string concatenation operator is **&**. Supported boolean or logical operators are the comparison operators **<**, **>**, **<=**, **>=**, **=**, and **<>** (not equal).

Supported Functions

Currently we support the following functions:

Cytoscape-specific functions

- Degree – the degree of a node.
- InDegree – the indegree of a node.
- OutDegree – the outdegree of a node.
- SourceID – the ID of the source node of an edge.
- TargetID – the ID of the target of an edge.

Numeric Functions

- Abs – Returns the absolute value of a number.
- ACos – Returns the arccosine of a number.
- ASin – Returns the arcsine of a number.
- ATan2 – Returns the arctangent of two numbers x and y.
- Average – Returns the average of a group of numbers.
- Cos – Returns the cosine of an angle given in radians.
- Cosh – Returns the hyperbolic sine of its argument.
- Count – Returns the number of numeric values in a list.
- Degrees – Returns its argument converted from radians to degrees.
- Exp – Returns e raised to a specified number.
- Ln – Returns the natural logarithm of a number.
- Log – Returns the logarithm of a number to a specified base.
- Max – Returns the maximum of a group of numbers.
- Median – Returns the median of a list of numbers.
- Min – Returns the minimum of a group of numbers.
- Mod – Calculates the modulus of a number.
- Pi – Returns an approximation of the value of p.
- Radians – Returns its argument converted from degrees to radians.
- Round – Rounds a number to a specified number of decimal places.
- Sin – Returns the sine of an angle given in radians.
- Sinh – Returns the hyperbolic sine of its argument.
- Sqrt – Calculates the square root of a number.
- Tan – returns the tangent of its argument in radians.
- Tanh – returns the hyperbolic tangent of its argument in radians.
- Trunc – Truncates a number.

String Functions

- Concatenate – Concatenates two or more pieces of text.
- Left – Returns a prefix of s string.
- Len – Returns the length of a string.
- Lower – Converts a string to lowercase.
- Mid – Selects a substring of some text.
- Right – Returns a suffix of a string.
- Substitute – Replaces some text with other text.
- Text – Format a number using the Java *DecimalFormat* class' conventions.
- Upper – Converts a string to uppercase.
- Value – Converts a string to a number.

Logical/Boolean Functions

- And – Returns the logical conjunction of any number of boolean values.
- Not – Returns the logical negation of a boolean value.
- Or – Returns the logical disjunction of any number of boolean values.

List Functions

- First – Returns the first entry in a list.
- Last – Returns the last entry in a list.
- Nth – Returns the n-th entry in a list.

Statistical Functions

- Largest – the kth largest value in a list.
- GeoMean – the geometric mean of a set of numbers.
- HarMean – the harmonic mean of a set of numbers.
- Mode – the mode of a set of numbers.
- NormDist – Returns the pdf or CDF of the normal distribution.
- Permut – Returns the number of permutations for a given number of objects.
- StDev **-** sample standard deviation.
- Var – sample variance.

Miscellaneous Functions

- Combin **-** Returns the number of combinations for a given number of objects.
- If – Returns one of two alternatives based on a boolean value.
- ListToString – Returns a string representation of a list.
- Now – Returns a string representation of the current date and time.
- Today – returns a string representation of the current date.

Pitfalls

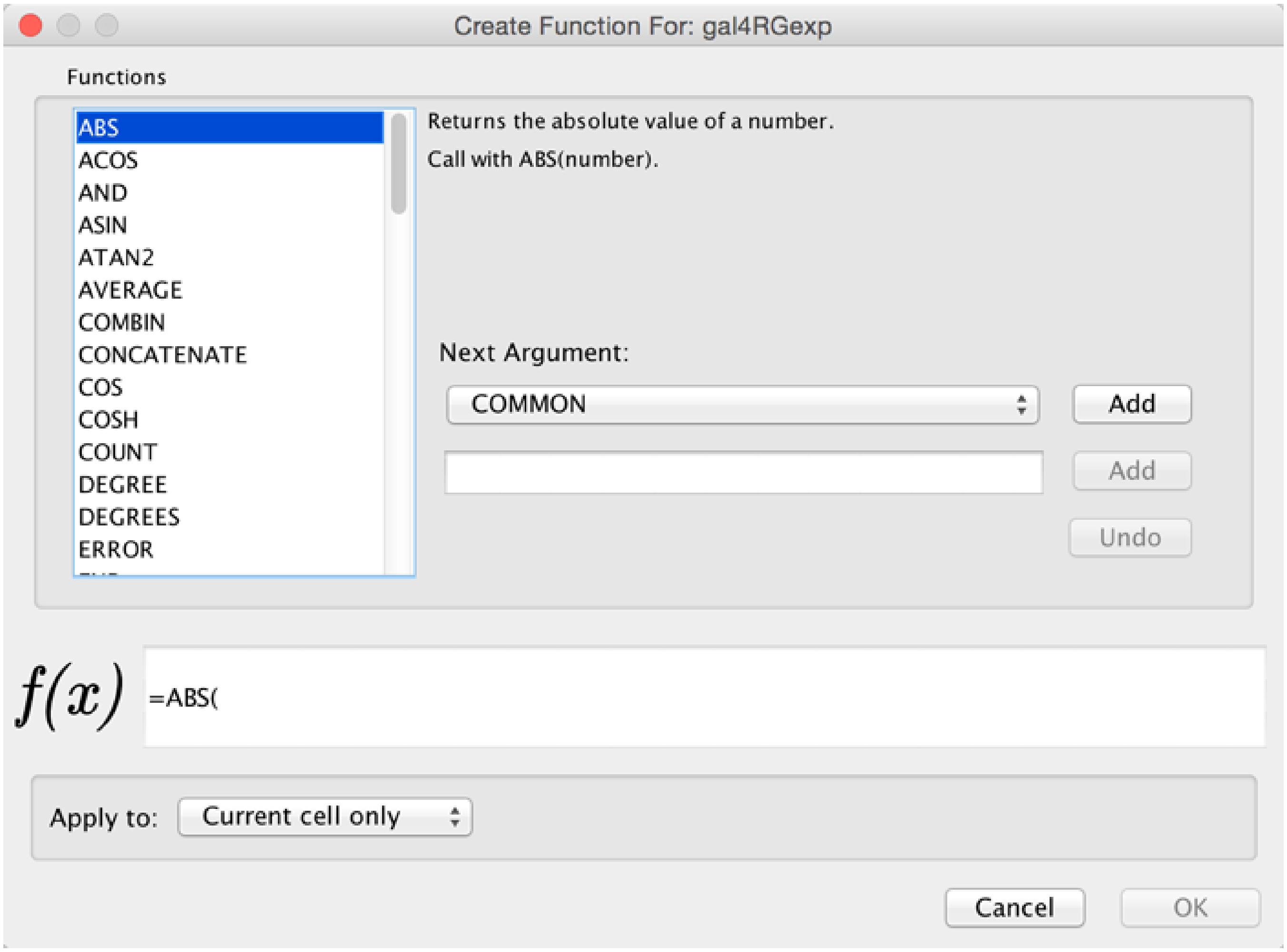
The possibly biggest problem is the referencing of other columns that have null values. This is not allowed and leads to errors. In order to mitigate this problem we support the following optional syntax for column references: **${columnName:defaultValue}**. The interpretation is that if **columnName** is null, then the default value will be used, otherwise the value of the referenced value will be used instead. The referenced column must still be a defined column and not an arbitrary name! The other potential problem is when there are circular column reference dependencies. Circular dependencies will be detected at formula evaluation time and lead to a run-time error.

Useful Tips

When working with formulas it can be very helpful to open the Developer's Log Console. Formula evaluation errors will be logged there.

The Formula Builder

In order to ease the creation of formulas as well as to facilitate discovery of built-in functions we provide a **Function Builder** in the Table Panel. After selecting a non-list column cell, you can invoke it by clicking on $f(x)$. This should bring up the Function Builder which looks like this:



Create Function For: gal4RGexp

Returns the absolute value of a number.

Call with ABS(number).

Next Argument: COMMON

$f(x)$ =ABS(

Apply to: Current cell only

Select a function on the left hand side of the dialog - here, we've selected the ABS function. Next to the list of functions, you can specify one or more arguments. This can either be a column (selected from the drop-down list) or a constant specified in the box below. If you select a column, the value of that column (in the row containing the formula) will be used, and the function result will be updated dynamically when that value changes. Click **Add** to add an argument - you can add one or more depending on how many arguments the function accepts. At the bottom of the dialog is a preview of the current formula. Under **Apply to**, you can select whether the formula will apply to the current cell only, the cell selection, or the entire column. Click OK when you are satisfied with the result, or Cancel to discard any changes.

The Function Builder is a useful tool for discovery of the list of built-in functions, which has the return type matching the data type of the column. Arguments can either be selected from a list of named columns, or constant values can be entered in a text entry field. A major shortcoming at this time is that the Formula Builder won't let you compose functions with function calls as arguments. If you need the most general functionality, please type the expression directly into a cell.

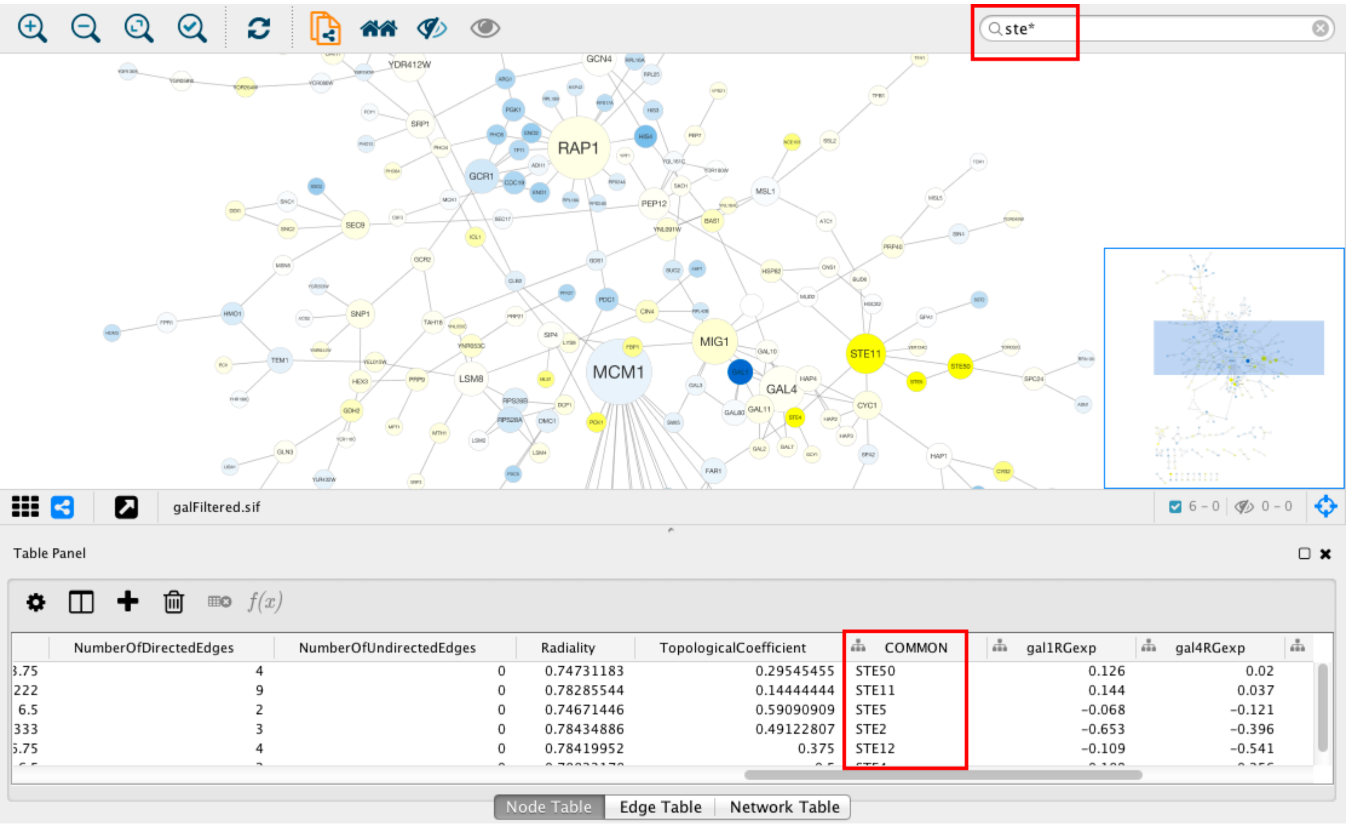
A Note for App Writers

It is relatively easy to add your own built-in formula functions. A simple function can probably be implemented in 15 to 20 minutes. It can then be registered via the parser and becomes immediately available to the user. It will of course also show up in the drop-down list in the Function Builder.

Finding and Filtering Nodes and Edges

Search Bar

You can search for nodes and edges by column value directly through Cytoscape's tool bar. For example, to select nodes or edges with a column value that starts with "STE", type `ste*` in the search bar. The search is case-insensitive. The `*` is a wildcard character that matches zero or more characters, while "?" matches exactly one character. So `ste?` would match "STE2" but would not match "STE12". Searching for `ste*` would match both.



	NumberOfDirectedEdges	NumberOfUndirectedEdges	Radiality	TopologicalCoefficient	COMMON	gal1RGexp	gal4RGexp
3.75	4	0	0.74731183	0.29545455	STE50	0.126	0.02
222	9	0	0.78285544	0.14444444	STE11	0.144	0.037
6.5	2	0	0.74671446	0.59090909	STE5	-0.068	-0.121
333	3	0	0.78434886	0.49122807	STE2	-0.653	-0.396
5.75	4	0	0.78419952	0.375	STE12	-0.109	-0.541

To search a specific column, you can prefix your search term with the column name followed by a `:`. For example, to select nodes and edges that have a "COMMON" column value that starts with "STE", use `common:ste*`. If you don't specify a particular column, all columns will be searched.

Columns with names that contain spaces, quotes, or characters other than letters and numbers currently do not work when searching a specific column. This will be fixed in a future release.

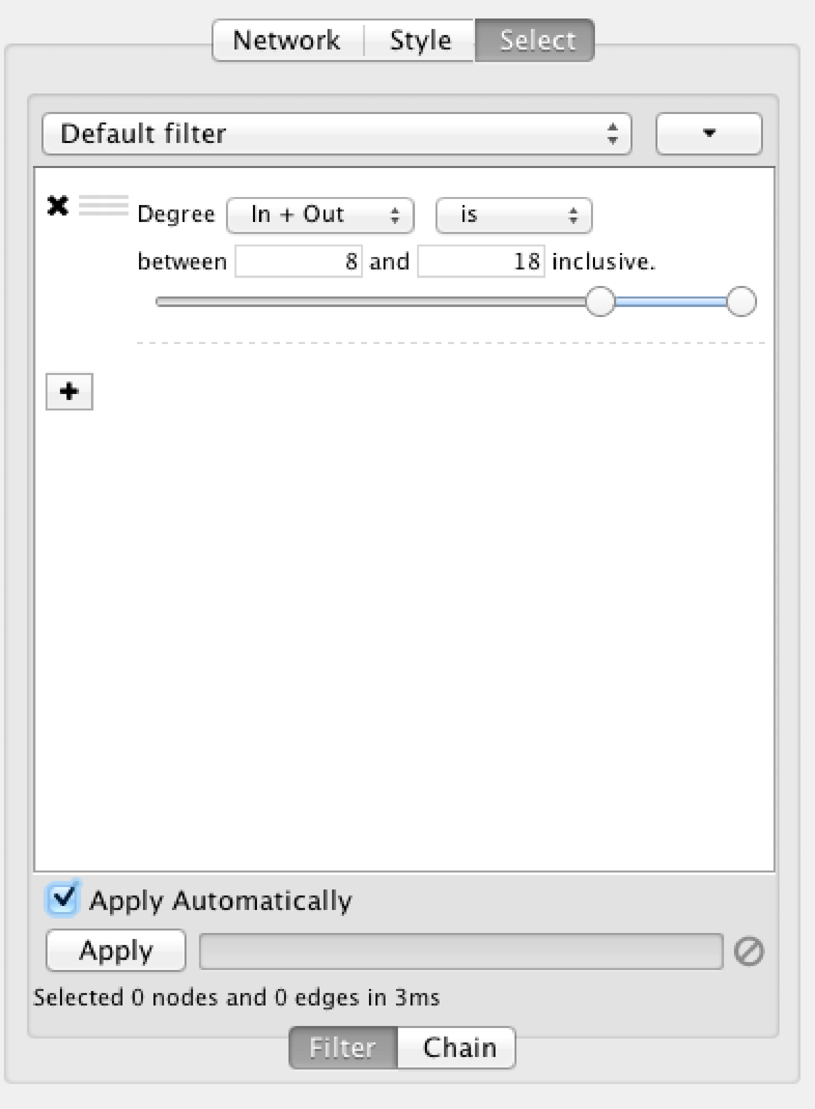
To search for column values that contain special characters you need to escape those characters using a "\". For example, to search for "GO:1232", use the query `GO\:1232`. The complete list of special characters is:

```
+ - & | ! ( ) { } [ ] ^ " ~ * ? : \
```

Note: Escaping characters only works when searching all columns. It currently does not work for column-specific searching. This will be fixed in a future release.

Filters

Cytoscape 3 provides a new user interface for filtering nodes and edges. These tools can be found in the **Select** panel:



Default filter

Degree In + Out is
between 8 and 18 inclusive.

☑ Apply Automatically

Selected 0 nodes and 0 edges in 3ms

There are two types of filters. On the **Filter** tab are *narrowing* filters, which can be combined into a tree. On the **Chain** tab are *chainable* filters, which can be combined in a linear chain.

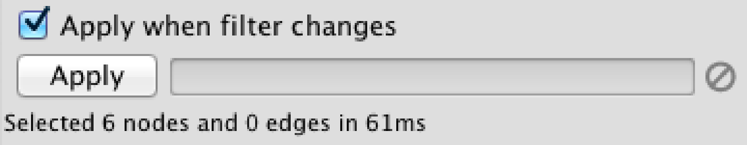
Narrowing Filters

Narrowing filters are applied to the entire network, and are used to select a subset of nodes or edges in a network based on user-specified constraints. For example, you can find edges with a weight between 0 and 5.5, or nodes with degree less than 3. A filter can contain an arbitrary number of sub-filters.

To add a filter click on the "+" button. To delete a filter (and all its sub-filters) click the "x" button. To move a filter grab the handle ≡ with the mouse and drag and drop the filter on its intended destination. Dropping a filter on top of another filter will group the filters into a composite filter.

Interactive Filter Application Mode

Due to the nature of narrowing filters, Cytoscape can apply them to a network efficiently and interactively. Some filters even provide slider controls to quickly explore different thresholds. This is the default behavior on smaller networks. For larger networks, Cytoscape automatically disables this interactivity. You can override this by manually checking the **Apply when filter changes** box above the **Apply** button:

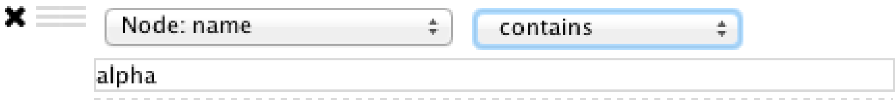

☑ Apply when filter changes
Apply
Selected 6 nodes and 0 edges in 61ms

Cytoscape comes packaged with the following narrowing filters:

Column Filter

This filter will match nodes or edges that have particular column values. For numeric columns sliders are provided to set minimum and maximum values, or the values may be entered manually.

From string columns, a variety of matching options are provided:



For example, column values can be checked to see if they contain or match exactly the text entered in the text box. More complex matching criteria can be specified by using a Java-style regular expression.

By default string matching is case insensitive. Case sensitive matching requires the use of a regular expression that starts with "(?-i)". For example to match the text "ABC" in a case sensitive way use the following regular expression: "(?-i)ABC".

Cytoscape uses Java regular expression syntax.

Degree Filter

The degree filter matches nodes with a degree that falls within the given minimum and maximum values, inclusive. You can choose whether the filter operates on the in-degree, out-degree or overall (in + out) degree.

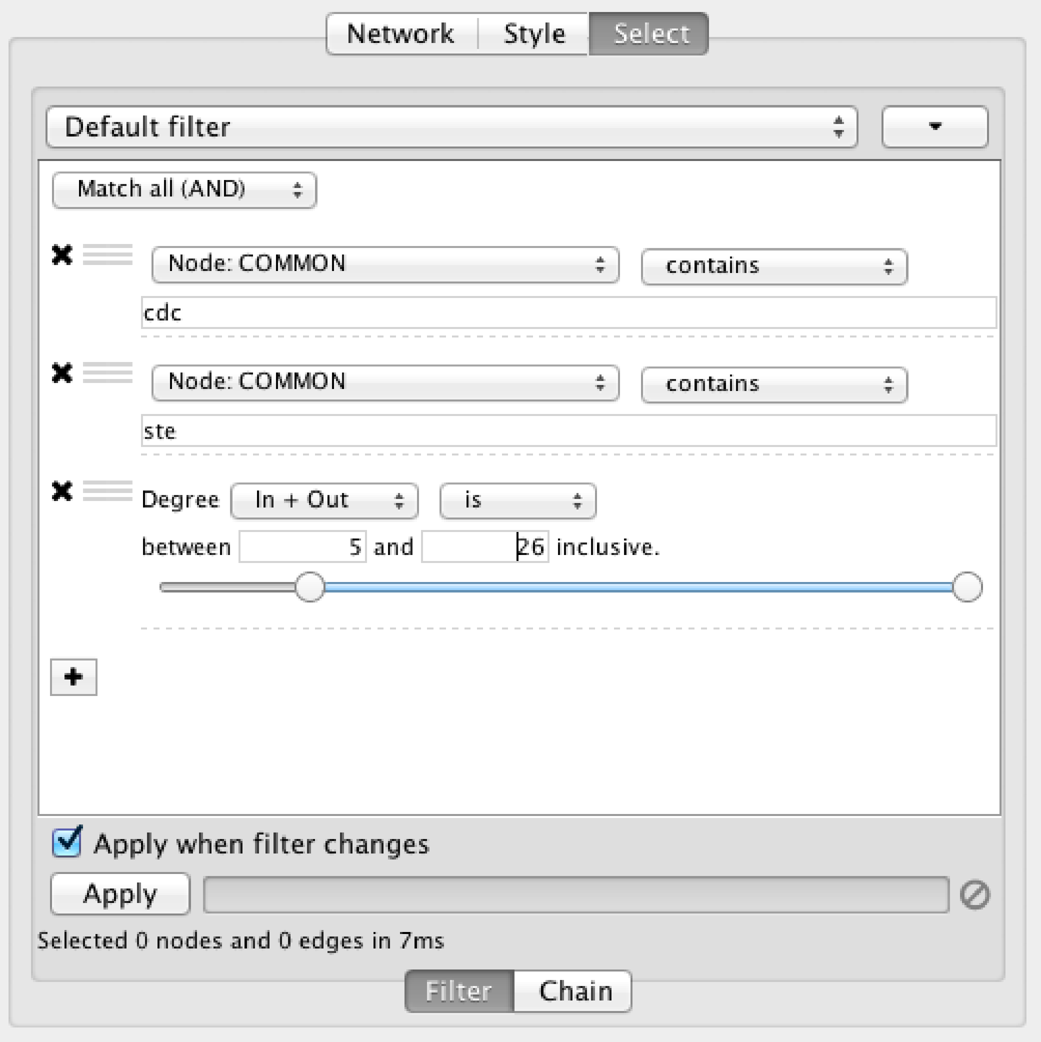
Topology Filter

The topology filter matches nodes having a certain number of neighbors which are within a fixed distance away, and which match a sub-filter. The thresholds for the neighborhood size and distance can be set independently, and the sub-filter is applied to each such neighbor node.

The topology filter will successfully match a node if the sub-filter matches against the required number of neighbor nodes.

Grouping and Organizing Filters

By default, nodes and edges need to satisfy the constraints of all your filters. You can change this so that instead, only the constraints of at least one filter needs to be met in order to match a node or edge. This behavior is controlled by the **Match all/any** drop-down box. This appears once your filter has more than one sub-filter. For example, suppose you wanted to match nodes with column *COMMON* containing `ste` or `cdc`, but you only want nodes with degree 5 or more, you'd first construct a filter that looks like this:



Default filter

Match all (AND)

Node: COMMON contains

cdc

Node: COMMON contains

ste

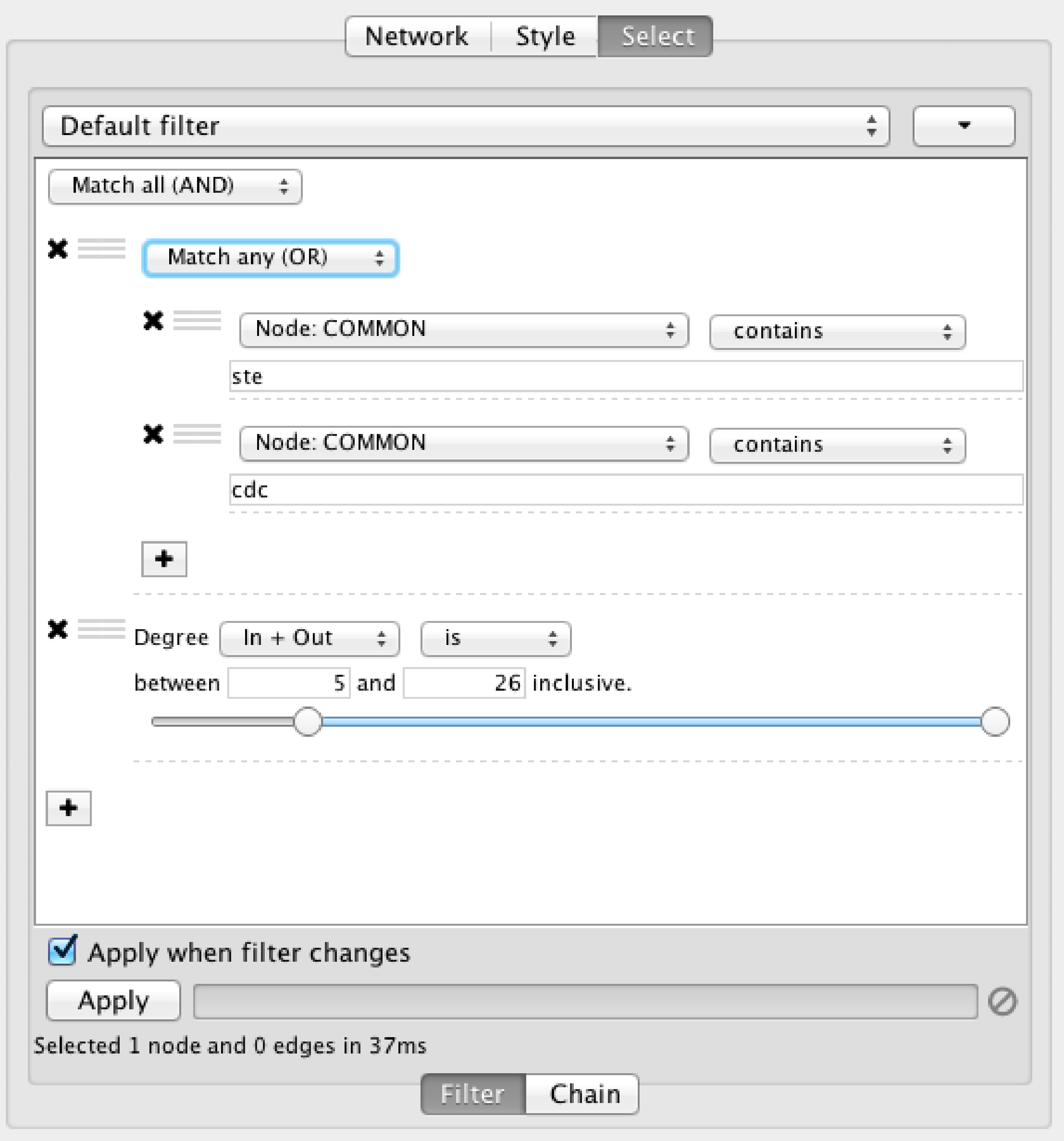
Degree In + Out is

between 5 and 26 inclusive.

☑ Apply when filter changes

Selected 0 nodes and 0 edges in 7ms

This filter will match nodes where **COMMON** contains `ste` **and** `cdc`. To change this to a logical **or** operation, drag either of the column filters by its handle ≡ onto the other column filter to create a new group. Now change the group’s matching behavior to **Match any**:



Default filter

Match all (AND)

Match any (OR)

Node: COMMON contains
ste

Node: COMMON contains
cdc

Degree In + Out is
between 5 and 26 inclusive.

☑ Apply when filter changes

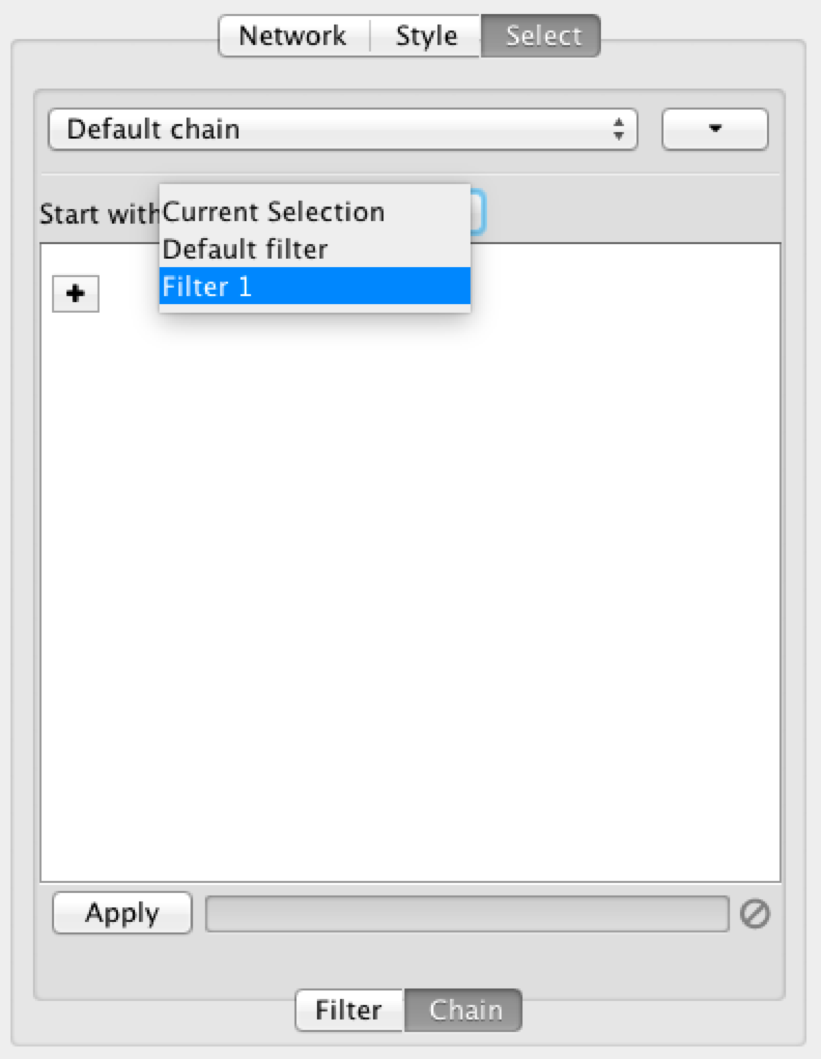
Selected 1 node and 0 edges in 37ms

You can also reorder filters by dropping them in-between existing filters.

Chainable Filters

Chainable filters are combined in an ordered list. The nodes and edges in the output of a filter become the input of the next filter in the chain. The first filter in the chain gets its input from the current selection or from a filter on the **Filter** tab. The output of the last filter becomes the new selection.

You can specify the input to the first filter in the chain by selecting a **Start with**, where **Current selection** refers to the nodes and edges currently selected. You can also choose a narrowing filter, which produces a different set of selected nodes and edges.



Chainable filters can be reordered by dragging one by the handle and dropping it between existing filters.

Cytoscape currently bundles the following chainable filters:

Edge Interaction Transformer

This transformer will go through all the input edges and selectively add their source nodes, target nodes, or both, to the output. This is useful for adding nodes that are connected to edges that match a particular filter.

Output options:

- Add (default): Automatically includes all input nodes and edges in the output, and adds source or target nodes from input edges to the output.
- Replace with: Does not automatically include input nodes and edges in the output. Only outputs nodes that match the filter.

A sub-filter may be added as well. When a sub-filter is present the source/target nodes must match the filter to be included in the output.

Node Adjacency Transformer

This transformer is used to add nodes and edges that are adjacent to the input nodes. A sub-filter may be specified as well.

Note that pressing the **Apply** button repeatedly may cause the selection to continuously expand. This allows adjacent nodes that are at greater distances to be added.

Output options:

- Add (default): Automatically includes all input nodes and edges in the output, and adds selected adjacent nodes and edges.
- Replace with: Only outputs the adjacent nodes/edges.

Select options:

- Adjacent nodes: Output nodes that are adjacent to the input nodes.
- Adjacent edges: Output edges that are adjacent (incident) to the input nodes.
- Adjacent nodes and edges (default): Output both nodes and edges that are adjacent to the input nodes.

Edge direction options. (Hidden by default, click the small arrow icon to reveal.):

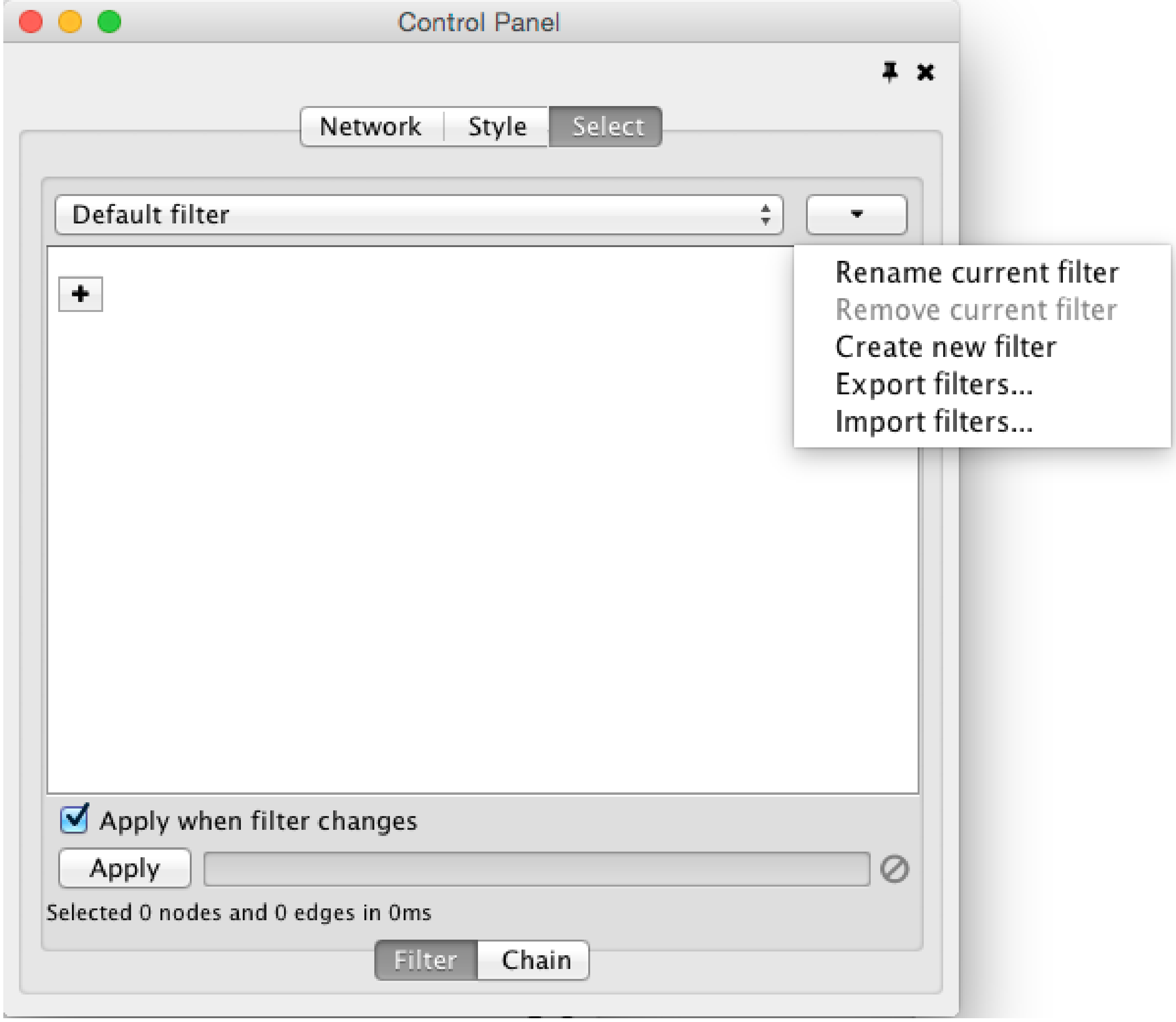
- Incoming: Only include adjacent nodes/edges when the adjacent edge is incoming.
- Outgoing: Only include adjacent nodes/edges when the adjacent edge is outgoing.
- Incoming and Outgoing (default): Ignore the directionality of adjacent edges.

Sub-filter options. (Available when a sub-filter has been added.):

- Adjacent nodes (default): The sub-filter is only applied to adjacent nodes. (Edges to the adjacent nodes are still included in the output.)
- Adjacent edges: The sub-filter is only applied to adjacent edges. (Nodes connected to the adjacent edges are still included in the output.)
- Adjacent nodes and edges: Both the adjacent edge and its connected node must match the filter. Note that for a filter to match an edge and a node at the same time it should be a compound filter that is set to “Match any (OR)”.

Working with Narrowing and Chainable Filters

The name of active filter appears in the drop-down box at the top of **Select** panel. Beside this is the options button which will allow you to rename, remove or export the active filter. It also lets you create a new filter, or import filters.



Selected 0 nodes and 0 edges in 0ms

At the bottom of the **Select** panel, there is an **Apply** button that will re-apply the active filter. On the opposite side of the progress bar is the cancel button, which will let you interrupt a long-running filter.

The Select Menu

The **Select** → **Nodes** and **Select** → **Edges** menus provide several mechanisms for selecting nodes and edges. Most options are fairly straightforward; however, some need extra explanation.

Select → **Nodes** → **From ID List File...** selects nodes based on node identifiers found in a specified file. The file format is simply one node id per line:

```
Node1
Node2
Node3
...
```

Navigation and Layout

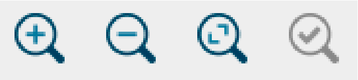
Basic Network Navigation

Cytoscape uses a Zoomable User Interface for navigating and viewing networks. ZUIs use two mechanisms for navigation: zooming and panning. Zooming increases or decreases the magnification of a view based on how much or how little a user wants to see. Panning allows users to move the focus of a screen to different parts of a view.

Zoom

Cytoscape provides four mechanisms for zooming: toolbar buttons, menu options, keyboard shortcuts and the scroll wheel.

Use the zooming buttons located on the toolbar to zoom in and out of the interaction network shown in the current network display. Zoom icons are detailed below:



From Left to Right:

- **Zoom In**
- Menu option: **View** → **Zoom In**
- Keyboard shortcut: `Ctrl-Plus` (`Command-Plus` on Mac OS X)

- **Zoom Out**
- Menu option: **View** → **Zoom Out**
- Keyboard shortcut: `Ctrl-Minus` (`Command-Minus` on Mac OS X)

- **Zoom Out to Display all of Current Network**
- Menu option: **View** → **Fit Content**
- Keyboard shortcut: `Ctrl-0` (`Command-0` on Mac OS X)

- **Zoom Selected Region**

- Menu option: **View** → **Fit Selected**
- Keyboard shortcut: `Ctrl-9` (`Command-9` on Mac OS X)

Using the scroll wheel, you can zoom in by scrolling up and zoom out by scrolling downwards. These directions are reversed on Macs with natural scrolling enabled (the default for Mac OS X Lion and newer versions).

Pan

There are two ways to pan the network:

- Left-Click and Drag - You can pan the network view by holding down the left mouse button and moving the mouse.
- Dragging Box on **Network Overview** - You can also pan the view by left-clicking and dragging the blue box in the overview panel in the lower part of the view.

Other Mouse Behaviors

Select

- Click the left mouse button on a node or edge to select that element.
- Hold down the `Shift` or `Ctrl` key (`Command` on Macs) and left-click a node or edge to add it to the selection. Doing the same on a selected element unselects it.
- Hold down the left mouse button on the canvas background and drag the mouse while holding down the `Shift` or `Ctrl` key (`Command` on Macs) to select groups of nodes/edges.

Context

Click the right mouse button (or Ctrl+left mouse button on Macs) on a node/edge to launch a context-sensitive menu with additional information about the node/edge.

Node Context Menu

This menu can change based on the current context. For nodes, it typically shows:

- Add
- Edit
- Select
- Group
- Nested Networks

- Apps
- External Links
- Preferences

Edges usually have the following menu:

- Edit
- Select
- Apps
- External Links
- Preferences

Apps can contribute their own items into node and edge context menus. These additions usually appear in the **Apps** section of the context menu.

Nested Network Node Context Menu

- **Add Nested Network**: Lets the user select any network in Cytoscape as the current node's nested network. If the current node already has a nested network it will be replaced.
- **Remove Nested Network**: Removes the currently associated nested network from a node. The associated network is not deleted. Only the association between the node and the network is removed.
- **Go to Nested Network**: The current node's nested network will be the current network view and have the focus. Should a network view for the nested network not exist, it will be created.

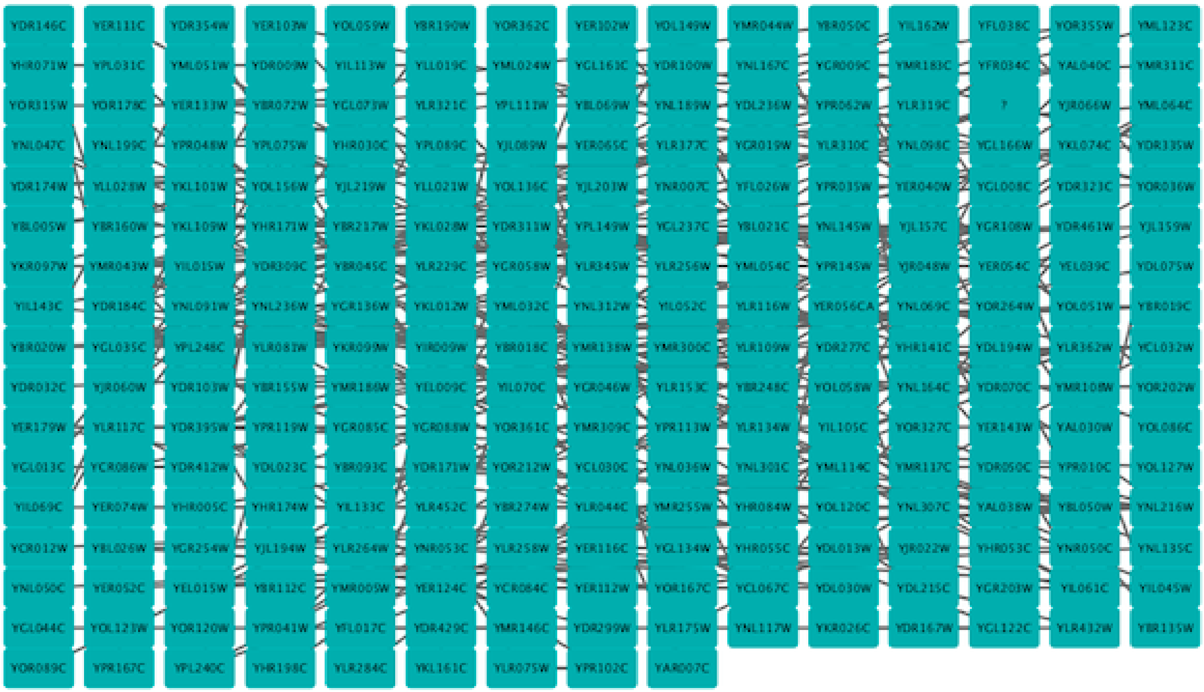
More information about nested networks can be found in the Nested Networks section.

Automatic Layout Algorithms

The Layout menu has an array of features for organizing the network visually according to one of several algorithms, aligning and rotating groups of nodes, and adjusting the size of the network. Cytoscape layouts have three different sources, which are reflected in the **Layout** menu.

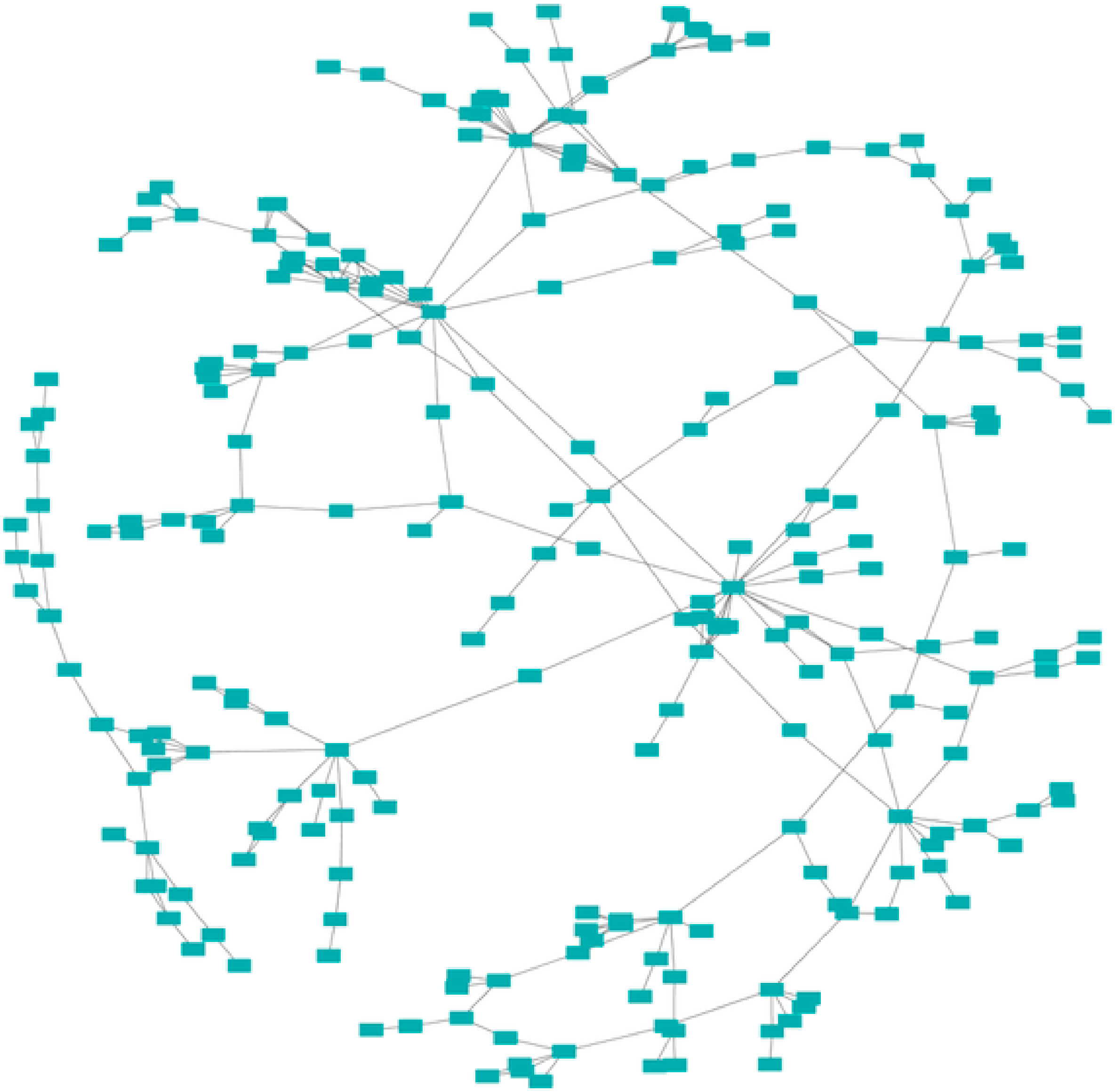
With the exception of the **yFiles** layouts (explained below), Cytoscape Layouts have the option to operate on only the selected nodes, and all provide a **Settings...** panel to change the parameters of the algorithm. Most of the Cytoscape layouts also partition the graph before performing the layout. In addition, many of these layouts include the option to take either node or edge columns into account. A few of the layout algorithms are:

Grid Layout



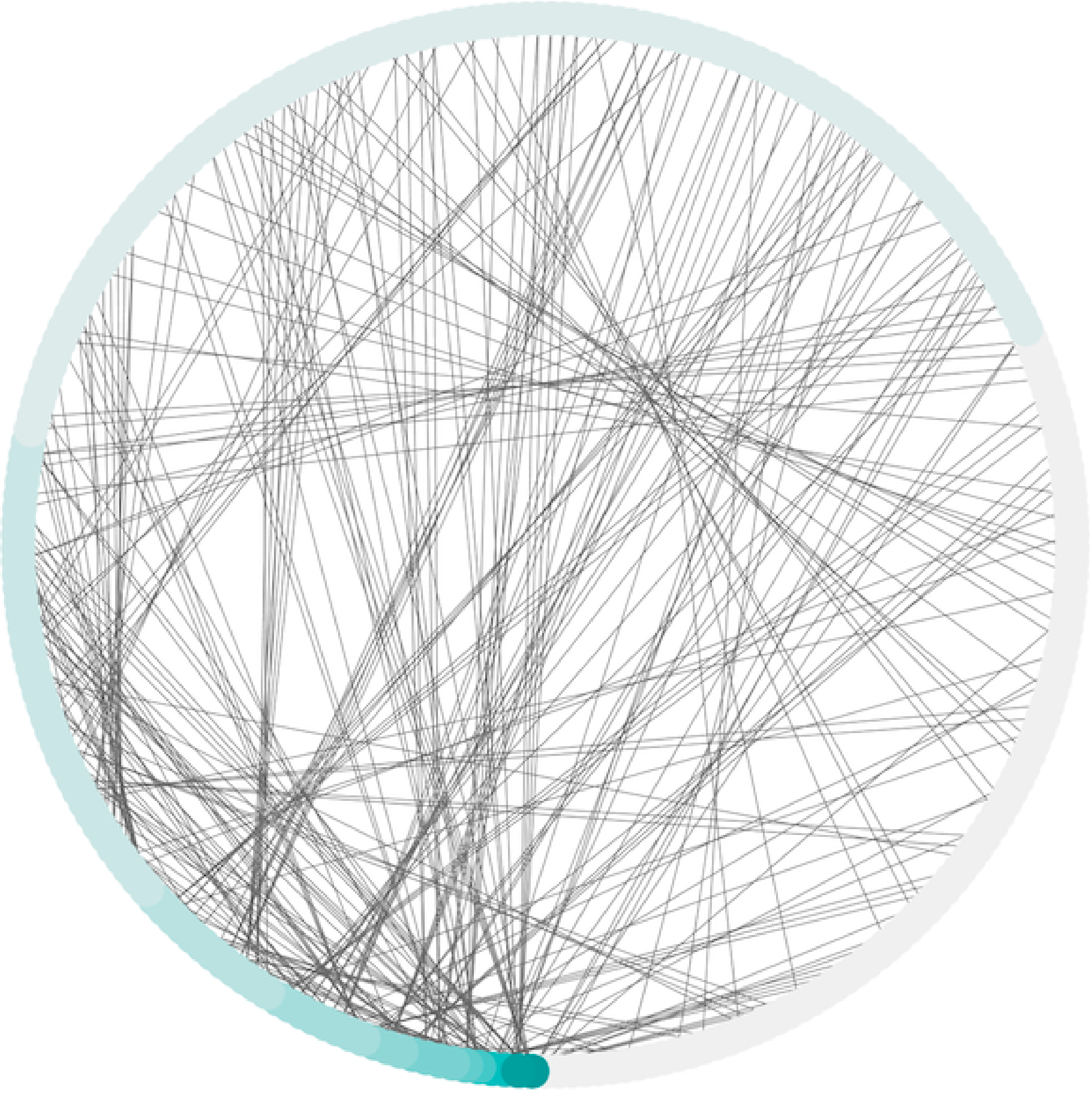
The grid layout is a simple layout the arranges all of the nodes in a square grid. This is the default layout and is always available as part of the Cytoscape core. It is available by selecting **Layout** → **Grid Layout**. A sample screen shot is shown above.

Edge-weighted Spring-Embedded Layout



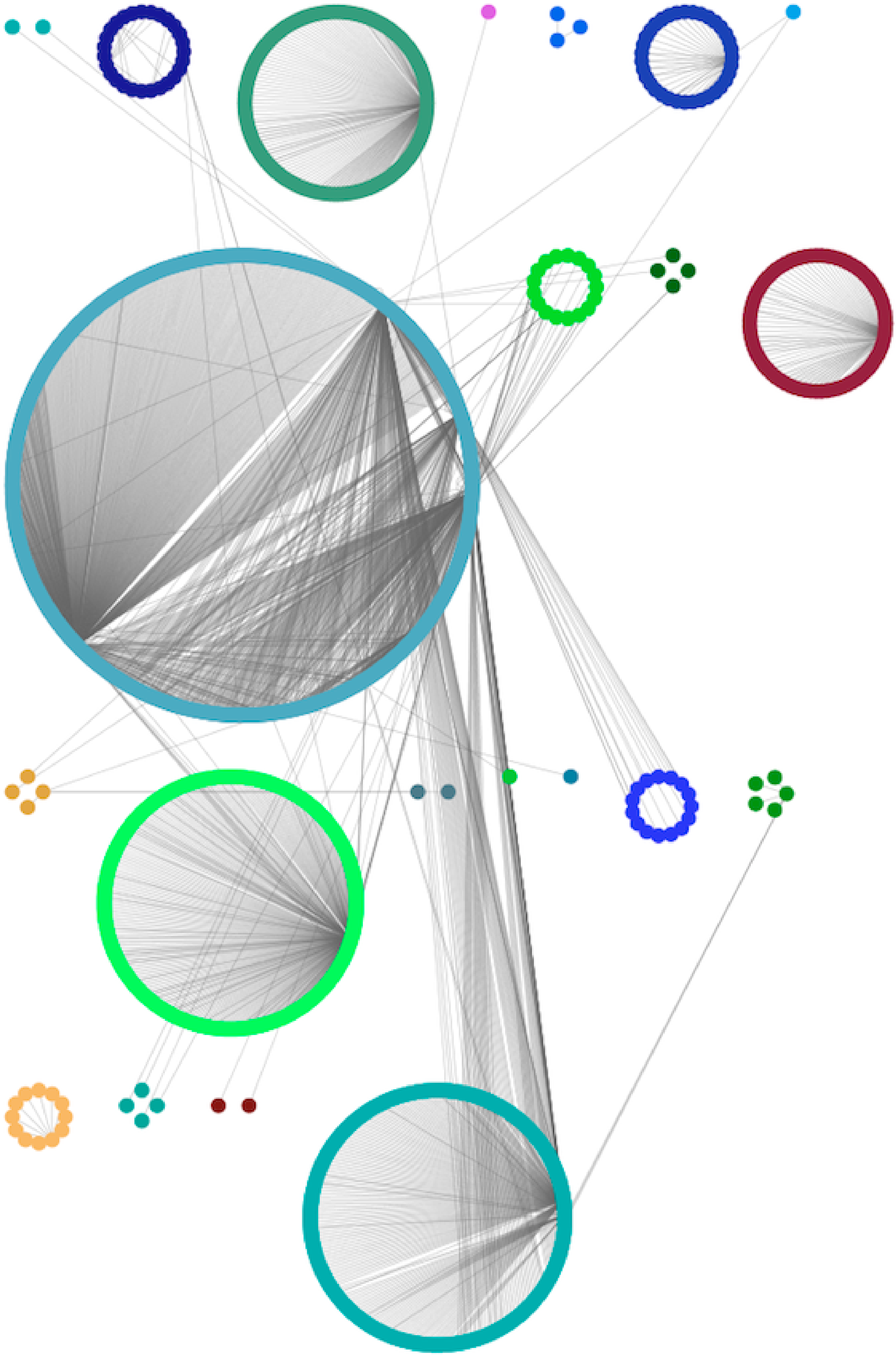
The spring-embedded layout is based on a “force-directed” paradigm as implemented by Kamada and Kawai (1988). Network nodes are treated like physical objects that repel each other, such as electrons. The connections between nodes are treated like metal springs attached to the pair of nodes. These springs repel or attract their end points according to a force function. The layout algorithm sets the positions of the nodes in a way that minimizes the sum of forces in the network. This algorithm can be applied to the entire network or a portion of it by selecting the appropriate options from **Layout** → **Edge-weighted Spring Embedded**.

Attribute Circle Layout



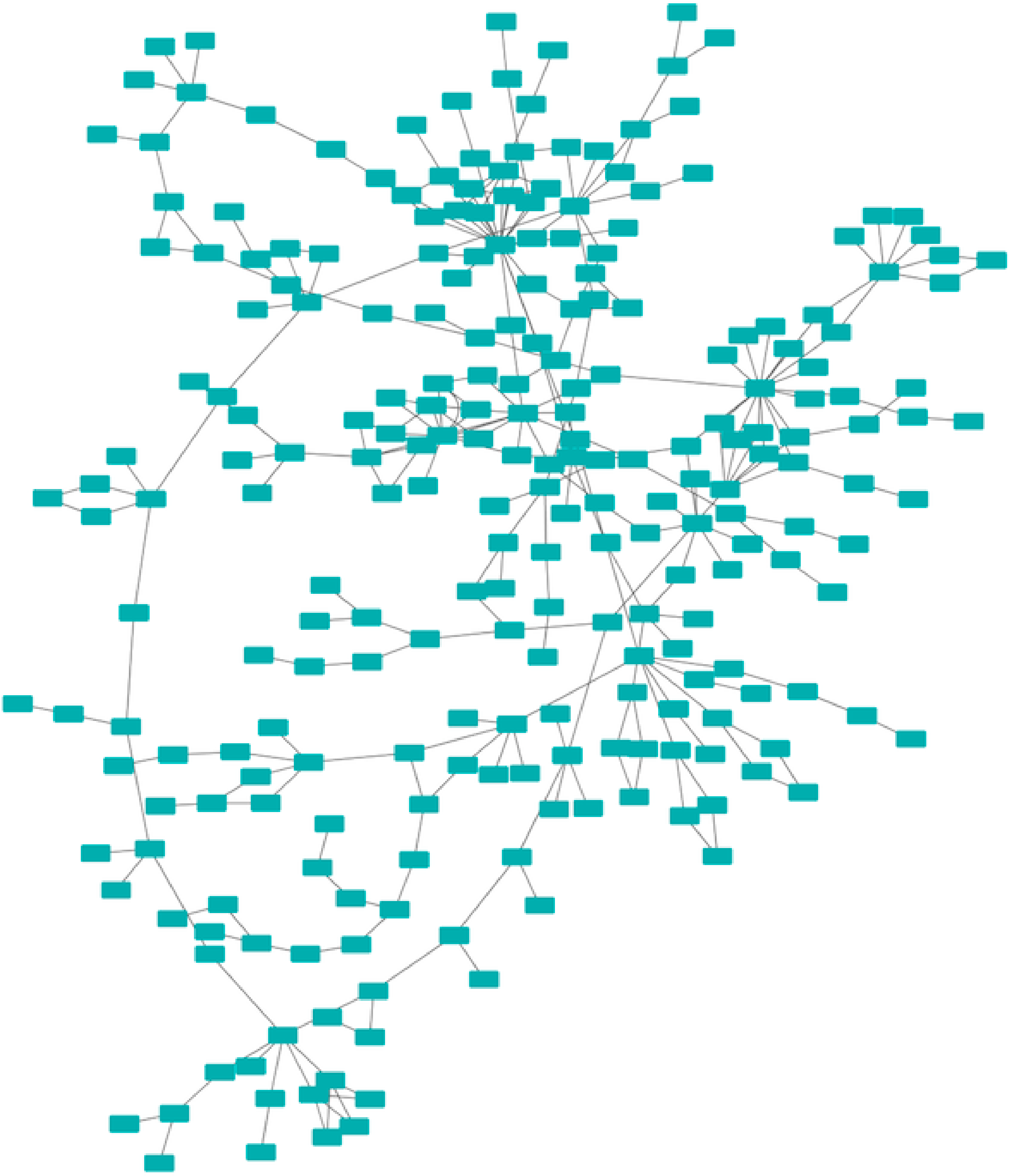
The **Attribute Circle** layout is a quick, useful layout, particularly for small networks, that will locate all of the nodes in the network around a circle. The node order is determined by a user-selected node column. The result is that all nodes with the same value for that column are located together around the circle. Using **Layout** → **Attribute Circle Layout** → ***column*** to put all nodes around a circle using ***column*** to position them. The sample screen shot above shows the a subset of the galFiltered network organized by node degree.

Group Attributes Layout



The **Group Attributes** layout is similar to the **Attribute Circle** layout described above except that instead of a single circle with all of the nodes, each set of nodes that share the same value for the column are laid out in a separate circle. The same network shown above (network generated by PSICQUIC Client) is shown above, using **Layout** → **Group Attributes Layout** → **taxonomy**.

Prefuse Force Directed Layout



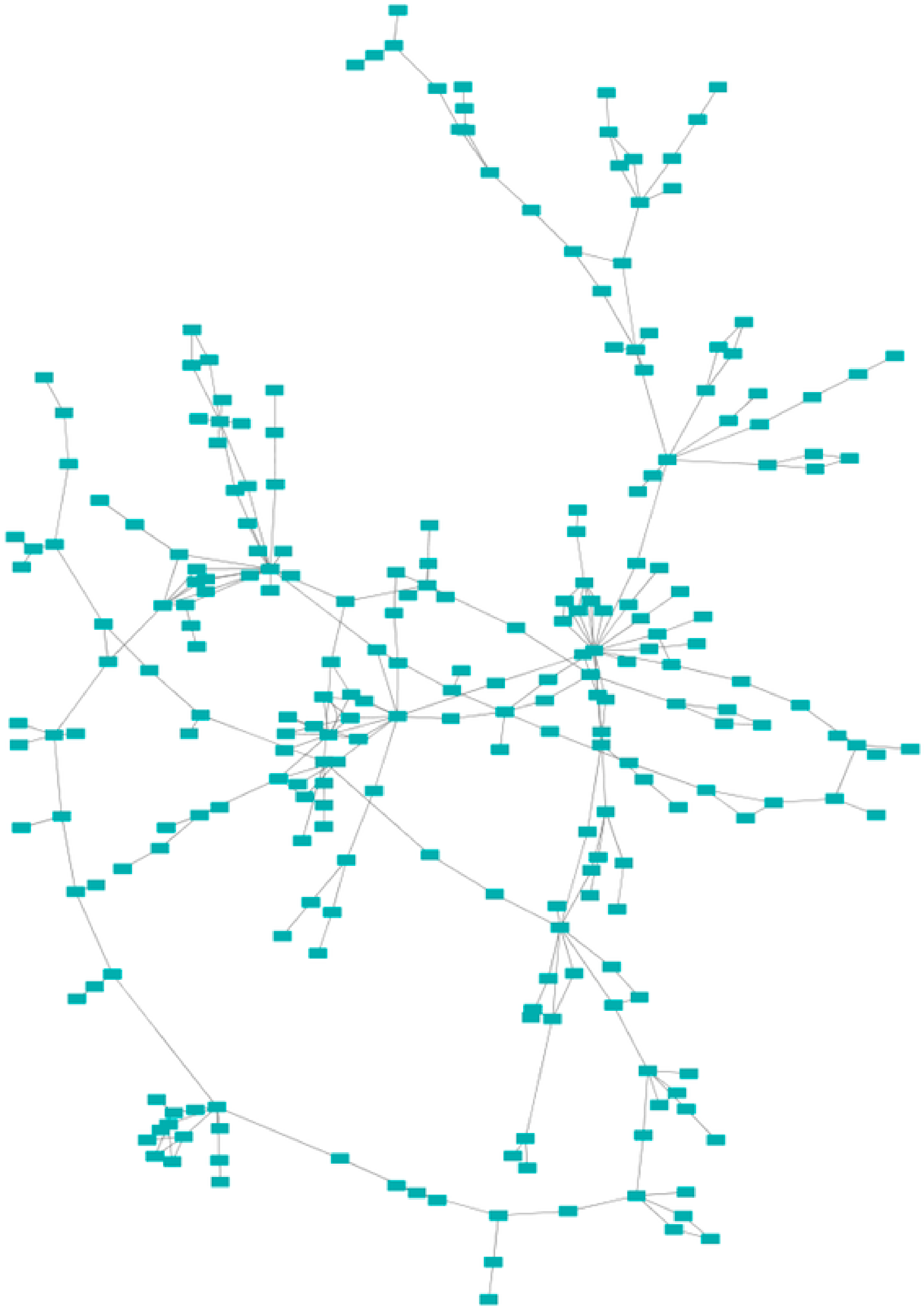
The force-directed layout is a layout based on the “force-directed” paradigm. This layout is based on the algorithm implemented as part of the prefuse toolkit (http://www.prefuse.org/) provided by Jeff Heer. The algorithm is very fast and with the right parameters can provide a very visually pleasing layout. The **Force Directed Layout** will also accept a numeric edge column to use as a weight for the length of the spring, although this will often require more use of the **Settings...** dialog to achieve the best layout. This algorithm is available by selecting

Layout → **Prefuse Force-Directed Layout** → **(unweighted)** or the edge column you want to use as a weight. A sample screen shot showing a portion of the galFiltered network provided in sample data is provided above.

yFiles Layouts

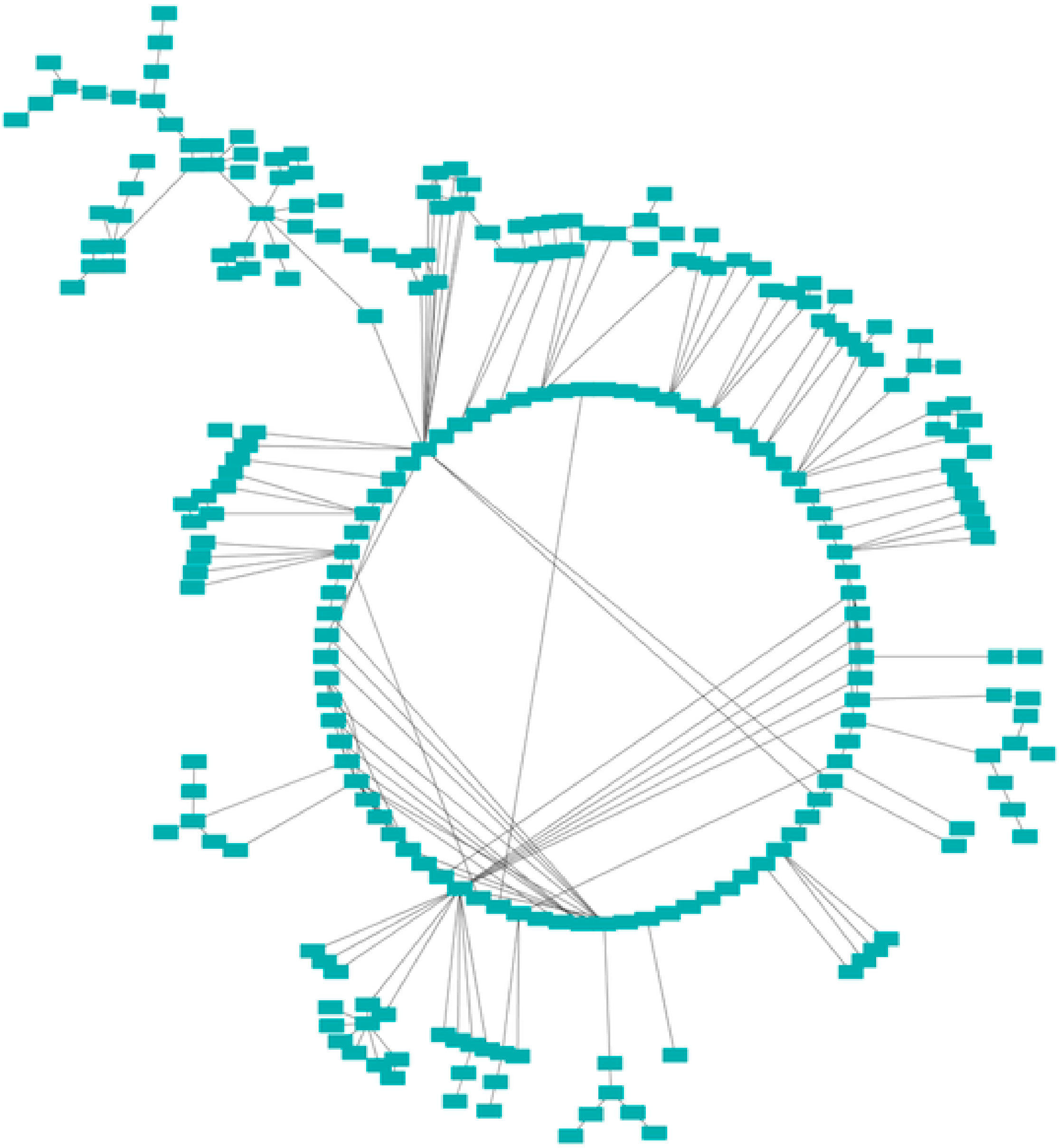
yFiles layouts are a set of commercial layout algorithms which are provided courtesy of yWorks (http://www.yworks.com). Due to license restrictions, the detailed parameters for these layouts are not available (there are no **yFiles** entries in the **Layout** → **Settings...**). The main layout algorithms provided by yFiles are:

yFiles Organic Layout



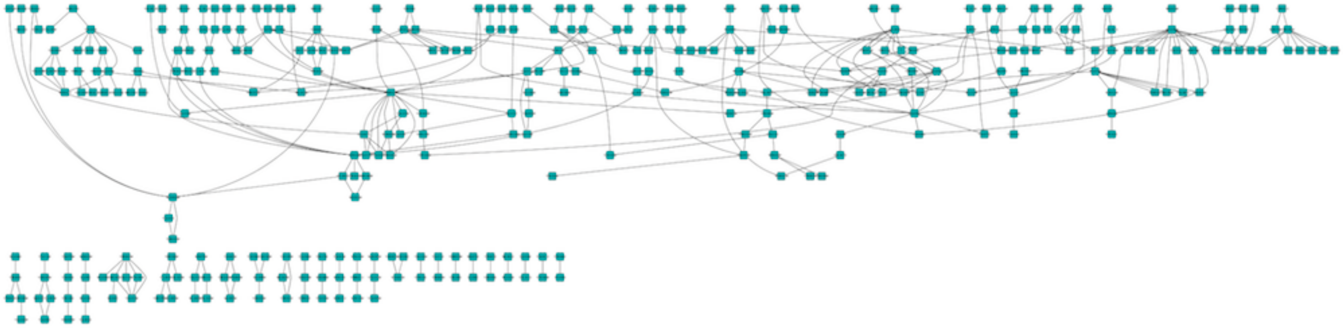
The organic layout algorithm is a kind of spring-embedded algorithm that combines elements of the other algorithms to show the clustered structure of a graph. This algorithm is available by selecting **Layout** → **yFiles Layouts** → **Organic**.

yFiles Circular Layout



This algorithm produces layouts that emphasize group and tree structures within a network. It partitions the network by analyzing its connectivity structure, and arranges the partitions as separate circles. The circles themselves are arranged in a radial tree layout fashion. This algorithm is available by selecting **Layout** → **yFiles Layouts** → **Circular**.

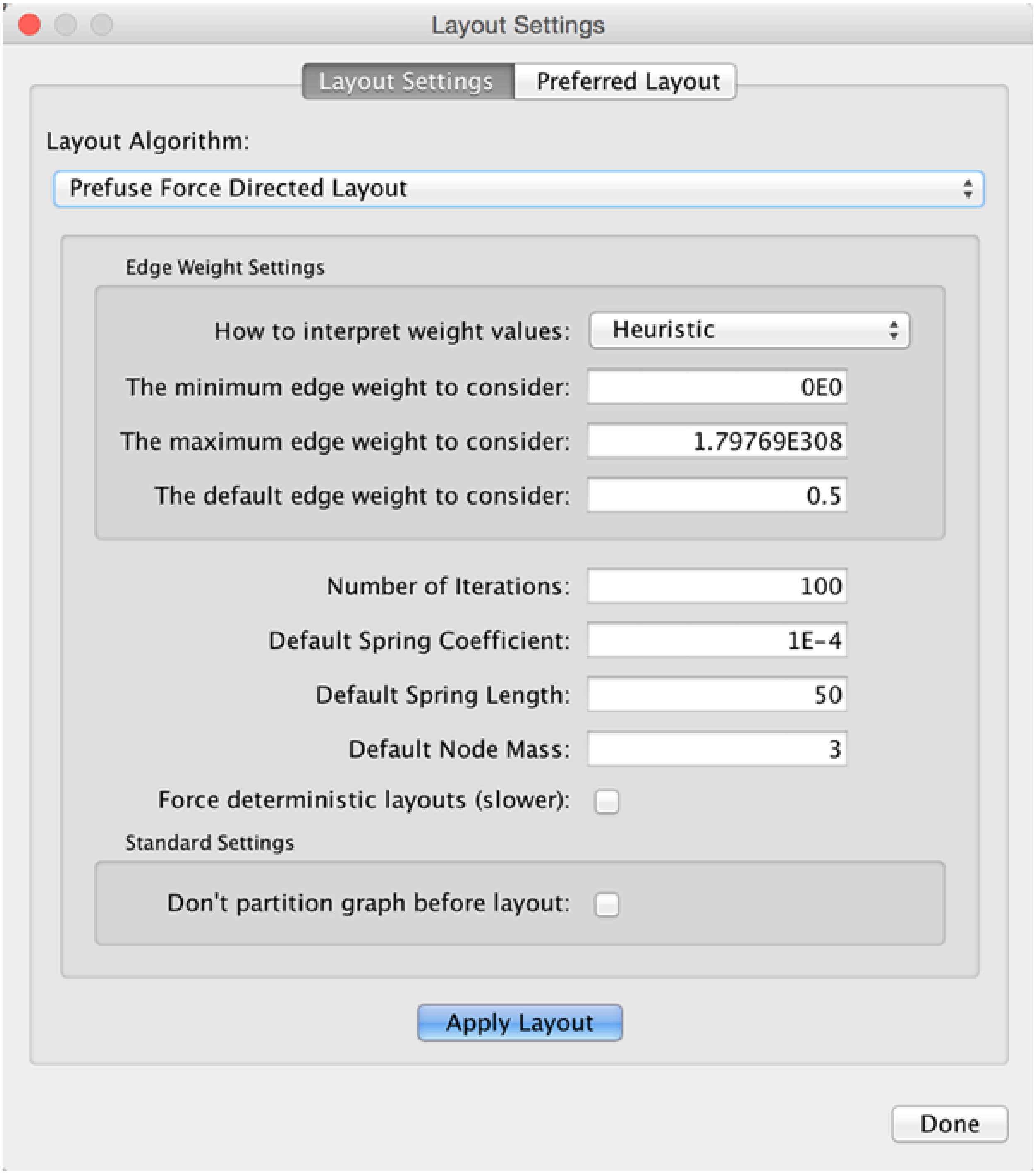
yFiles Hierarchical Layout



The hierarchical layout algorithm is good for representing main direction or “flow” within a network. Nodes are placed in hierarchically arranged layers and the ordering of the nodes within each layer is chosen in such a way that minimizes the number of edge crossings. This algorithm is available by selecting **Layout** → **yFiles Layouts** → **Hierarchical**.

Layout Parameters

Many layouts have adjustable parameters that are exposed through the **Layouts** → **Settings...** menu option. The **Layout Settings** dialog, which allows you to choose which layout algorithm settings to adjust, is shown below. The settings presented vary by algorithm and only those algorithms that allow access to their parameters will appear in the drop-down menu at the top of the dialog. Once you’ve modified a parameter, clicking the **Execute Layout** button will apply the layout.



Layout Algorithm:

Prefuse Force Directed Layout

Edge Weight Settings

How to interpret weight values: Heuristic

The minimum edge weight to consider: 0E0

The maximum edge weight to consider: 1.79769E308

The default edge weight to consider: 0.5

Number of Iterations: 100

Default Spring Coefficient: 1E-4

Default Spring Length: 50

Default Node Mass: 3

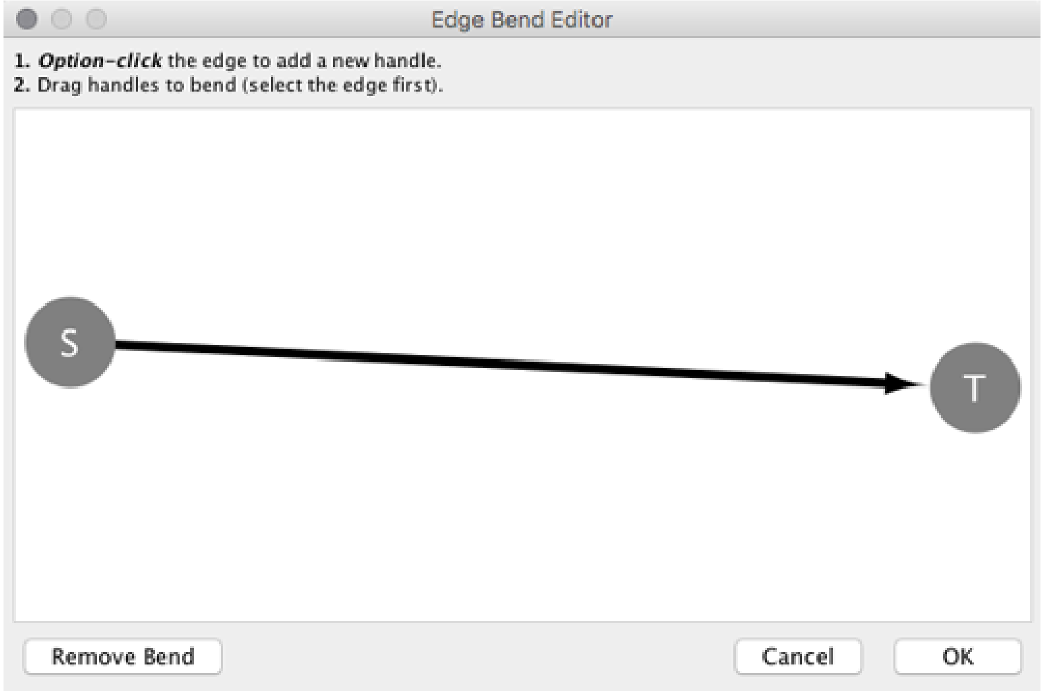
Force deterministic layouts (slower): □

Standard Settings

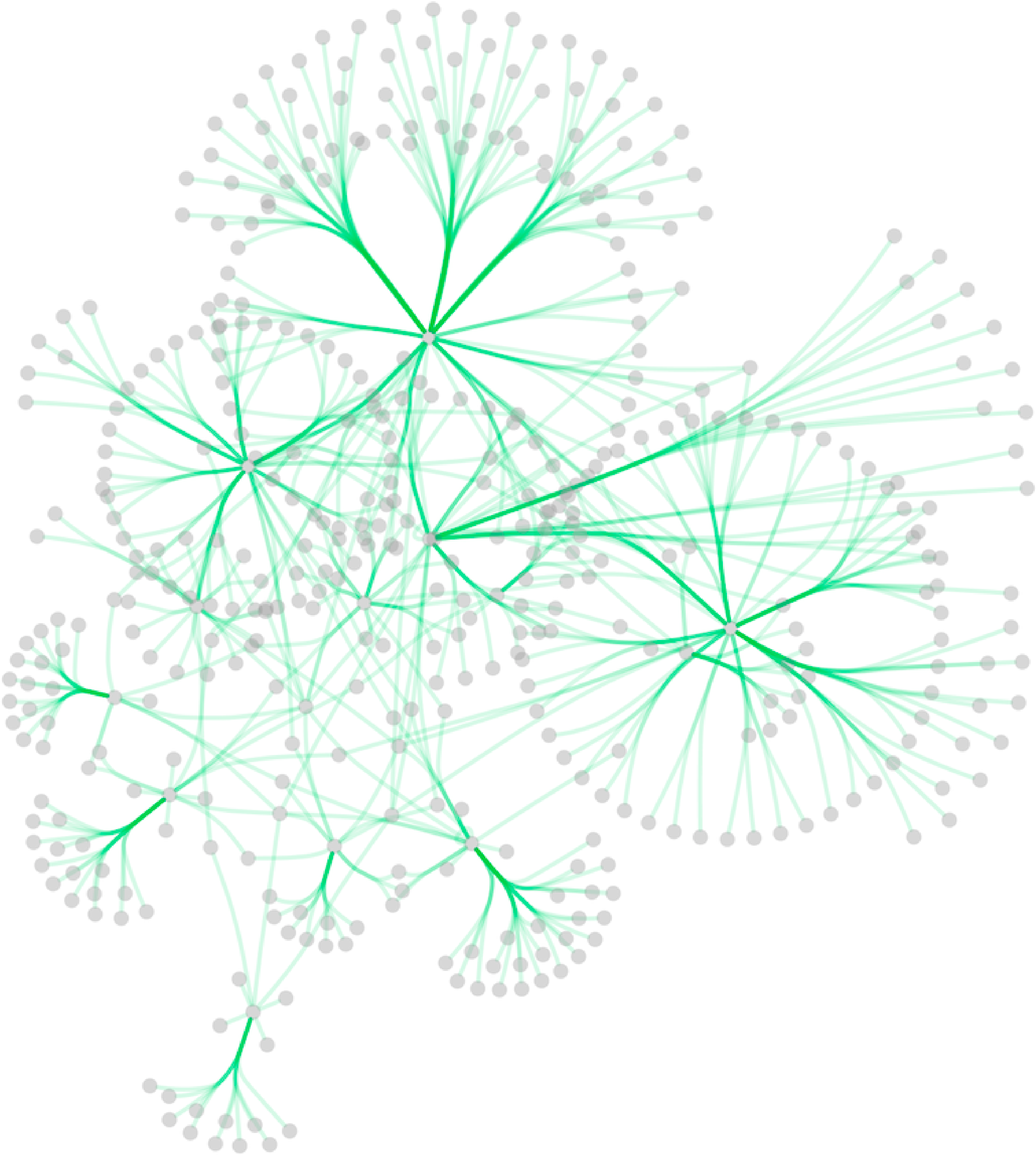
Don't partition graph before layout: □

Edge Bend and Automatic Edge Bundling

From Cytoscape 3.0, **Edge Bend** is a regular edge property and can be used as a part of a **Style**. Just like any other edge property, you can select a Default Value, a Mapping and use Bypass for select nodes. In the Styles tab, select the **Bend** property from the **Properties** drop-down and click on either the Default Value, Mapping or Bypass cell to bring up the **Edge Bend Editor**. In the editor, you can add as many handles as you want to the edge using Alt-Click on Windows, Option-Click on Mac, or Ctrl-Alt-Click on Linux.

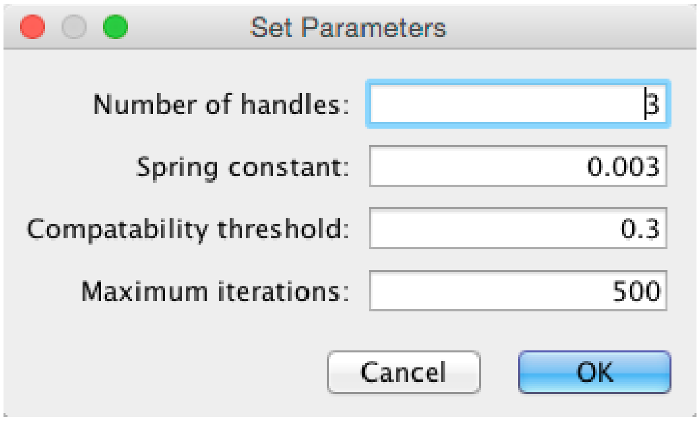


1. ***Option-click*** the edge to add a new handle.
2. Drag handles to bend (select the edge first).



To clear all edge bends, select **Layout** → **Clear All Edge Bends**.

In addition to adding handles manually, you can use the **Bundle Edges** function to bundle all or selected edges automatically.



Number of handles: 3
Spring constant: 0.003
Compatability threshold: 0.3
Maximum iterations: 500

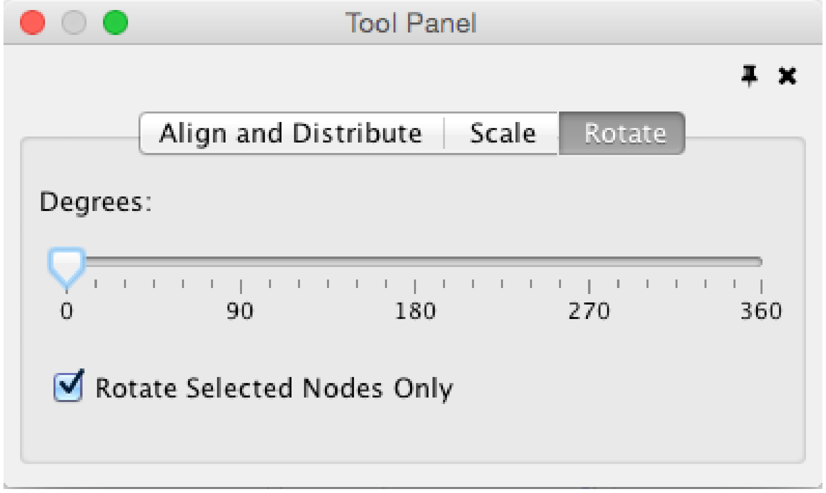
1. Select **Layout** → **Bundle Edges** → **All Nodes and Edges**.
2. Set parameters.
   - Details of the algorithm is described in this paper (http://www.win.tue.nl/~dholten/papers/forcebundles_eurovis.pdf).
3. Press OK to run. Edge bundling may take a long time if the number of edges is large.
   - If it takes too long, try decreasing **Maximum Iterations.**
   - For large, dense networks, try setting **Maximum iterations** in the range of 500 - 1000.

Note: The handle locations will be optimized for current location of nodes. If you move node positions, you need to run the function again to get proper result.

Manual Layout

The simplest method to manually organize a network is to click on a node and drag it. If you select multiple nodes, all of the selected nodes will be moved together.

Rotate

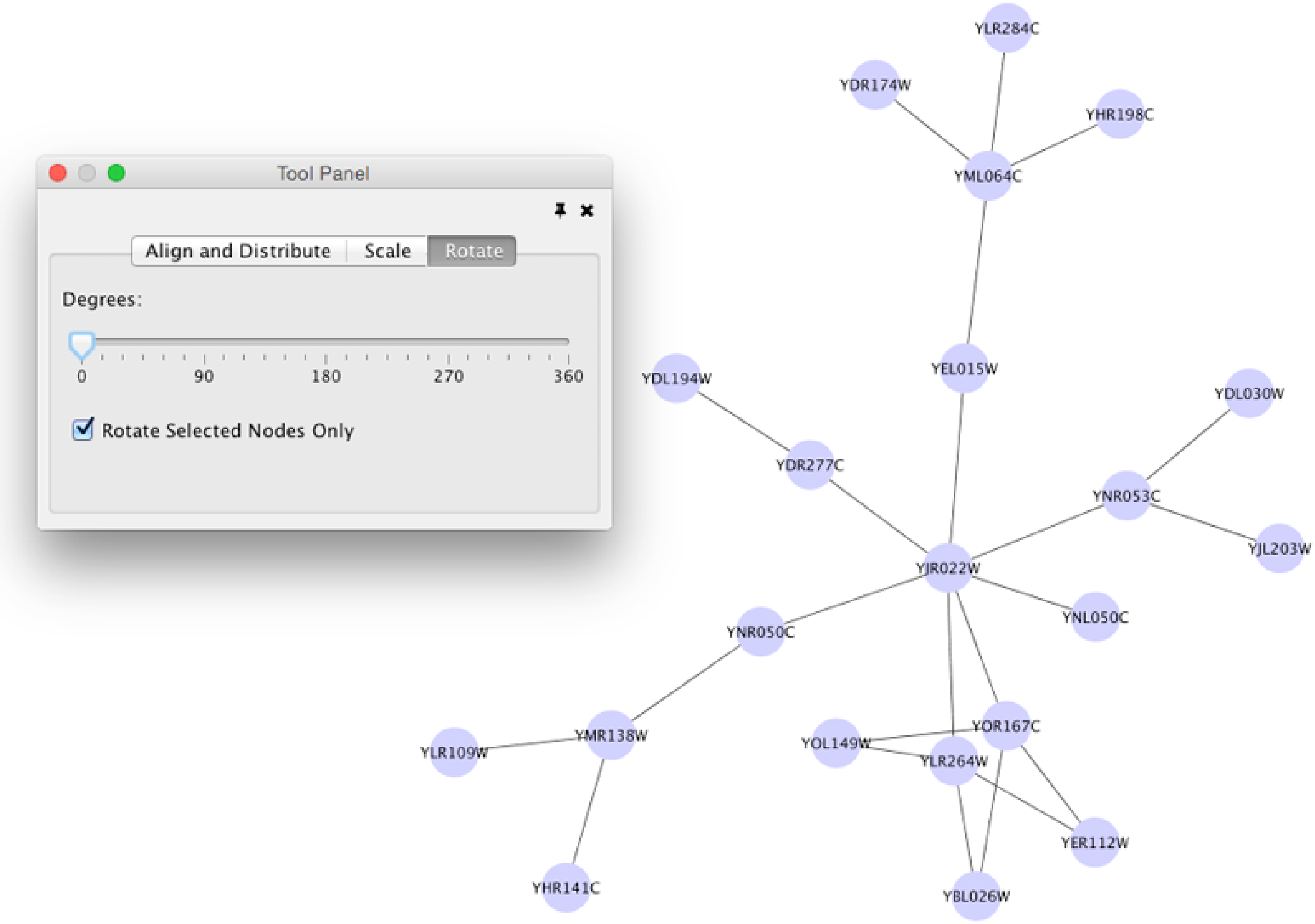


Degrees: 0

☑ Rotate Selected Nodes Only

Selecting the **Layout** → **Rotate** option will show the **Rotate** window in the **Tool Panel**. This function will either rotate the entire network or a selected portion of the network. The image below shows a network with selected nodes rotated.

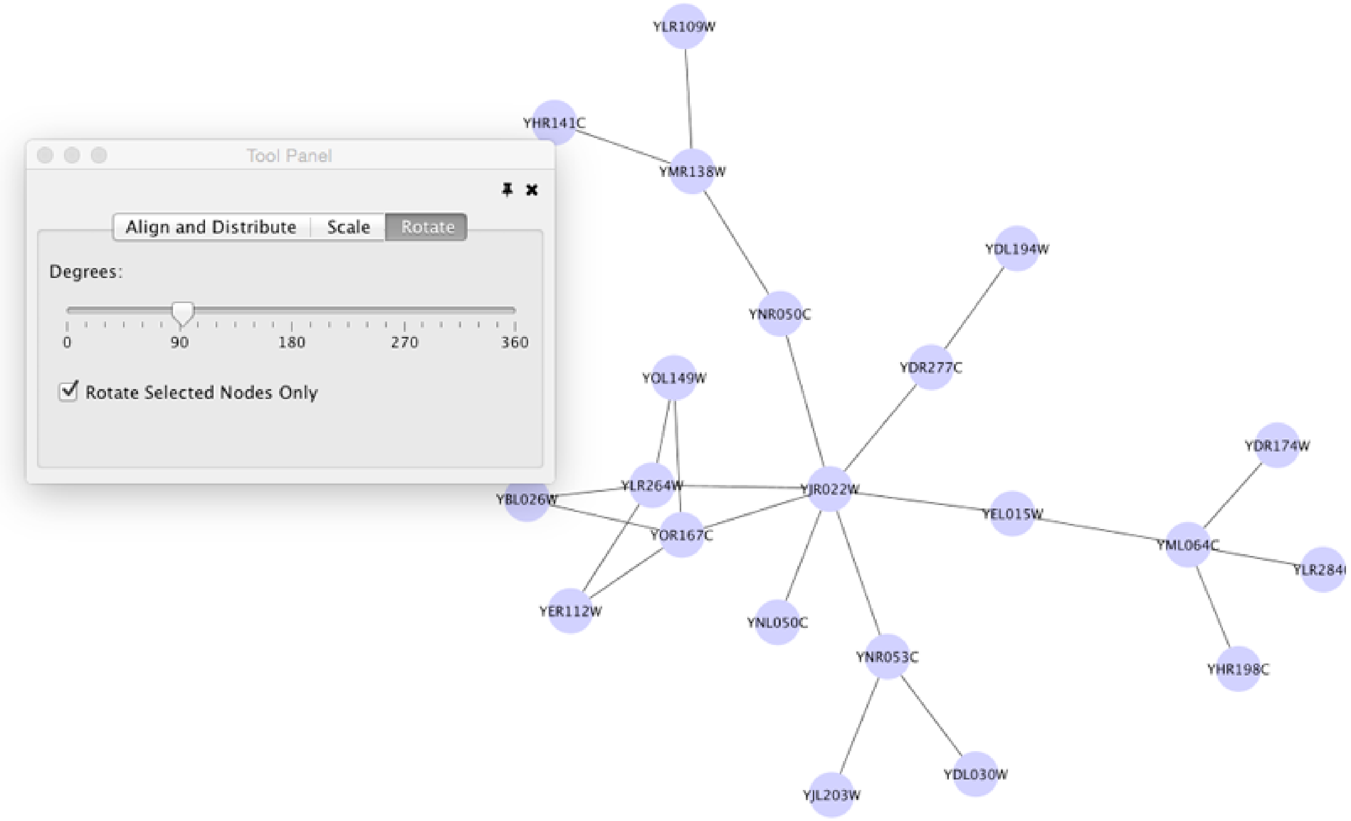
Before



Degrees:

☑ Rotate Selected Nodes Only

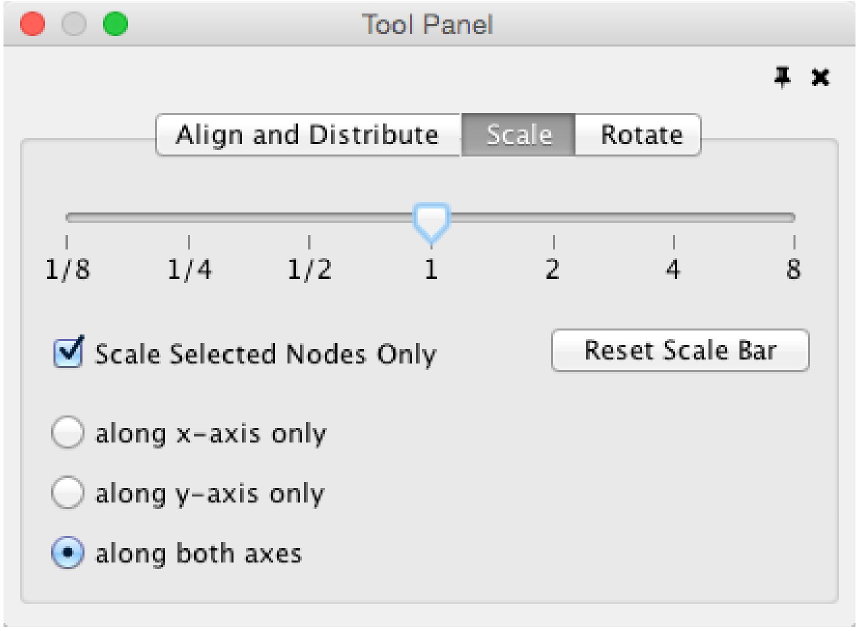
After



Degrees:

☑ Rotate Selected Nodes Only

Scale



☑ Scale Selected Nodes Only

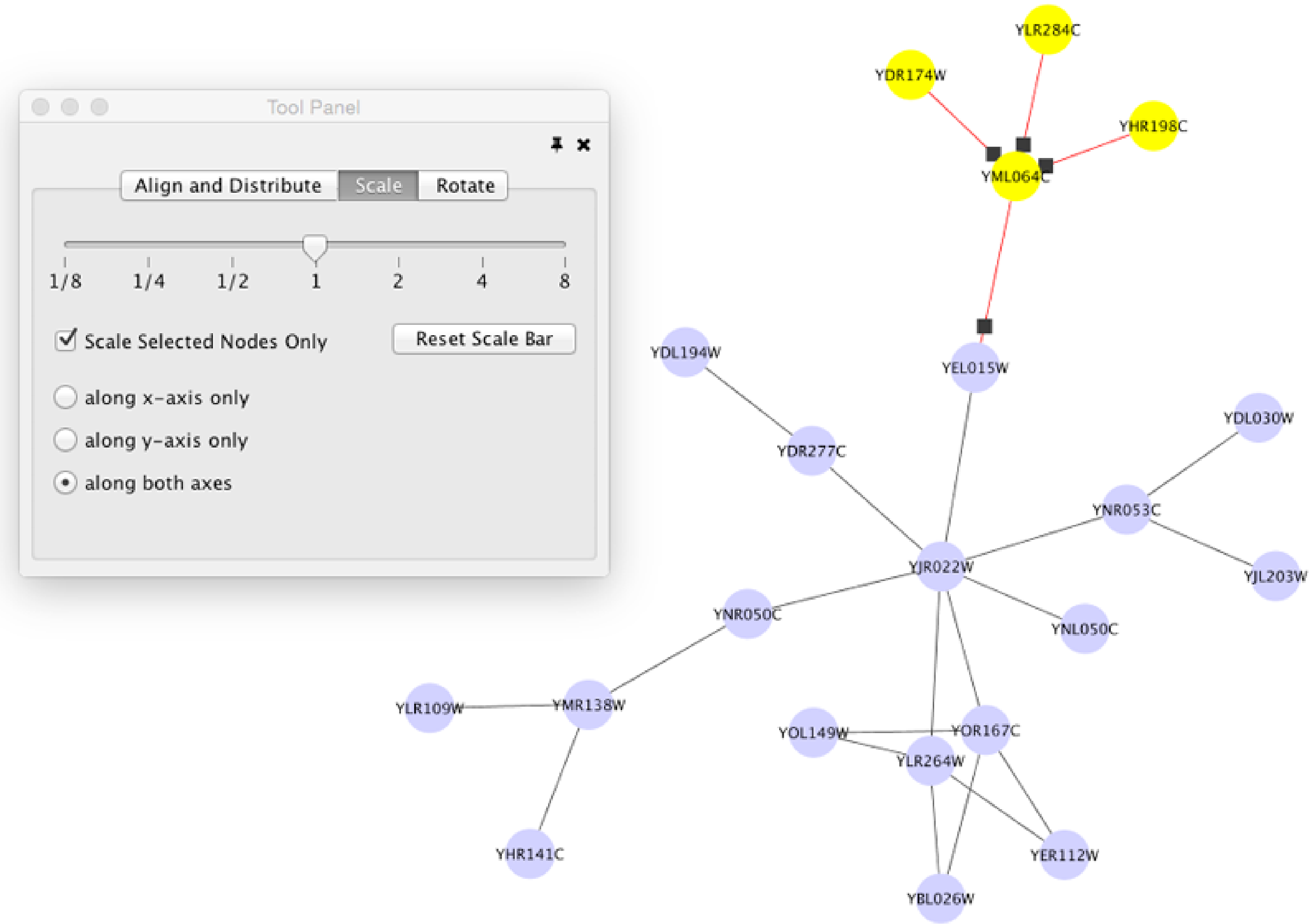
○ along x-axis only

○ along y-axis only

● along both axes

Selecting the **Layout** → **Scale** option will open the **Scale** window in the **Tool Panel**. This function will scale the position of the entire network or of the selected portion of the network. Note that only the position of the nodes will be scaled, not the node sizes. Node size can be adjusted using **Styles**. The image below shows selected nodes scaled.

Before



Tool Panel

Align and Distribute | Scale | Rotate

1/8 1/4 1/2 1 2 4 8

☑ Scale Selected Nodes Only

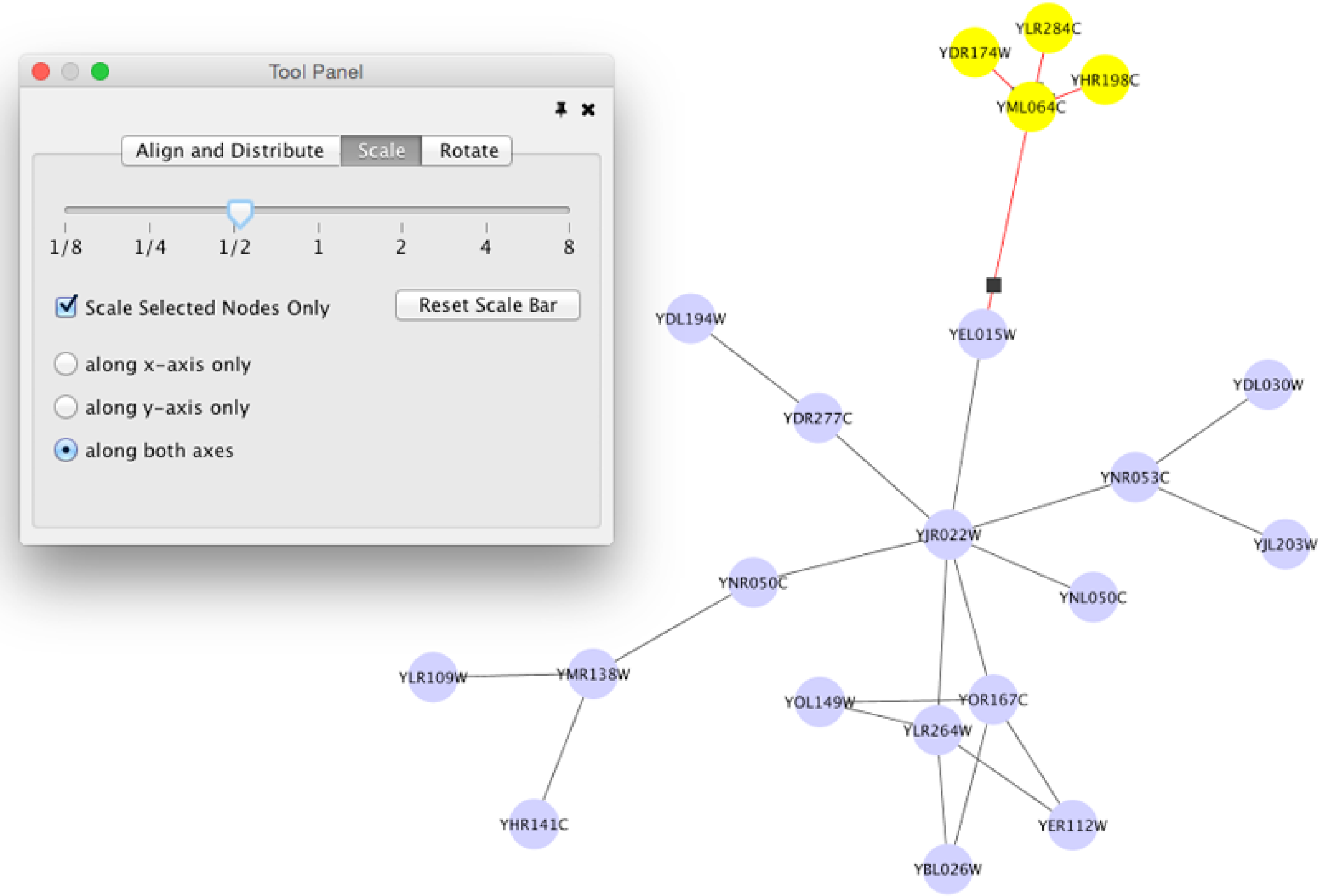
Reset Scale Bar

○ along x-axis only

○ along y-axis only

● along both axes

After



Tool Panel

Align and Distribute | Scale | Rotate

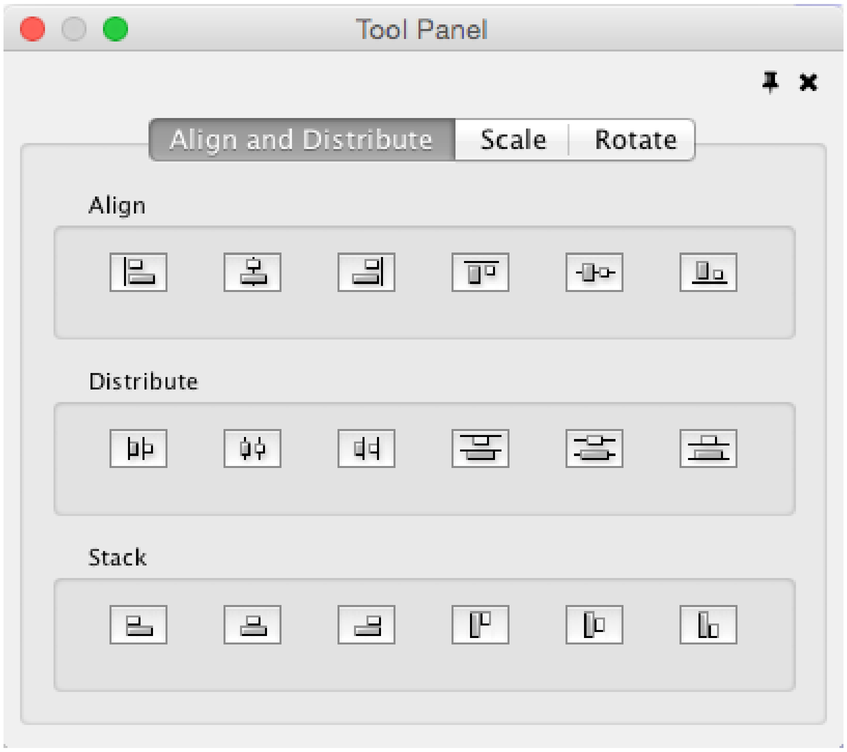
☑ Scale Selected Nodes Only

○ along x-axis only

○ along y-axis only

● along both axes

Align, Distribute and Stack

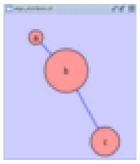
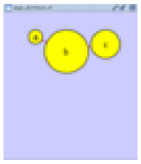
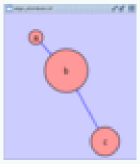
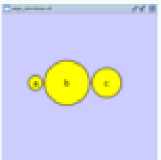
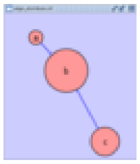
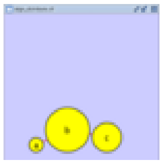
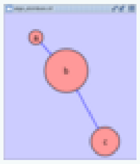
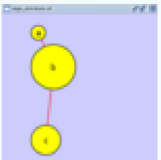
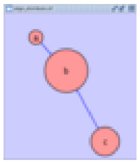
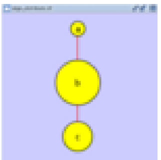
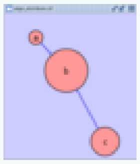
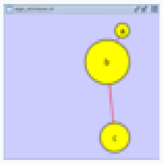


Tool Panel

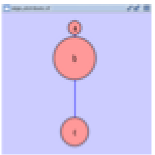
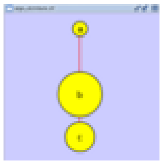
Align and Distribute | Scale | Rotate

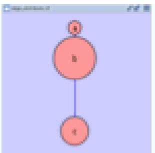
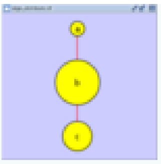
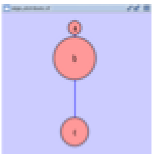
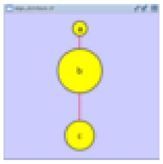
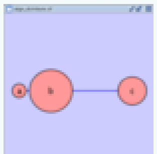
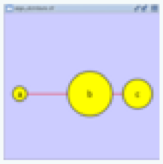
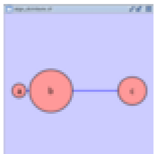
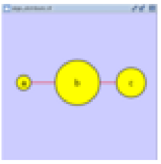
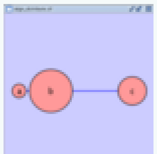
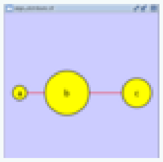
Selecting the **Layout** → **Align/Distribute** option will open the **Align and Distribute** window in the **Tool Panel**. **Align** provides different options for either vertically or horizontally aligning selected nodes against a line. The differences are in what part of the node gets aligned, e.g. the center of the node, the top of the node, the left side of the node. **Distribute** evenly distributes selected nodes between the two most distant nodes along either the vertical or horizontal axis. The differences are again a function what part of the node is used as a reference point for the distribution. **Stack** vertically or horizontally stacks selected nodes with the full complement of alignment options. The table below provides a description of what each button does.

Align Options

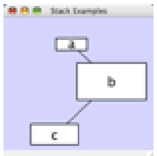
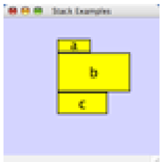
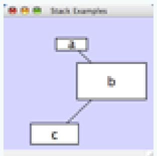
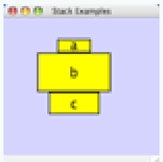
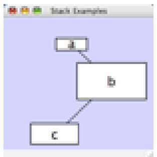
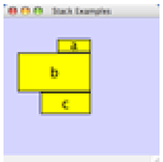
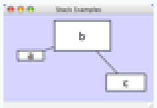
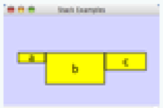
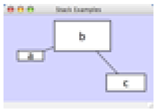
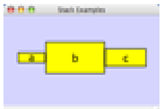
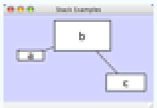
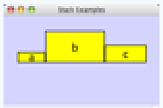
BUTTON	BEFORE	AFTER	DESCRIPTION OF ALIGN OPTIONS
			Vertical Align Top - The tops of the selected nodes are aligned with the top-most node.
			Vertical Align Center - The centers of the selected nodes are aligned along a line defined by the midpoint between the top and bottom-most nodes.
			Vertical Align Bottom - The bottoms of the selected nodes are aligned with the bottom-most node.
			Horizontal Align Left - The left hand sides of the selected nodes are aligned with the left-most node.
			Horizontal Align Center - The centers of the selected nodes are aligned along a line defined by the midpoint between the left and right-most nodes.
			Horizontal Align Right - The right hand sides of the selected nodes are aligned with the right-most node.

Distribute Options

BUTTON	BEFORE	AFTER	DESCRIPTION OF ALIGN OPTIONS
			Vertical Distribute Top - The tops of the selected nodes are distributed evenly between the top-most and bottom-most nodes, which should stay stationary.

			Vertical Distribute Center - The centers of the selected nodes are distributed evenly between the top-most and bottom-most nodes, which should stay stationary.
			Vertical Distribute Bottom - The bottoms of the selected nodes are distributed evenly between the top-most and bottom-most nodes, which should stay stationary.
			Horizontal Distribute Left - The left hand sides of the selected nodes are distributed evenly between the left-most and right-most nodes, which should stay stationary.
			Horizontal Distribute Center - The centers of the selected nodes are distributed evenly between the left-most and right-most nodes, which should stay stationary.
			Horizontal Distribute Right - The right hand sides of the selected nodes are distributed evenly between the left-most and right-most nodes, which should stay stationary.

Stack Options

BUTTON	BEFORE	AFTER	DESCRIPTION OF ALIGN OPTIONS
			Vertical Stack Left - Vertically stacked below top-most node with the left-hand sides of the selected nodes aligned.
			Vertical Stack Center - Vertically stacked below top-most node with the centers of selected nodes aligned.
			Vertical Stack Right - Vertically stacked below top-most node with the right-hand sides of the selected nodes aligned.
			Horizontal Stack Top - Horizontally stacked to the right of the left-most node with the tops of the selected nodes aligned.
			Horizontal Stack Center - Horizontally stacked to the right of the left-most node with the centers of selected nodes aligned.
			Horizontal Stack Bottom - Horizontal Stack Center - Horizontally stacked to the right of the left-most node with the bottoms of the selected nodes aligned.

Node Movement and Placement

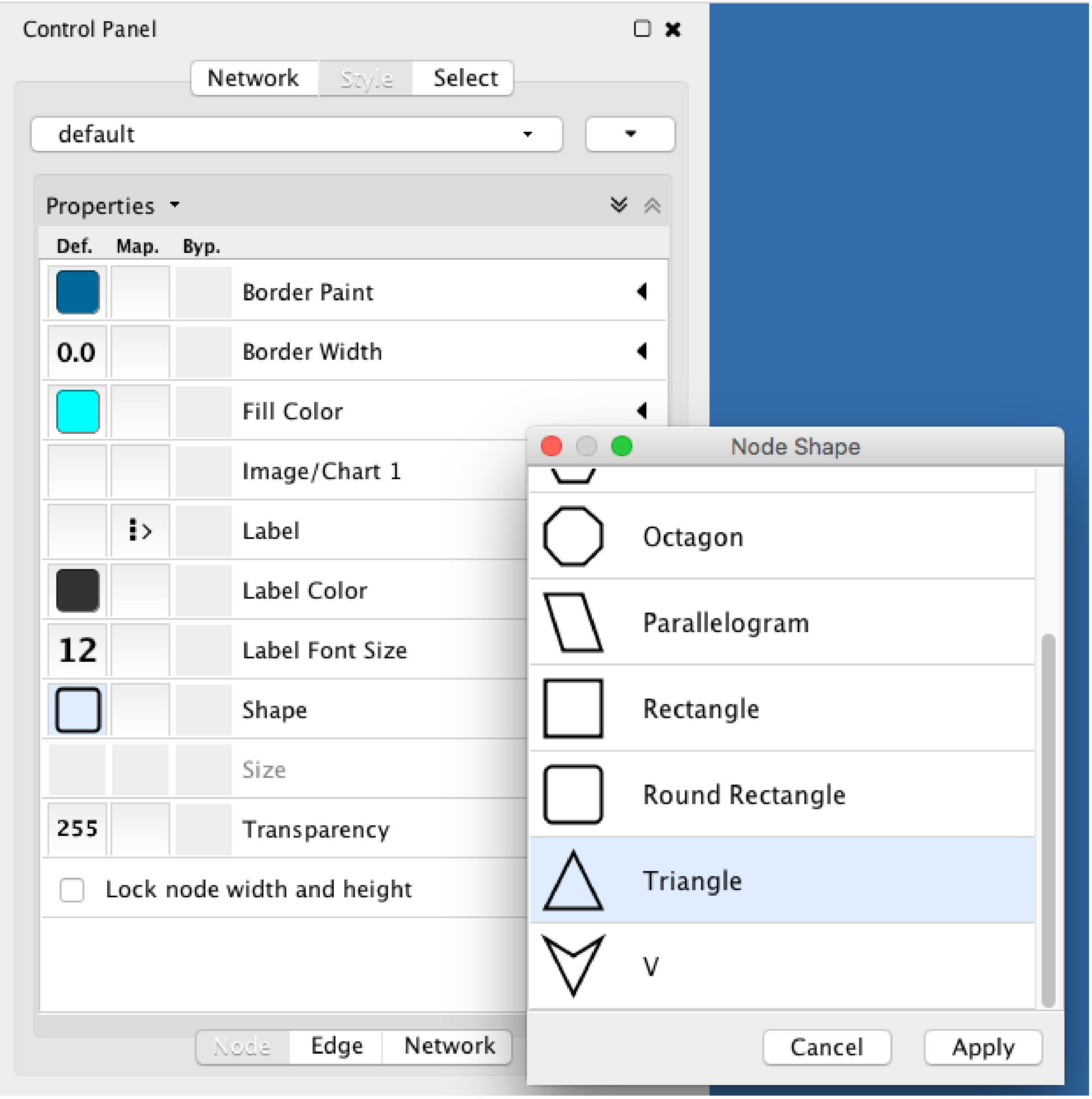
In addition to the ability to click on a node and drag it to a new position, Cytoscape now has the ability to move nodes using the arrow keys on the keyboard. By selecting one or more nodes using the mouse and clicking one of the arrow keys (←, ↑, →, ↓) the selected nodes will move one pixel in the chosen direction. If an arrow key is pressed while holding the Shift key down, the selected nodes will 15 pixels in the chosen direction.

Styles

What are Styles?

One of Cytoscape's strengths in network visualization is the ability to allow users to encode any table data (name, type, degree, weight, expression data, etc.) as a property (such as color, size of node, transparency, or font type) of the network. A set of these encoded or mapped table data sets is called a **Style** and can be created or edited in the **Style** panel of the **Control Panel**. In this interface, the appearance of your network is easily customized. For example, you can:

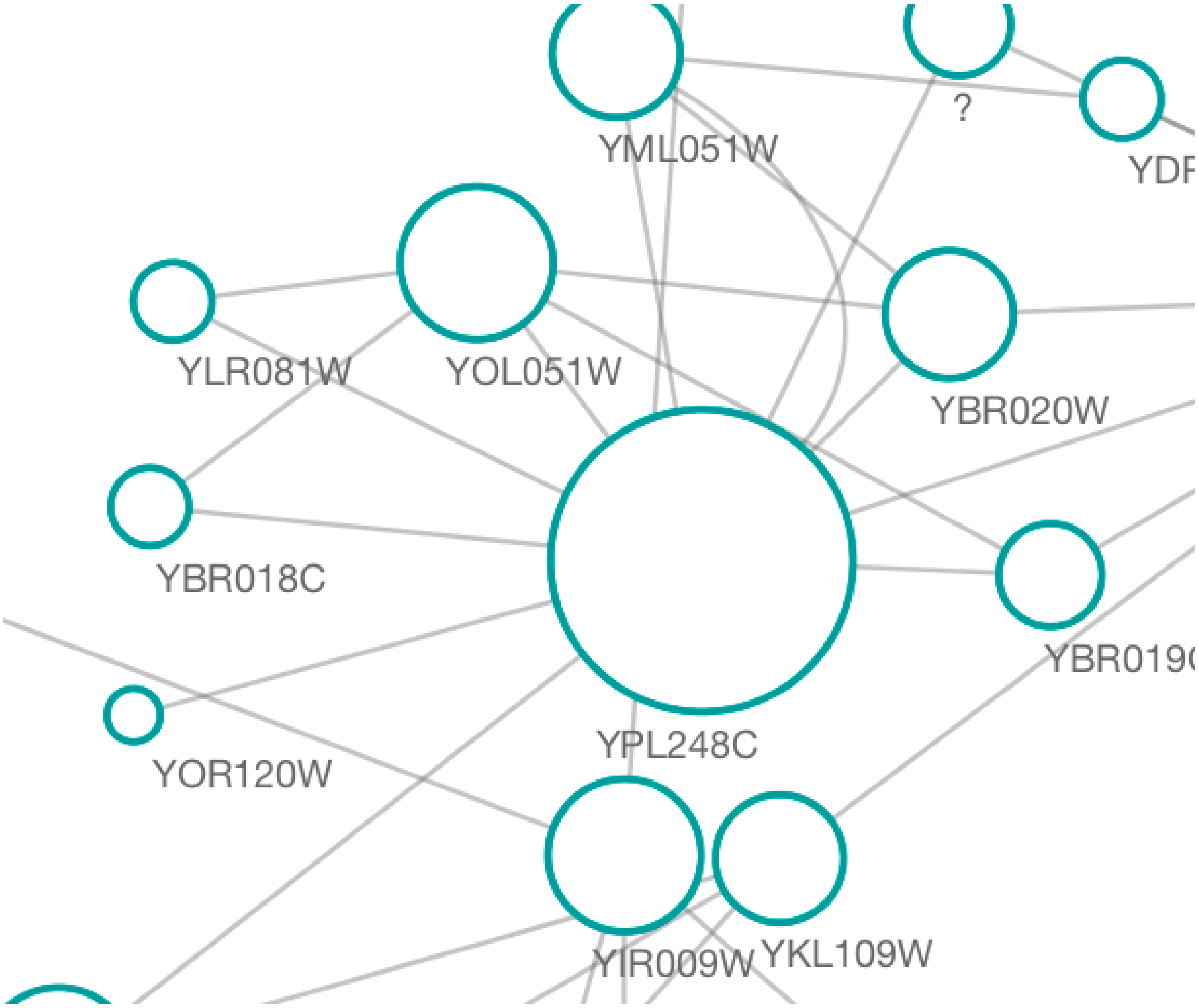
Specify a default color and shape for all nodes.



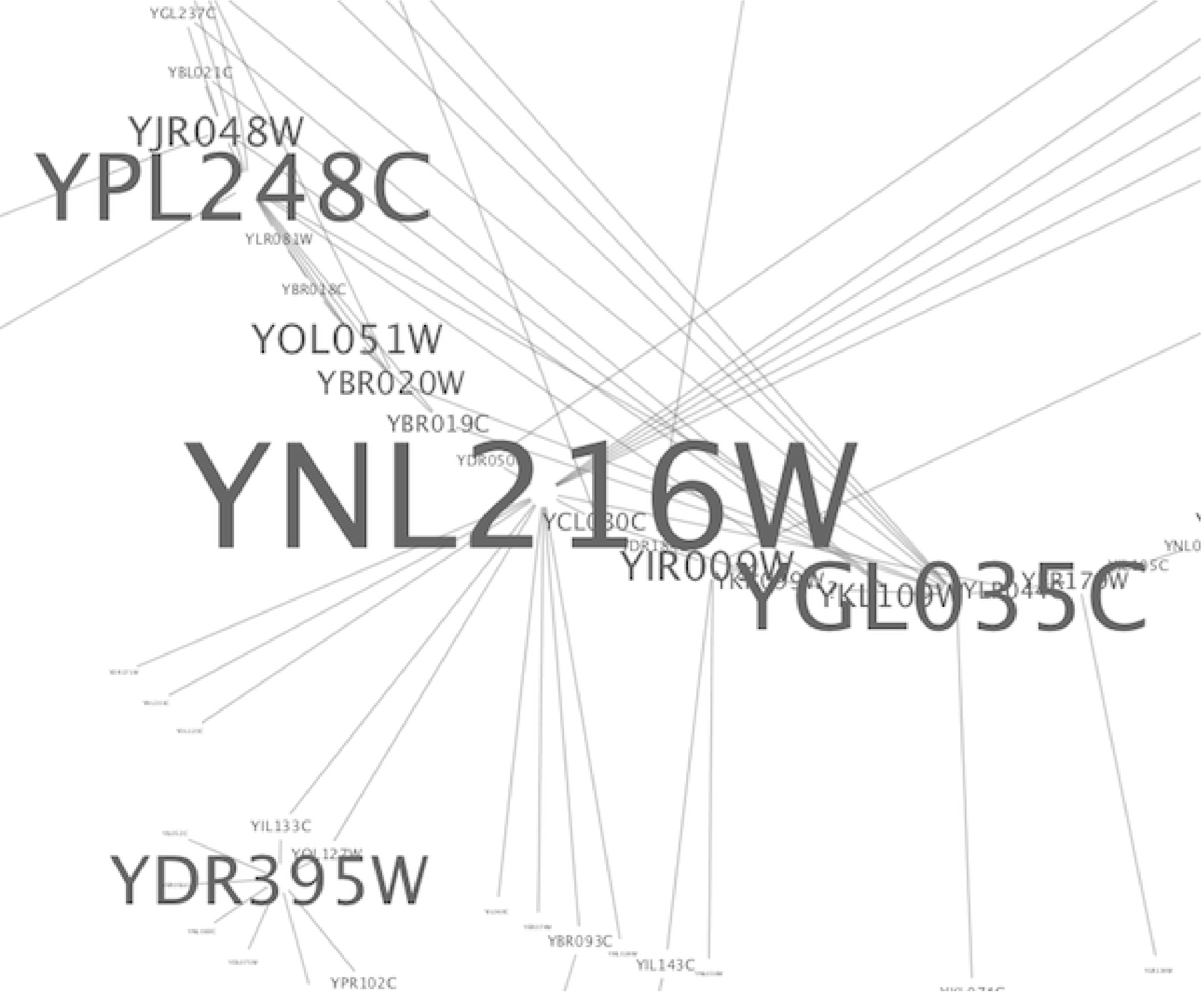
Def.	Map.	Byp.	
🟦			Border Paint
0.0			Border Width
🟦			Fill Color
			Image/Chart 1
	⁝>		Label
■			Label Color
12			Label Font Size
□			Shape
			Size
255			Transparency

☐ Lock node width and height

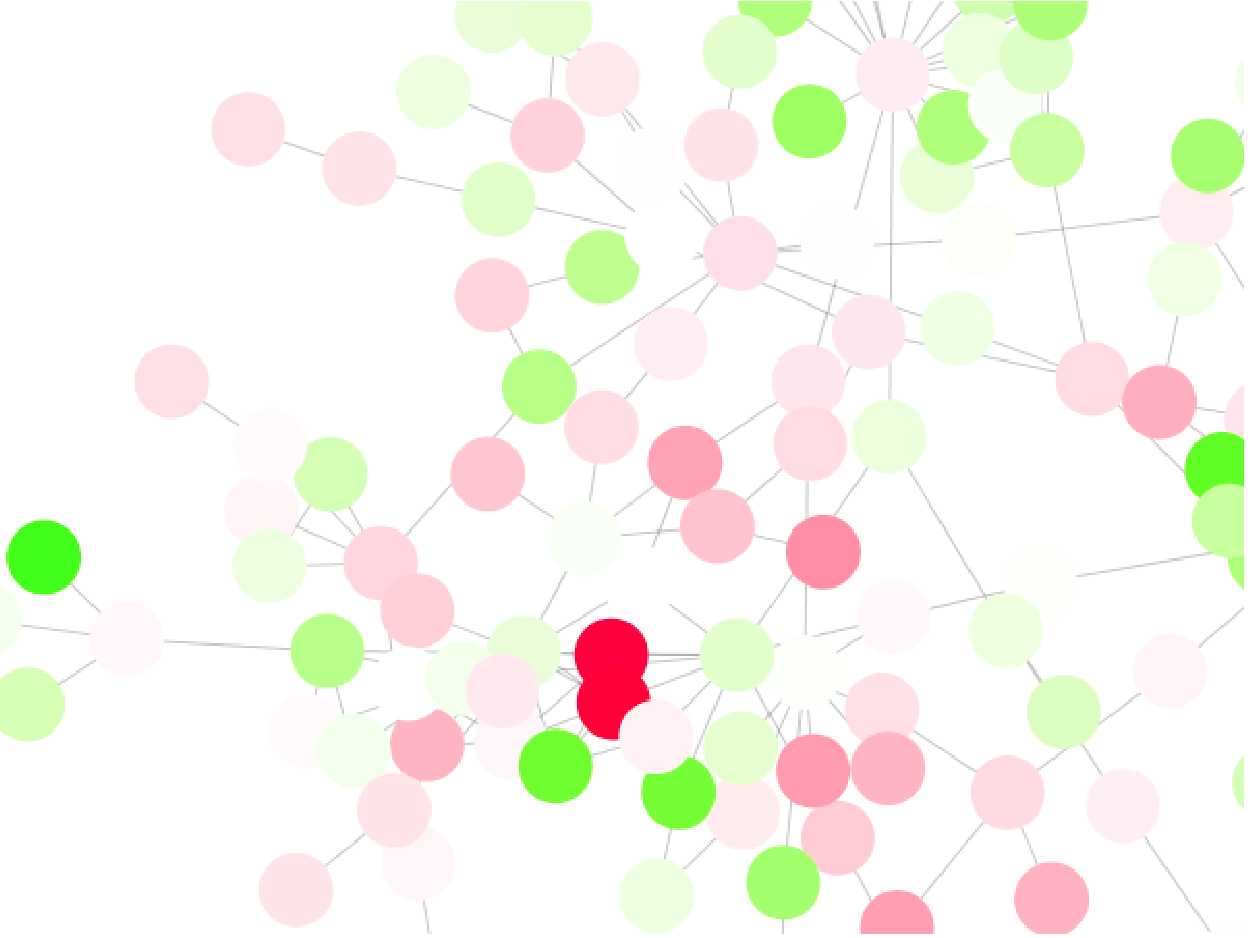
Set node sizes based on the degree of connectivity of the nodes. You can visually see the hub of a network...



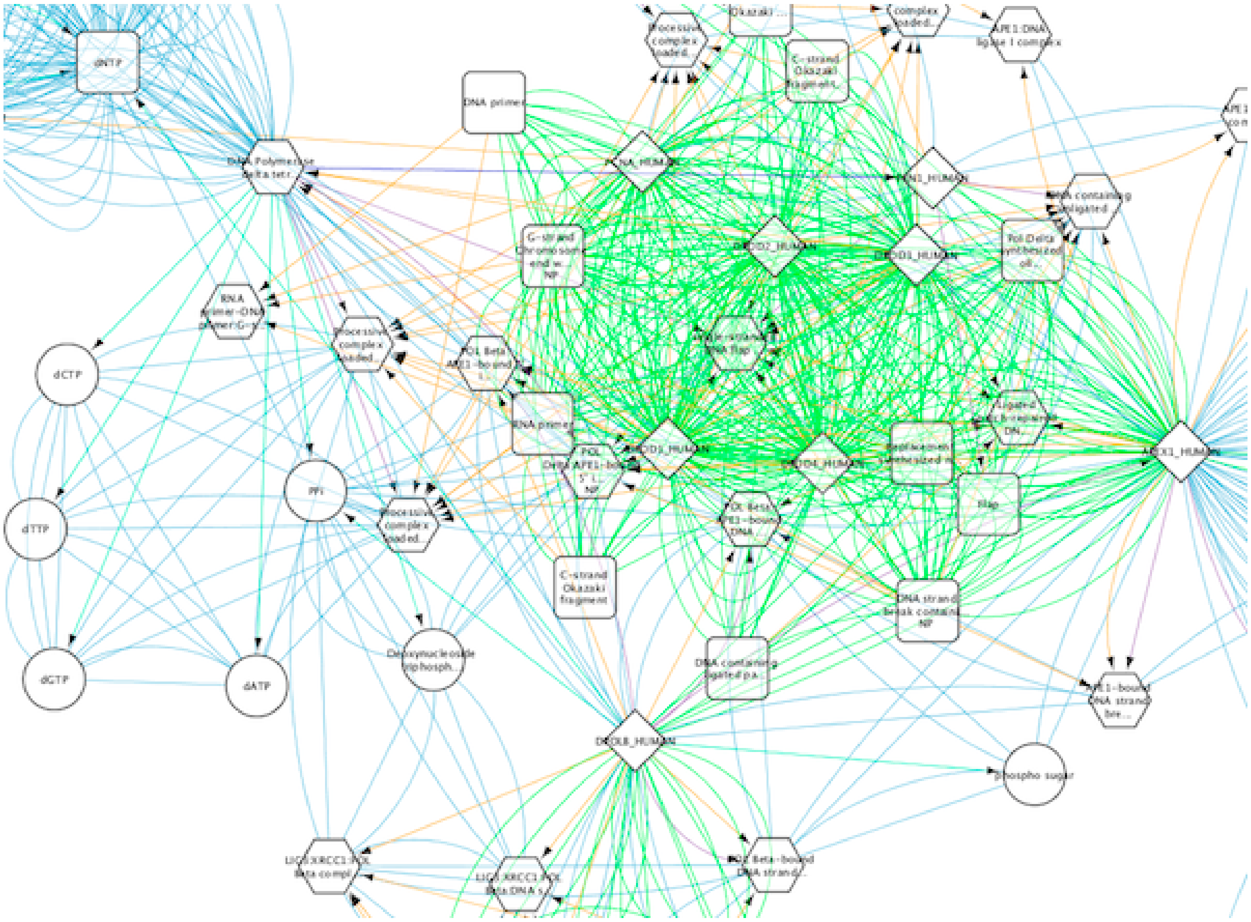
...or, set the font size of the node labels instead.



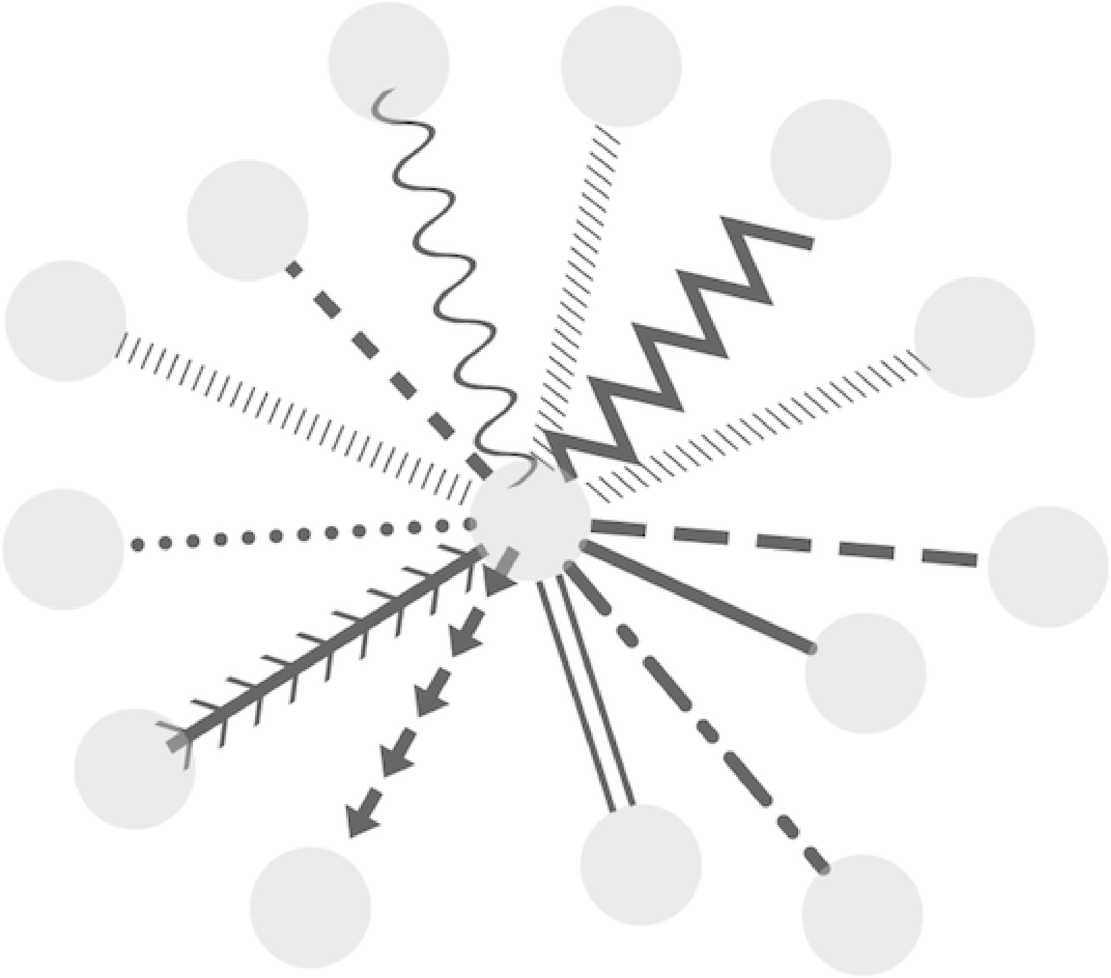
Visualize gene expression data along a color gradient.



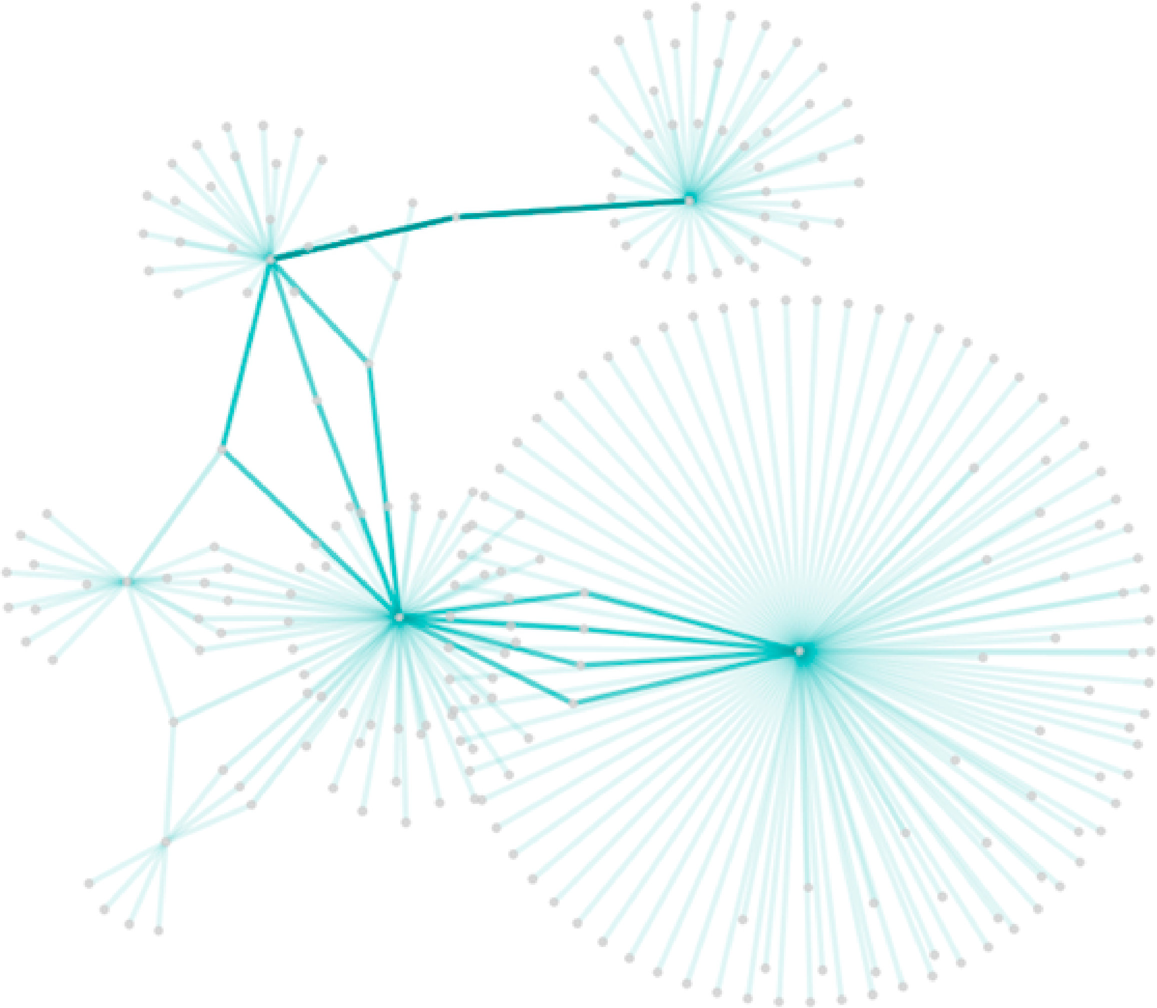
Encode specific physical entities as different node shapes.


dNTP
DNA primer
Processive complex loaded...
Okazaki ...
C-strand Okazaki fragment...
complex loaded...
APE1:DNA ligase I complex
DNA Polymerase delta tetr...
PCNA_HUMAN
FEN1_HUMAN
DNA containing unligated...
G-strand Chromosome end w... NP
DPOD2_HUMAN
DPOD3_HUMAN
Pol Delta synthesized oli...
RNA primer-DNA primer:G-s...
Processive complex loaded...
POL Beta: APE1-bound 1...
Single-stranded DNA flap
dCTP
RNA primer
Ligated patch-repaired DN...
DPOD1_HUMAN
DPOD4_HUMAN
Displacement synthesized o...
APEX1_HUMAN
POL Delta:APE1-bound 5'-1... NP
PPi
Processive complex loaded...
POL Beta: APE1-bound DNA...
flap
dTTP
C-strand Okazaki fragment
DNA strand break containi... NP
Deoxynucleoside triphosph...
DNA containing ligated pa...
APE1-bound DNA strand bre...
dGTP
dATP
DPOLB_HUMAN
Phospho sugar
LIG3:XRCC1:POL Beta compl...
LIG3:XRCC1:POL Beta:DNA s...
POL Beta-bound DNA strand...

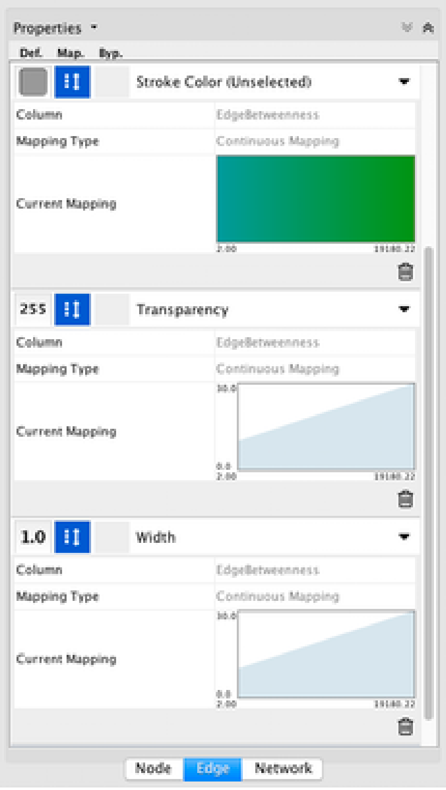
Use specific line types to indicate different types of interactions.



Control edge transparency (opacity) using edge weights.



Control multiple edge properties using edge score.



Properties

Def. Map. Byp.

Stroke Color (Unselected)

Column: EdgeBetweenness

Mapping Type: Continuous Mapping

Current Mapping

255 Transparency

Column: EdgeBetweenness

Mapping Type: Continuous Mapping

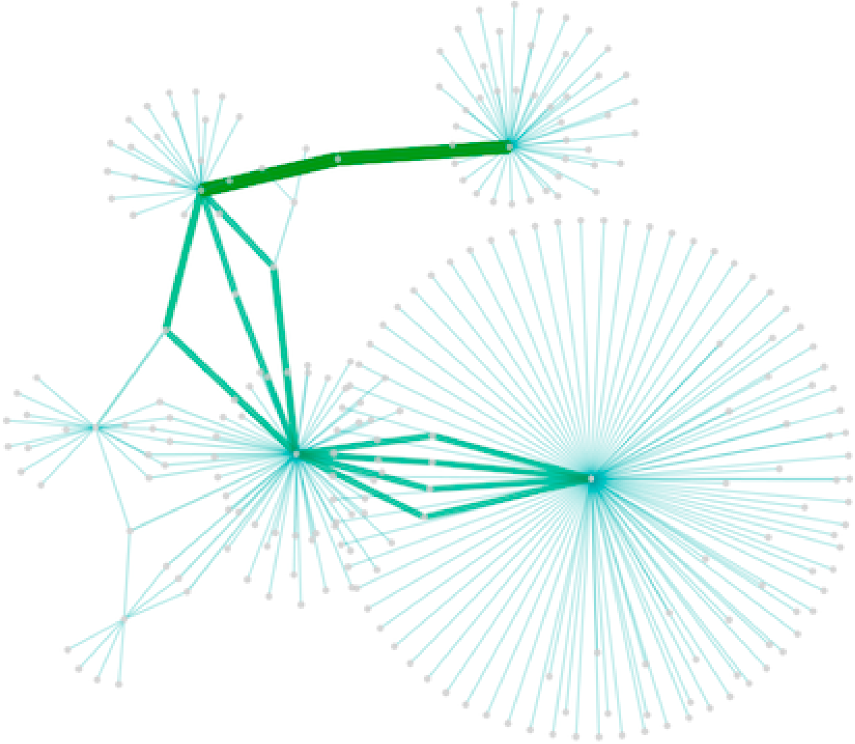
Current Mapping

1.0 Width

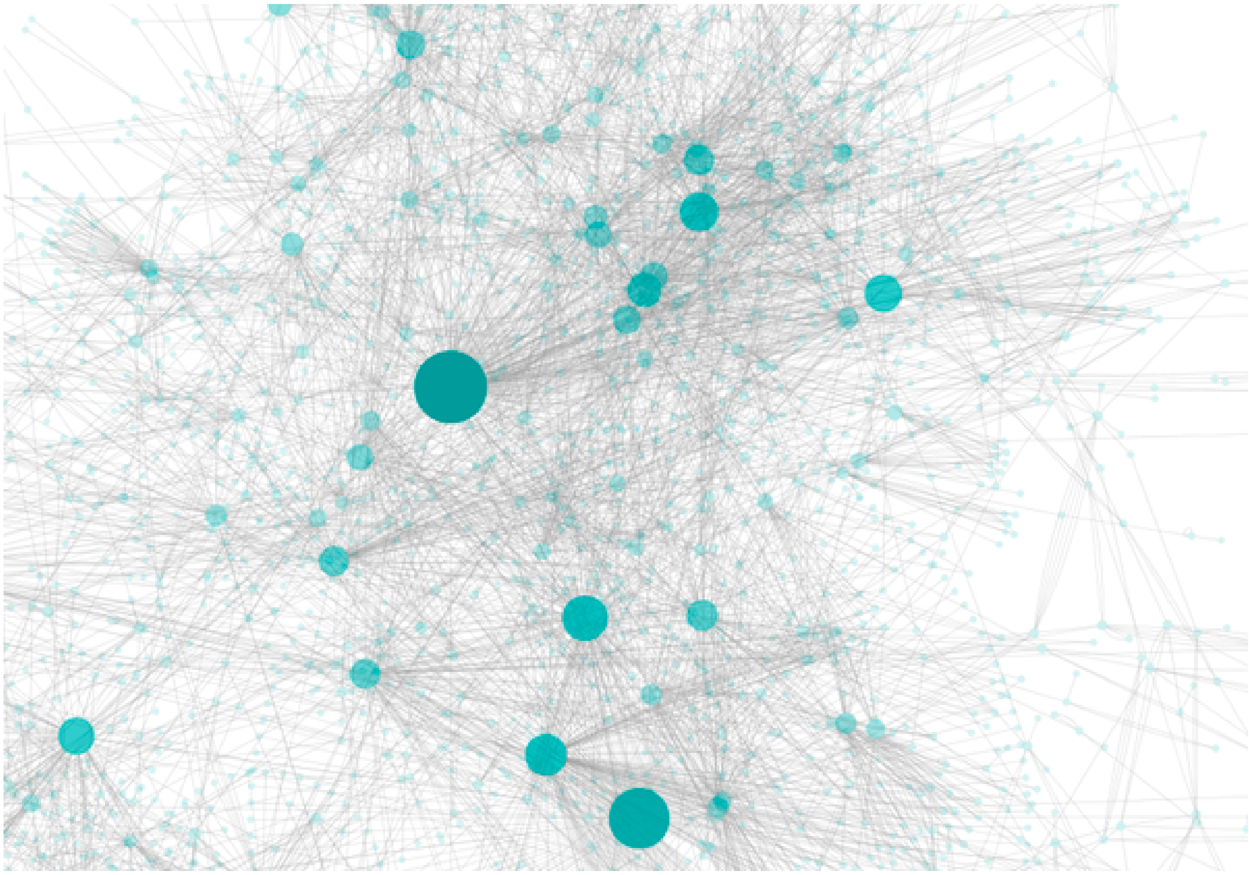
Column: EdgeBetweenness

Mapping Type: Continuous Mapping

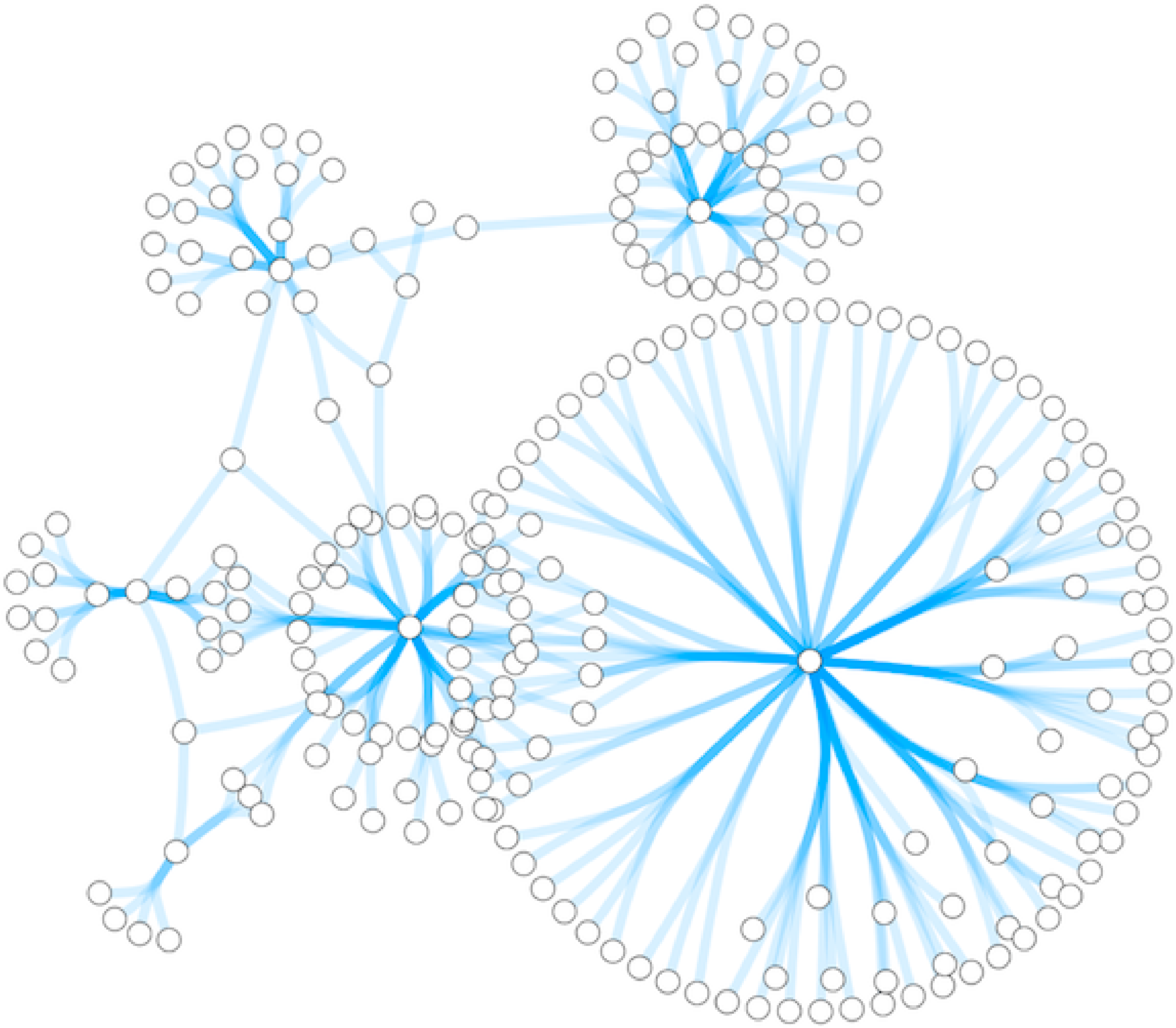
Current Mapping



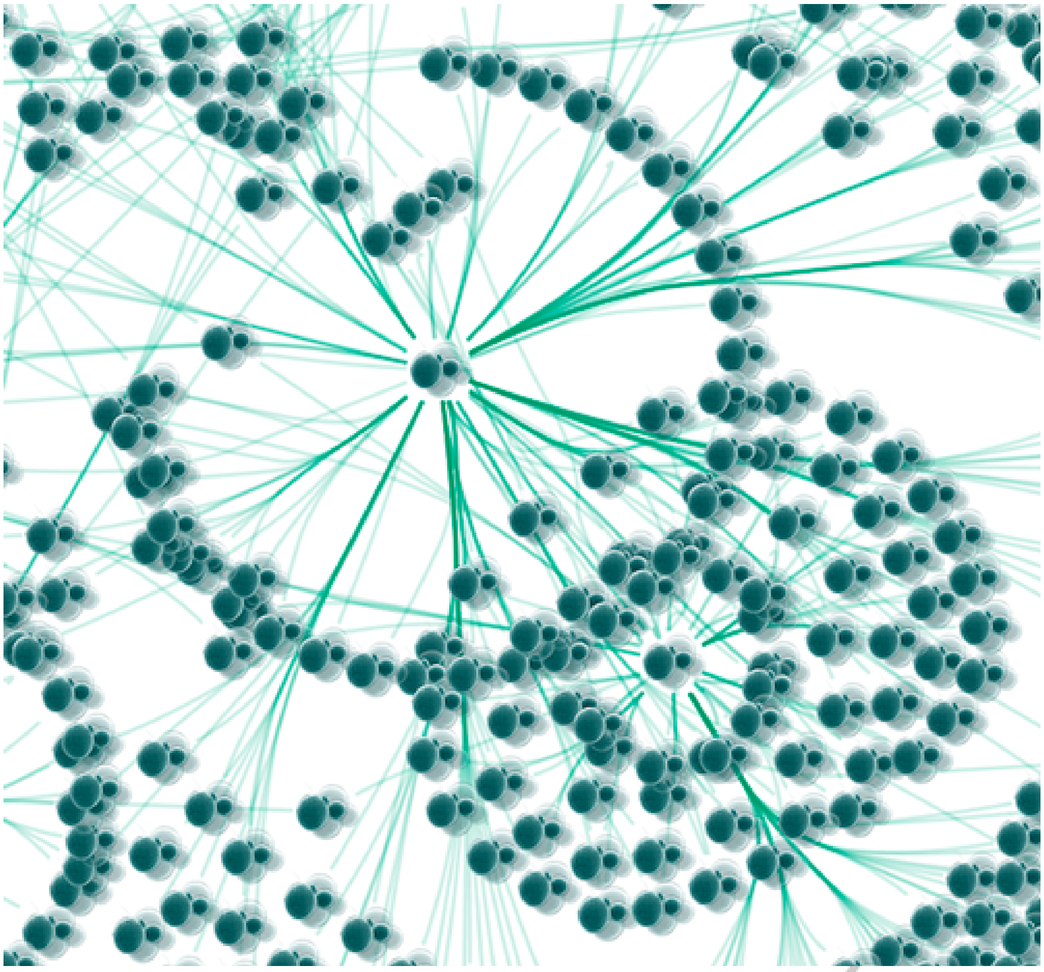
Browse extremely-dense networks by controlling the opacity of nodes.

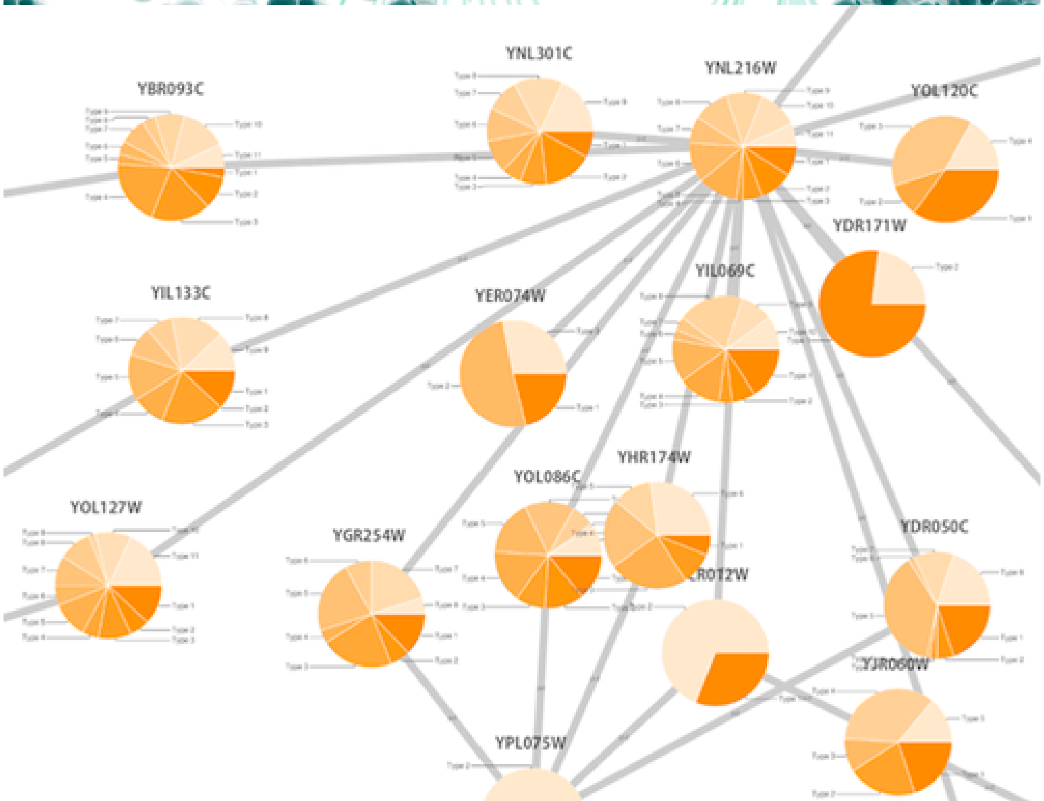


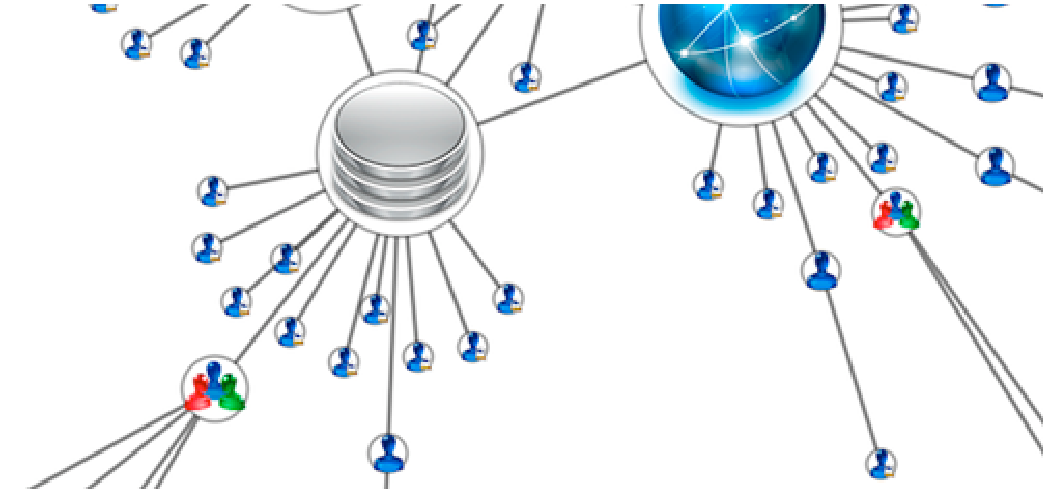
Show highly-connected region by edge bundling and opacity.



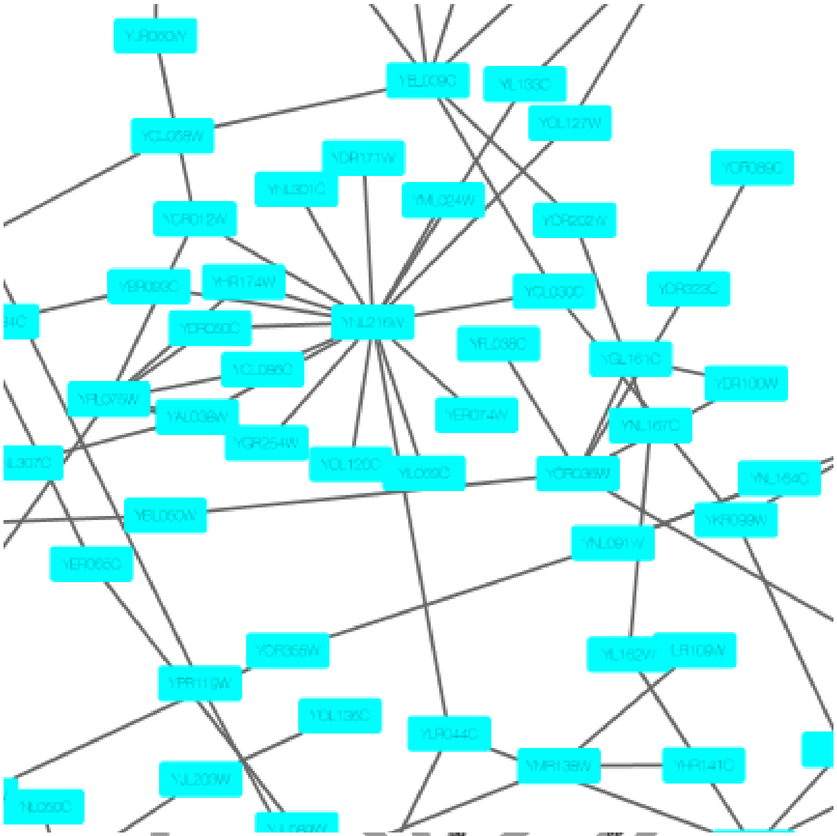
Add photo/image/graphics on top of nodes.

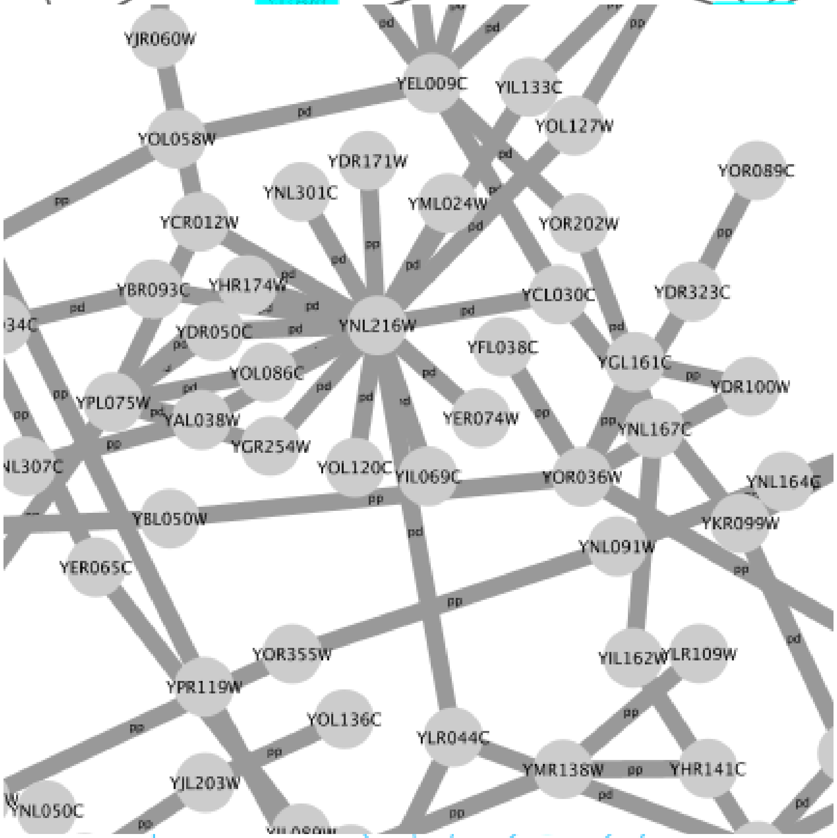


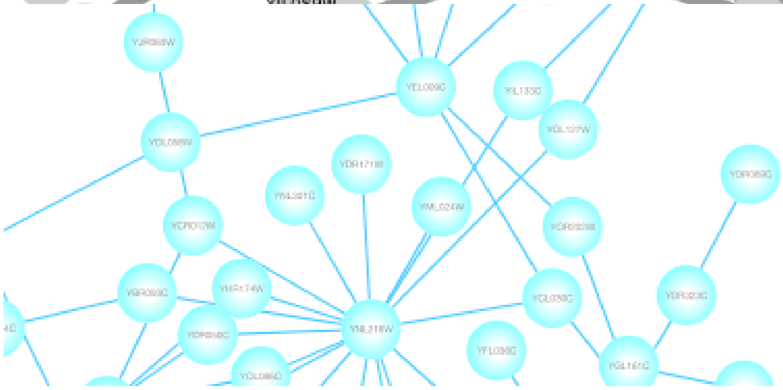

YNL301C
YNL216W
YBR093C
YOL120C
YDR171W
YIL133C
YER074W
YIL069C
YHR174W
YOL086C
YOL127W
YGR254W
YDR050C
ER012W
YJR060W
YPL075W

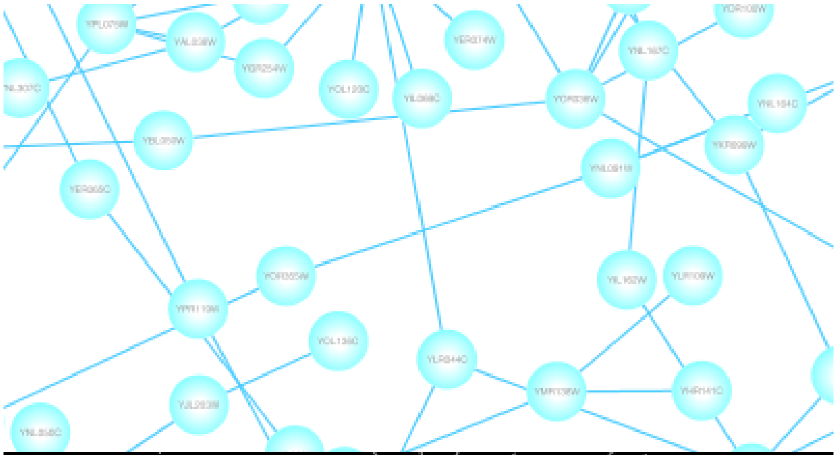


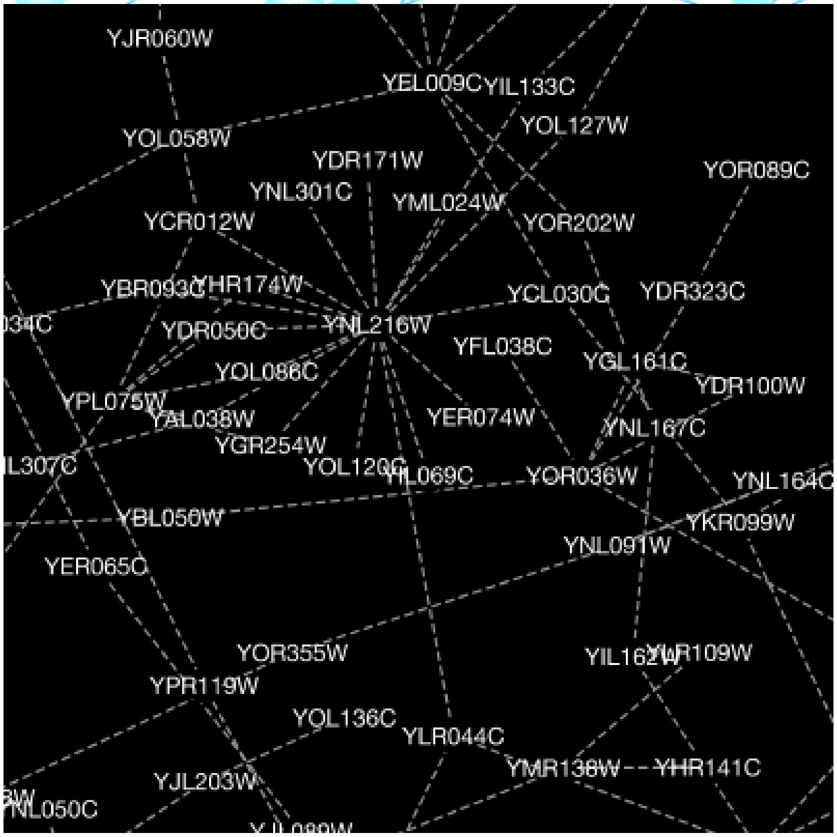
Cytoscape 3 has several sample styles. Below are a few examples of these applied to the *galFiltered.sif* network :


YJR060W
YEL009C
YIL133C
YOL127W
YOL058W
YDR171W
YNL301C
YML024W
YOR089C
YOR202W
YCR012W
YBR093C
YHR174W
YCL030C
YDR323C
YNL216W
YDR050C
YFL038C
YGL161C
YOL086C
YDR100W
YPL075W
YAL038W
YER074W
YNL167C
YGR254W
YOL120C
YIL069C
YOR036W
YNL164C
YBL050W
YKR099W
YNL091W
YER065C
YOR355W
YIL162W
YLR109W
YPR119W
YOL136C
YLR044C
YMR138W
YHR141C
YJL203W
YNL050C


YJR060W
YEL009C
YIL133C
YOL127W
YOL058W
YDR171W
YNL301C
YML024W
YOR089C
YOR202W
YCR012W
YBR093C
YHR174W
YCL030C
YDR323C
YNL216W
YDR050C
YFL038C
YGL161C
YOL086C
YDR100W
YPL075W
YAL038W
YER074W
YNL167C
YGR254W
YOL120C
YIL069C
YOR036W
YNL164C
NL307C
YBL050W
YKR099W
YNL091W
YER065C
YOR355W
YIL162W
YLR109W
YPR119W
YOL136C
YLR044C
YMR138W
YHR141C
YJL203W
YNL050C
pp
pd

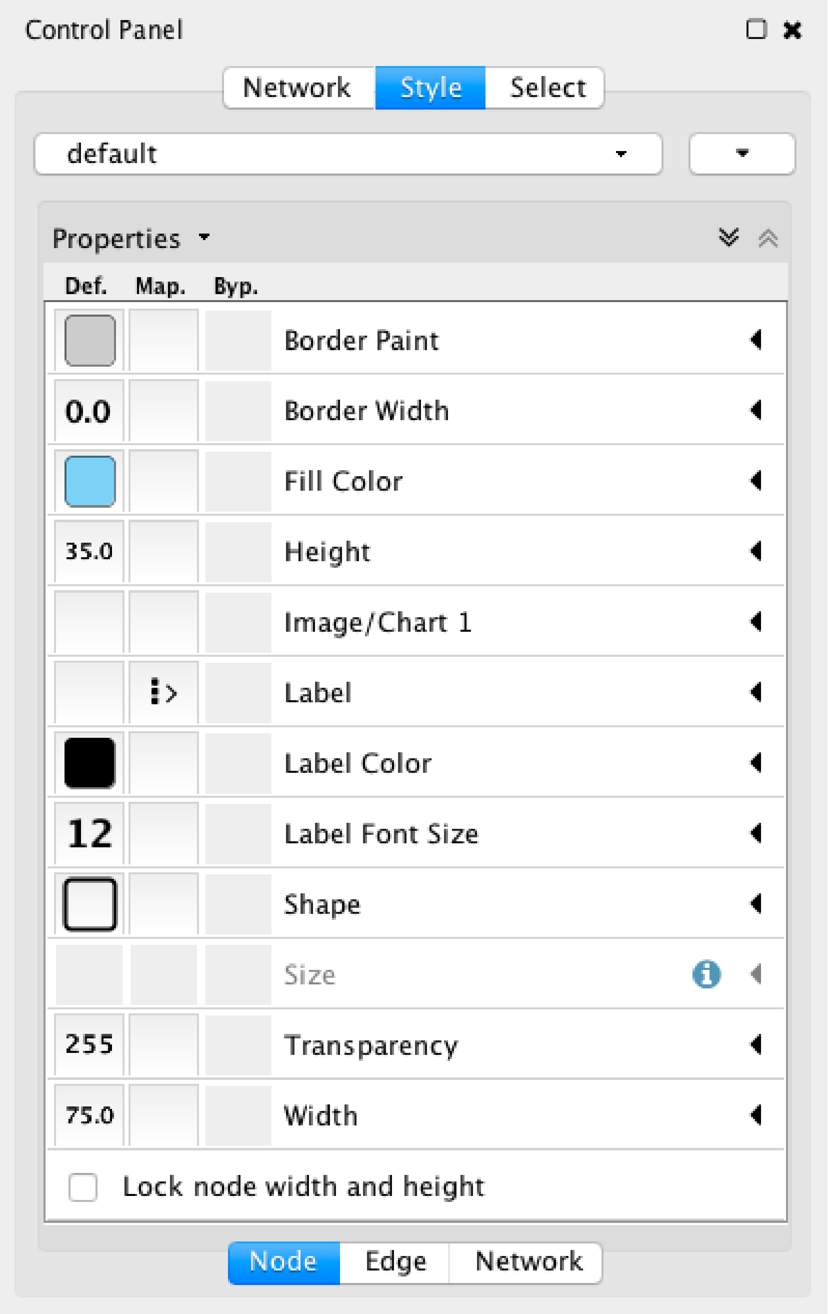

YJR060W
YEL009C
YIL133C
YOL127W
YOL058W
YDR171W
YNL301C
YML024W
YOR089C
YOR202W
YCR012W
YBR093C
YHR174W
YCL030C
YDR323C
YNL216W
YDR050C
YFL038C
YGL161C
YOL086C





Introduction to the Style Interface

The **Style** interface is located under the **Style** panel of the **Control Panel**.



Def.	Map.	Byp.	
▢			Border Paint
0.0			Border Width
🟦			Fill Color
35.0			Height
			Image/Chart 1
			Label
■			Label Color
12			Label Font Size
□			Shape
			Size
255			Transparency
75.0			Width

□ Lock node width and height

This interface allows you to create/delete/view/switch between different styles using the Current Style options. The panel displays the mapping details for a given style and is used to edit these details as well.

- At the top of the interface, there is a drop-down menu for selecting a pre-defined style. There is also an **Options** drop-down with options to rename, remove, create and copy a Style, and an option to create a legend for the selected Style.
- The main area of the interface is composed of three tabs, for Node, Edge and Network.
- Each tab contains a list of properties relevant to the current style. At the top of the list a **Properties** drop-down allows you to add additional properties to the list.
- Each property entry in the list has 3 columns:
  - The **Default Value** shows just that, the default value for the property. Clicking on the **Default Value** column for any property allows you to change the default value.

- **Mapping** displays the type of mapping currently in use for the property. Clicking on the **Mapping** column for any property expands the property entry to show the interface for editing the mapping. Details on the mapping types provided here.
- **Bypass** displays any style bypass for a selected node or edge. Note that a node/edge or subset of nodes/edges must be selected to activate the **Bypass** column. Clicking on the **Bypass** column for selected node(s)/edge(s) allows you to enter a bypass for that property for selected node(s)/edge(s).

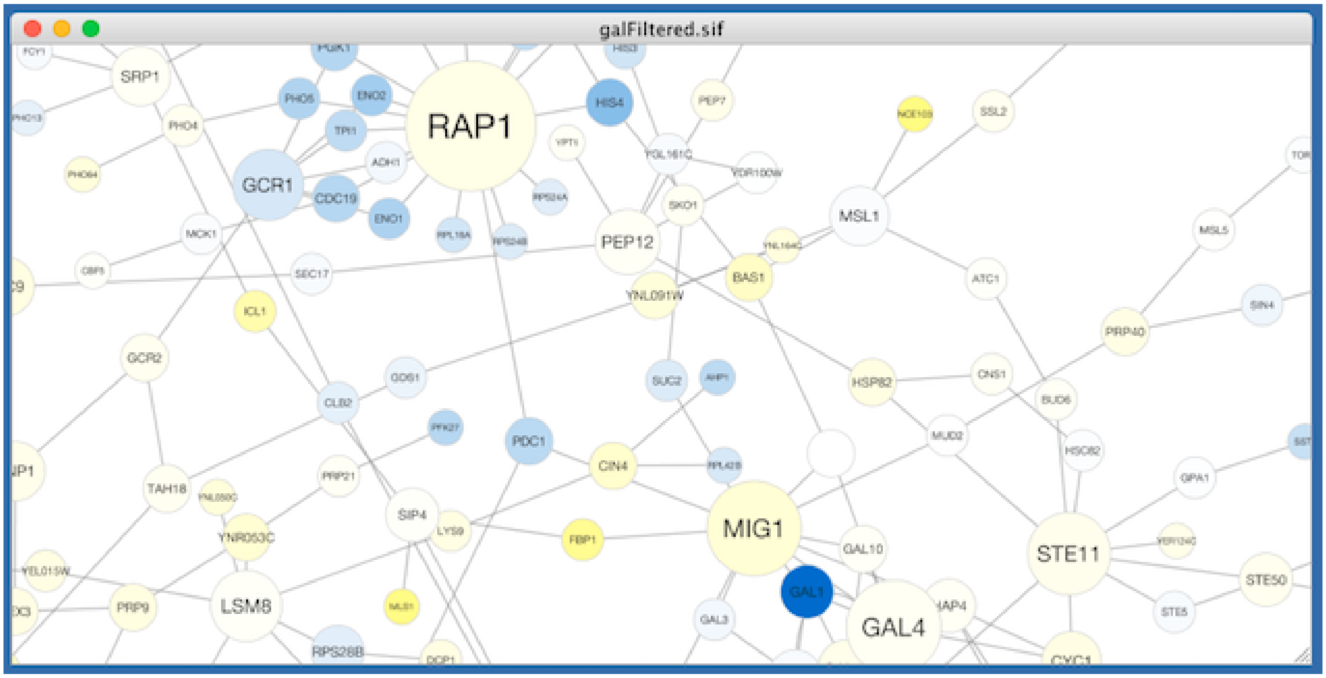
The **Default Value** is used when no mapping is defined for a property, or for nodes/edges not covered by a mapping for a particular property. If a **Mapping** is defined for a property, this defines the style for all or a subset of nodes/edges, depending on how the mapping is defined. A **Bypass** on a specific set of nodes/edges will bypass and override both the default value and defined mapping.

Introduction to Style

The Cytoscape distribution includes several predefined styles to get you started. To examine a few styles, try out the following example:

Step 1. Load some sample data

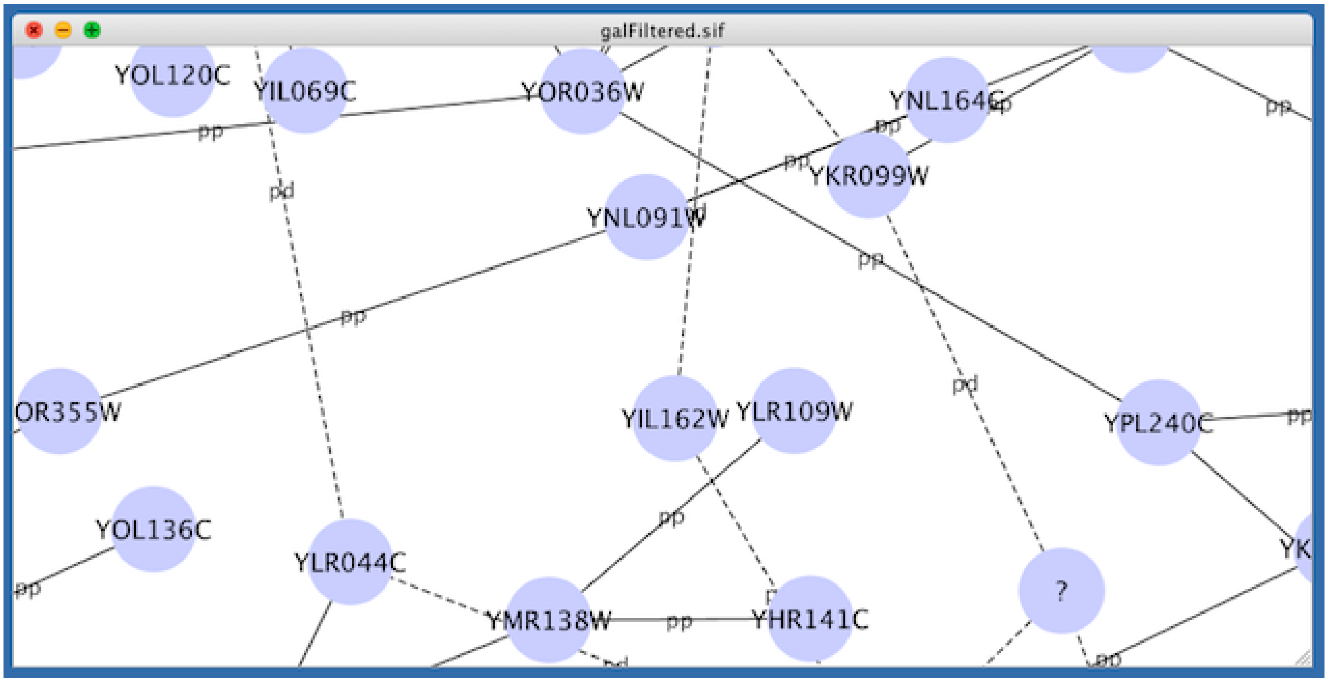
- Load a sample session file: From the main menu, select **File** → **Open...**, and select the file *sampleData/galFiltered.cys*.
- The session file includes a network, some annotations, and sample styles. By default, the style **galFiltered Style** is selected. Gene expression values for each node are colored along a color gradient between blue and yellow (where blue represents a low expression ratio and yellow represents a high expression ratio, using thresholds set for the **gal1RGexp** experiment bundled with Cytoscape in the *sampleData/galExpData.csv* file). Also, node size is mapped to the degree of the node (number of edges connected to the node) and you can see the hubs of the network as larger nodes. See the sample screenshot below:



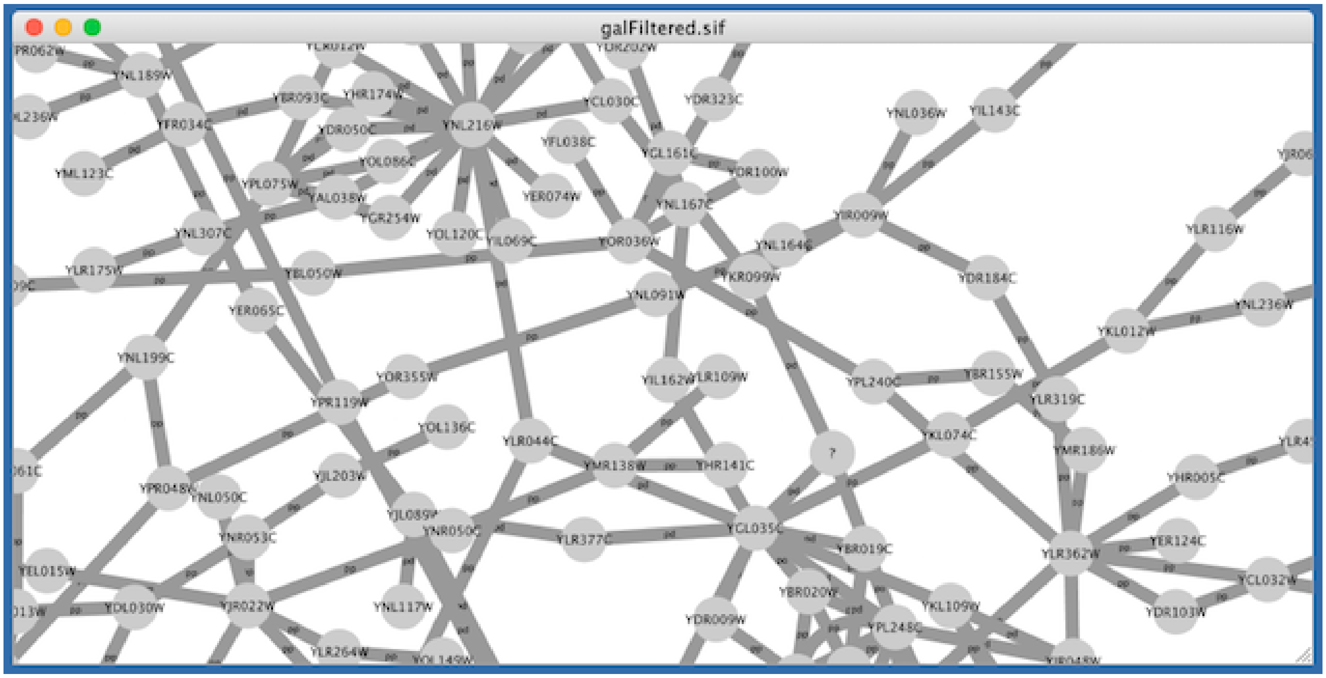
Step 2. Switch between different styles

You can change the style by making a selection from the **Current Style** drop-down list, found at the top of the **Style** panel.

For example, if you select **Sample1**, a new style will be applied to your network, and you will see a white background and round blue nodes. If you zoom in closer, you can see that protein-DNA interactions (specified with the label "pd") are drawn with dashed edges, whereas protein-protein interactions (specified with the label "pp") are drawn with solid edges (see sample screenshot below).



Finally, if you select **Solid**, you can see the graphics below:



This style does not have mappings except node/edge labels, but you can modify the network graphics by editing the *Default Value* for any property.

Additional sample styles are available in the `sampleStyles.xml` file in the *sampleData* directory. You can import the sample file from **File** → **Import** → **Styles...**.

List of Node, Edge and Network Properties

Cytoscape allows a wide variety of properties to be controlled. These are summarized in the tables below.

Node Properties

NODE PROPERTIES	DESCRIPTION
Border Line Type	The type of line used for the border of the node.
Border Transparency	The opacity of the color of the border of the node. *Zero* means totally transparent, and *255* means totally opaque.
Border Width	The width of the node border.
Label	The text used for the node label.
Label Font Face	The font used for the node label.
Label Font Size	The size of the font used for the node label.
Label Position	The position of the node label relative to the node.
Label Transparency	The opacity of the node label. *Zero* means totally transparent, and *255* means totally opaque.

Label Width	The maximum width of the node label. If the node label is wider than the specified width, Cytoscape will automatically wrap the label on space characters. Cytoscape will not hyphenate words, meaning that if a single word (i.e. no spaces) is longer than maximum width, the word will be displayed beyond the maximum width.
Nested Network Image Visible	A boolean value that indicates whether a nested network should be visualized (assuming a nested network is present for the specified node).
Padding (Compound Node)	Internal padding of the compound node (a node that contains other nodes).
Paint	The color of the whole node, including its border, label and selected paint. This property can be added to the list from the drop-down menu **Properties** → **Paint** → **Paint**.
Border Paint	The color of the border of the node. This property can be added to the list from the drop-down menu **Properties** → **Paint** → **Border Paint**.
Image/Chart *1-9*	A user-defined graphic (image, chart or gradient) that is displayed on the node. These properties (maximum of nine) can be added to the list from the drop-down menu **Properties** → **Paint** → **Custom Paint *n*** → **Image/Chart *n***.
Image/Chart Position *1-9*	The position of each graphic (image, chart or gradient). These properties (maximum of nine) can be added to the list from the drop-down menu **Properties** → **Paint** → **Custom Paint *n*** → **Image/Chart Position *n***.
Fill Color	The color of the node. This property can be added to the list from the drop-down menu **Properties** → **Paint** → **Fill Color**.
Label Color	The color of the node label. This property can be added to the list from the drop-down menu **Properties** → **Paint** → **Label Color**.
Selected Paint	The fill color of the node when selected. This property can be added to the list from the drop-down menu **Properties** → **Paint** → **Selected Paint**.
Shape	The shape of the node.
Shape (Compound Node)	The shape of the compound node (a node that contains other nodes).
Size	The size of the node. Width and height will be equal. This property is mutually exclusive of *Node Height* and *Node Width*. It can be added to the list from the drop-down menu **Properties** → **Size** → **Size**.
Image/Chart Size *1-9*	The size of the related node *Image/Chart*. It can be added to the list from the drop-down menu **Properties** → **Size** → **Image/Chart Size *n***.
Height	The height of the node. Height will be independent of width. This property is mutually exclusive of *Node Size*. It can be added to the list from the drop-down menu **Properties** → **Size** → **Height**.

Width	The width of the node. Width will be independent of height. This property is mutually exclusive of *Node Size*. It can be added to the list from the drop-down menu **Properties** → **Size** → **Width**.
Fit Custom Graphics to node	Toggle to fit Image/Chart size to node size. It can be added to the list from the drop-down menu **Properties** → **Size** → **Fit Custom Graphics to node**.
Lock node width and height	Toggle to ignore *Width* and *Height*, and to use *Size* for both values. It can be added to the list from the drop-down menu **Properties** → **Size** → **Lock node width and height**.
Tooltip	The text of the tooltip that appears when a mouse hovers over the node.
Transparency	The opacity of the color of the node. *Zero* means totally transparent, and *255* means totally opaque.
Visible	Hides the node if set to *false*. By default, this value is set to *true*.
X Location	X location of the node. Default value of this will be ignored. The value will be used only when mapping function is defined.
Y Location	Y location of the node. Default value of this will be ignored. The value will be used only when mapping function is defined.
Z Location	Z location of the node. Default value of this will be ignored. The value will be used only when mapping function is defined.

Edge Properties

EDGE PROPERTIES	DESCRIPTION
Bend	The edge bend. Defines how the edge is rendered. Users can add multiple handles to define how to bend the edge line.
Curved	If *Edge Bend* is defined, edges will be rendered as straight or curved lines. If this value is set to *true*, edges will be drawn as curved lines.
Label	The text used for the edge label.
Label Font Face	The font used for the edge label.
Label Font Size	The size of the font used for the edge label.
Label Transparency	The opacity of the color of the edge label. *Zero* means totally transparent, and *255* means totally opaque.
Line Type	The type of stoke used to render the line (solid, dashed, etc.)
Paint	The color of the whole edge (including the stroke and arrows) when it is selected or unselected. This property can be added to the list from the drop-down menu **Properties** → **Paint** → **Paint**.
Color (Selected)	The color of the whole edge (stroke and arrows) when selected. This property can be added to the list from the drop-down menu **Properties** → **Paint** → **Color (Selected)** → **Color (Selected)**.

Source Arrow Selected Paint	The selected color of the arrow on the source node end of the edge. It can be added to the list from the drop-down menu **Properties** → **Paint** → **Color (Selected)** → **Source Arrow Selected Paint**.
Stroke Color (Selected)	The color of the edge line when selected. It can be added to the list from the drop-down menu **Properties** → **Paint** → **Color (Selected)** → **Stroke Color (Selected)**.
Target Arrow Selected Paint	The selected color of the arrow on the target node end of the edge. It can be found in the drop-down menu **Properties** → **Paint** → **Color (Selected)** → **Target Arrow Selected Paint**.
Color (Unselected)	The color of the whole edge (stroke and arrows) when it is not selected. It can be found in the drop-down menu **Properties** → **Paint** → **Color (Unselected)** → **Color (Unselected)**.
Source Arrow Unselected Paint	The color of the arrow on the source node end of the edge. It can be found in the drop-down menu **Properties** → **Paint** → **Color (Unselected)** → **Source Arrow Unselected Paint**.
Stroke Color (Unselected)	The color of the edge line. It can be found in the drop-down menu **Properties** → **Paint** → **Color (Unselected)** → **Stroke Color (Unselected)**.
Target Arrow Unselected Paint	The color of the arrow on the target node end of the edge. It can be found in the drop-down menu **Properties** → **Paint** → **Color (Unselected)** → **Target Arrow Unselected Paint**.
Label Color	The color of the edge label. It can be found in the drop-down menu **Properties** → **Paint** → **Label Color**.
Source Arrow Shape	The shape of the arrow on the source node end of the edge.
Target Arrow Shape	The shape of the arrow on the target node end of the edge.
Tooltip	The text of the tooltip that appears when a mouse hovers over the edge.
Transparency	The opacity of the of the edge. Zero means totally transparent, and 255 means totally opaque.
Visible	Hides the edge if set to *false*. By default, this value is set to *true*.
Width	The width of the edge line.
Edge color to arrows	If *true* then **Color (Unselected)** is used for the whole edge, including its line and arrows. It can be found in the drop-down menu **Properties** → **Paint** → **Color (Unselected)** → **Edge color to arrows**.

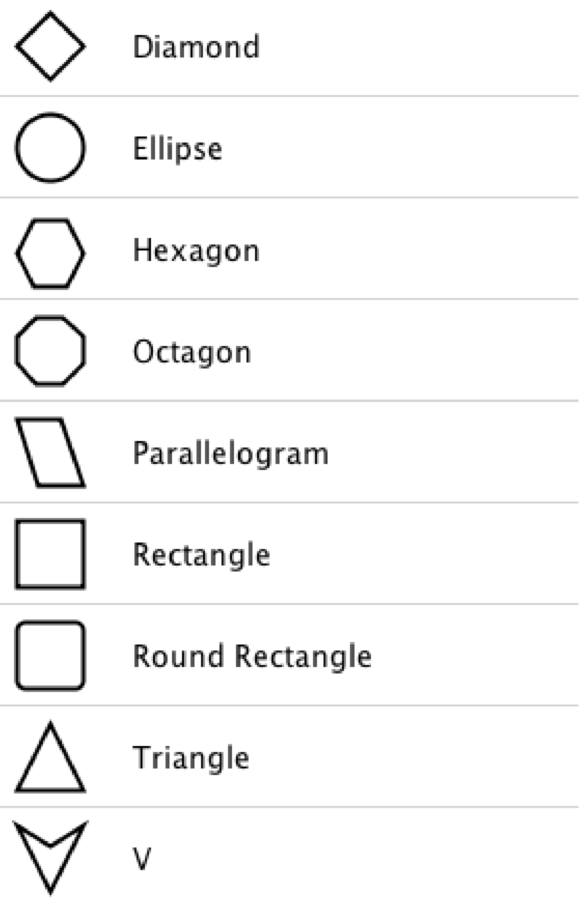
Network Properties

NETWORKPROPERTIES	DESCRIPTION
Background Paint	The background color of the network view.

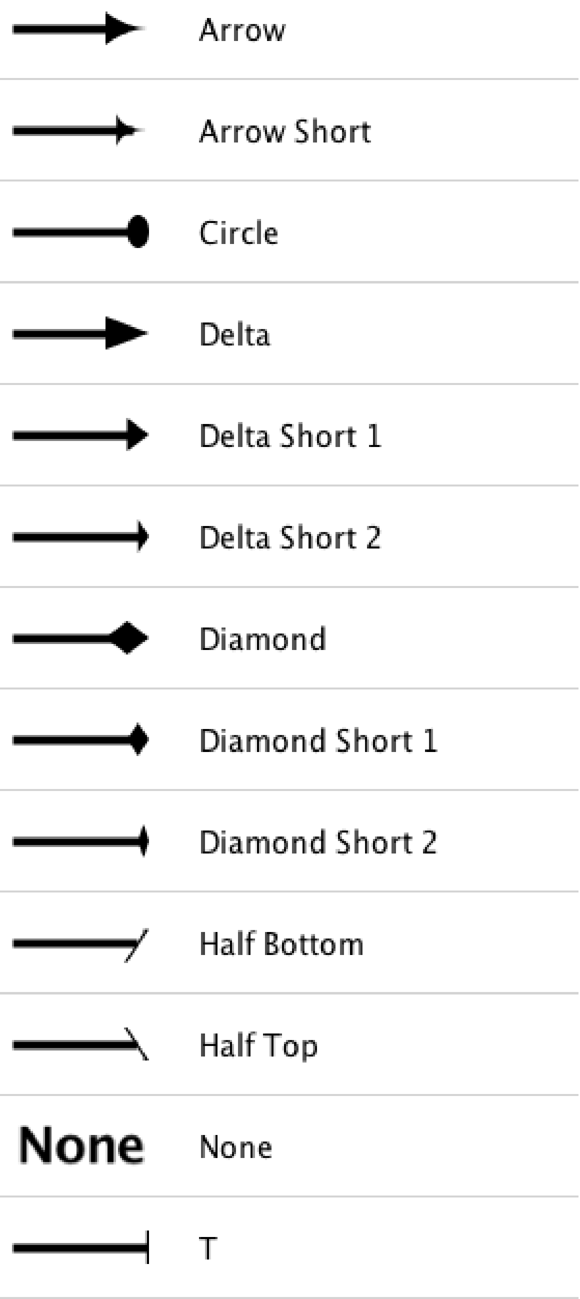
Center X Location	The X location of network view center.
Center Y Location	The Y location of network view center.
Edge Selection	Edges are selectable or not. If this is *false*, users cannot select edges.
Node Selection	Nodes are selectable or not. If this is *false*, users cannot select nodes.
Scale Factor	The zoom level of the network view.
Size	The size (width and height) of the network view. It can be found in the drop-down menu **Properties** → **Size** → **Size**.
Height	The height of the network view. It can be found in the drop-down menu **Properties** → **Size** → **Height**.
Width	The width of the network view. It can be found in the drop-down menu **Properties** → **Size** → **Width**.
Title	The title of the network view.

Available Shapes and Line Styles

Available Shapes and Line Styles

AVAILABLE SHAPES AND LINE STYLES	SAMPLE
Node Shapes	 Diamond Ellipse Hexagon Octagon Parallelogram Rectangle Round Rectangle Triangle V

Line Types	\\\\\\\\\\\\	Backward Slash
	>>>>>>>>>>	Contiguous Arrow
	– – – – – -	Dash
	–·–·–·–	Dash Dot
	··················	Dots
	- - - - - - - -	Equal Dash
	////////////	Forward Slash
	– – – – – -	Marquee Dash
	–·–·–·–	Marquee Dash Dot
	- - - - - - - -	Marquee Equal Dash
	═════	Parallel Lines
	→→→→→→→	Separate Arrow
	∿∿∿∿	Sinewave
	─────	Solid
	▬▬▬▬	Vertical Slash
	/\/\/\	Zigzag

Arrow Shapes	 Arrow Arrow Short Circle Delta Delta Short 1 Delta Short 2 Diamond Diamond Short 1 Diamond Short 2 Half Bottom Half Top None T

How Mappings Work

For each property, you can specify a default value or define a dynamic mapping. Cytoscape currently supports three different types of mappings:

1. **Passthrough Mapping**
    - The values of network column data are passed directly through to properties. A passthrough mapping is typically used to specify node/edge labels. For example, a passthrough mapping can label all nodes with their common gene names.
2. **Discrete Mapping**
    - Discrete column data are mapped to discrete properties. For example, a discrete

mapping can map different types of molecules to different node shapes, such as rectangles for gene products and ellipses for metabolites.

3. **Continuous Mapping**
   - Continuous data are mapped to properties. Depending on the property, there are three kinds of continuous mapping:

     i. **Continuous-to-Continuous Mapping**: for example, you can map a continuous numerical value to node size.

     ii. **Color Gradient Mapping**: This is a special case of continuous-to-continuous mapping. Continuous numerical values are mapped to a color gradient.

     iii. **Continuous-to-Discrete Mapping**: for example, all values below 0 are mapped to square nodes, and all values above 0 are mapped to circular nodes.
   - However, note that there is no way to smoothly morph between circular nodes and square nodes.

The table below shows mapping support for each property.

Legend

Legend

SYMBOL	DESCRIPTION
-	Mapping is not supported for the specified property.
+	Mapping is fully supported for the specified property.
o	Mapping is partially supported for the specified property. Support for "continuous to continuous" mapping is not supported.

Node Mappings

Node Mappings

NODE PROPERTY		PASSTHROUGH MAPPING	DISCRETE MAPPING	CONTINUOUS MAPPING
Color	*Fill Color*	+	+	+
	Transparency	+	+	+
	Border Paint	+	+	+
	Border Transparency	+	+	+
	Label Color	+	+	+
	Label Transparency	+	+	+
	Size/Width/Height	+	+	+

Numeric	Label Font Size	+	+	+
	Border Width	+	+	+
	Label Width	+	+	+
	Padding (Compound Node)	+	+	+
	Image/Chart Size	+	+	+
Other	Border Line Type	+	+	O
	Shape	+	+	O
	Shape (Compound Node)	+	+	O
	Label	+	+	O
	Tooltip	+	+	O
	Label Font Face	+	+	O
	Label Position	-	+	O
	Nested Network Image Visible	+	+	O
	Image/Chart	O	+	O
	Image/Chart Position	-	+	O

Edge Mappings

Edge Mappings

EDGE PROPERTY		PASSTHROUGH MAPPING	DISCRETE MAPPING	CONTINUOUS MAPPING
Color	Color	+	+	+
	Transparency	+	+	+
	Target Arrow Color	+	+	+
	Source Arrow Color	+	+	+
	Label Color	+	+	+
	Label Transparency	+	+	+
Numeric	Width	+	+	+
	Label Font Size	+	+	+
	Label Width	+	+	+
	Line Type	+	+	O
	Bend	-	+	O

Other	*Curved*	+	+	o
	Source Arrow Shape	+	+	o
	Target Arrow Shape	+	+	o
	Label	+	+	o
	Tooltip	+	+	o
	Label Font Face	-	+	o

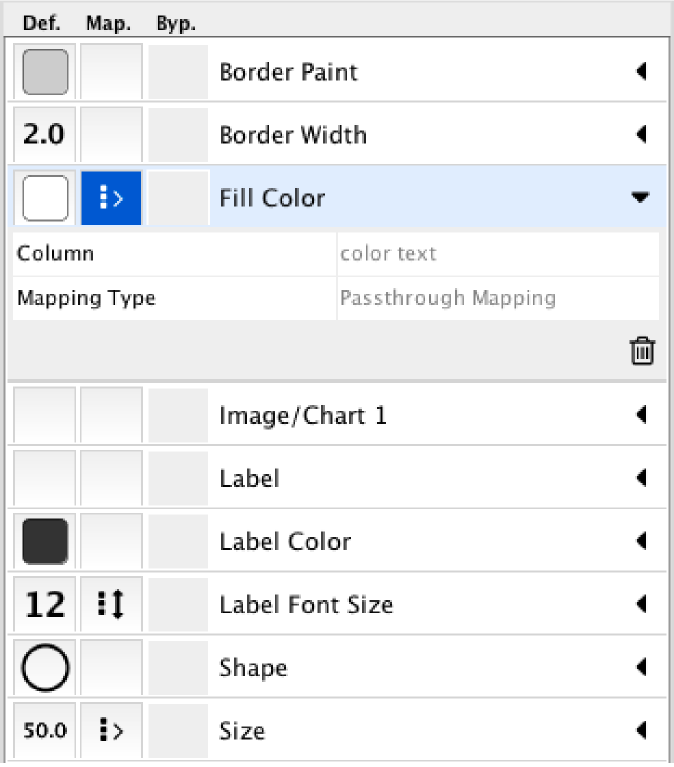
Text Passthrough Mapping

In Cytoscape 2.8.0 and later versions, the Passthrough Mapping can recognize some text representations of values. This means, if you have a string column named *Node Size Values*, you can directly map those values as the Node Size by setting “Node Size Values” as controlling column with **Node Size** “Passthrough Mapping”. The following value types are supported:

- **Colors:** Standard color names supported by all browsers or RGB representation in hex
- **Numerical Values:** Automatically mapped to the specified property.
- **Images:** URL String. If the URL is valid and an actual image data exists there, Cytoscape automatically downloads the image and maps it to the node.

Examples

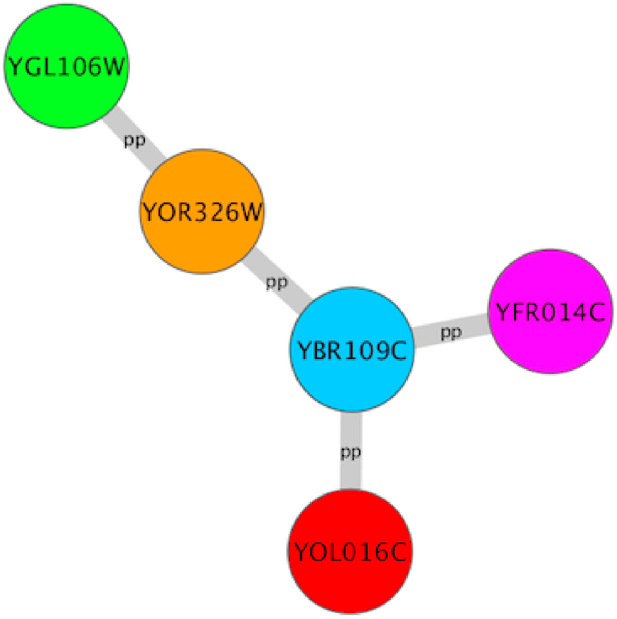
Color Passthrough Mapping



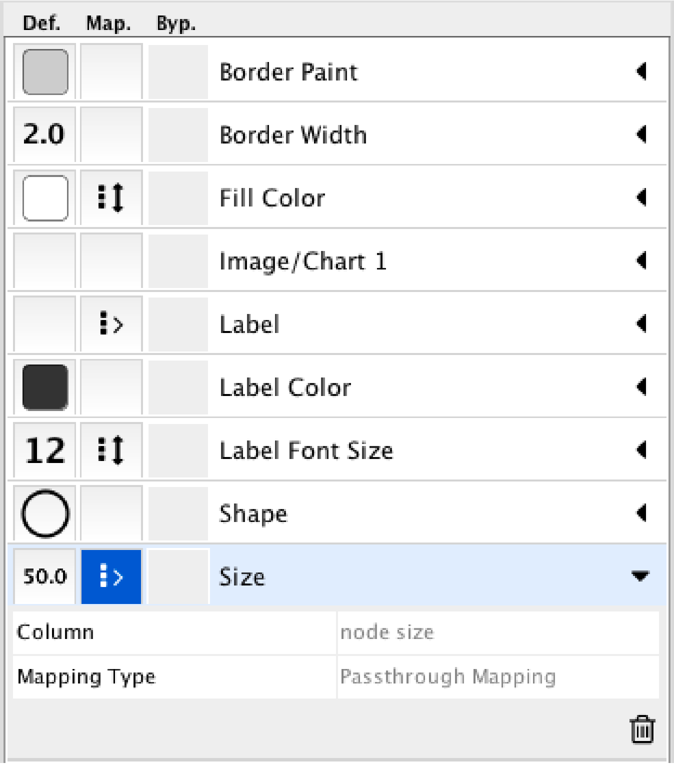
Def.	Map.	Byp.	
			Border Paint
2.0			Border Width
			Fill Color
			Image/Chart 1
			Label
			Label Color
12			Label Font Size
○			Shape
50.0			Size

Column: color text

Mapping Type: Passthrough Mapping



Node Size Passthrough Mapping



Def.	Map.	Byp.	
			Border Paint
2.0			Border Width
			Fill Color
			Image/Chart 1
			Label
			Label Color
12			Label Font Size
○			Shape
50.0			Size

Column	node size
Mapping Type	Passthrough Mapping

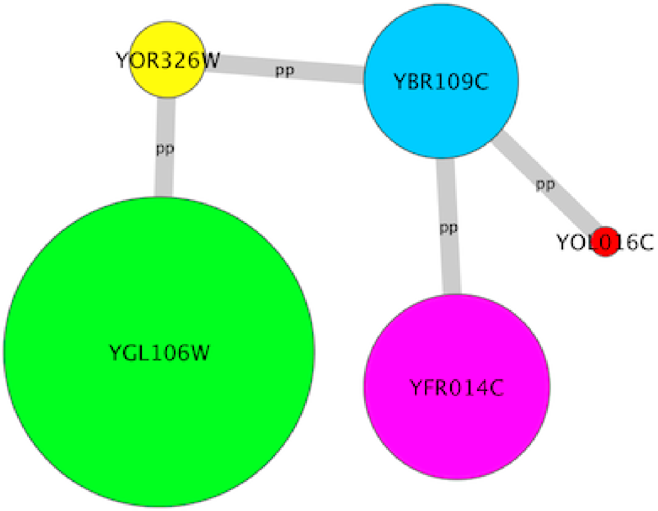
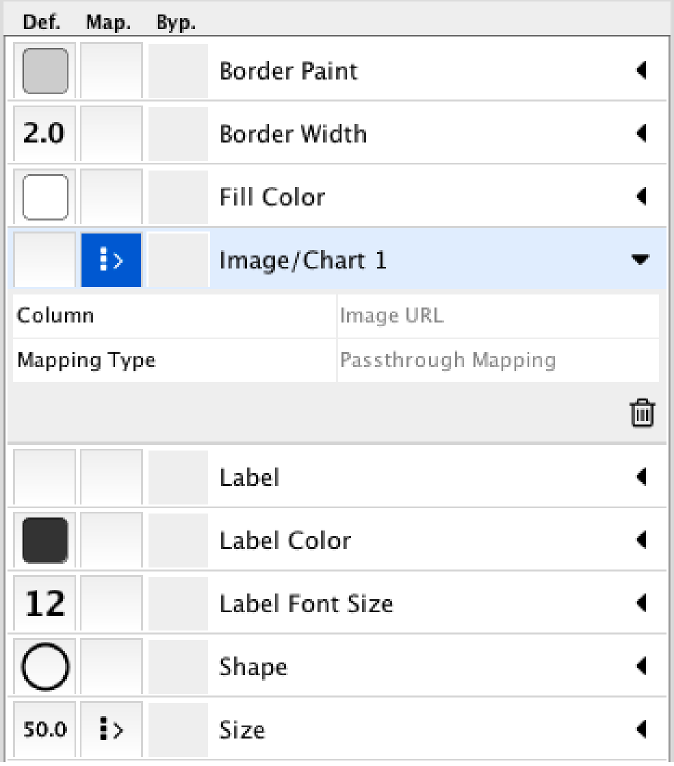


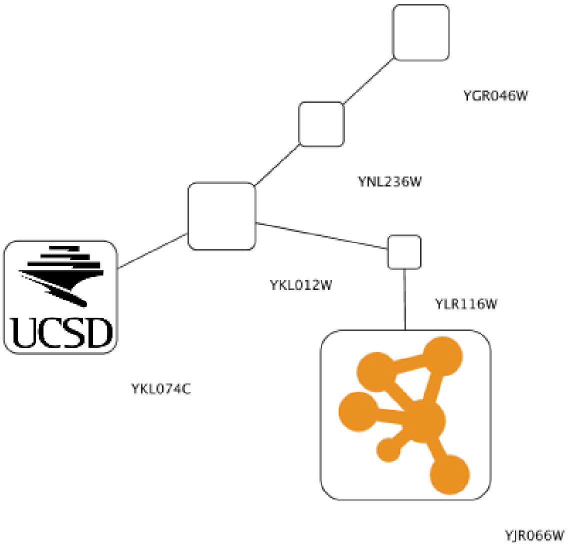
Image Passthrough Mapping



Def.	Map.	Byp.	
			Border Paint
2.0			Border Width
□			Fill Color
	⁝>		Image/Chart 1

Column	Image URL
Mapping Type	Passthrough Mapping

Def.	Map.	Byp.	
			Label
■			Label Color
12			Label Font Size
○			Shape
50.0	⁝>		Size



Images, Charts and Gradients

Cytoscape allows you to set custom graphics to nodes. Using the Style interface, you can map **Image/Chart** properties to nodes like any other property. Cytoscape provides a set of images and you can also add your own images in the **Image Manager**, as well as remove or modify existing ones.

Taxonomy Icon (http://biosciencedbc.jp/taxonomy_icon/taxonomy_icon.cgi?lng=en) set used in this section is created by Database Center for Life Science (DBCLS) and is distributed under Creative Commons License (CC BY 2.1.)

Managing Images

The **Image Manager** is available under the menu option **View** → **Open Image Manager...**:

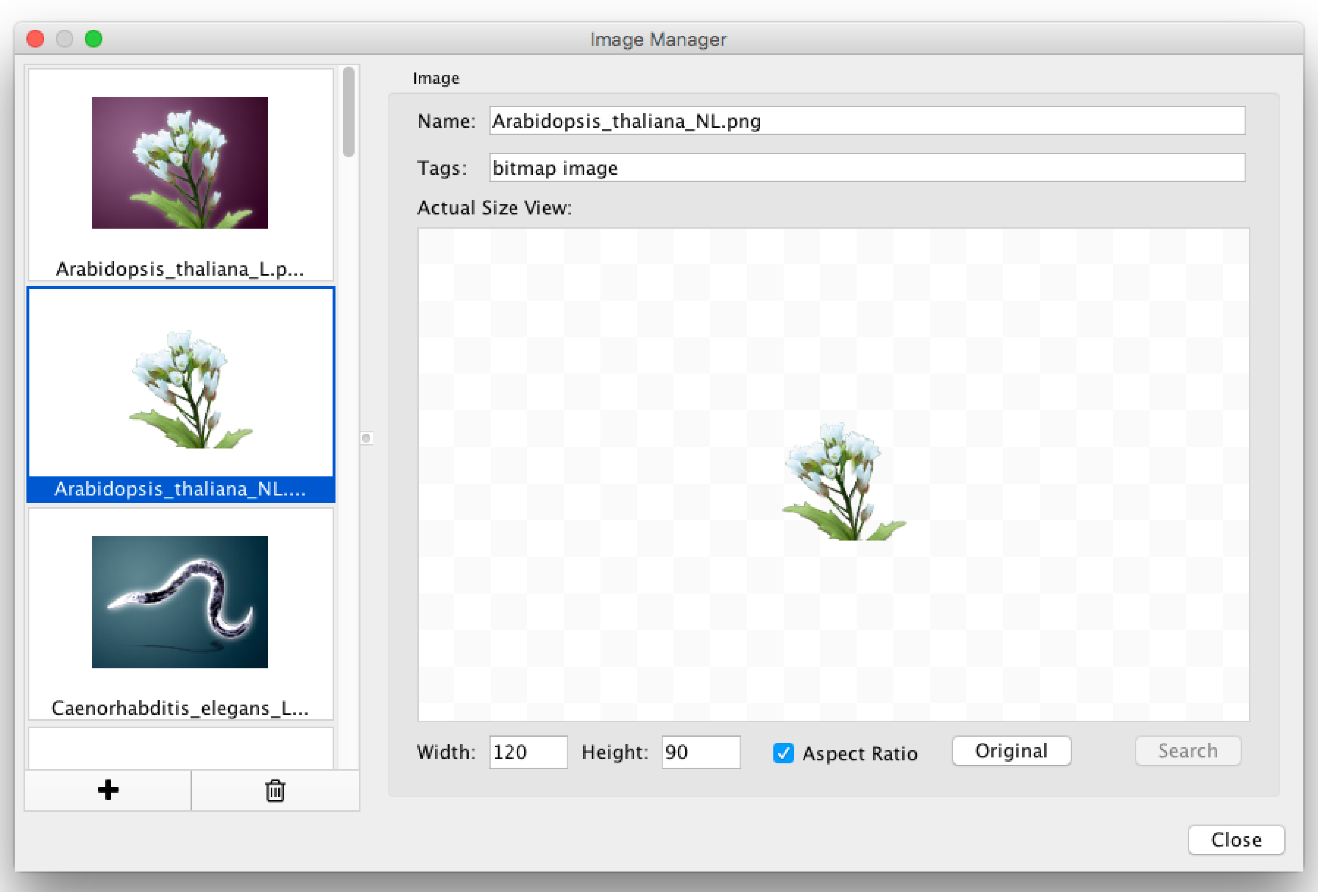


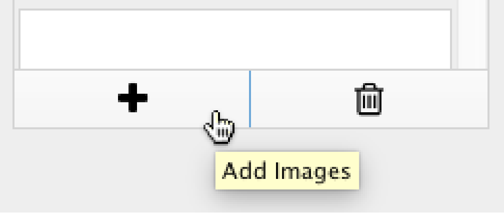
Image Manager
Arabidopsis_thaliana_L.p...
Arabidopsis_thaliana_NL....
Caenorhabditis_elegans_L...
+
Image
Actual Size View:
Original
Search
Close

Name: Arabidopsis_thaliana_NL.png

Tags: bitmap image

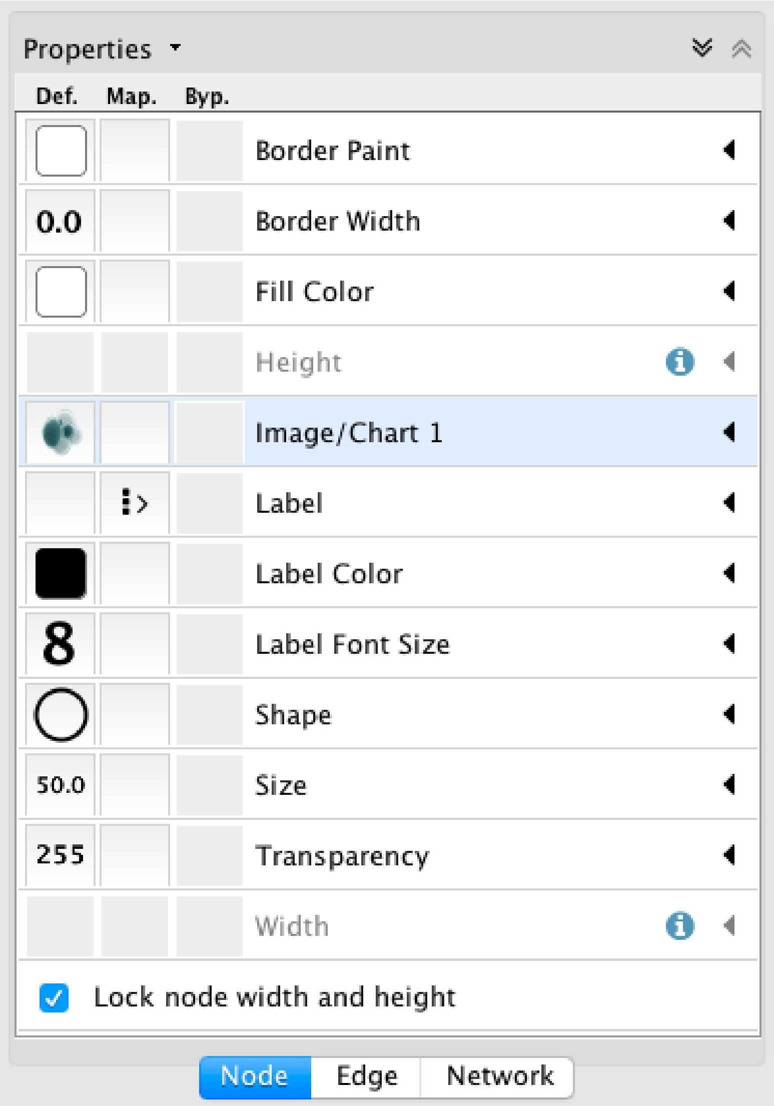
Width: 120 Height: 90 ☑ Aspect Ratio

- You can add images by drag-and-drop of image files and URLs. If you want to add images from a web browser or local file system, you can drag images from them and drop those images onto the list of images on the left.
    - **Note:** When you drag and drop images from web browser, make sure that you are actually dragging the URL for the image. In some cases, images are linked to an HTML page or scripts, and in such cases, this drag and drop feature may not work.
- If you want to add one or more images from a folder, press the **+** button on the bottom of the Image Manager window and then select the images you want to add.



- To remove images from the current session's image library, simply select one or more images from the list and press the **Remove Selected Images** button (trash icon).
- Images can be resized by defining specific **Width** and **Height** values. If the **Aspect Ratio** box is checked, the width-height ratio is always synchronized. You can resize the image to the original size by pressing the **Original** button.

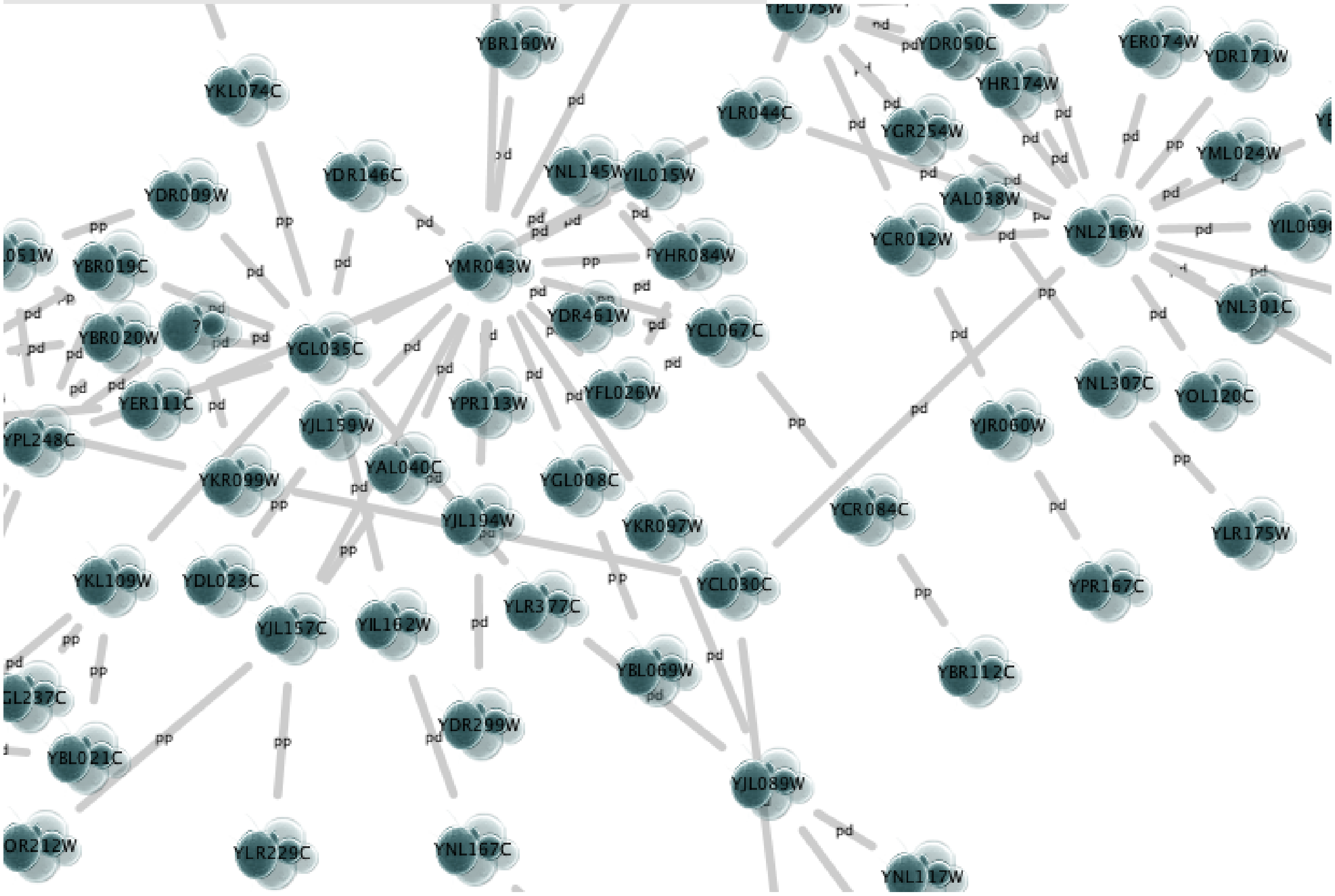
Using Graphics in Styles



Properties ▾

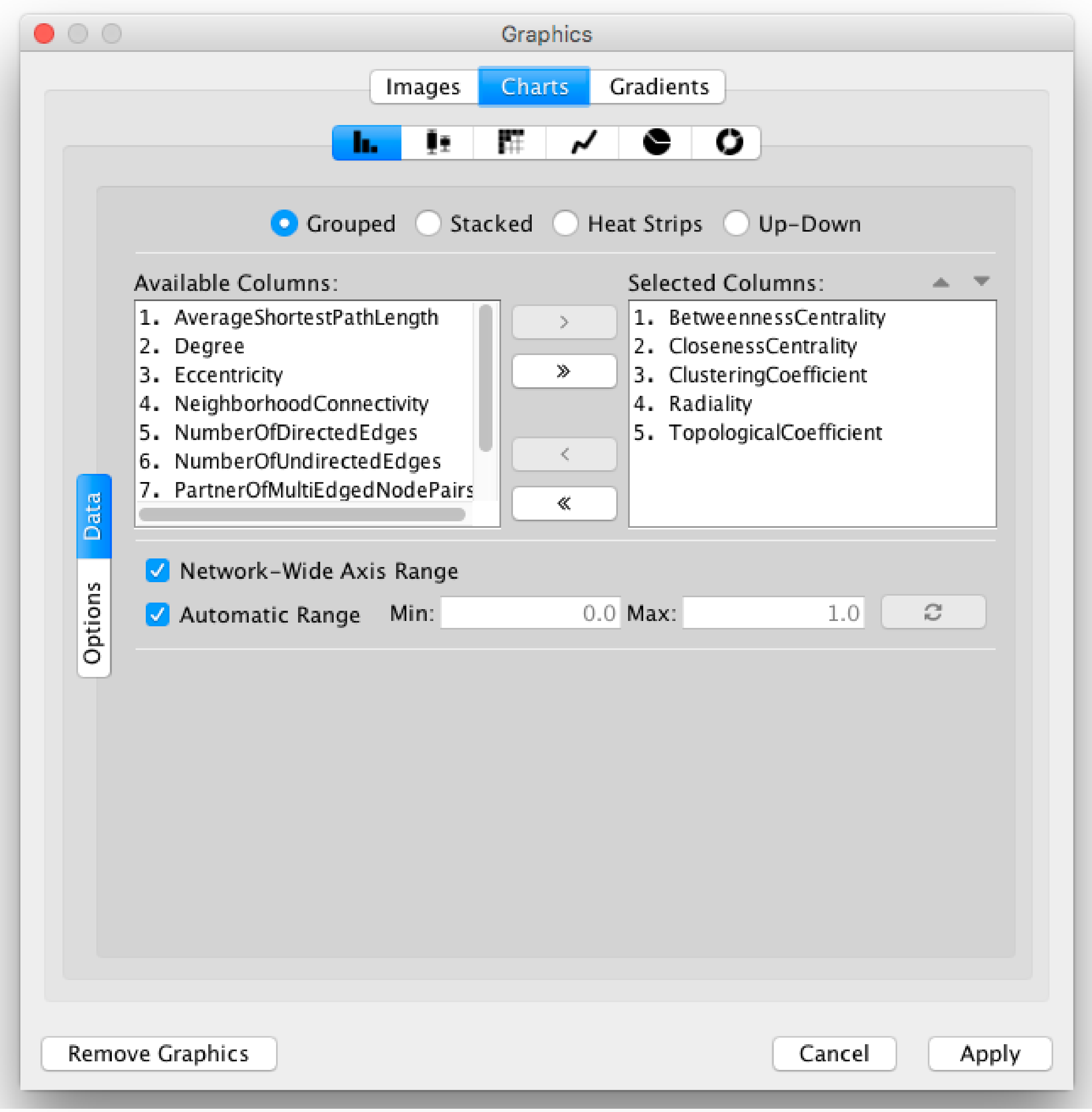
Def.	Map.	Byp.	
□			Border Paint
0.0			Border Width
□			Fill Color
			Height
			Image/Chart 1
			Label
■			Label Color
8			Label Font Size
○			Shape
50.0			Size
255			Transparency
			Width

☑ Lock node width and height



Node graphics are used and defined like any other property, through the **Style** interface. There are nine **Image/Chart** properties.

- Cytoscape provides three kinds of graphics (selectable via tabs on the **Graphics** dialog):
  - **Images:** You can select one of the provided images or add your own (click the **Open Image Manager...** button to add more images to the list).
  - **Charts:** The following chart types are available: *Bar* , *Box*, *Heat Map*, *Line*, *Pie*, *Ring*.
  - **Gradients:** You can also set *Linear* and *Radial* gradients to nodes.



● Grouped ○ Stacked ○ Heat Strips ○ Up-Down

Available Columns:

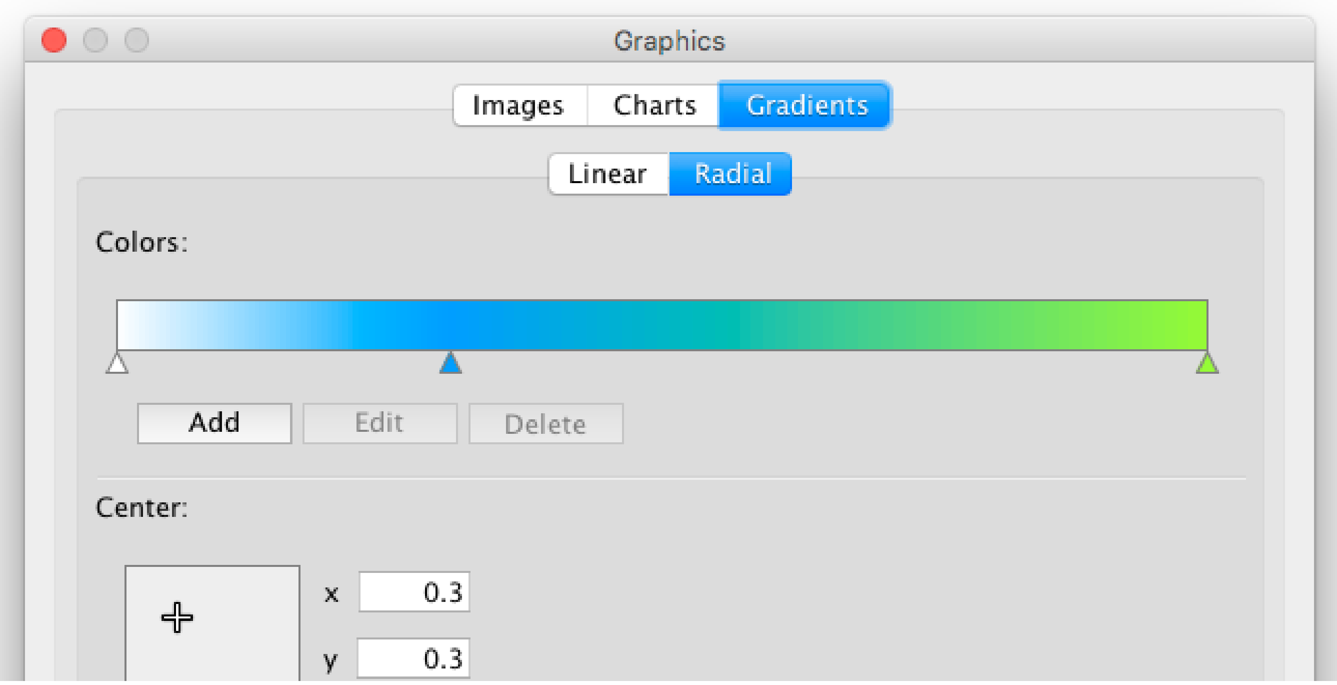
1. AverageShortestPathLength
2. Degree
3. Eccentricity
4. NeighborhoodConnectivity
5. NumberOfDirectedEdges
6. NumberOfUndirectedEdges
7. PartnerOfMultiEdgedNodePairs

Selected Columns:

1. BetweennessCentrality
2. ClosenessCentrality
3. ClusteringCoefficient
4. Radiality
5. TopologicalCoefficient

☑ Network-Wide Axis Range

☑ Automatic Range Min: 0.0 Max: 1.0

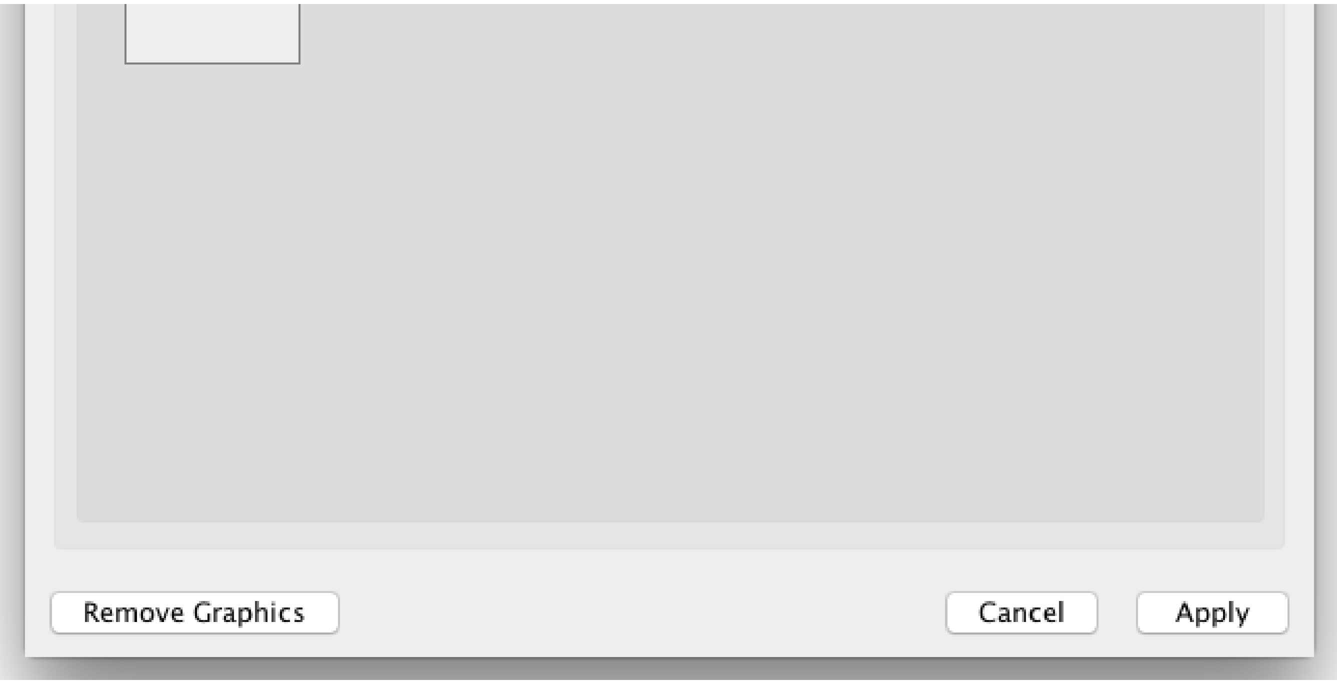


Colors:

Center:

x 0.3

y 0.3



- To add a graphic, first add one **Image/Chart** property to the properties list in the **Style** interface (on the **Node** tab, select **Properties** → **Paint** → **Custom Paint** ***n*** → **Image/Chart** ***n***). Next, click the **Default Value** column of the **Image/Chart** property to bring up the **Graphics** dialog. Select an image, a chart or a gradient and then click **Apply**.
  - By default, graphics are automatically resized to be consistent with the **Node Size** property.
- To remove an image, chart or gradient, click the **Remove Graphics** button on the **Graphics** dialog.

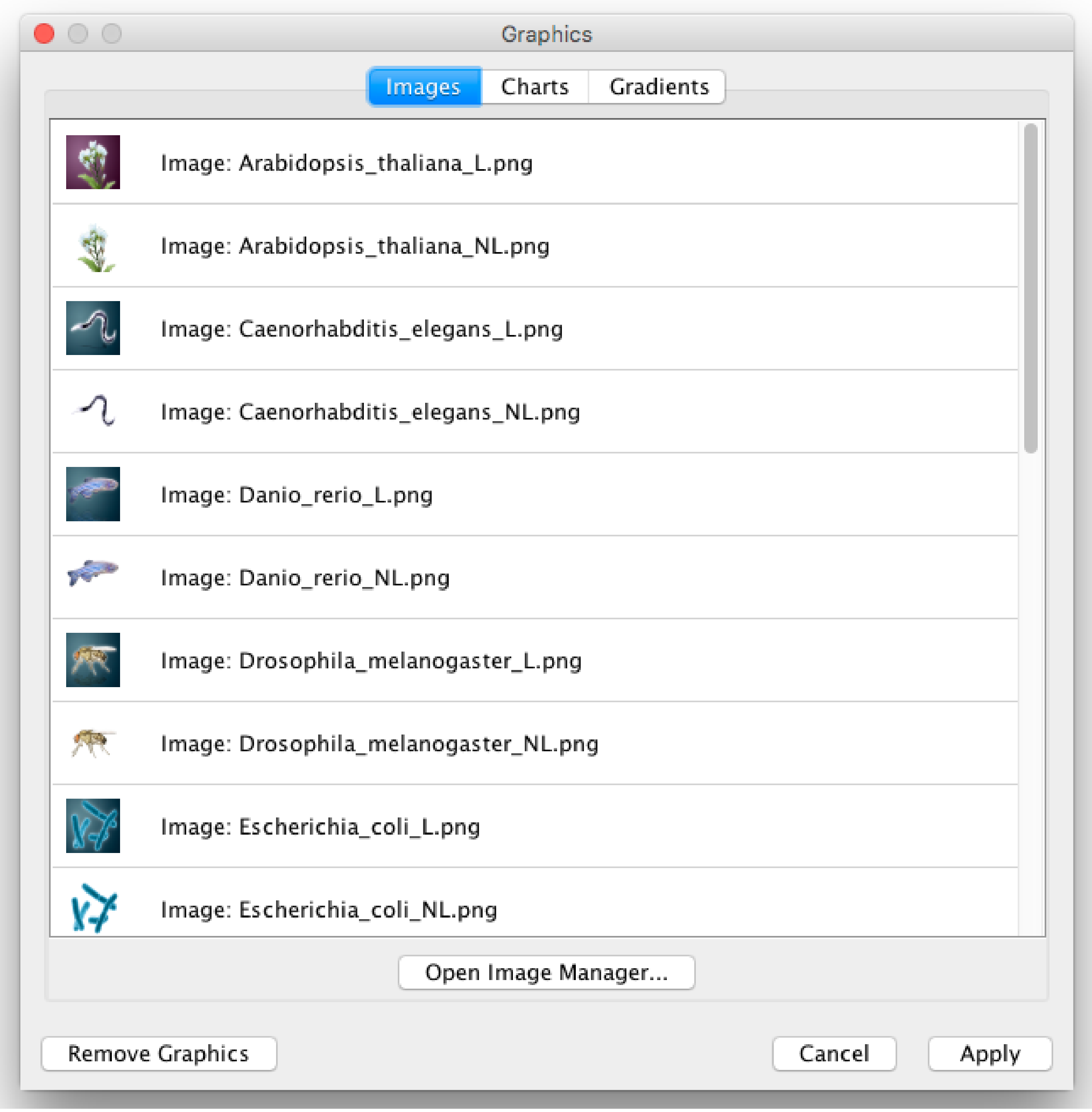


Image: Arabidopsis_thaliana_L.png

Image: Arabidopsis_thaliana_NL.png

Image: Caenorhabditis_elegans_L.png

Image: Caenorhabditis_elegans_NL.png

Image: Danio_rerio_L.png

Image: Danio_rerio_NL.png

Image: Drosophila_melanogaster_L.png

Image: Drosophila_melanogaster_NL.png

Image: Escherichia_coli_L.png

Image: Escherichia_coli_NL.png

Graphics Positions

Each **Image/Chart** property is associated with a position. You can edit its position by using the UI available in the **Default Value** column for the **Image/Chart Position** property that has the same number. For instance, the **Image/Chart Position 2** value modifies the position of **Image/Chart 2**.

- **Note:** Setting graphics positions for *Linear* or *Radial* gradients has no effect, as they are always centered on the node.

Z-Ordering

The number that appears with the **Image/Chart** property represents an ordering of layers. Basic node color and shape are always rendered first, then node Image/Chart 1, 2, ..., through 9.

Saving and Loading Images

In general, saving and loading images is automatic. When you quit Cytoscape, all of the images in the **Image Manager** will be saved automatically. There are two types of saving:

1. To a session file
    - When you save the current session to a file, the images used in the current styles will be saved to that file. For example, if you have a style with a discrete mapping for **Image/Chart 1**, all images used in the style will be saved to the session file. Other images will **not** be saved in your session file. This is because your image library can be huge when you add thousands of images to the Image Manager and it takes a very long time to save and load the session file.
2. Automatic saving to `CytoscapeConfiguration/images3` directory
    - When you select **File** → **Quit** (Windows and Linux) or **Cytoscape** → **Quit Cytoscape** (Mac OS X), all of the images in the Image Manager will be saved automatically to your Cytoscape settings directory. Usually, they are saved in `YOUR_HOME_DIRECTORY/CytoscapeConfiguration/images3`.

In any case, images will be saved automatically to your system or session and will be restored when you restart Cytoscape or load a session.

Styles Tutorials

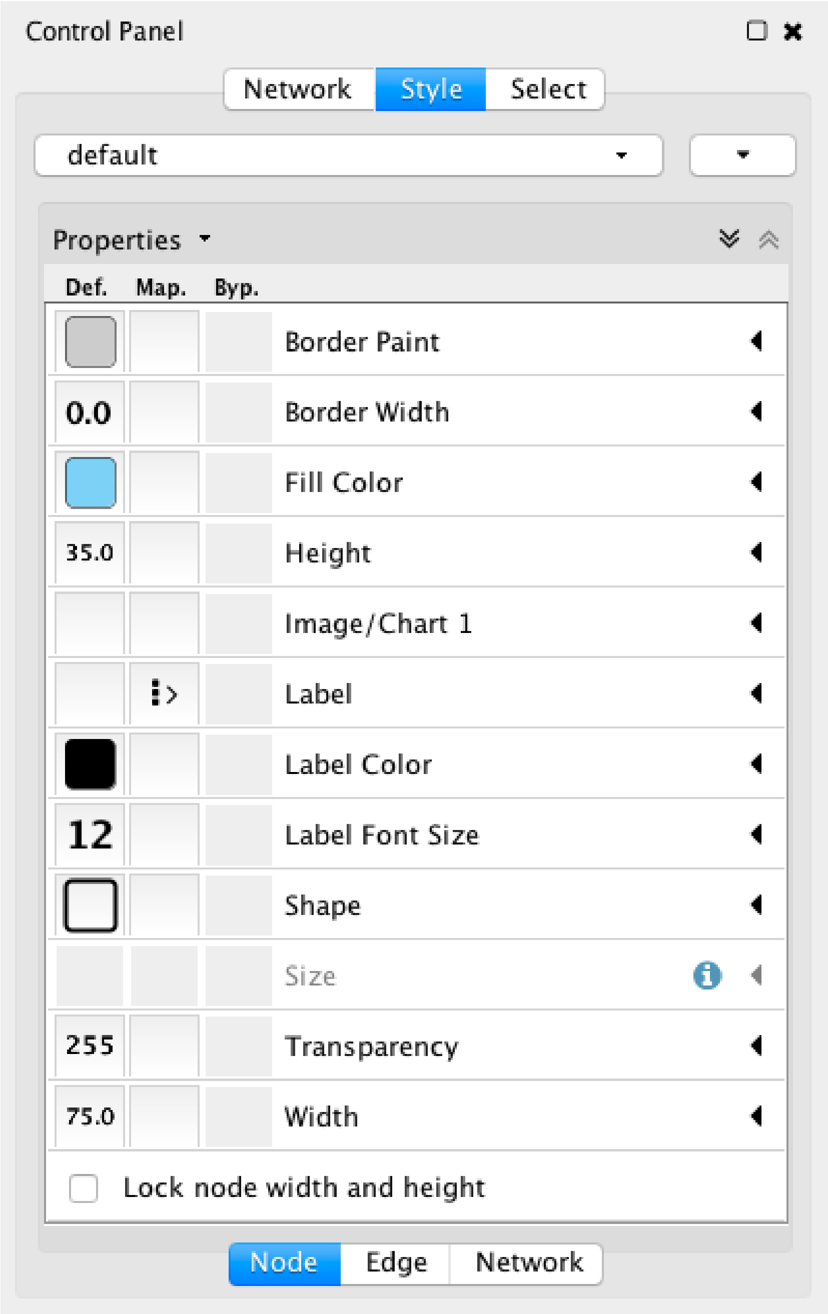
The following tutorials demonstrate some of the basic **Style** features. Each tutorial is independent of the others.

Tutorial 1: Creating a Basic Style and Setting Default Values

The goal of this tutorial is to learn how to create a new Style and set some default values.

1. **Load a sample network:** From the main menu, select **File** → **Import** → **Network** → **File...**, and select `sampleData/galFiltered.sif`.
2. **Create some node/edge statistics:** The **Network Analyzer** calculates some basic statistics for nodes and edges. From the main menu, select **Tools** → **Network Analyzer** → **Network Analysis** → **Analyze Network**, and click **OK**. Once the result is displayed, simply close the window. All statistics are stored as regular table data.

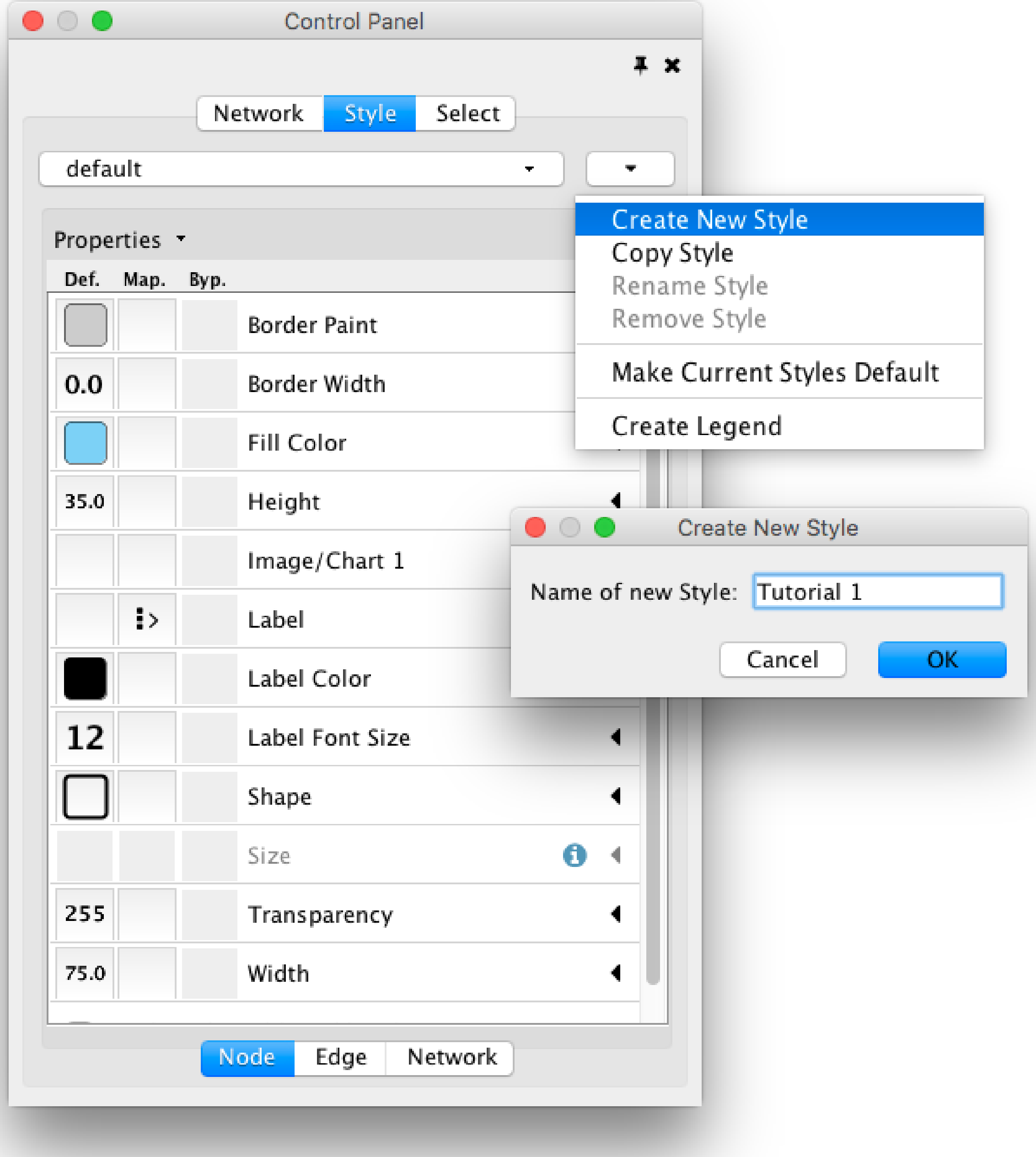
3. Select the **Style** panel in the Control Panel.



Def.	Map.	Byp.	
▢			Border Paint
0.0			Border Width
▢			Fill Color
35.0			Height
			Image/Chart 1
	⋮>		Label
■			Label Color
12			Label Font Size
□			Shape
			Size
255			Transparency
75.0			Width

□ Lock node width and height

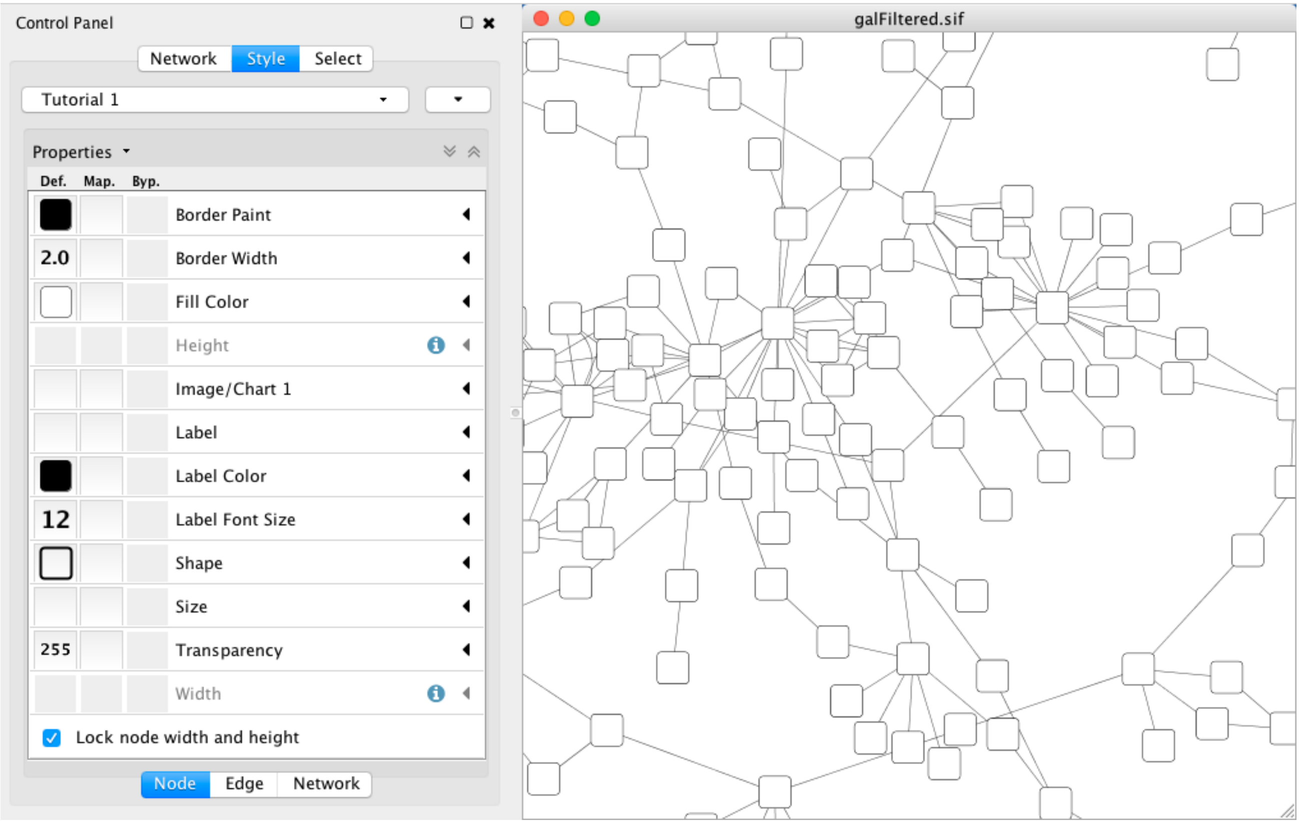
4. **Create a new style:** Click the **Options** ▾ drop-down, and select **Create New Style**. Enter a name for your new style when prompted.



Name of new Style: Tutorial 1

Since no mappings are set up yet, only default values are defined for some of the properties. From this panel, you can create node/edge mappings for all properties.

1. **Change the default node color and shape:** To set the default node shape to triangles, click the **Default Value** column for the **Shape** property. A list of available node shapes will be shown. Select the **Triangle** item and click the **Apply** button. You can edit other default values in the same way. In the example shown below, the node shape is set to **Round Rectangle**, while **Fill Color** is set to white. The new Style is automatically applied to the current network, as shown below.



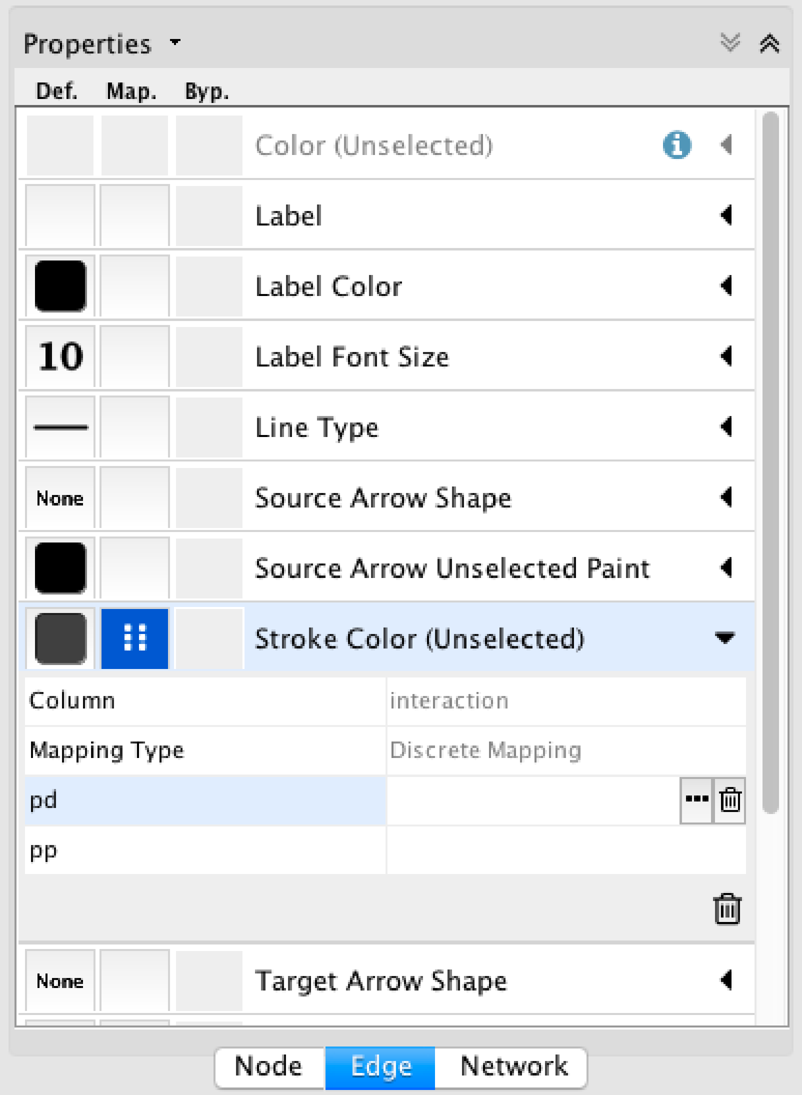
Def.	Map.	Byp.	
■			Border Paint
2.0			Border Width
□			Fill Color
			Height
			Image/Chart 1
			Label
■			Label Color
12			Label Font Size
□			Shape
			Size
255			Transparency
			Width

☑ Lock node width and height

Tutorial 2: Creating a New Style with a Discrete Mapping

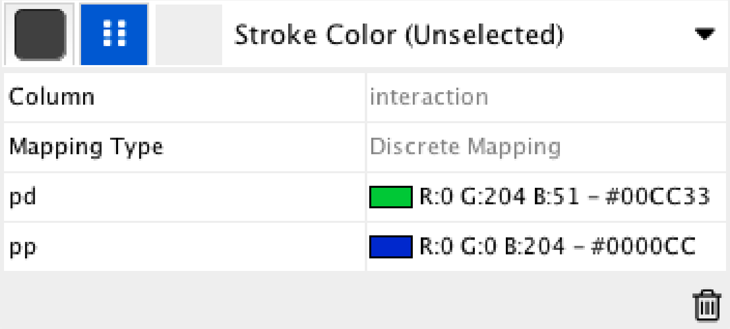
Now you have a network with a new Style. The following section demonstrates how to create a new style that has a discrete mapping. The goal is to draw protein-DNA interactions as dashed lines, and protein-protein interactions as solid lines.

1. **Find the property:** In the Edge tab of the Style panel, find the **Stroke Color (Unselected)** property. If it is not already visible in the properties sheet, add it by selecting the drop-down item **Properties** → **Paint** → **Color (Unselected)** → **Stroke Color (Unselected)**.
2. **Choose a data column to map to:** Expand the entry for **Stroke Color (Unselected)** by clicking the arrow icon on the right. Click the **Column** entry and select “interaction” from the drop-down list that appears.
3. **Set the mapping type:** Under **Mapping Type**, select “Discrete Mapping”. All available column values for “interaction” will be displayed, as shown below.



Column	interaction
Mapping Type	Discrete Mapping
pd	
pp	

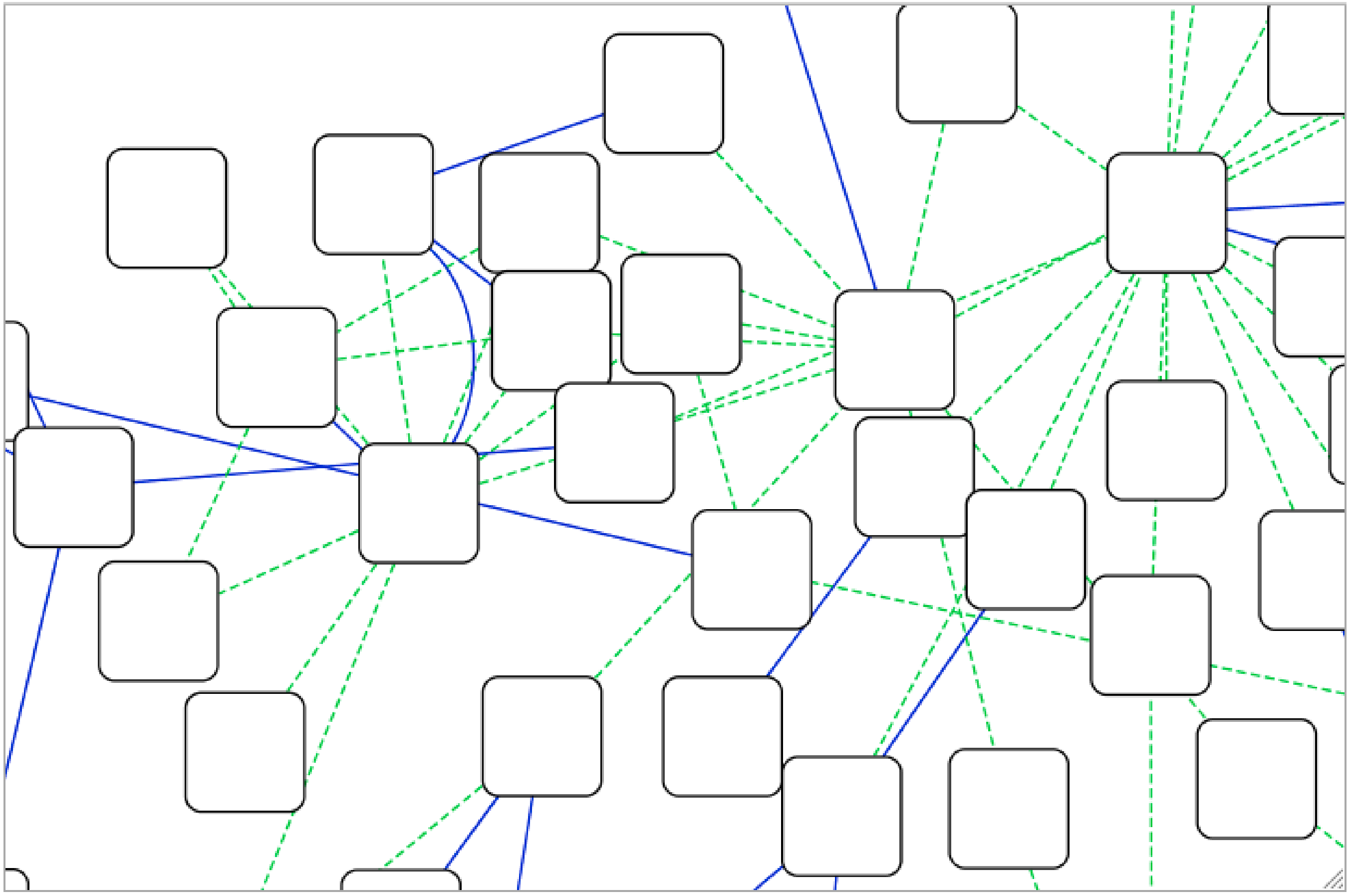
4. **Set the mapped values:** Click the empty cell next to “pd” (protein-DNA interactions). On the right side of the cell, click on the ... button that appears. A popup window will appear; select green or similar, and the change will immediately appear on the network window.



Column	interaction
Mapping Type	Discrete Mapping
pd	🟩 R:0 G:204 B:51 – #00CC33
pp	🟦 R:0 G:0 B:204 – #0000CC

Repeat step 4 for “pp” (protein-protein interactions), but select a darker color. Then repeat steps 3 through 4 for the *Line Type* property, by selecting the correct line style (“Dash” or “Solid”) from the list.

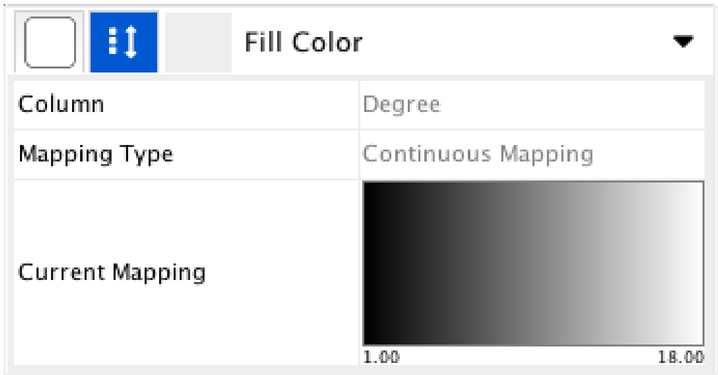
Now your network should show “pd” interactions as dashed green lines and “pp” interactions as solid lines. A sample screenshot is provided below.



Tutorial 3: Creating a New Style with a Continuous Mapping

At this point, you have a network with some edge mappings. Next, let's create mappings for nodes. The following section demonstrates how to create a new style using a continuous mapping. The goal is to superimpose node statistics (in this example, node degree) onto a network and display it along a color gradient.

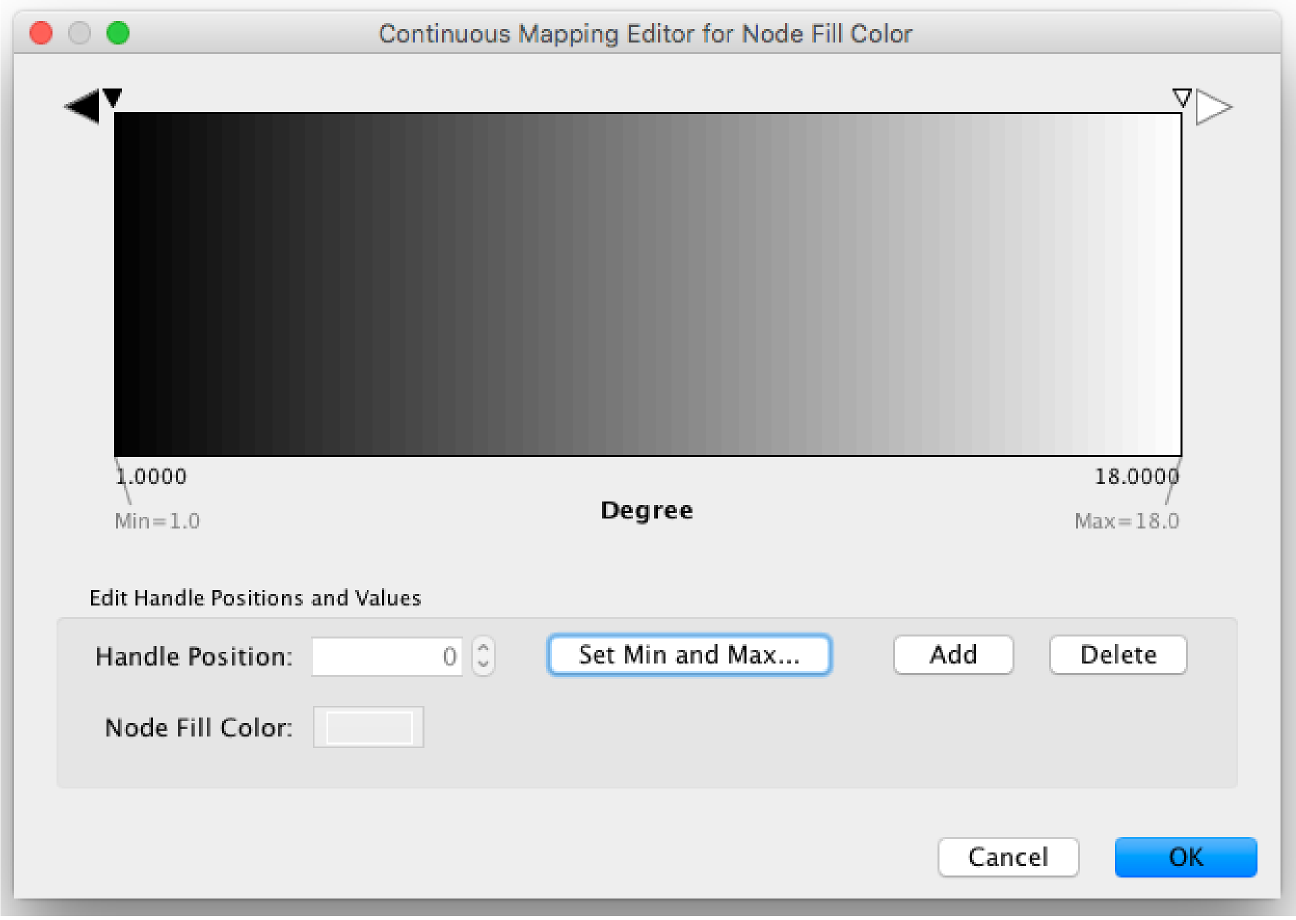
1. **Find the property:** In the Node tab of the Style panel, find the **Fill Color** property. If it is not already visible in the properties sheet, add it by selecting the drop-down item **Properties** → **Paint** → **Fill Color**.
2. **Set the node table column:** Expand the entry for **Fill Color** by clicking the arrow icon on the right. Click the **Column** entry and select "Degree" from the drop-down list that appears.
3. **Set the mapping type:** Set the "Continuous Mapping" option as the **Mapping Type**. This automatically creates a default mapping.



Fill Color

Column	Degree
Mapping Type	Continuous Mapping
Current Mapping	1.00 18.00

4. **Define the points where colors will change:** Double-click on the black-and-white gradient rectangle next to **Current Mapping** to open the **Continuous Mapping Editor**. Note the two smaller triangles at the top of the gradient.



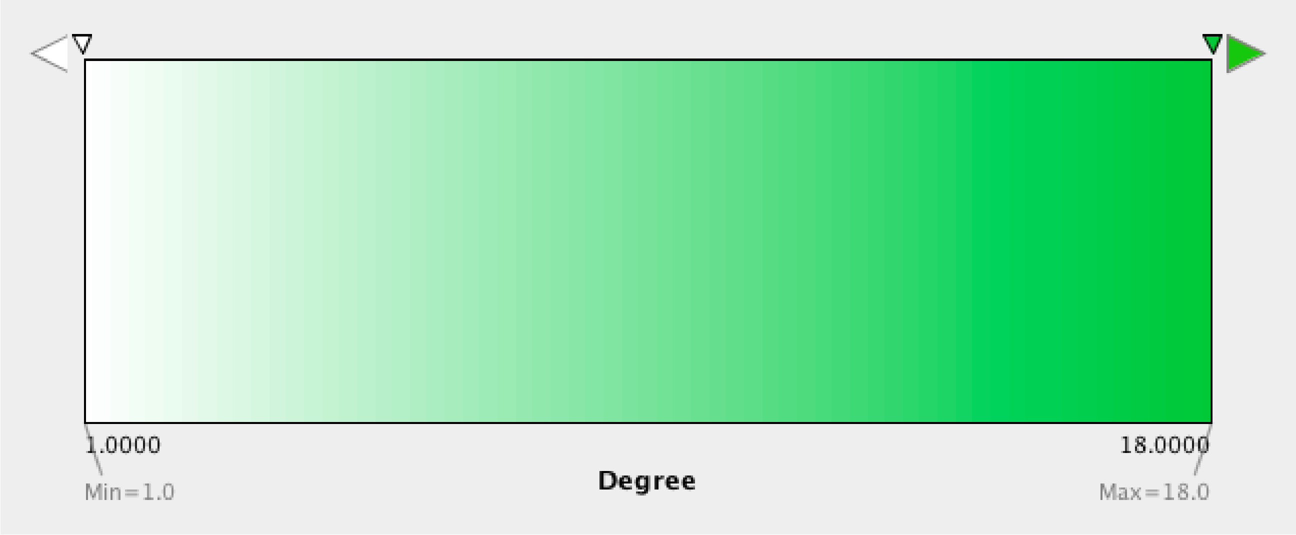
Continuous Mapping Editor for Node Fill Color

Edit Handle Positions and Values

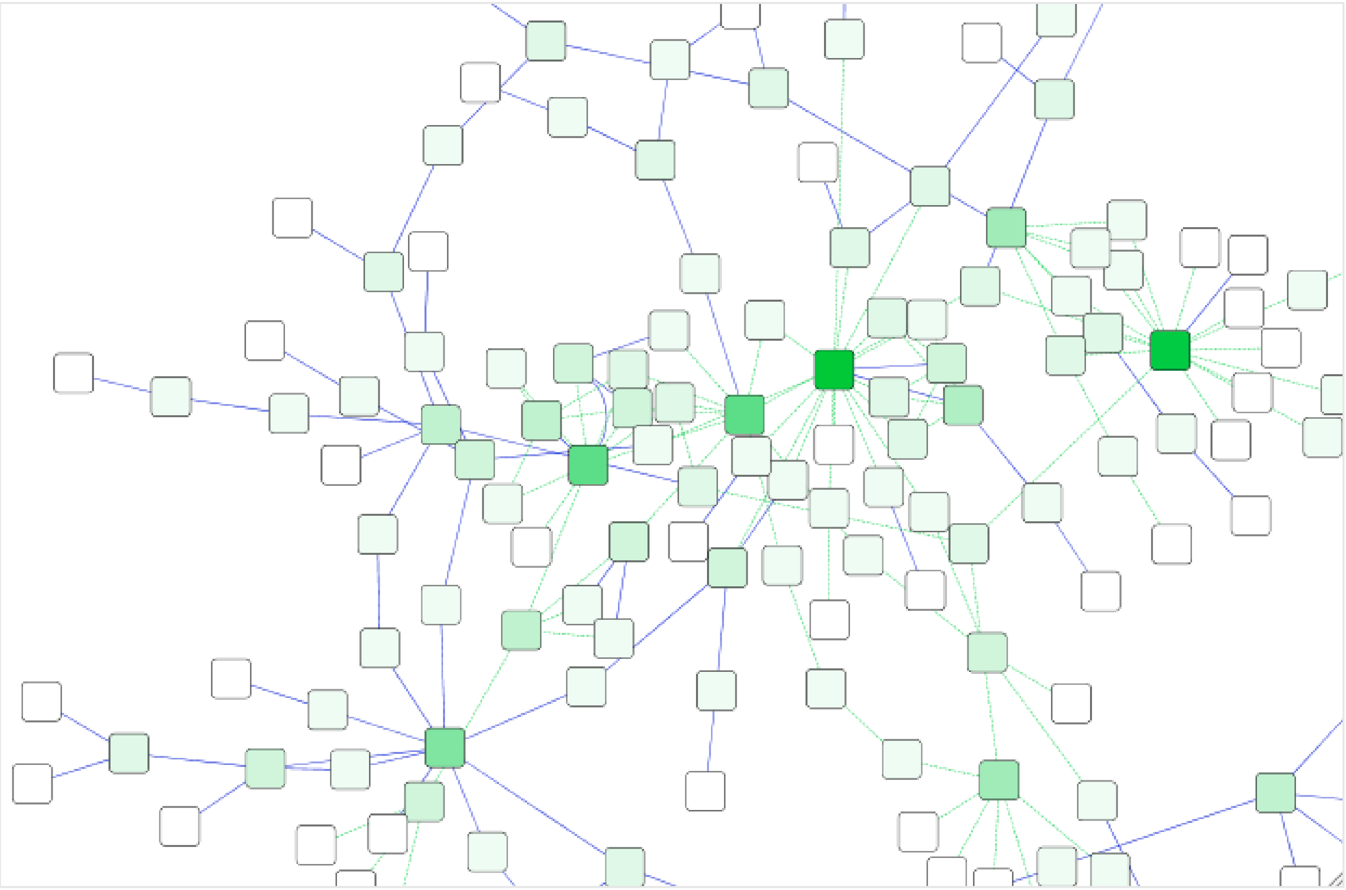
Handle Position: 0

Node Fill Color:

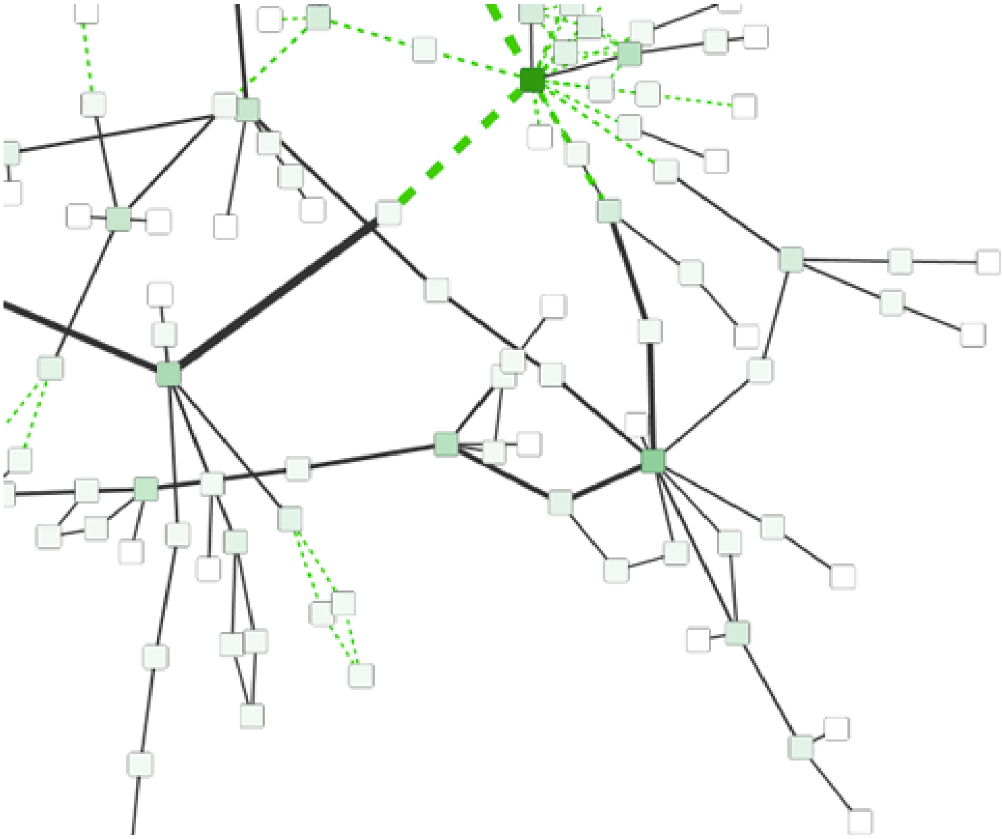
5. **Define the colors between points:** Double-click on the larger leftmost triangle (facing left) and a color palette will appear. Set the color white and repeat for the smaller left-side triangle. For the triangle on the right, set the color green and then choose the same color for the smaller right-side triangle.



The color gradients will immediately appear on the network. All nodes with degree *1* will be set to white, and all values between *1* and *18* will be painted with a white/green color gradient (see the sample screenshot below).



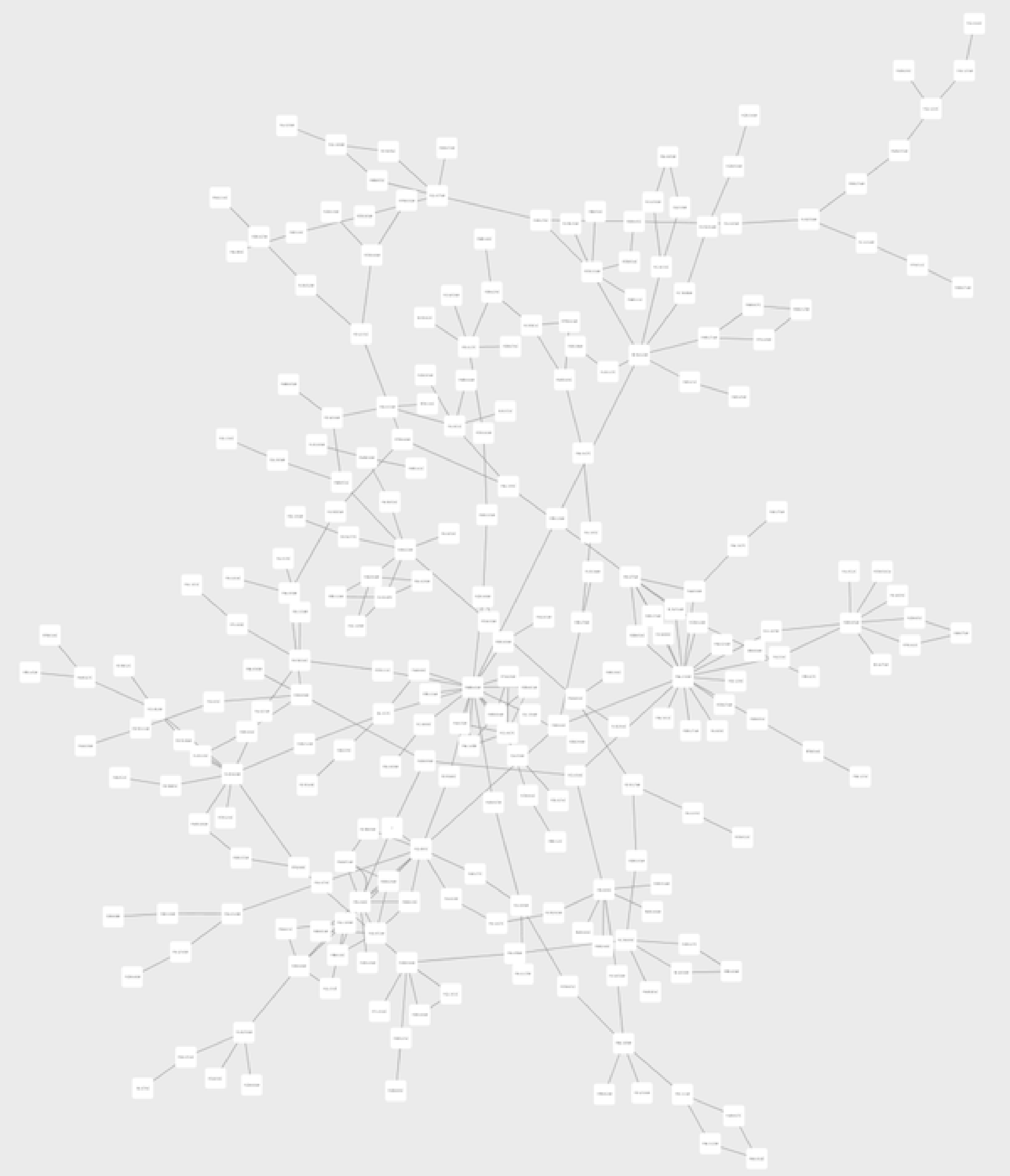
- **Repeat for other properties:** You can create more continuous mappings for other numeric table data. For example, edge data table column “EdgeBetweenness” is a number, so you can use it for continuous mapping. The following is an example visualization which mapps *Edge Width* to “EdgeBetweenness”.



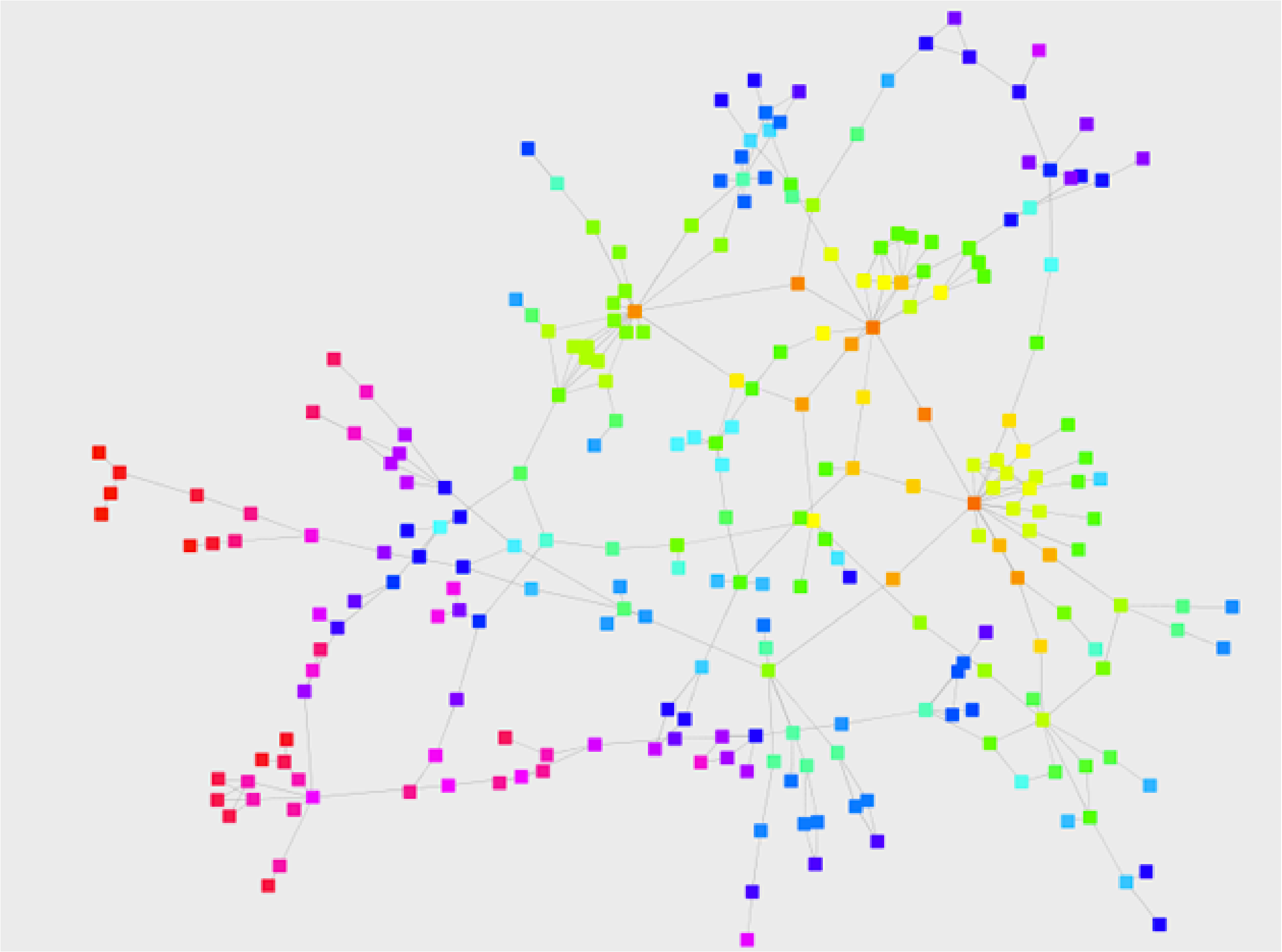
Tutorial 4: Setting Automatic Values to a Discrete Mapping

The goal of this section is to learn how to generate automatic values for discrete mappings.

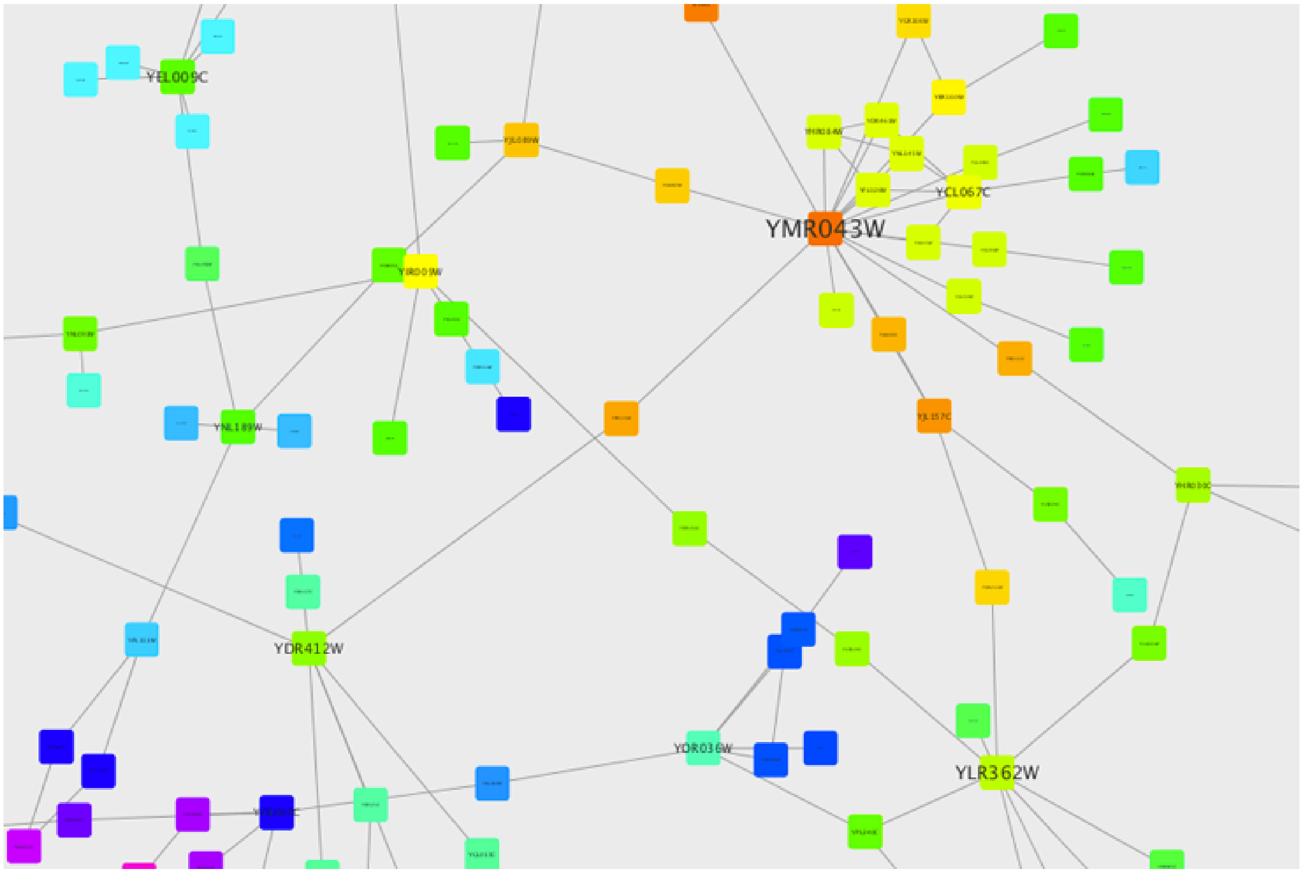
1. Switch the Current Style to **Minimal**. Now your network looks like the following:



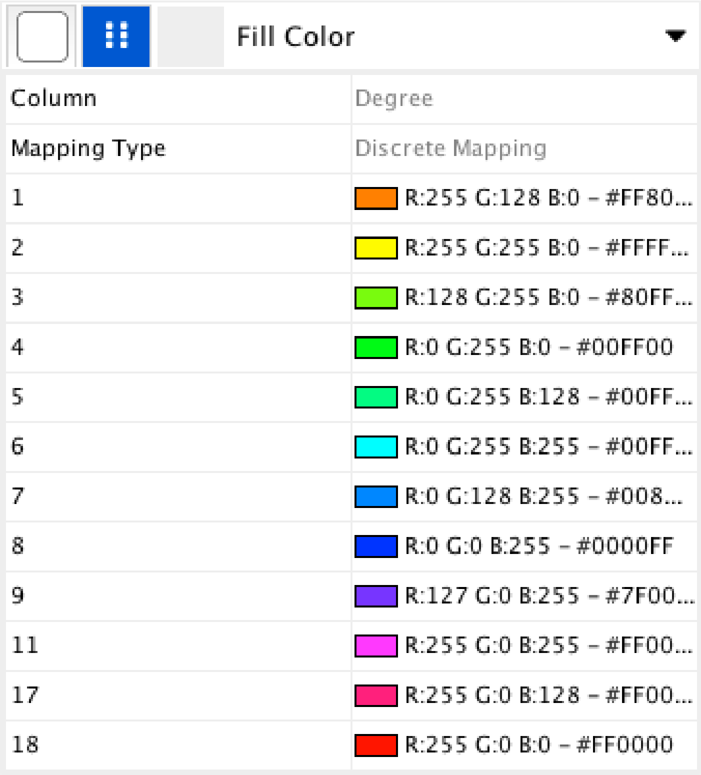
2. Create a discrete mapping for **Fill Color**. Select “AverageShortestPathLength” (generated by the Network Analyzer) as the controlling property.
3. Right-click the **Fill Color** cell, then select **Mapping Value Generators** → **Rainbow**. Cytoscape will automatically generate different colors for all the property values as shown below:



4. Create a discrete mapping for **Label Font Size**. Select “AverageShortestPathLength” as controlling property (**Column** field).
5. Right-click the **Label Font Size** cell, then select **Mapping Value Generators** → **Number Series**. Type **3** for the first value and click OK. Enter *3* for increment.
6. Apply **Layout** → **yFiles Layouts** → **Organic**. The final view is shown below:



This mapping generator utility is useful for categorical data. The following example shows a discrete mapping that maps the species column to node color.

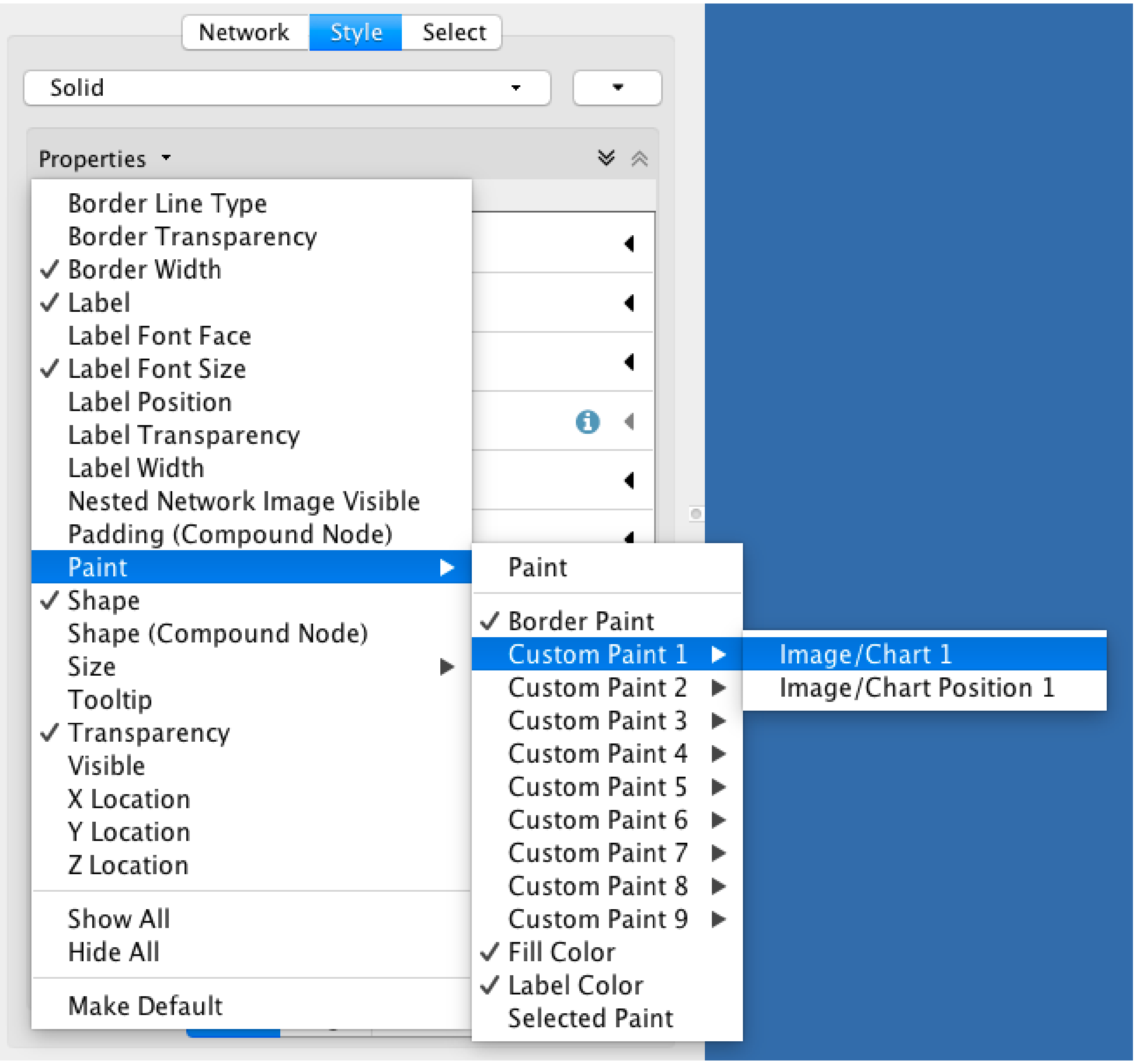

Fill Color ▼

Column	Degree
Mapping Type	Discrete Mapping
1	🟧 R:255 G:128 B:0 – #FF80...
2	🟨 R:255 G:255 B:0 – #FFFF...
3	🟩 R:128 G:255 B:0 – #80FF...
4	🟩 R:0 G:255 B:0 – #00FF00
5	🟩 R:0 G:255 B:128 – #00FF...
6	🟦 R:0 G:255 B:255 – #00FF...
7	🟦 R:0 G:128 B:255 – #008...
8	🟦 R:0 G:0 B:255 – #0000FF
9	🟪 R:127 G:0 B:255 – #7F00...
11	🟪 R:255 G:0 B:255 – #FF00...
17	🟥 R:255 G:0 B:128 – #FF00...
18	🟥 R:255 G:0 B:0 – #FF0000

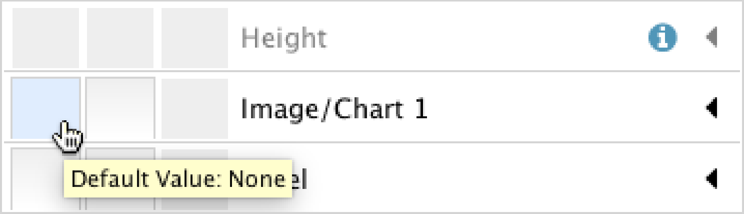
Tutorial 5: Using Images in Styles

This tutorial is a quick introduction to the node image feature. You can assign up to nine images per node as a part of a Style.

1. In this first example, we will use the presets that Cytoscape 3 has. In general, you can use any type of bitmap graphics. User images should be added to the **Image Manager** before using them in a Style.
2. If you are continuing from the previous tutorials, skip to the next step. Otherwise, load a network and run the Network Analyzer (**Tools** → **Network Analyzer** → **Network Analysis** → **Analyze Network...**). This creates several new table columns (statistics for nodes and edges).
3. Click the **Style** panel in the **Control Panel**, and select the **Solid** style.
4. If the property **Image/Chart 1** is not in the properties sheet yet, add it from the drop-down **Properties** → **Paint** → **Custom Paint 1** → **Image/Chart 1**.



5. Click the **Default Value** cell of the **Image/Chart 1** entry in order to open the **Graphics** dialog.



6. Select any of the images from the list and click **Apply**.

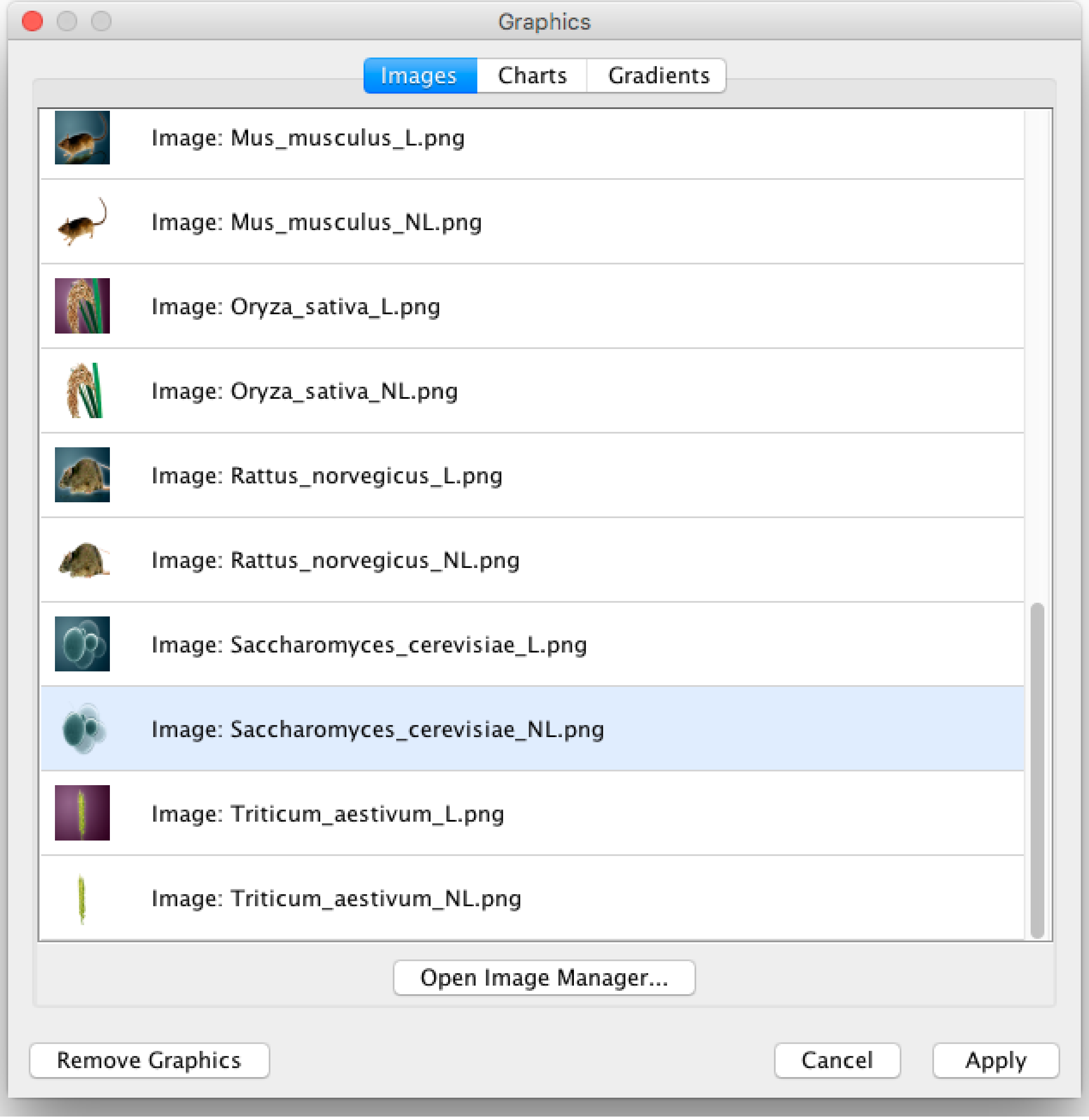
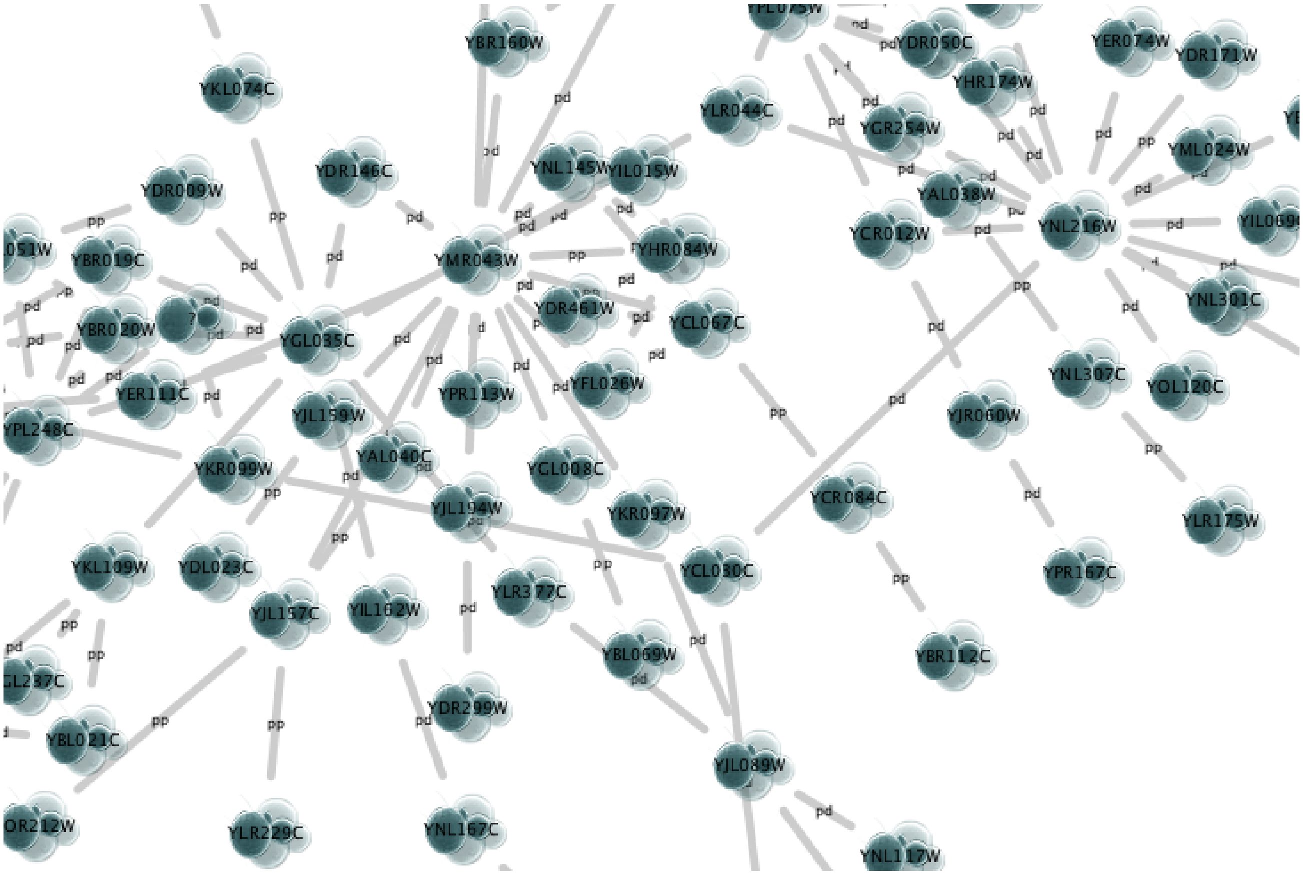
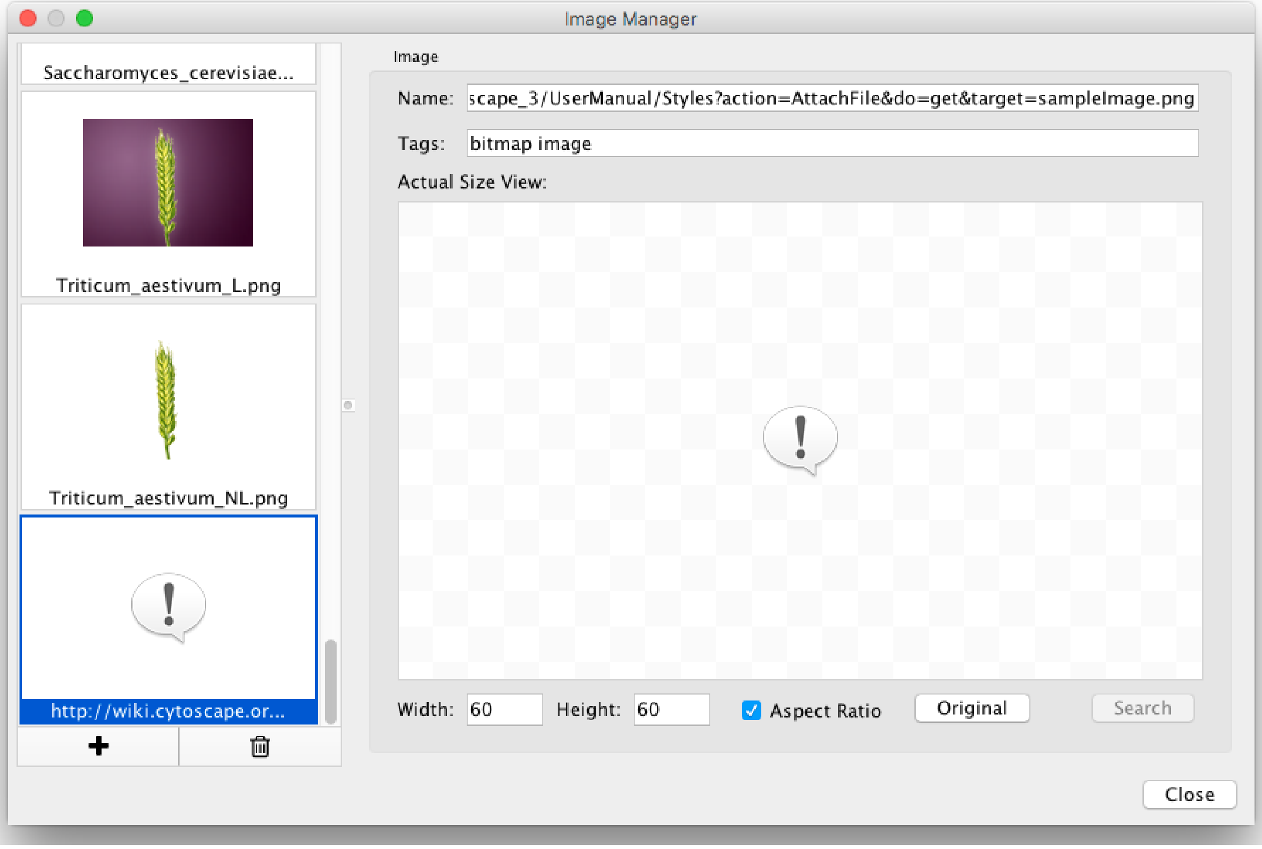


Image: Mus_musculus_L.png
Image: Mus_musculus_NL.png
Image: Oryza_sativa_L.png
Image: Oryza_sativa_NL.png
Image: Rattus_norvegicus_L.png
Image: Rattus_norvegicus_NL.png
Image: Saccharomyces_cerevisiae_L.png
Image: Saccharomyces_cerevisiae_NL.png
Image: Triticum_aestivum_L.png
Image: Triticum_aestivum_NL.png

7. Click the **Default Value** cell of node **Transparency** and set the value to *zero*.
8. Set the **Default Value** of node **Size** to *80*.
9. Set the **Default Value** of node **Label Font Size** to *10*, to increase readability.
10. Also change the edge **Width** to *6*. Now your network looks like the following:



11. Open the Image Manager under **View** → **Open Image Manager....** Drag and Drop this ! icon to the image list which automatically adds it to the manager.

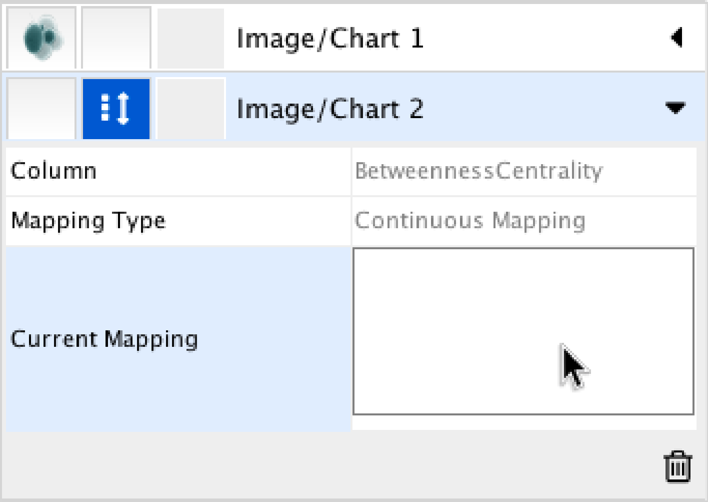


Name: scape_3/UserManual/Styles?action=AttachFile&do=get&target=sampleImage.png

Tags: bitmap image

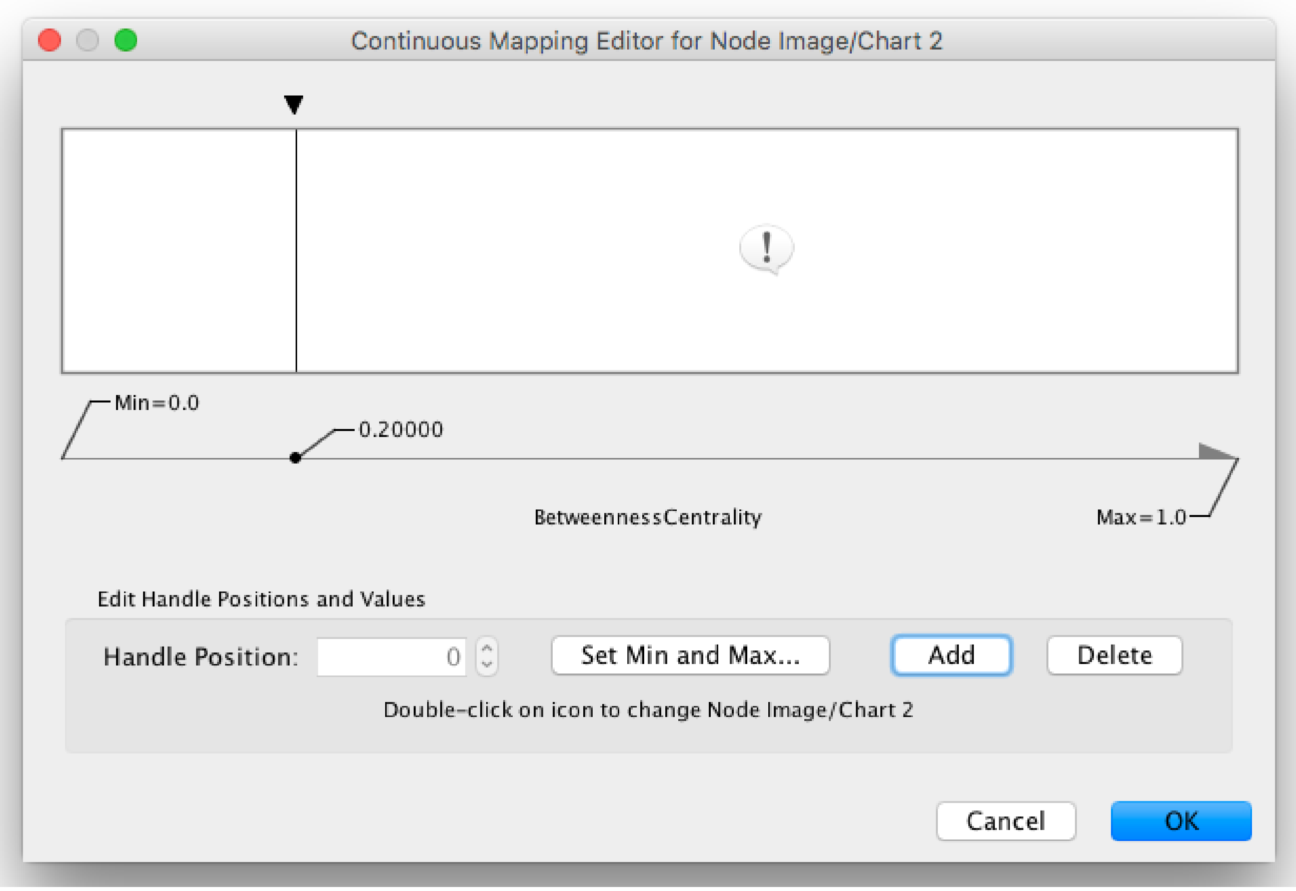
Width: 60 Height: 60 ☑ Aspect Ratio

12. Create a Continuous Mapping for **Image/Chart 2** and select "BetweennessCentrality" as its controlling property. Double-click the **Current Mapping** value cell to open the Continuos Mapping Editor.



Column	BetweennessCentrality
Mapping Type	Continuous Mapping
Current Mapping	

13. In the **Continuos Mapping Editor**, add a handle position by clicking in the **Add** button, and move the handle to *0.2*. Double-click the region over **0.2** and set the new icon you have just added in the last step.



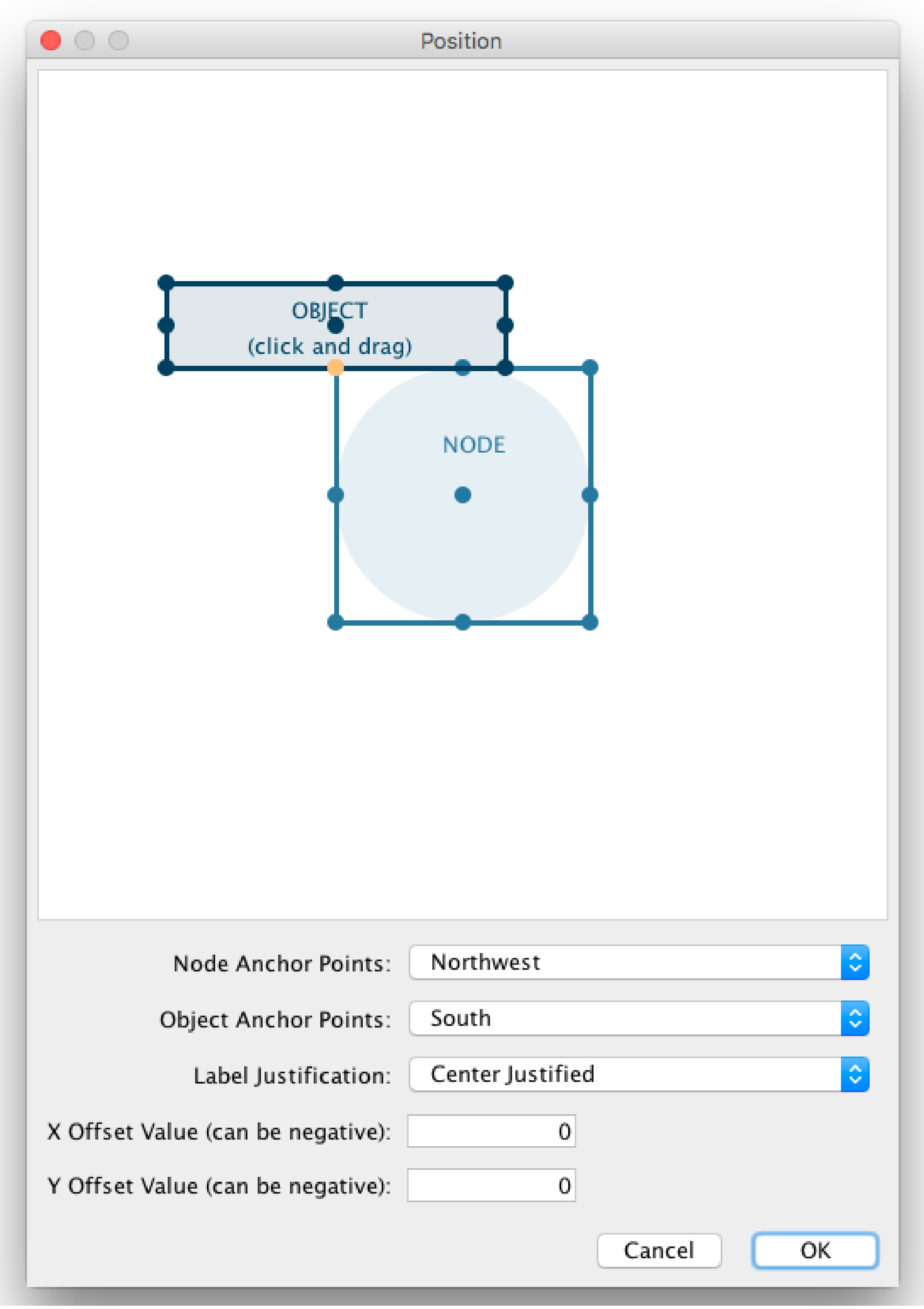
Continuous Mapping Editor for Node Image/Chart 2

Edit Handle Positions and Values

Handle Position: 0

Double-click on icon to change Node Image/Chart 2

14. Add the property **Image/Chart Position 2** from the drop-down option **Properties** → **Paint** → **Custom Paint 2** → **Image/Chart Position 2**. Click its **Default Value** cell to move the position of the graphics to upper left.



Node Anchor Points: Northwest

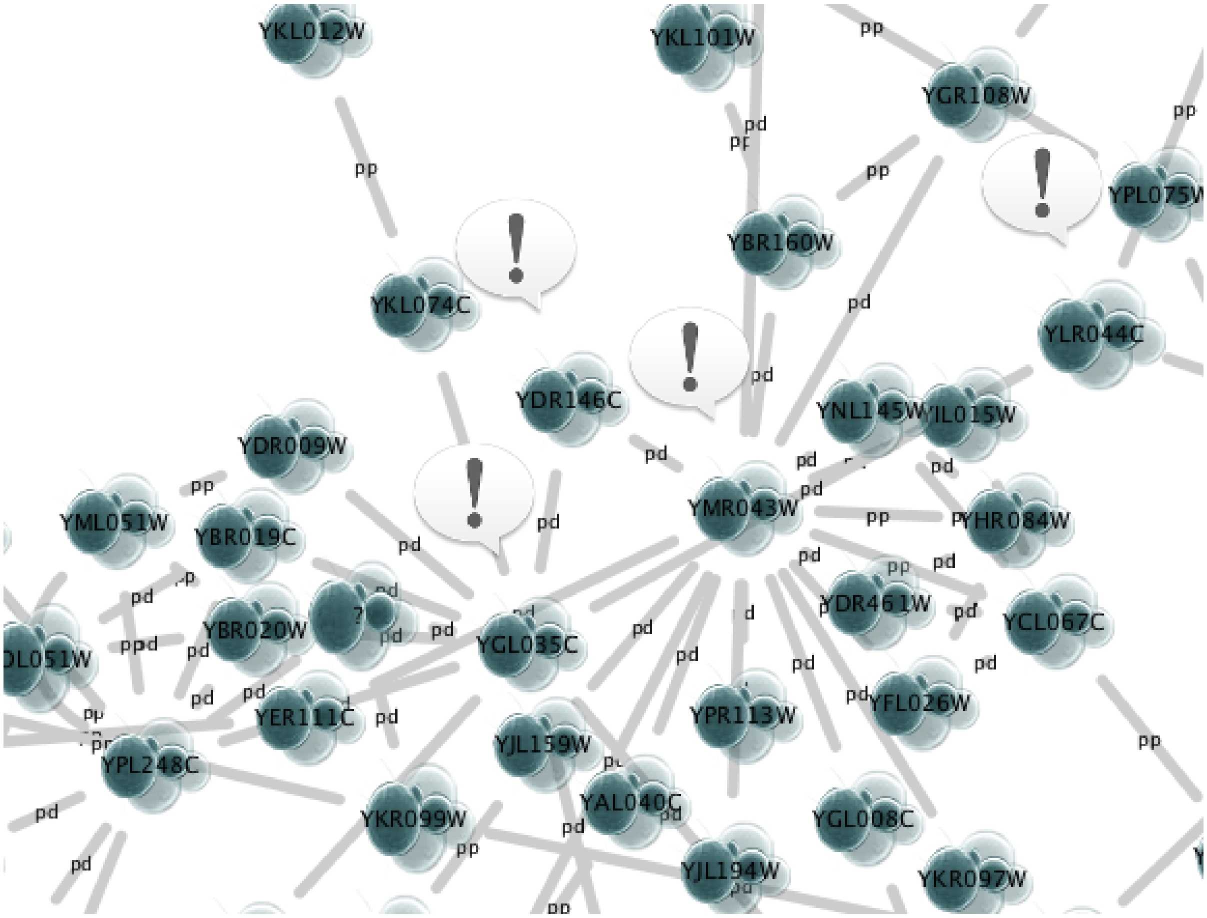
Object Anchor Points: South

Label Justification: Center Justified

X Offset Value (can be negative): 0

Y Offset Value (can be negative): 0

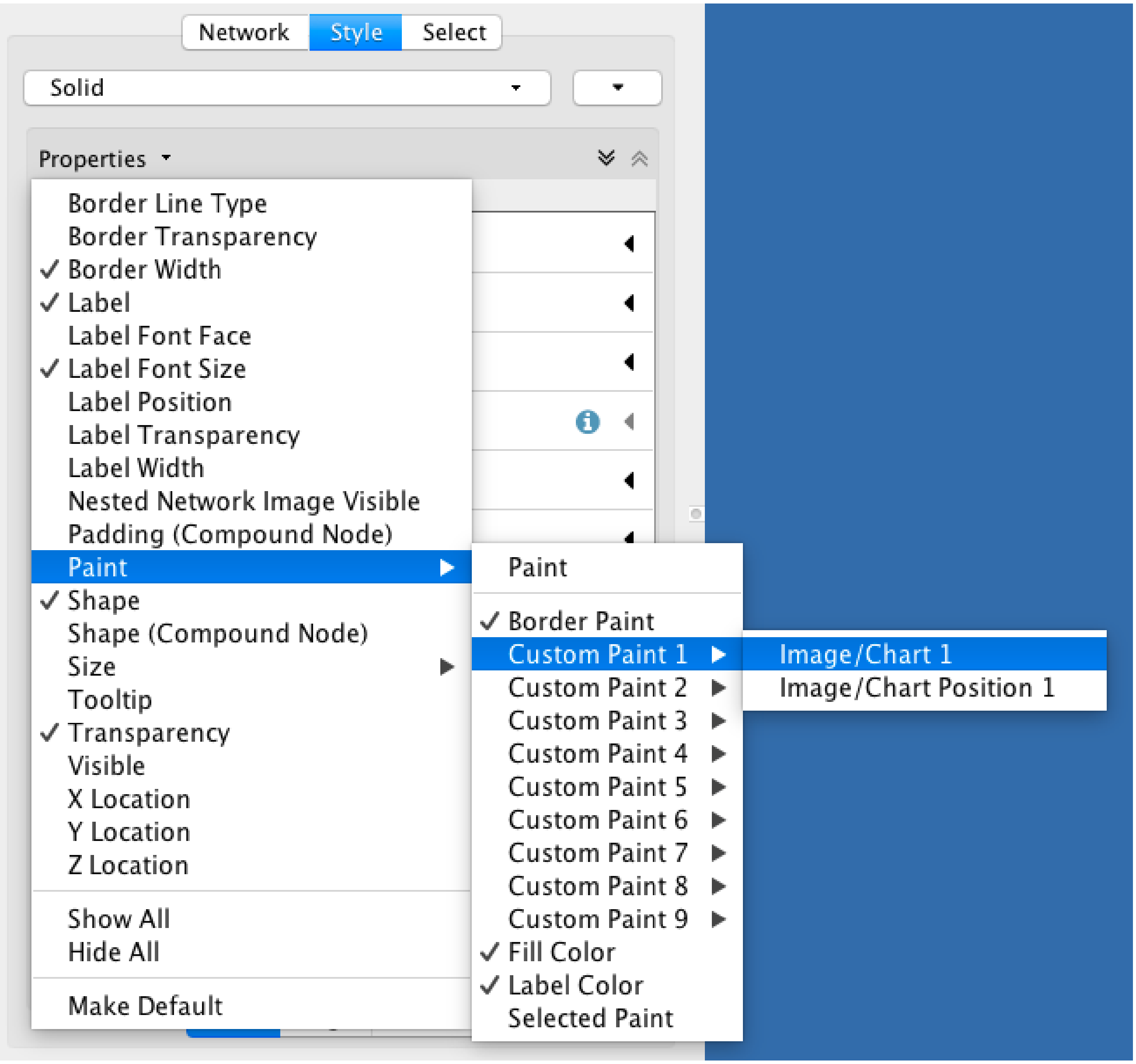
Now the important nodes in the network (nodes with high betweenness centrality) are annotated with the icon:



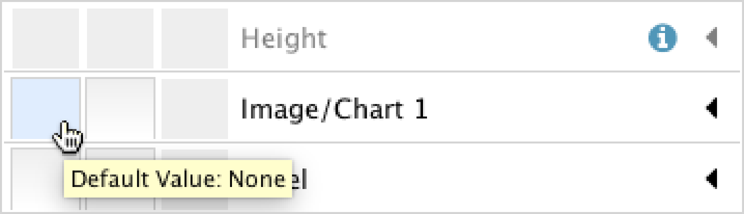
Tutorial 6: Creating Node Charts

The goal of this tutorial is to learn how to create and customize node charts from data stored in the Node tables.

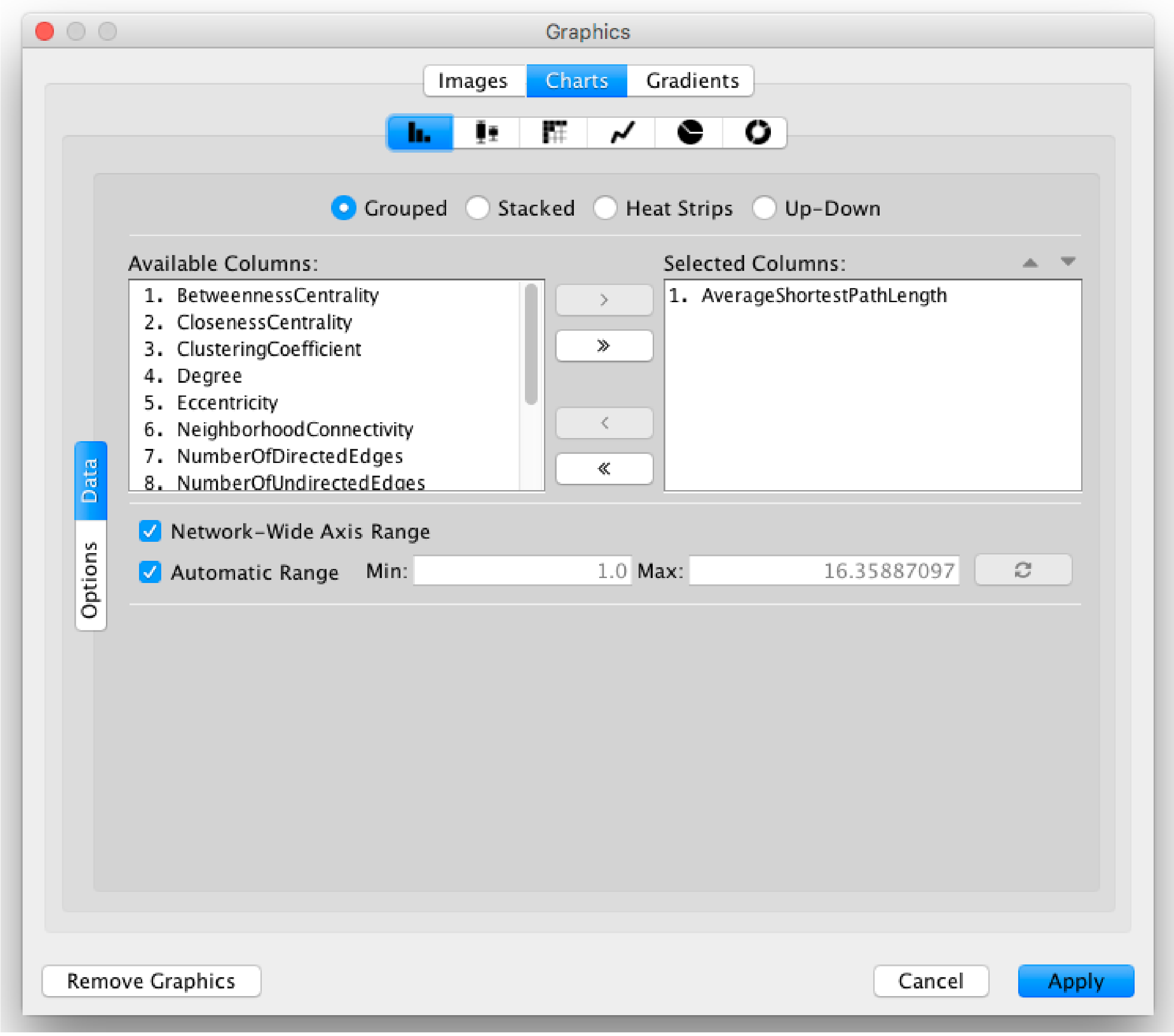
1. Start a new session and load a sample network. From the main menu, select **File** → **Import** → **Network** → **File...**, and select `sampleData/galFiltered.sif`.
2. Create some node/edge statistics using the **Network Analyzer**. Network Analyzer calculates some basic statistics for nodes and edges. From the main menu, select **Tools** → **Network Analyzer** → **Network Analysis** → **Analyze Network...**, and click **OK**. Once the result is displayed, simply close the window. All statistics are stored as regular table data.
3. Select the **Style** panel in the Control Panel.
4. **Create a new style:** Click the **Options** ▾ drop-down, and select **Create New Style**. Enter a name for your new style when prompted.
5. If the property **Image/Chart 1** is not in the properties sheet yet, add it from the drop-down **Properties** → **Paint** → **Custom Paint 1** → **Image/Chart 1**.



6. Click the **Default Value** cell of the **Image/Chart 1** entry in order to open the **Graphics** dialog.



7. Click the **Charts** tab and make sure the **Bar Chart** option is selected.



● Grouped ○ Stacked ○ Heat Strips ○ Up-Down

Available Columns:

1. BetweennessCentrality
2. ClosenessCentrality
3. ClusteringCoefficient
4. Degree
5. Eccentricity
6. NeighborhoodConnectivity
7. NumberOfDirectedEdges
8. NumberOfUndirectedEdges

Selected Columns:

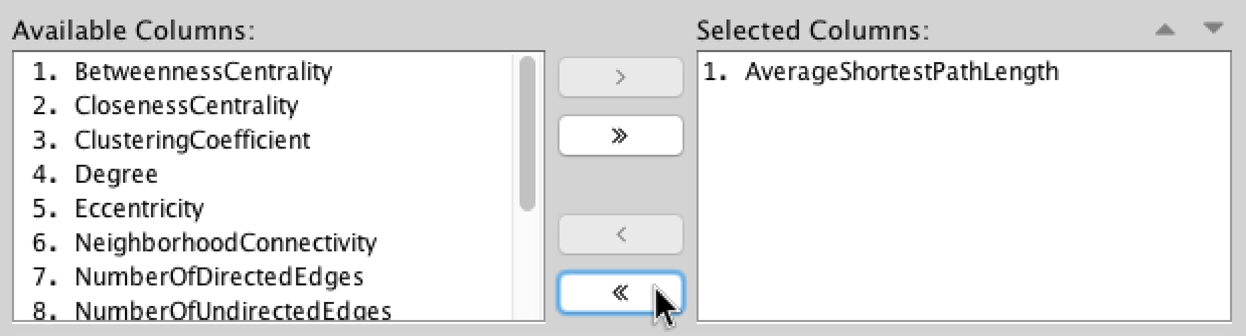
1. AverageShortestPathLength

☑ Network-Wide Axis Range

☑ Automatic Range Min: 1.0 Max: 16.35887097

8. **Select data columns:** Now you have to choose the columns in the Node table that contains the data you want to be displayed as charts. The **Available Columns** list displays all node columns that can be used as chart data (i.e. *single* or *list* columns of numerical types).

- First click the **Remove All** button to remove the current selected columns (by default, Cytoscape selects the first column in the **Available Columns** list).



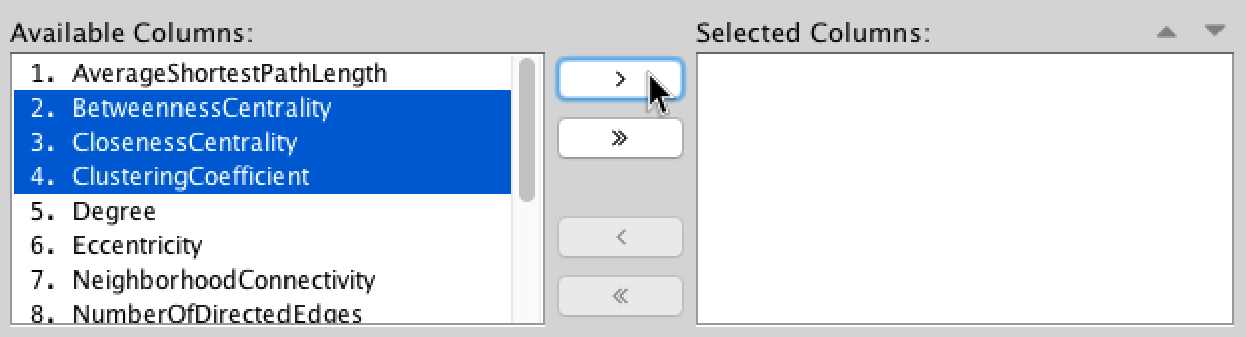
Available Columns:

1. BetweennessCentrality
2. ClosenessCentrality
3. ClusteringCoefficient
4. Degree
5. Eccentricity
6. NeighborhoodConnectivity
7. NumberOfDirectedEdges
8. NumberOfUndirectedEdges

Selected Columns:

1. AverageShortestPathLength

- Now select all *centrality* and *coefficient* columns from **Available Columns** list and click the **Add Selected** button.

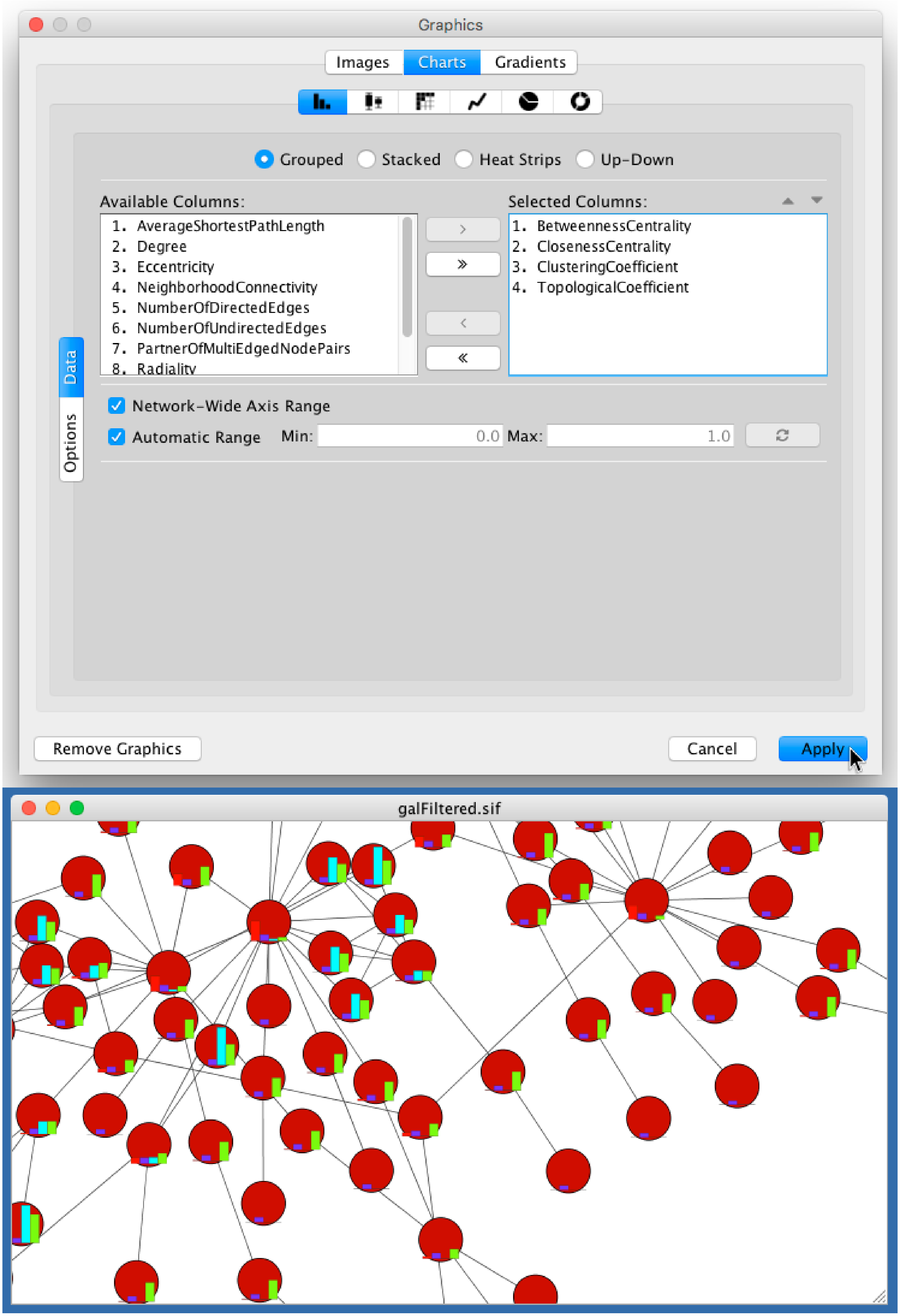


Available Columns:

1. AverageShortestPathLength
2. BetweennessCentrality
3. ClosenessCentrality
4. ClusteringCoefficient
5. Degree
6. Eccentricity
7. NeighborhoodConnectivity
8. NumberOfDirectedEdges

Selected Columns:

9. Click the **Apply** button to create bar charts with the selected data columns and default options.



● Grouped ○ Stacked ○ Heat Strips ○ Up-Down

Available Columns:

1. AverageShortestPathLength
2. Degree
3. Eccentricity
4. NeighborhoodConnectivity
5. NumberOfDirectedEdges
6. NumberOfUndirectedEdges
7. PartnerOfMultiEdgedNodePairs
8. Radiality

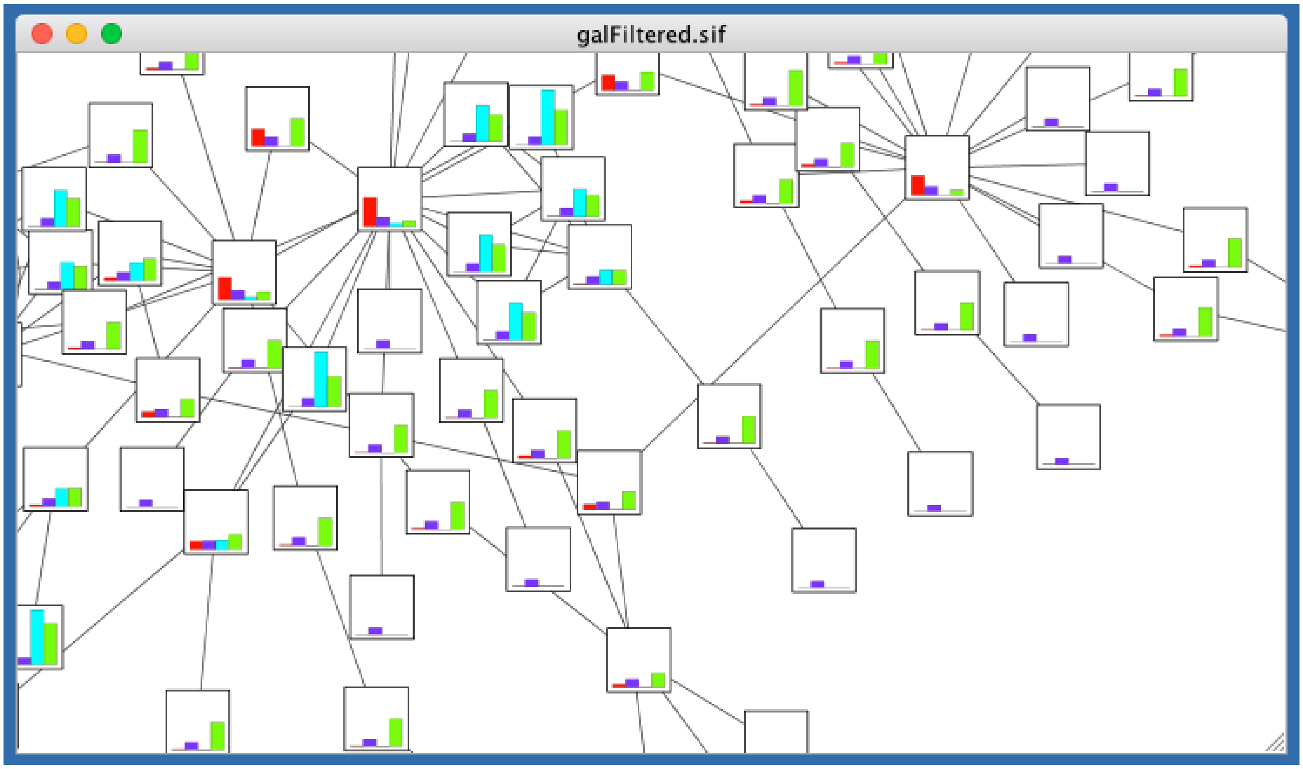
Selected Columns:

1. BetweennessCentrality
2. ClosenessCentrality
3. ClusteringCoefficient
4. TopologicalCoefficient

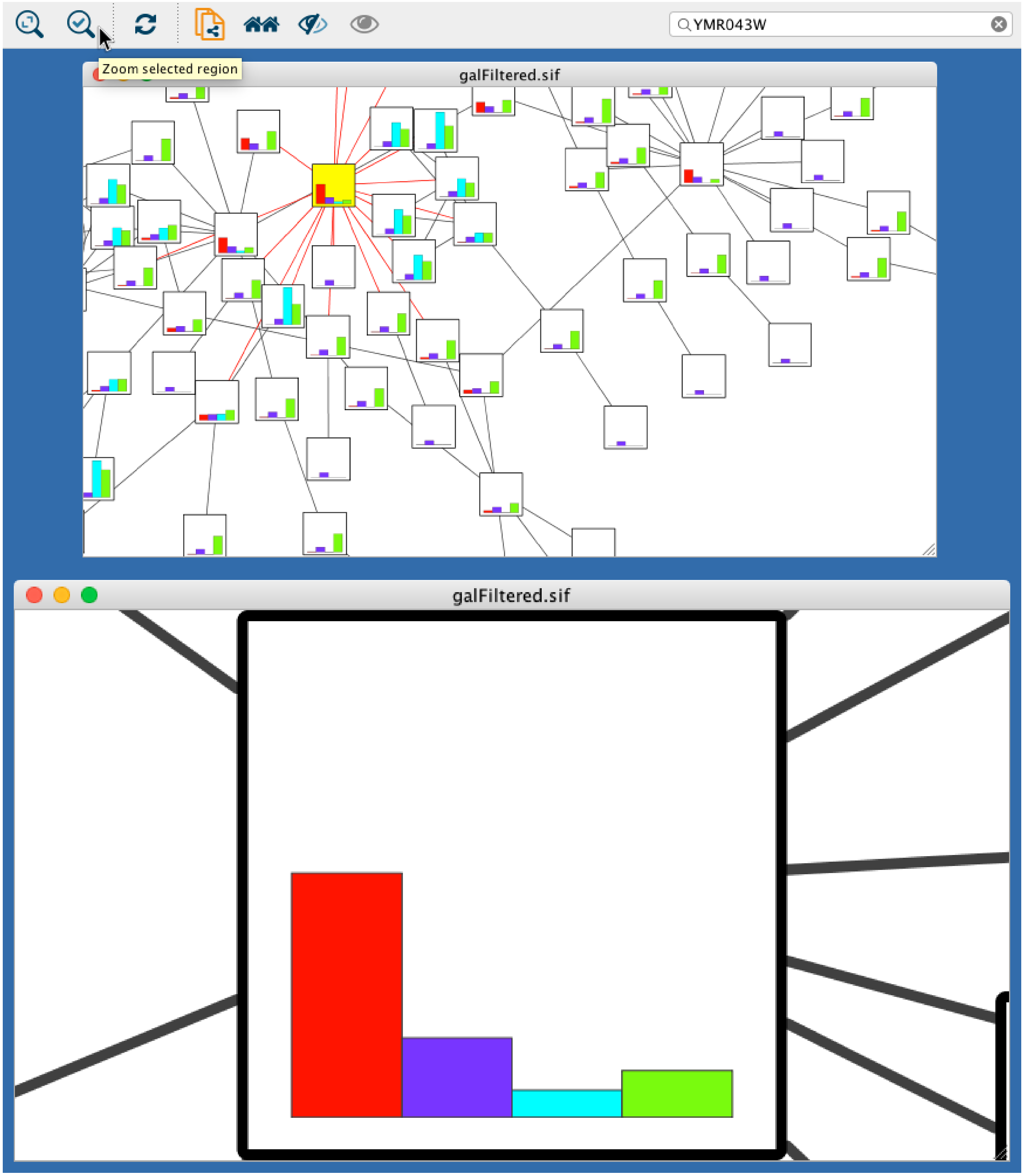
☑ Network-Wide Axis Range

☑ Automatic Range Min: 0.0 Max: 1.0

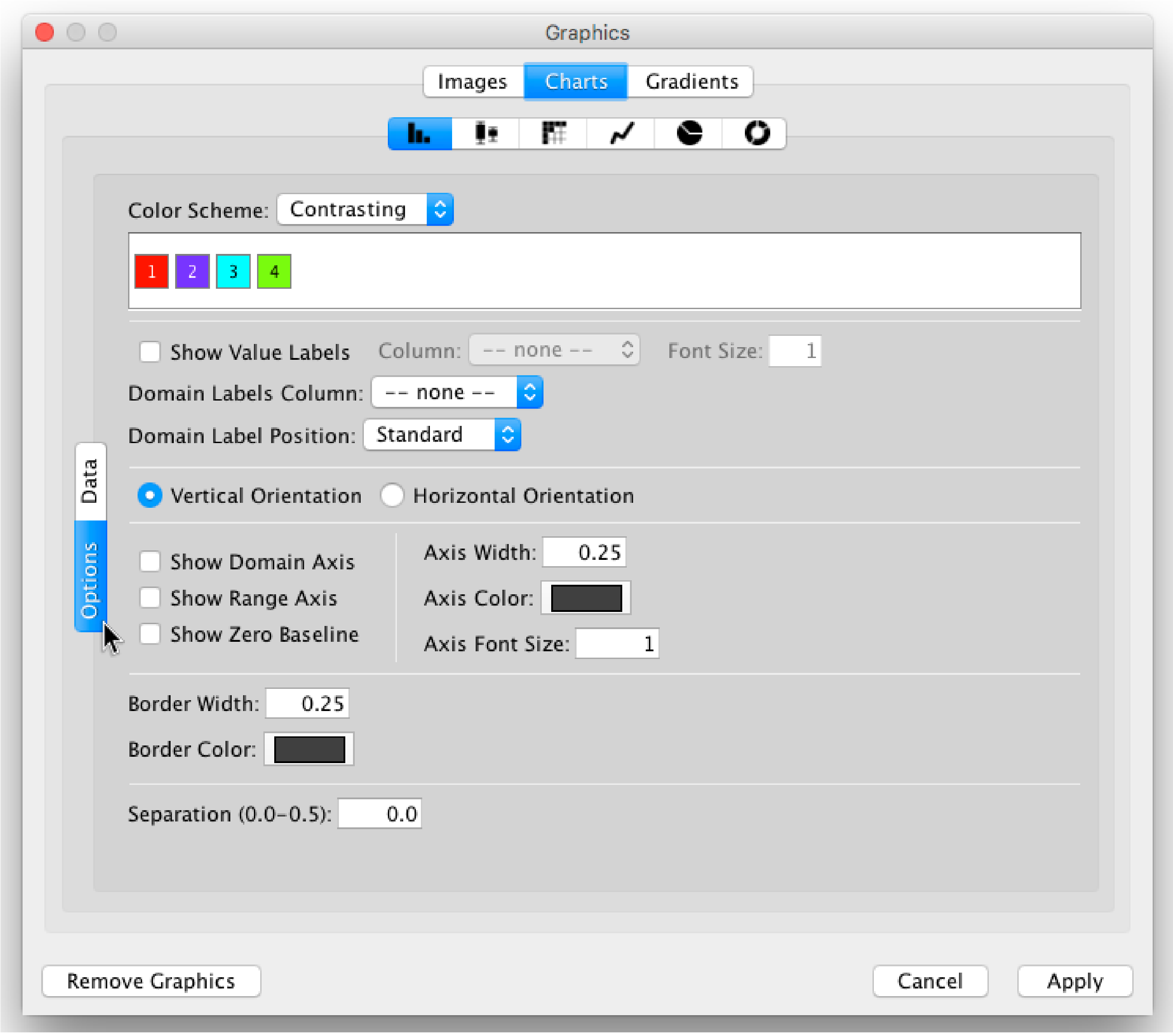
10. The network view doesn't look so good yet, so let's make a few changes to its Style before we continue. In the example shown below, the node **Shape** is set to *Rectangle*, while the node **Fill Color** is set to *white*.



11. Focus on one node to see the chart details. For example search for and then focus on node "YMR043W".



12. **Change other chart options:** Click the **Default Value** cell of the **Image/Chart 1** property again in order to open the **Graphics** dialog, and then select the **Options** tab on the **Bar Chart** editor.



Color Scheme: Contrasting

1 2 3 4

□ Show Value Labels Column: -- none -- Font Size: 1

Domain Labels Column: -- none --

Domain Label Position: Standard

● Vertical Orientation ○ Horizontal Orientation

□ Show Domain Axis

□ Show Range Axis

□ Show Zero Baseline

Axis Width: 0.25

Axis Color:

Axis Font Size: 1

Border Width: 0.25

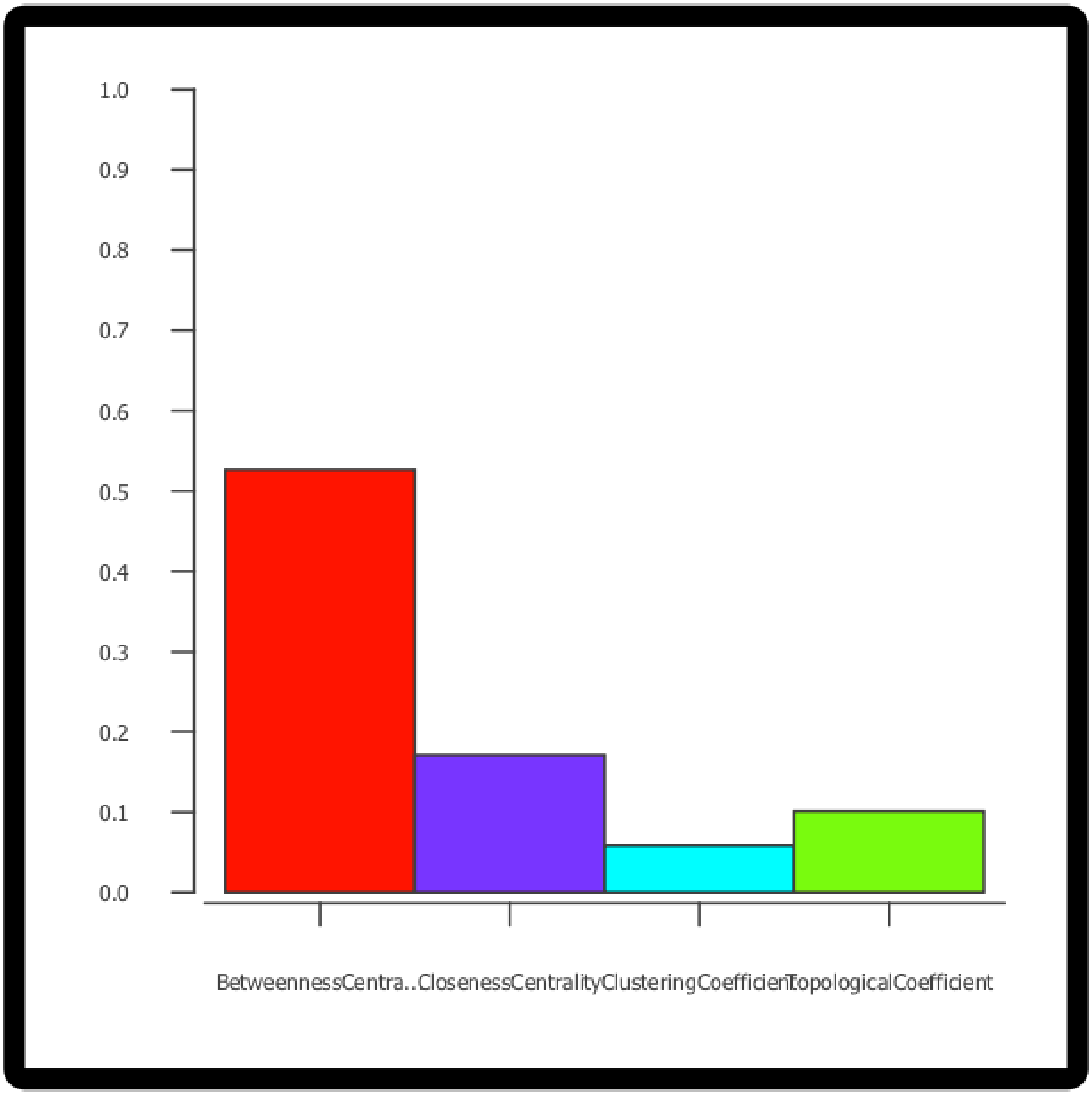
Border Color:

Separation (0.0–0.5): 0.0

On this panel, you can:

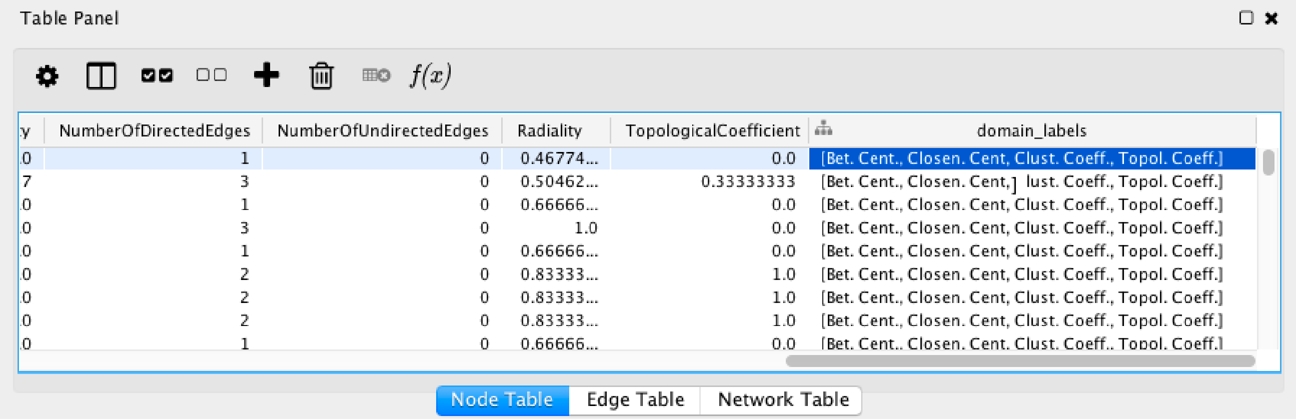
- Choose another **Color Scheme** or set all the colors individually (just click on each color).
- Show/Hide Value and Domain **Labels** and also set the **Domain Label Position**.
- Change the chart **Orientation**.
- Show/Hide Axes.
- Change the line width and color for axes and bars.
- Increase or reduce the separation between bars (up to 50% of the total chart width).
- **Note:** Other charts provide different options.

13. Check both **Show Domain Axis** and **Show Range Axis** and apply the graphics again. Now the node chart should look like this:



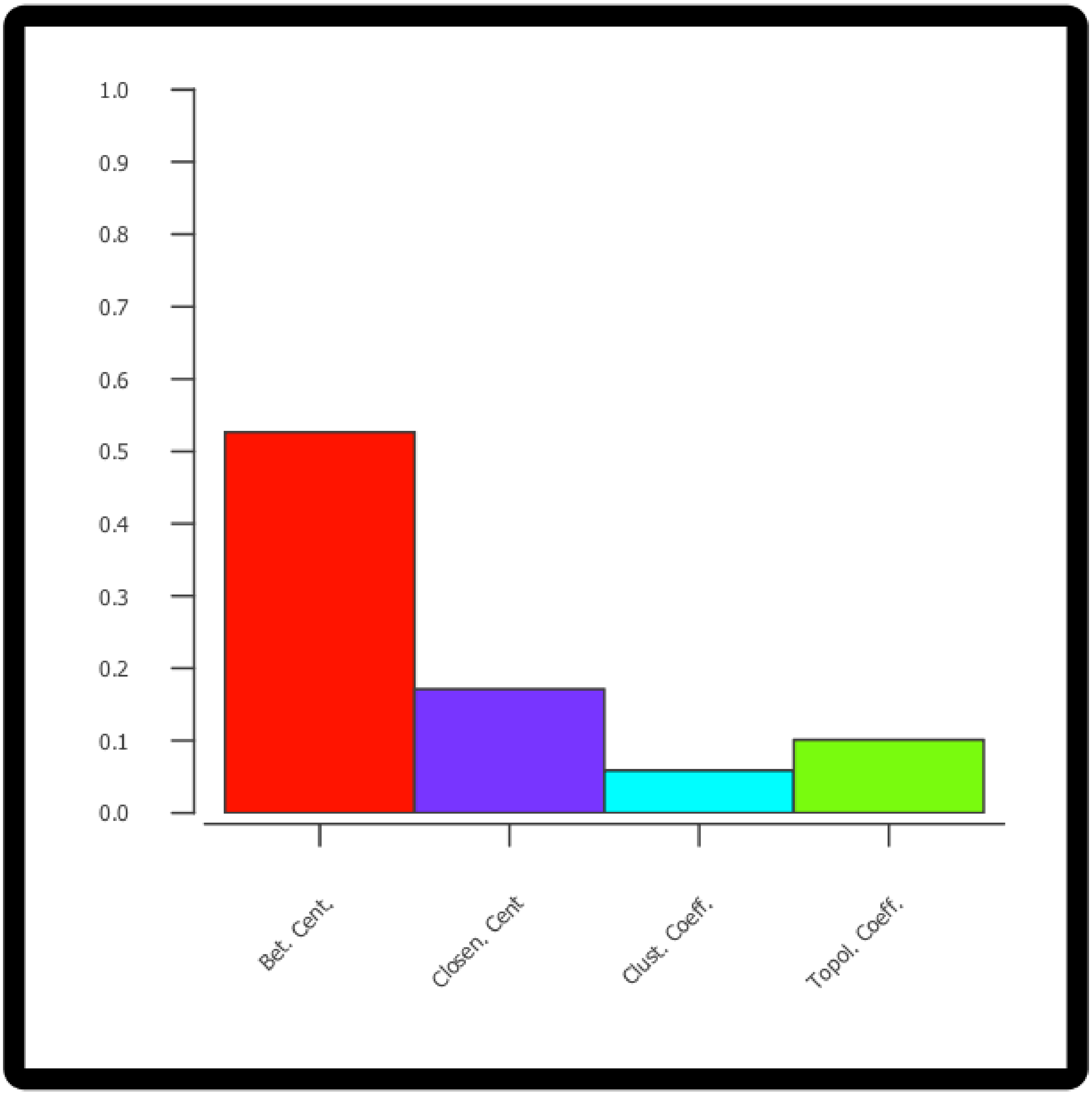
14. The default domain labels are not very useful, so let's set better labels:

- On the **Node Table** (Table Panel), create a new *List Column* of type *String* and name it "domain_labels".
- Edit any of the cells of the created column (double-click it) and type `["Bet. Cent.","Closen. Cent","Clust. Coeff.","Topol. Coeff."]`.
- Right-click the same cell and select the option **Apply to entire column**.



y	NumberOfDirectedEdges	NumberOfUndirectedEdges	Radiality	TopologicalCoefficient	domain_labels
.0	1	0	0.46774...	0.0	[Bet. Cent., Closen. Cent, Clust. Coeff., Topol. Coeff.]
7	3	0	0.50462...	0.33333333	[Bet. Cent., Closen. Cent,] lust. Coeff., Topol. Coeff.]
.0	1	0	0.66666...	0.0	[Bet. Cent., Closen. Cent, Clust. Coeff., Topol. Coeff.]
.0	3	0	1.0	0.0	[Bet. Cent., Closen. Cent, Clust. Coeff., Topol. Coeff.]
.0	1	0	0.66666...	0.0	[Bet. Cent., Closen. Cent, Clust. Coeff., Topol. Coeff.]
.0	2	0	0.83333...	1.0	[Bet. Cent., Closen. Cent, Clust. Coeff., Topol. Coeff.]
.0	2	0	0.83333...	1.0	[Bet. Cent., Closen. Cent, Clust. Coeff., Topol. Coeff.]
.0	2	0	0.83333...	1.0	[Bet. Cent., Closen. Cent, Clust. Coeff., Topol. Coeff.]
.0	1	0	0.66666...	0.0	[Bet. Cent., Closen. Cent, Clust. Coeff., Topol. Coeff.]

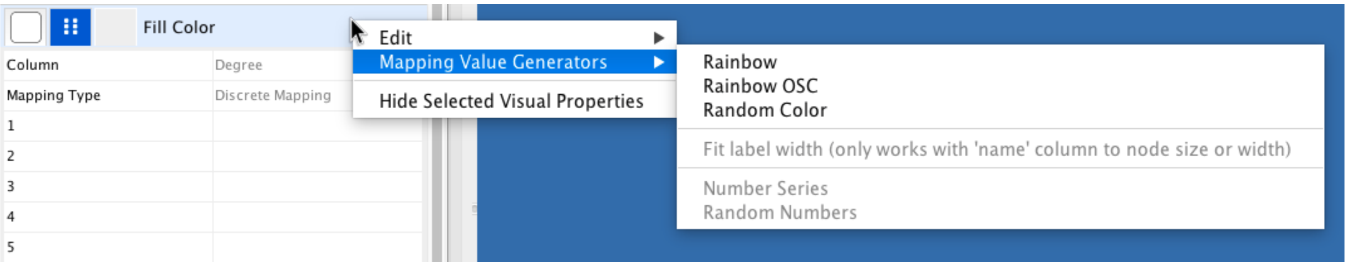
- Open the chart editor again and select the **Options** panel.
- Select “domain_labels” on the **Domain Labels Column** drop-down button.
- Select “Up 45^o^” on the **Domain Labels Position** drop-down button. The labels should look like this now:



Advanced Topics

Discrete Mappings

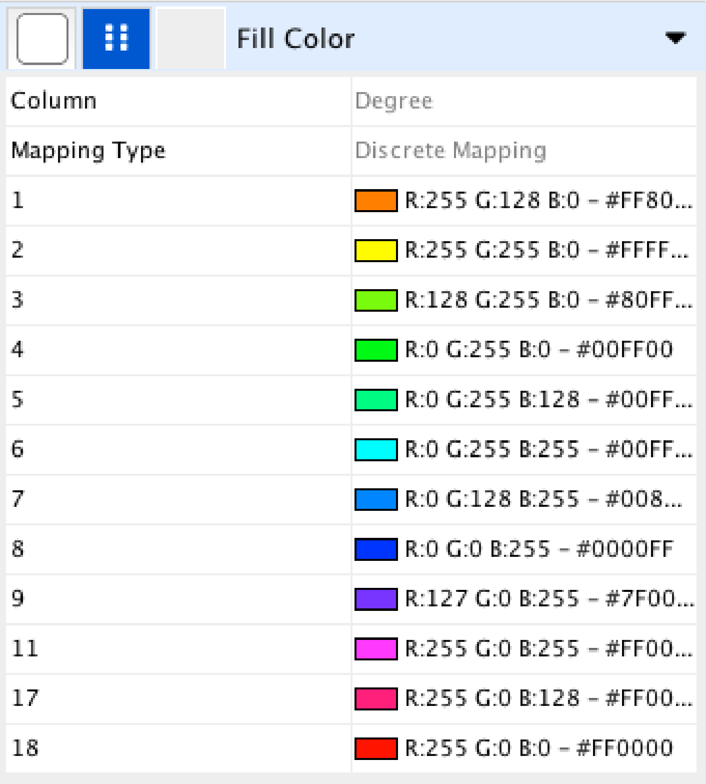
Several utility functions are available for Discrete Mappings. You can use these functions by right-clicking on any property entry (shown below).


Fill Color
Edit
Mapping Value Generators
Hide Selected Visual Properties
Rainbow
Rainbow OSC
Random Color
Fit label width (only works with 'name' column to node size or width)
Number Series
Random Numbers

Column	Degree
Mapping Type	Discrete Mapping
1	
2	
3	
4	
5	

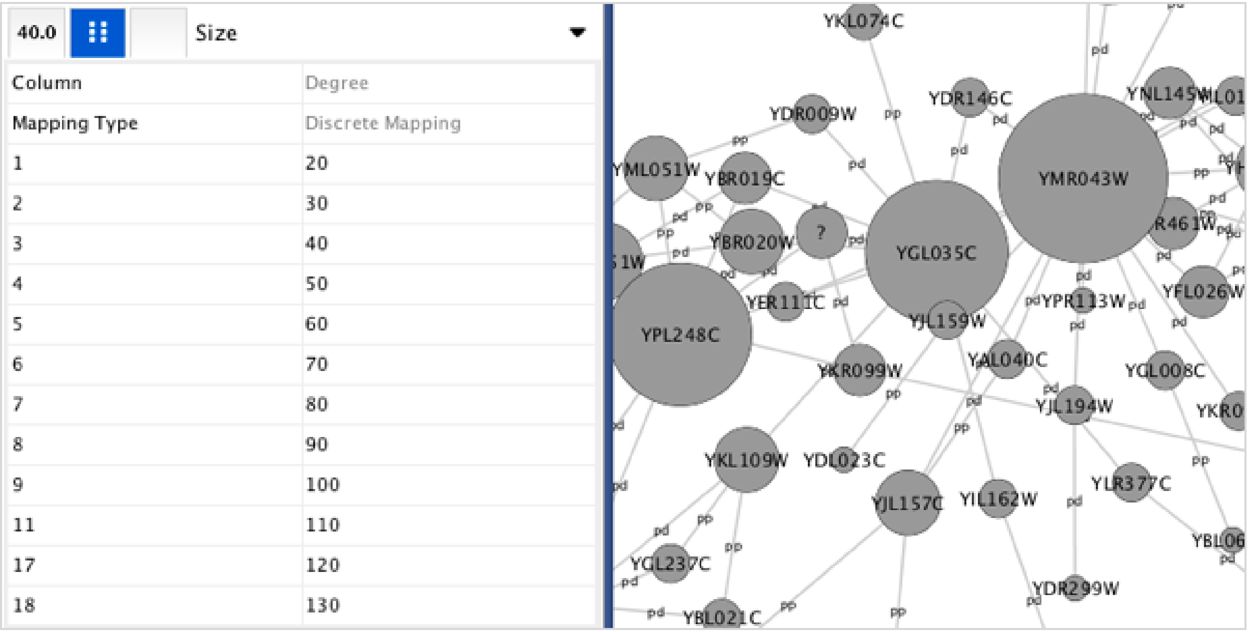
Automatic Value Generators

- **Mapping Value Generators -** Functions in this menu category are value generators for discrete mappings. Users can set values for discrete mappings automatically by selecting these functions.
  - **Rainbow** and **Rainbow OSC -** These functions try to assign as diverse a set of colors as possible for each data value.



Column	Degree
Mapping Type	Discrete Mapping
1	🟧 R:255 G:128 B:0 – #FF80...
2	🟨 R:255 G:255 B:0 – #FFFF...
3	🟩 R:128 G:255 B:0 – #80FF...
4	🟩 R:0 G:255 B:0 – #00FF00
5	🟩 R:0 G:255 B:128 – #00FF...
6	🟦 R:0 G:255 B:255 – #00FF...
7	🟦 R:0 G:128 B:255 – #008...
8	🟦 R:0 G:0 B:255 – #0000FF
9	🟪 R:127 G:0 B:255 – #7F00...
11	🟪 R:255 G:0 B:255 – #FF00...
17	🟥 R:255 G:0 B:128 – #FF00...
18	🟥 R:255 G:0 B:0 – #FF0000

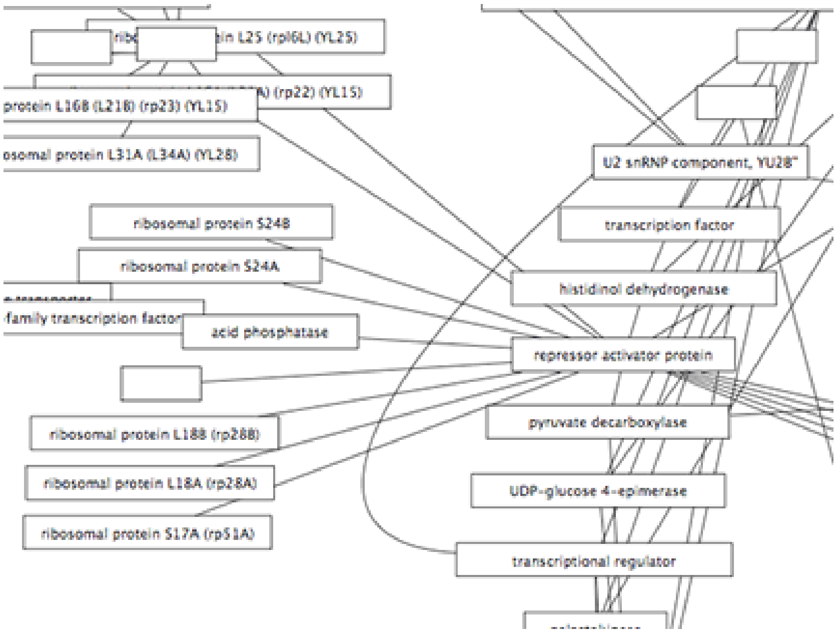
  - **Random Numbers** and **Random Colors -** Randomized numbers and colors.
  - **Number Series -** Set a series of numbers to the specified mapping. Requires a starting number and increment.



40.0 Size

Column	Degree
Mapping Type	Discrete Mapping
1	20
2	30
3	40
4	50
5	60
6	70
7	80
8	90
9	100
11	110
17	120
18	130

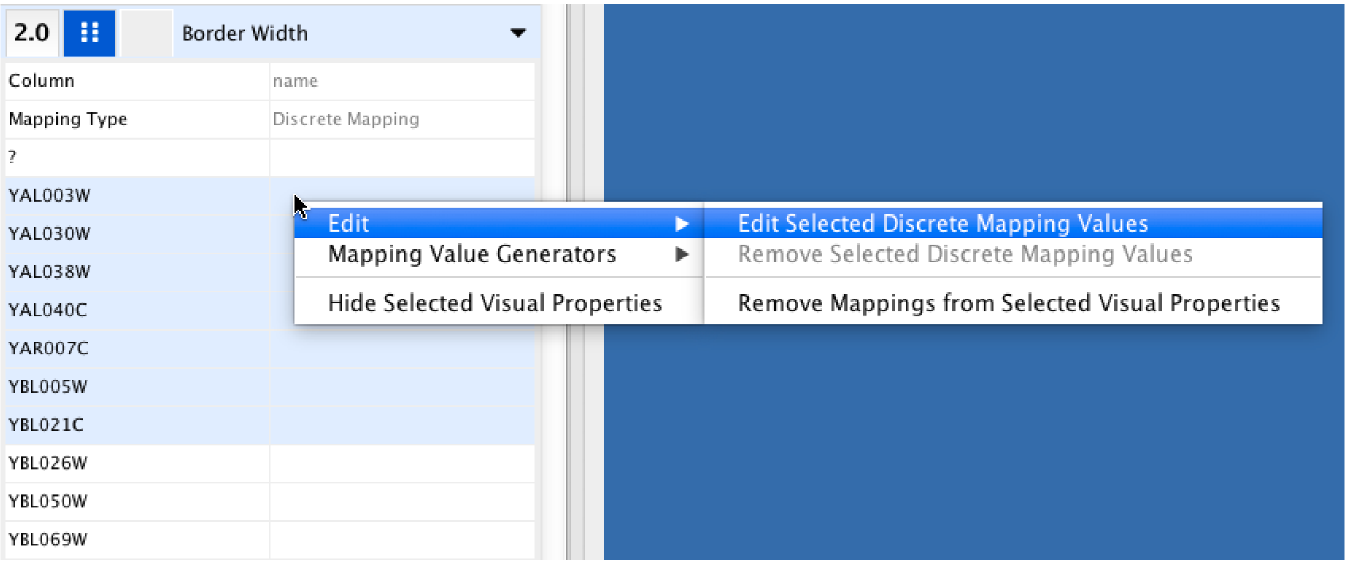
- **Fit label width** - This function is only for node **Width** and **Size**. When a discrete mapping for node **Width** or **Size** is available, you can fit the size of each node to its label automatically by selecting this function. See the example below:



ribosomal protein L25 (rpl6L) (YL25)
(rp22) (YL15)
protein L16B (L21B) (rp23) (YL15)
osomal protein L31A (L34A) (YL28)
ribosomal protein S24B
ribosomal protein S24A
family transcription factor
acid phosphatase
ribosomal protein L18B (rp28B)
ribosomal protein L18A (rp28A)
ribosomal protein S17A (rp51A)
U2 snRNP component, YU28"
transcription factor
histidinol dehydrogenase
repressor activator protein
pyruvate decarboxylase
UDP-glucose 4-epimerase
transcriptional regulator

Edit Selected Values at Once

You can set multiple values at once. First, you need to select discrete mapping rows in which you want to change values then right-click and select **Edit** → **Edit Selected Discrete Mapping Values**. A dialog pops up and you can enter the new value for the selected rows.



2.0 Border Width

Column	name
Mapping Type	Discrete Mapping
?	
YAL003W	
YAL030W	
YAL038W	
YAL040C	
YAR007C	
YBL005W	
YBL021C	
YBL026W	
YBL050W	
YBL069W	

Edit
Mapping Value Generators
Hide Selected Visual Properties
Edit Selected Discrete Mapping Values
Remove Selected Discrete Mapping Values
Remove Mappings from Selected Visual Properties

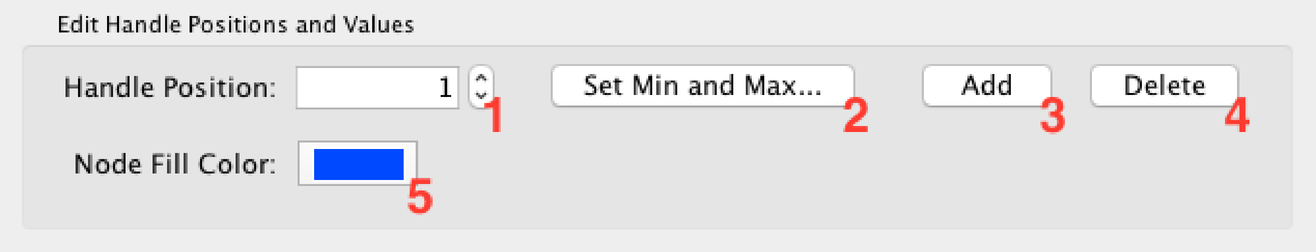
Working with Continuous Mapping Editors

There are three kinds of **Continuous Mapping Editors**. Each of them are associated with a specific property type:

Editor Type

EDITOR TYPE	SUPPORTED DATA TYPE	PROPERTIES
Color Gradient Editor	Color	node/edge/border/label colors
Continuous-Continuous Editor	Numbers	size/width/transparency
Continuous-Discrete Editor	All others	font/shape/text/graphics/position/etc.

Range Settings Panel



Edit Handle Positions and Values

Handle Position: 1

Node Fill Color: 🟦

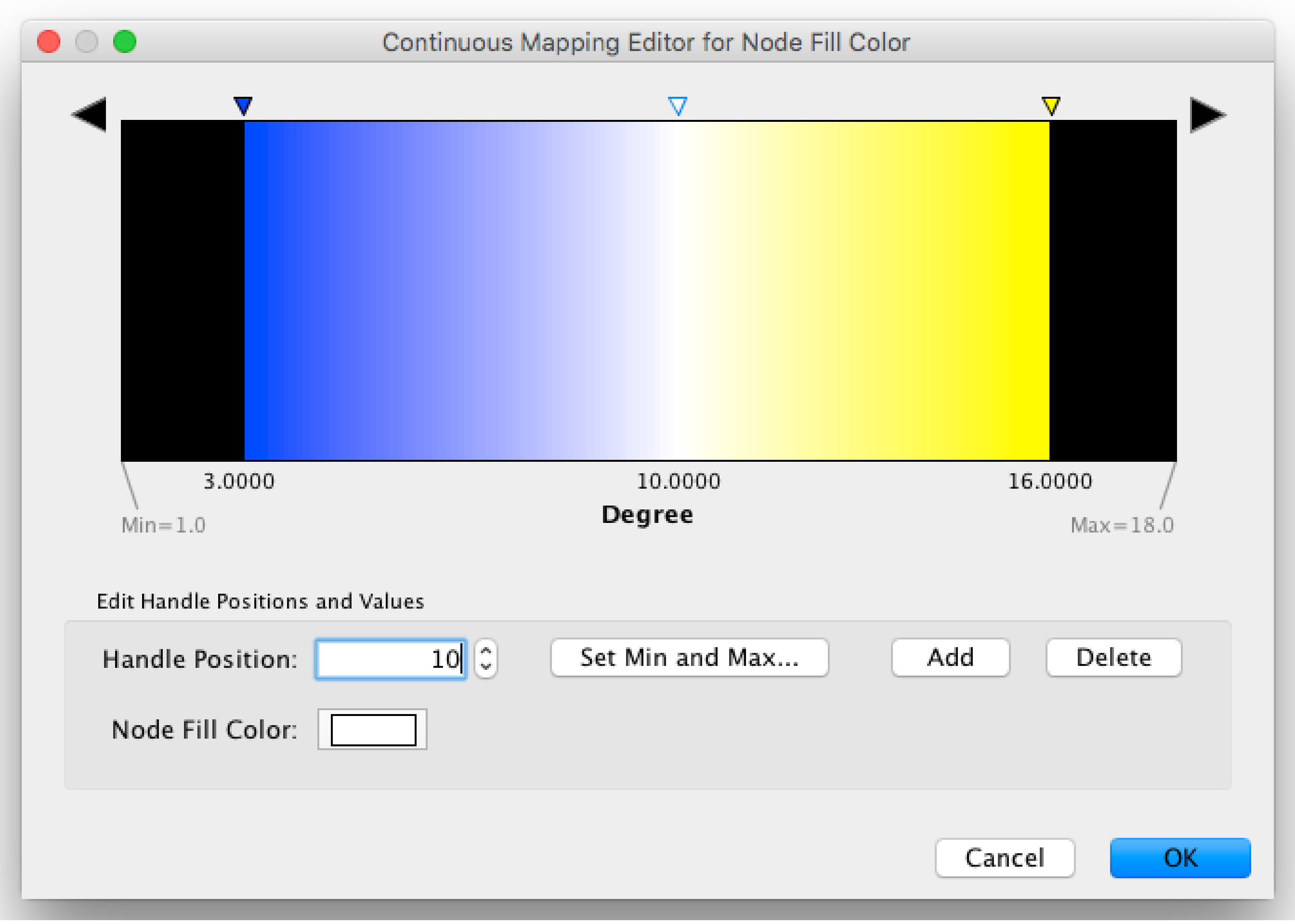
Each continuous mapping editor has a range settings section (labelled **Edit Handle Positions and Values**) with the following fields and buttons.

1. **Handle Position -** This box displays the current value for the selected slider handle. You can also directly type the value in this box to move the slider to an exact location.
2. **Set Min and Max... -** Click this button to set the overall range of this editor. The first time you open the editor, the *Min* and *Max* values are set by the range of the data column you

selected (i.e. the minimum and maximum values of the mapped column). Once you change this, you can restore the default numbers by clicking *Reset* button.

3. **Add** - Adds a new handle to the editor.
4. **Delete** - Deletes the selected handle from the slider widget.
5. *Handle Value* (e.g. **Node Fill Color**) - Click this button to edit the value (e.g. a color) assigned to the selected handle.

Gradient Editor



Continuous Mapping Editor for Node Fill Color

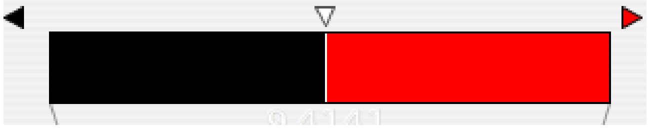
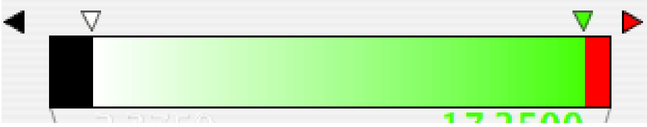
Edit Handle Positions and Values

Handle Position: 10

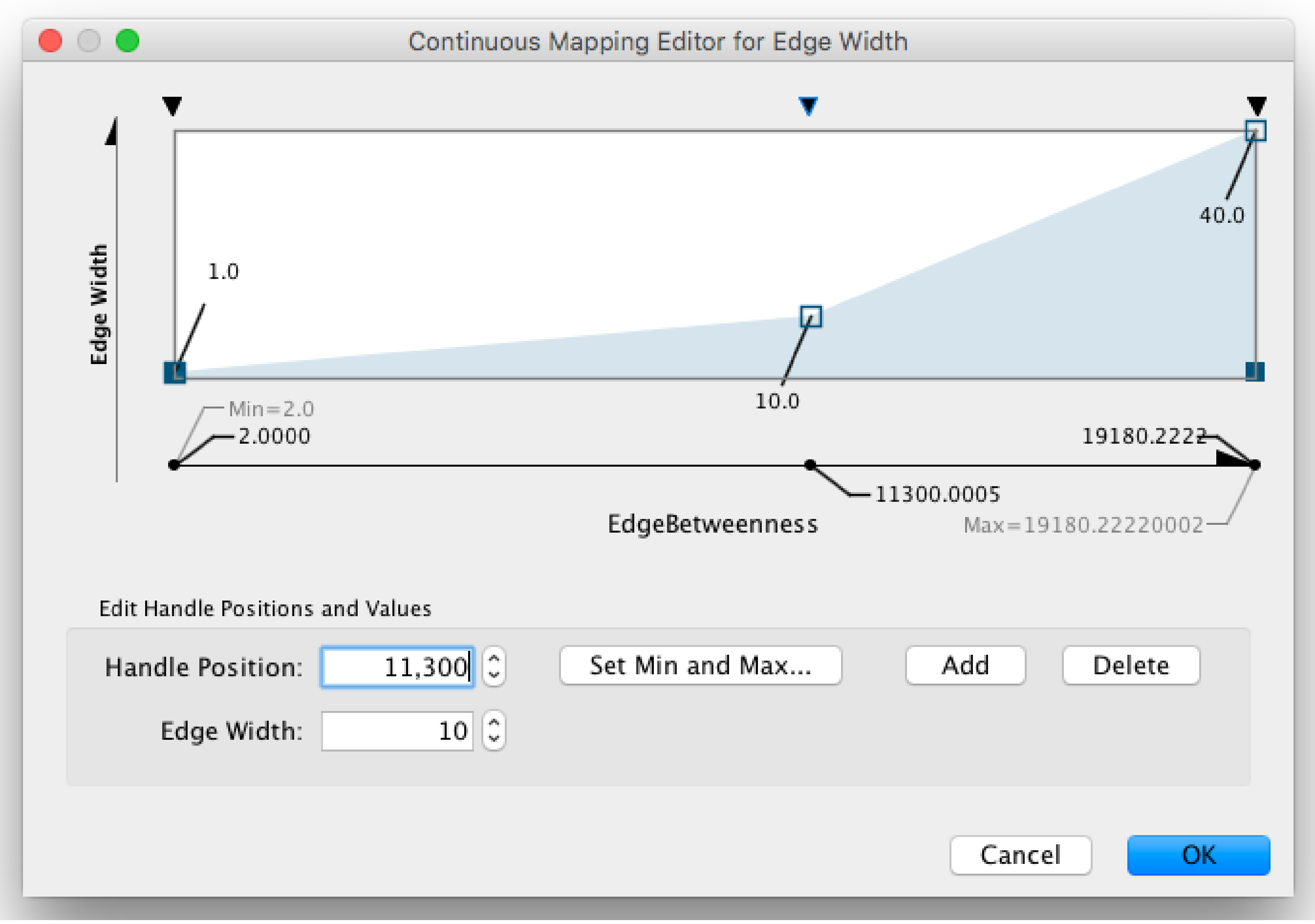
Node Fill Color:

The Gradient Editor is an editor for creating continuous mappings for colors. To change the color of each region, just double-click the handles (small triangles on the top). A Color gradient will be created only when the editor has two or more handles (see the example below).

Gradient Editor

1 HANDLE (NO GRADIENT)	2 HANDLES
	

Continuous-Continuous Editor



Continuous Mapping Editor for Edge Width

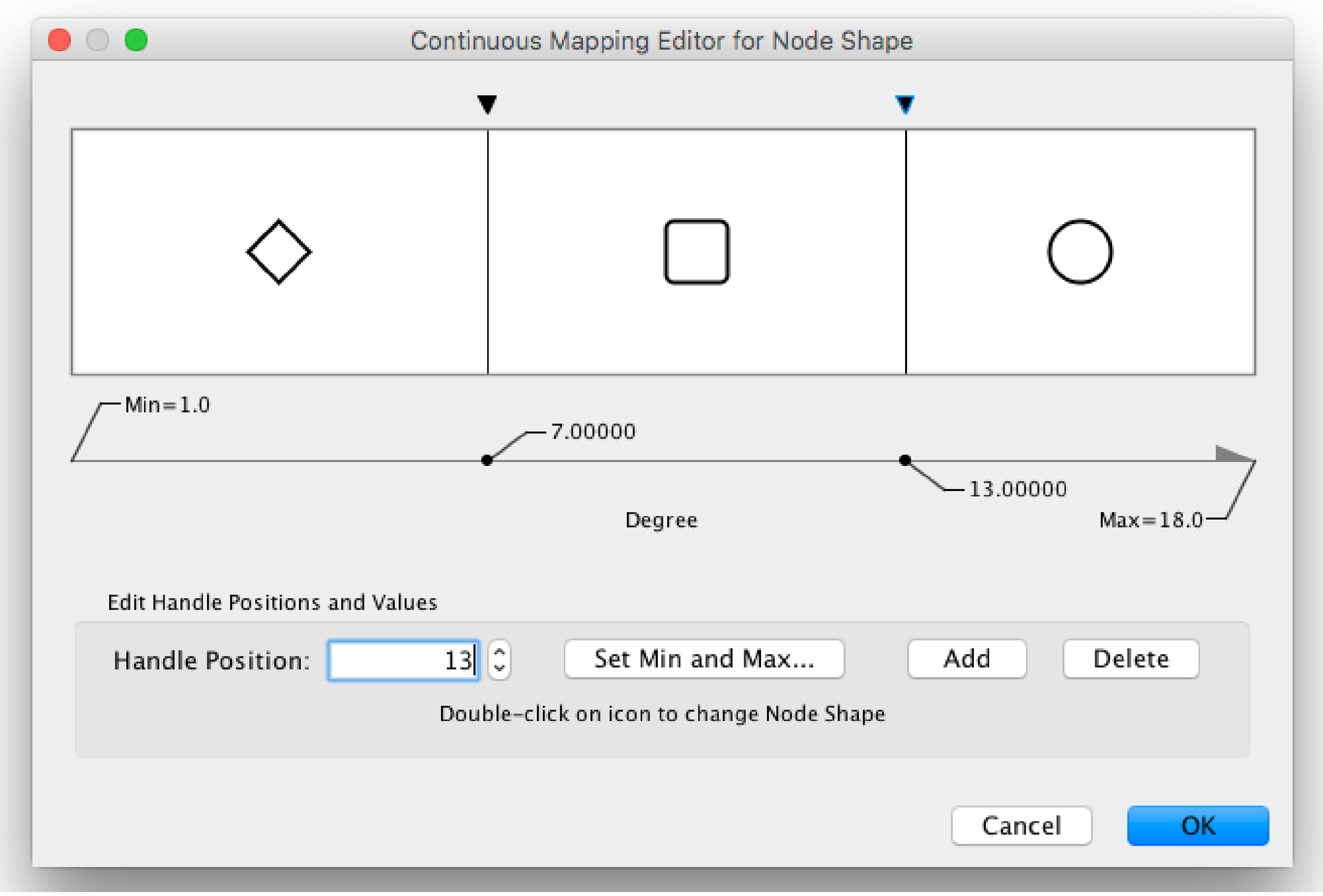
Edit Handle Positions and Values

Handle Position: 11,300

Edge Width: 10

The Continuous-Continuous Editor is for creating mappings between numerical data and numerical properties (e.g. size, transparency). To change the value assigned on the Y-axis (the property shown in the example above is edge **Width**), drag the small squares or double-click them to directly type an exact value.

Continuous-Discrete Editor


Continuous Mapping Editor for Node Shape
Min=1.0
7.00000
13.00000
Max=18.0
Degree
Edit Handle Positions and Values
Set Min and Max...
Add
Delete
Double-click on icon to change Node Shape
Cancel
OK

Handle Position: 13

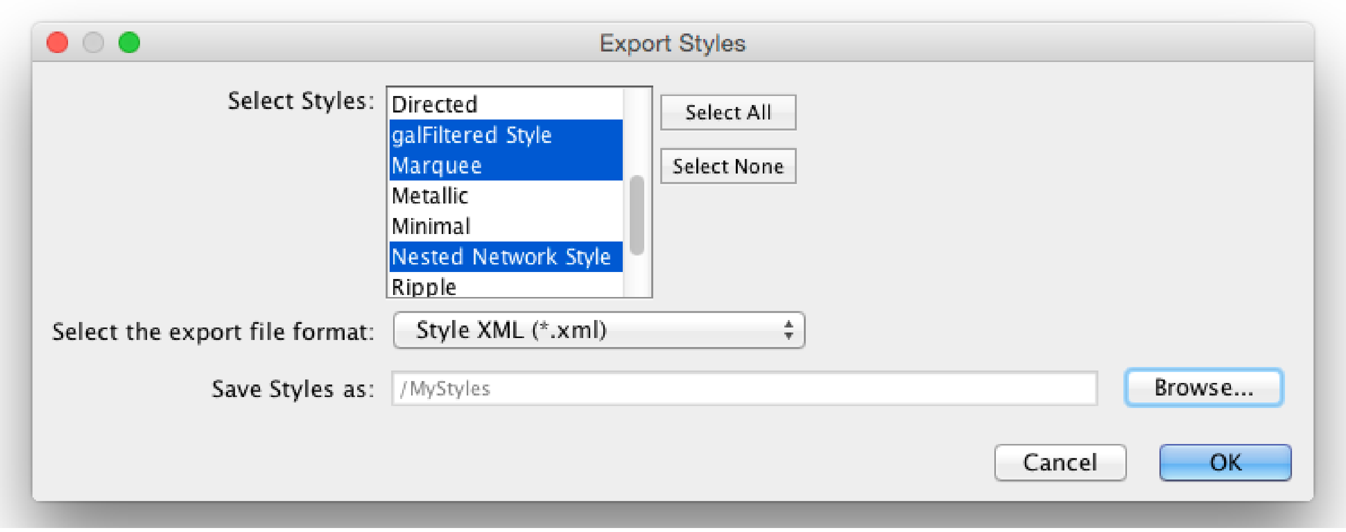
The Continuous-Discrete Editor is used to create mappings from numerical column values to discrete properties, such as fonts, shapes, or line styles. To edit a value for a specific region, double-click the icon on the track.

Managing Styles

All Cytoscape Style settings are initially loaded from a default file that cannot be altered by users. When users make changes to the properties, a `session_syle.xml` file is saved in the session file. This means that if you save your session, you will not lose your properties. No other style files are saved during normal operation.

Saving Styles

Styles are automatically saved with the session they were created in. Before Cytoscape exits, you will be prompted to make sure you save the session before quitting. It is also possible to save your styles in a file separate from the session file. To do this, navigate to the menu option **File** → **Export** → **Styles...**, and save the selected styles to a file. This feature can be used to share styles with other users.



Select the export file format: Style XML (*.xml)

Save Styles as: /MyStyles

You can also change the default list of styles for all future sessions of Cytoscape. To do this, click the **Options** ▾ drop-down in the **Style** section, and select **Make Current Styles Default**. This will save the current styles as a `default_vizmap.xml` file to your `CytoscapeConfiguration` directory (found in your home directory). These styles will then be loaded each time Cytoscape is started.

Style File Formats

The Cytoscape-native Style format is *Style XML*. If you want to share Style files with other Cytoscape users, you need to export them to this format.

From version 3.1.0 on, Cytoscape can also export Cytoscape.js compatible JSON file. Since Cytoscape.js is an independent JavaScript library, and there are some differences between Cytoscape and Cytoscape.js, not all properties are mapped to JSON. The following properties are not supported by the exporter:

- Custom Graphics and their locations
- Edge Bends
- Nested Networks
- Network Background (Note: This can be set manually as standard CSS in Cytoscape.js)

The Continuous-Discrete Editor is used to create mappings from numerical data values to discrete properties, such as fonts, shapes, or line styles. To edit a value for a specific region, double-click on the icon on the track.

Importing Styles

To import existing styles, navigate to the menu option **File** → **Import** → **Styles...** and select a `styles.xml` (Cytoscape 3 format) file. Imported properties will supplement existing properties or override existing ones if the properties have the same name. You can also specify a style file using the -V command line option. Properties loaded from the command line will override any default properties. Note that legacy `.props` files can only be loaded via the **File** → **Import** → **Styles...** menu, but not by command line.

App Manager

What are Apps?

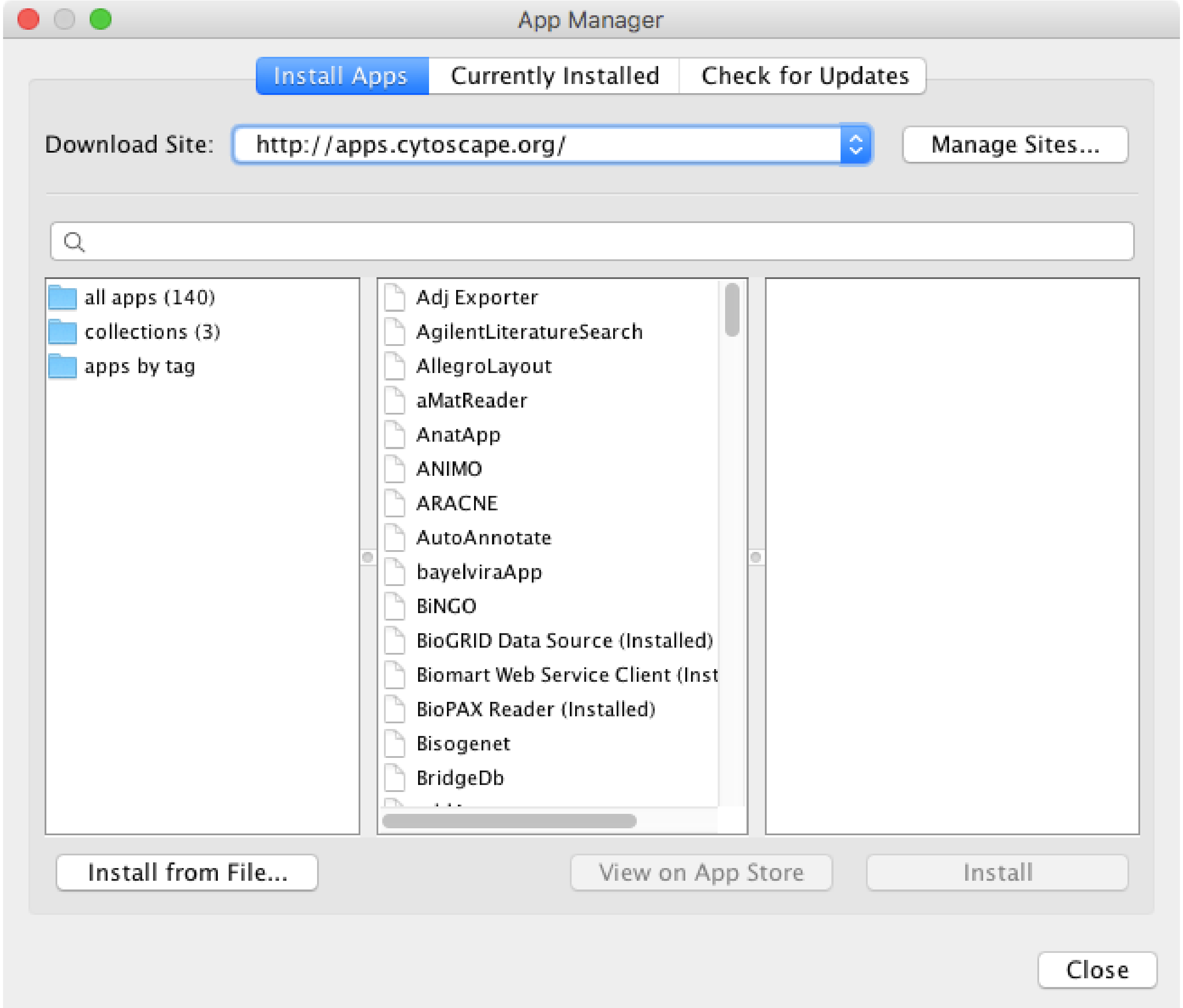
Cytoscape's capabilities are not fixed. They can be expanded with **apps**. They can extend Cytoscape in a variety of ways. One app can have the ability to import data from an online database. Another app could provide a new method for analyzing networks. You can install apps after you have installed Cytoscape. Most apps were made by Cytoscape users like you.

If you're familiar with Cytoscape 2.x, you probably know that Cytoscape apps were called **plugins**. Starting in Cytoscape 3.0, we are calling them **apps**. Cytoscape 2.x plugins cannot be used in Cytoscape 3.0.

Installing Apps

You can install apps through the App Store or within Cytoscape. In this section, we'll talk about installing apps through Cytoscape. You can learn how to install apps through the App Store here (http://apps3.nrnb.org/help/getstarted_app_install).

To install apps within Cytoscape, go to the menu bar and choose **Apps** → **App Manager**. At the top of the **App Manager** window, make sure you have the **Install** tab selected.



Download Site: http://apps.cytoscape.org/

There are four ways you can find apps:

- If you know the name of an app you're looking for, enter it in the **Search** field. The App Manager will list the apps whose names or descriptions match the **Search** field in the middle panel.
- If you're not sure what sort of app you and want to see everything, click the **all apps** folder. In the middle pane, you will see a list of all the apps.
- If you want to install a collection of apps for a specific use case, click on the **collections** folder. This will display the available collections in the middle pane. A collection is simply an app that installs other apps for a specific use case.
- If you have a general idea of what sort of app you're looking for, double-click on the **apps by tag** folder, then click on one of the tags that interests you. The apps with that tag are listed in the middle pane.

When you click on an app (or collection) in the middle panel, the App Manager shows its short description and icon in the right panel. If you want more information, click the **View on App Store** button on the bottom-right. If you want to go ahead and install, click the **Install** button.

If you've downloaded an app to your computer, you can install it by clicking the **Install from File** button on the bottom-left.

Managing your Installed Apps

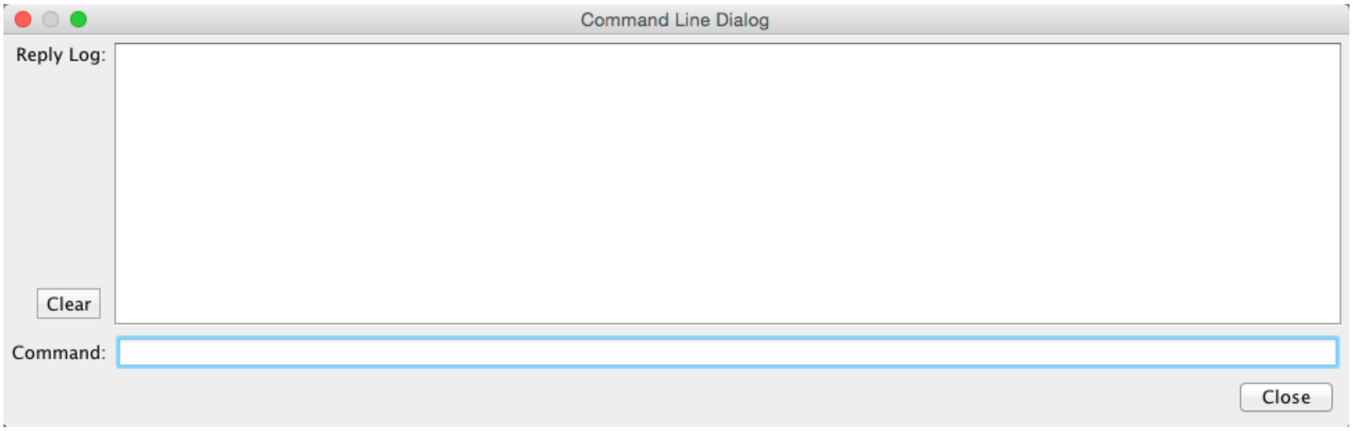
You can see a list of all apps you have installed by clicking the **Currently Installed** tab at the top. When you click on an app in the list, you'll see a description of your app at the bottom. At the bottom, you'll see a couple buttons where you can:

- **Uninstall** an app. This deletes the app from your computer. If you want to reinstall the app, you will have to find it again in the **Install from App Store** tab or in the App Store site and reinstall it there.
- **Disable** an app. This temporarily disables the app. The app stays on your computer, but Cytoscape does not load it. You can enable the app by first selecting the disabled app in the list, then click **Enable**.

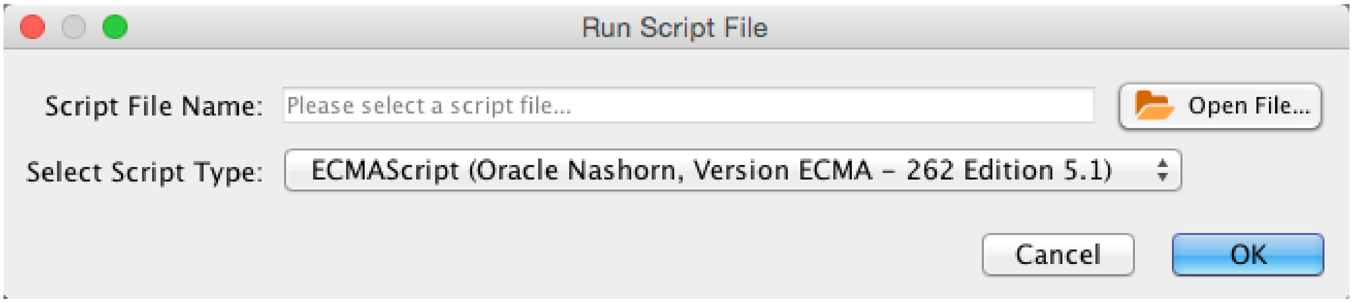
Note that uninstalling or disabling a collection will not uninstall or disable any apps installed by the collection.

Command Tool

The **Command Line Tool** provides a simple command-line interface to Cytoscape using the Commands API. Any app that registers commands will be available through the Command Tool. The Command Tool provides two main functions: first, a Command Line Dialog is available from **Tools** → **Command Line Dialog**, that allows the user to type commands into Cytoscape and see the results in a "Reply Log".



Second, and arguably more useful, it will read script files and execute them. Each line in the script file is a command that is sent to a app. Script files may be entered on the Cytoscape command line using the "-S" flag to Cytoscape, through the **File** → **Run Script File...** menu item, or through **Tools** → **Execute Command File**.



Run Script File

Script File Name: Please select a script file...

Select Script Type: ECMAScript (Oracle Nashorn, Version ECMA – 262 Edition 5.1)

Cytoscape commands consist of three parts: a command class, or namespace; a command within that namespace; and a series of arguments or options provided as a series of **name=value** pairs. For example, to import an XGMML format file from the **Command Line Dialog** or a command script, you would use:

```
network import file filePath="path-to-file"
```

where *network* is the namespace, *import file* is the command, and there is only one argument: *filePath="path-to-file"*. If there were more arguments they would appear on the same line separated by spaces.

The Command Tool also uses the Command API to provide help. "help" by itself will list all of the command classes (or namespaces) and "help " followed by a namespace will list all of the commands supported by that namespace. Details of a specific command are available by typing "help " followed by the namespace and command (e.g. "help layout force-directed"). The Command Tool registers the "command" namespace and supports a single command: run, which takes a file argument. Here is the help for the command run command from the command namespace:

```
help command run
       command run file=<File>
```

Similarly, the help for the "network import file" example from above is:

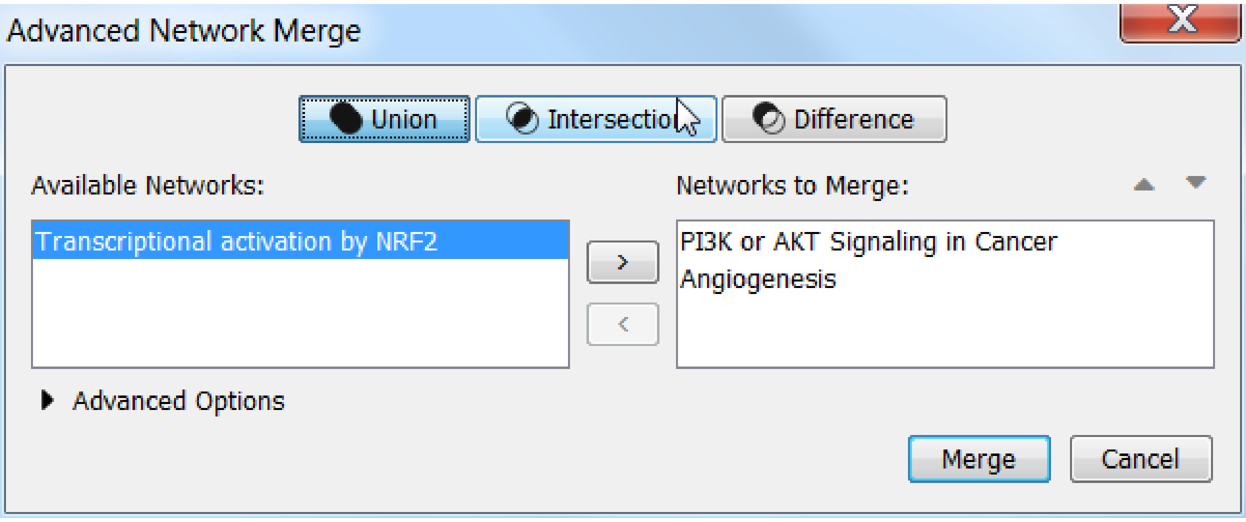
```
help network import file
 network import file arguments:
 dataTypeList=<String>: List of column data types ordered by column index
   (e.g. "string,int,long,double,boolean,intlist" or just "s,i,l,d,b,il")
 defaultInteraction=<String>: Default interaction type
 delimiters=<ListMultipleSelection [,,;, ,\t]>: Text Delimiters
 delimitersForDataList=<ListSingleSelection (\||\|/|,)>:
   Text Delimiters for data list type
 file=<File>: Data Table file
 firstRowAsColumnNames=true|false: First row used for column names
 indexColumnSourceInteraction=<int>: Column for source interaction
 indexColumnTargetInteraction=<int>: Column for target interaction
 indexColumnTypeInteraction=<int>: Column for interaction type
 NetworkViewRendererList=<ListSingleSelection ()>: Network View Renderer
 RootNetworkList=<ListSingleSelection (-- Create new network collection
   --|Network)>: Network Collection
 startLoadRow=<int>: Start Load Row
 TargetColumnList=<ListSingleSelection ()>: Node Identifier Mapping Column
```

Merge

Cytoscape allows for merging of both network and table data, through **Tools** → **Merge**.

Merge Networks

The **Advanced Network Merge** interface is available from **Tools** → **Merge** → **Networks...** and allows for merging of two or more networks.



Available Networks:

Transcriptional activation by NRF2

Networks to Merge:

PI3K or AKT Signaling in Cancer
Angiogenesis

Basic Operations

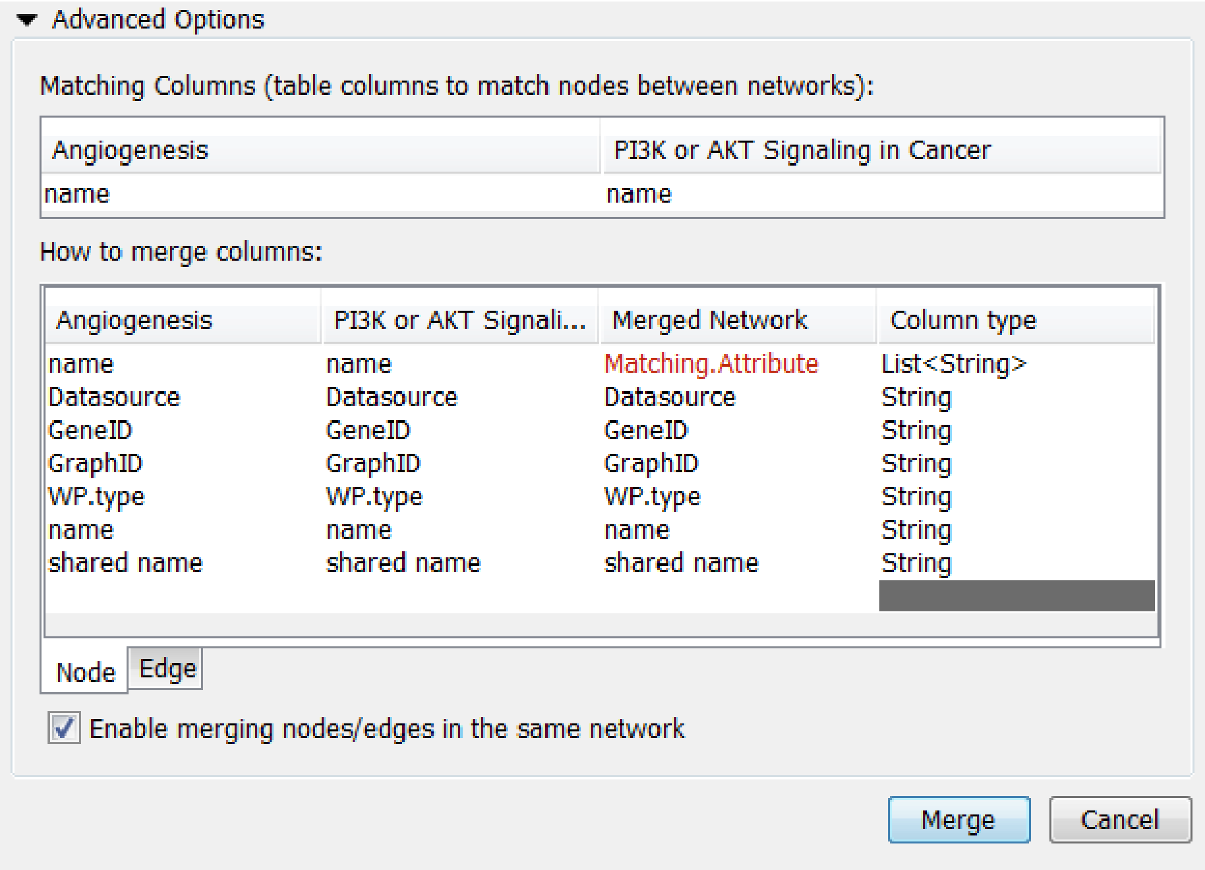
- With the buttons select either “union”, “intersection” or “difference”.
- Networks available for merge are listed under **Networks to merge**. Select a network from

the list and click the right arrow to transfer the network to **Selected networks**. Click **Merge** to continue. The merged network will be displayed as a separate network.

Advanced Options

The **Advanced Network Merge** interface includes an expandable **Advanced Network Merge** panel, where you can specify the details of how to merge the networks. The options available here are:

- **Matching columns**: This specifies the network columns that should be used for merging. Typically, the "name" column or some other column containing identifier information is used here.
- **How to merge columns?**: A table lets the user specify for each of the individual network columns, what the corresponding column in the merged network should be named and its data type.



Matching Columns (table columns to match nodes between networks):

Angiogenesis	PI3K or AKT Signaling in Cancer
name	name

How to merge columns:

Angiogenesis	PI3K or AKT Signali...	Merged Network	Column type
name	name	Matching.Attribute	List<String>
Datasource	Datasource	Datasource	String
GeneID	GeneID	GeneID	String
GraphID	GraphID	GraphID	String
WP.type	WP.type	WP.type	String
name	name	name	String
shared name	shared name	shared name	String

☑ Enable merging nodes/edges in the same network

Merge Tables

NetworkAnalyzer

NetworkAnalyzer computes a comprehensive set of topological parameters for undirected and directed networks, including:

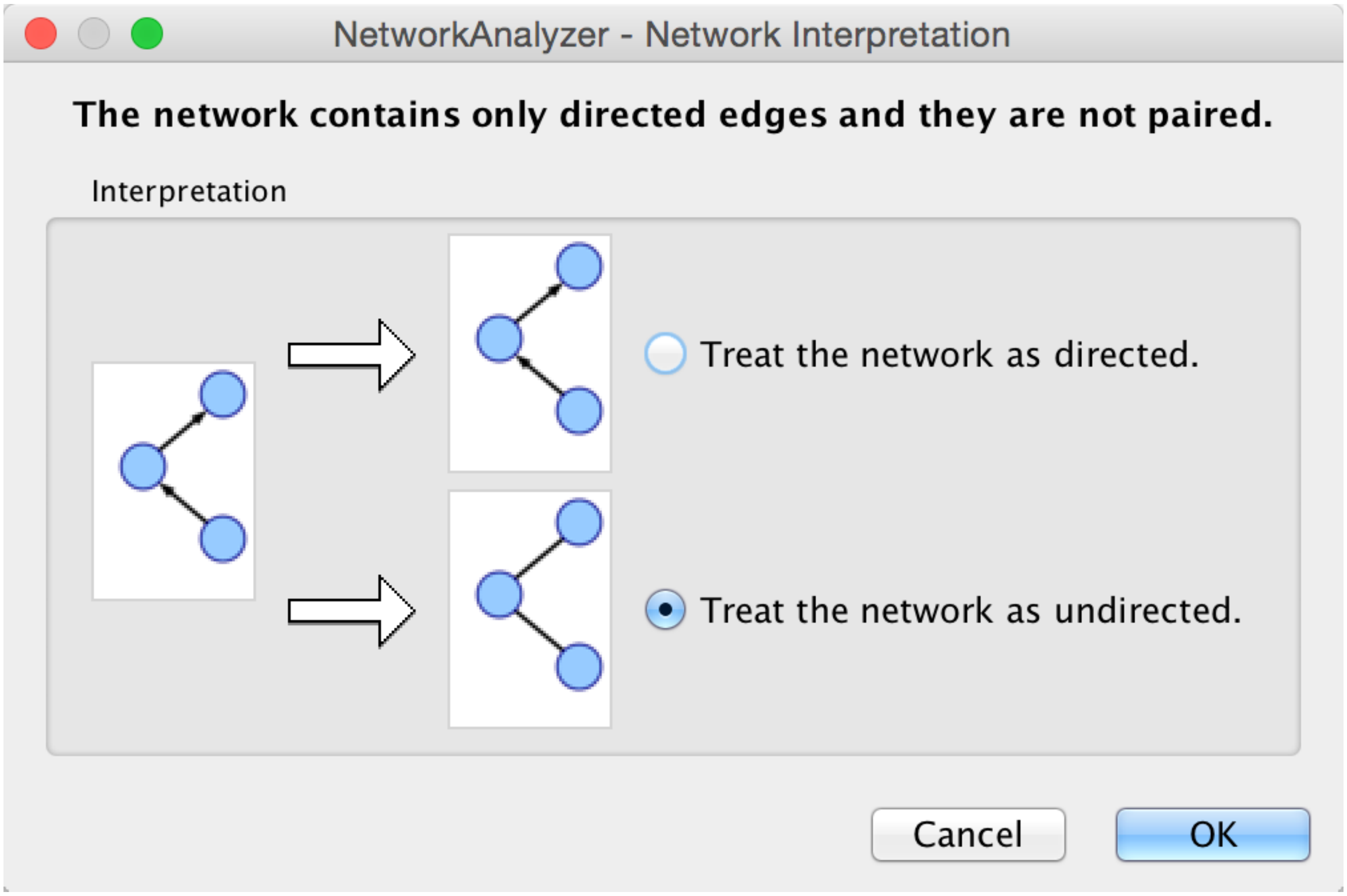
- Number of nodes, edges and connected components.
- Network diameter, radius and clustering coefficient, as well as the characteristic path length.
- Charts for topological coefficients, betweenness, and closeness.
- Distributions of degrees, neighborhood connectiveness, average clustering coefficients, shortest path lengths, number of shared neighbors and stress centrality.

NetworkAnalyzer also constructs the intersection, union and difference of two networks. It supports the extraction of connected components as separate networks and the removal of self-loops.

Network Analysis

Analyze Network

To run NetworkAnalyzer, select **Tools** → **NetworkAnalyzer** → **Network Analysis** → **Analyze Network**.


NetworkAnalyzer - Network Interpretation

The network contains only directed edges and they are not paired.

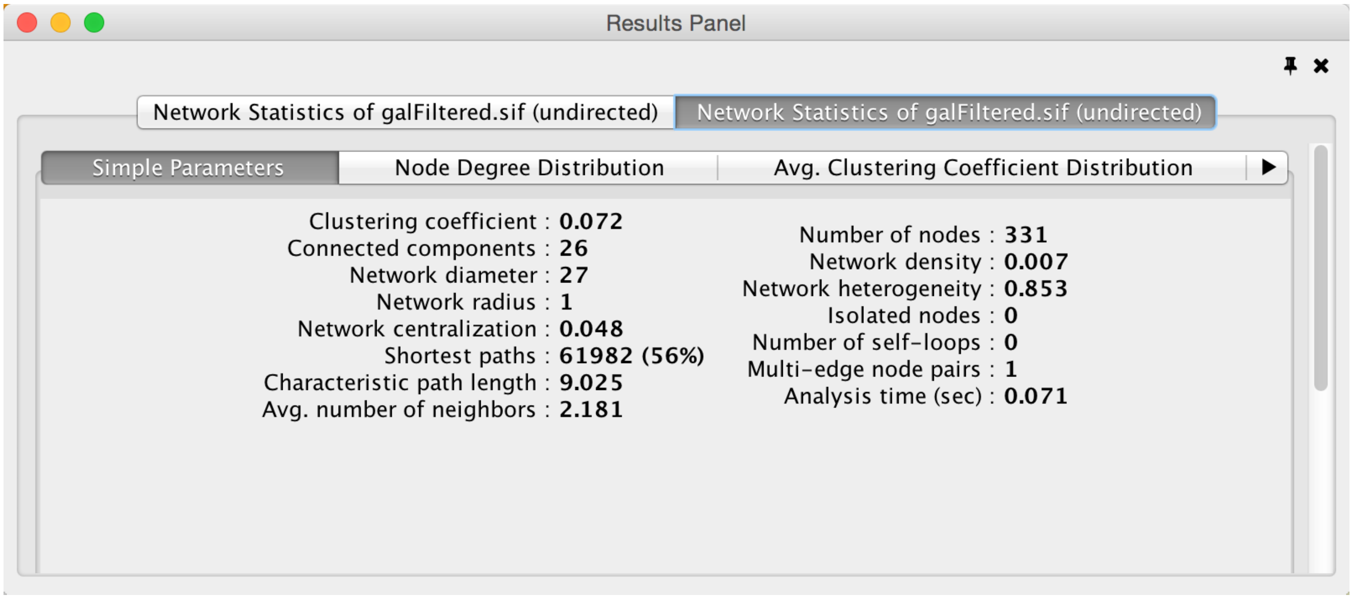
Interpretation

○ Treat the network as directed.

● Treat the network as undirected.

The NetworkAnalyzer will determine whether your network contains directed or undirected edges. At this point, you can choose to ignore edge direction information.

When results are calculated, they will appear in the **Results Panel**.


Results Panel
Network Statistics of galFiltered.sif (undirected)
Network Statistics of galFiltered.sif (undirected)
Simple Parameters
Node Degree Distribution
Avg. Clustering Coefficient Distribution

```
     Clustering coefficient : 0.072            Number of nodes : 331
     Connected components : 26                Network density : 0.007
         Network diameter : 27           Network heterogeneity : 0.853
           Network radius : 1                   Isolated nodes : 0
   Network centralization : 0.048        Number of self-loops : 0
           Shortest paths : 61982 (56%)  Multi-edge node pairs : 1
Characteristic path length : 9.025          Analysis time (sec) : 0.071
Avg. number of neighbors : 2.181
```

The results have multiple tabs. Details on the network parameters can be found here.

- **Simple Parameters**
- **Node Degree Distribution**
- **Avg. Clustering Coefficient Distribution**
- **Topological Coefficients**
- **Shortest Path Distribution**
- **Shared Neighbors Distribution**
- **Neighborhood Connectivity Distribution**
- **Betweenness Centrality**
- **Closeness Centrality**
- **Stress Centrality Distribution**

You can also save the statistics for later use by using the **Save Statistics** button.

Analyze Subset of Nodes

An exhaustive topological analysis of very large networks can be a time consuming task. The computation of local parameters for the nodes is significantly faster than the computation of global (path-related) parameters. Examples of local parameters are node degree, neighborhood connectivity, topological and clustering coefficients. Betweenness and closeness centralities, as well as stress, are global parameters.

NetworkAnalyzer provides the **Analyze Subset of Nodes** option for computing local parameters for a subset of nodes. If one or more nodes in the network are selected before starting an analysis, only the sub-network induced by the selected nodes is analyzed. Moreover, only local parameters are computed. Shared neighbors distribution and shortest path lengths distribution, among others, are not displayed in the results.

Batch Analysis

The **Batch Analysis** option is used to perform topological analysis on all networks stored in specific directory, using all possible interpretations for every network. Batch analysis consists of three simple steps:

- **Selecting directories**: The user selects the input and output directories. The input directory should contain network files that can be loaded into Cytoscape. Sub-directories of the input directory are not considered. The output directory is the one that will contain all analysis results after the batch analysis. In order to avoid file overwriting, NetworkAnalyzer requires that the output directory is empty (contains no files) before the batch analysis starts.
- **Analysis**: NetworkAnalyzer scans the input directory and loads all supported networks into Cytoscape, one at a time. Each loaded network is inspected and then it is analyzed considering all possible interpretations for it. The analysis step is complete after all networks are analyzed. Note that depending on the number of networks and their sizes, this might be a very time-consuming step.
- **Inspection of results**: After the analysis is complete, the button **Show Results** is enabled, and it displays the results dialog. The dialog contains a table of all topological analyses performed. Every row in the results table lists the loaded network, its interpretation and the resulting network statistics file that was saved in the output directory. By clicking on a network name and on statistics file name, the user can load the network and topology analysis results, respectively.

Load Network Statistics

Existing network statistics can be loaded from a file saved previously in NetworkAnalyzer.

Plot Parameters

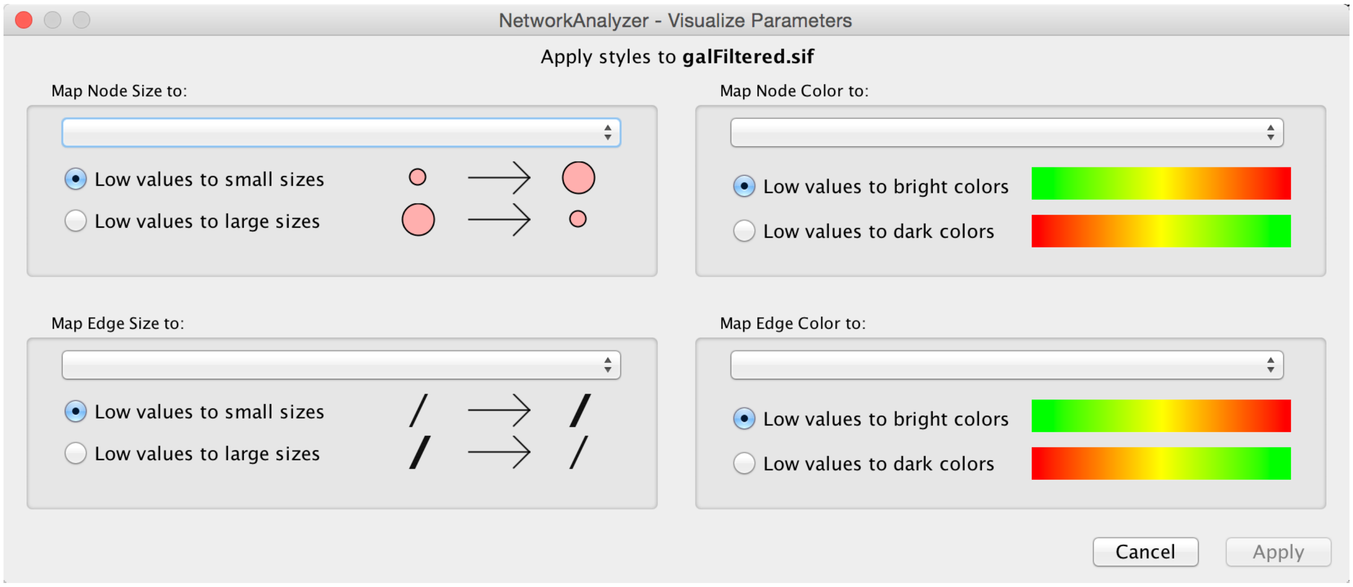
The **Plot Parameters** dialog offers a possibility to plot two parameters against each other. The parameters to be plotted can be chosen from two drop-down menus. The **Table Column 1** menu provides the values for the domain/category axis, and the **Table Column 2** menu specifies the values for the range/value axis. The plot is updated each time a different parameter is selected in one of the menus.

Generate Style from Statistics

NetworkAnalyzer computed parameters can be visualized as node/edge size and color, if the **Store node / edge parameters in node / edge table** option in **NetworkAnalyzer Settings** is enabled. Parameters loaded from a .netstats file cannot be visualized because the network itself is not stored in the network statistics file. If, after performing topological analysis, the network is modified by introducing or removing nodes or edges, it is recommended (and sometimes required) to run NetworkAnalyzer again before visualizing any parameters.

The visualization is initiated by the **Generate Style from Statistics...** menu option. There are two ways of mapping computed parameters.

- Map to node / edge size: The computed parameter is mapped to the size of the nodes or edges. Mapping can be straight or inverse, that is, low parameter values can be mapped to small sizes or to large sizes. The smallest node size is set to 10 and the largest one to 100. Regarding edges, size reflects the edge line width and varies between 1 and 8. Refer to the Styles section for details.
- Map to node / edge color: A computed parameter is mapped to the color of the nodes or edges. Two mapping styles are possible - mapping low parameter values to bright colors or to dark colors. By default, the brightest color is green and the darkest color is red. The mapping also uses a middle (intermediate) color, which allows for fine-grained perception of differing values through the color gradient. The default middle color is yellow.


NetworkAnalyzer - Visualize Parameters

Apply styles to **galFiltered.sif**

Map Node Size to:

● Low values to small sizes
○ Low values to large sizes

Map Node Color to:

● Low values to bright colors
○ Low values to dark colors

Map Edge Size to:

● Low values to small sizes
○ Low values to large sizes

Map Edge Color to:

● Low values to bright colors
○ Low values to dark colors

NetworkAnalyzer Settings

The following settings can be configured by the user:

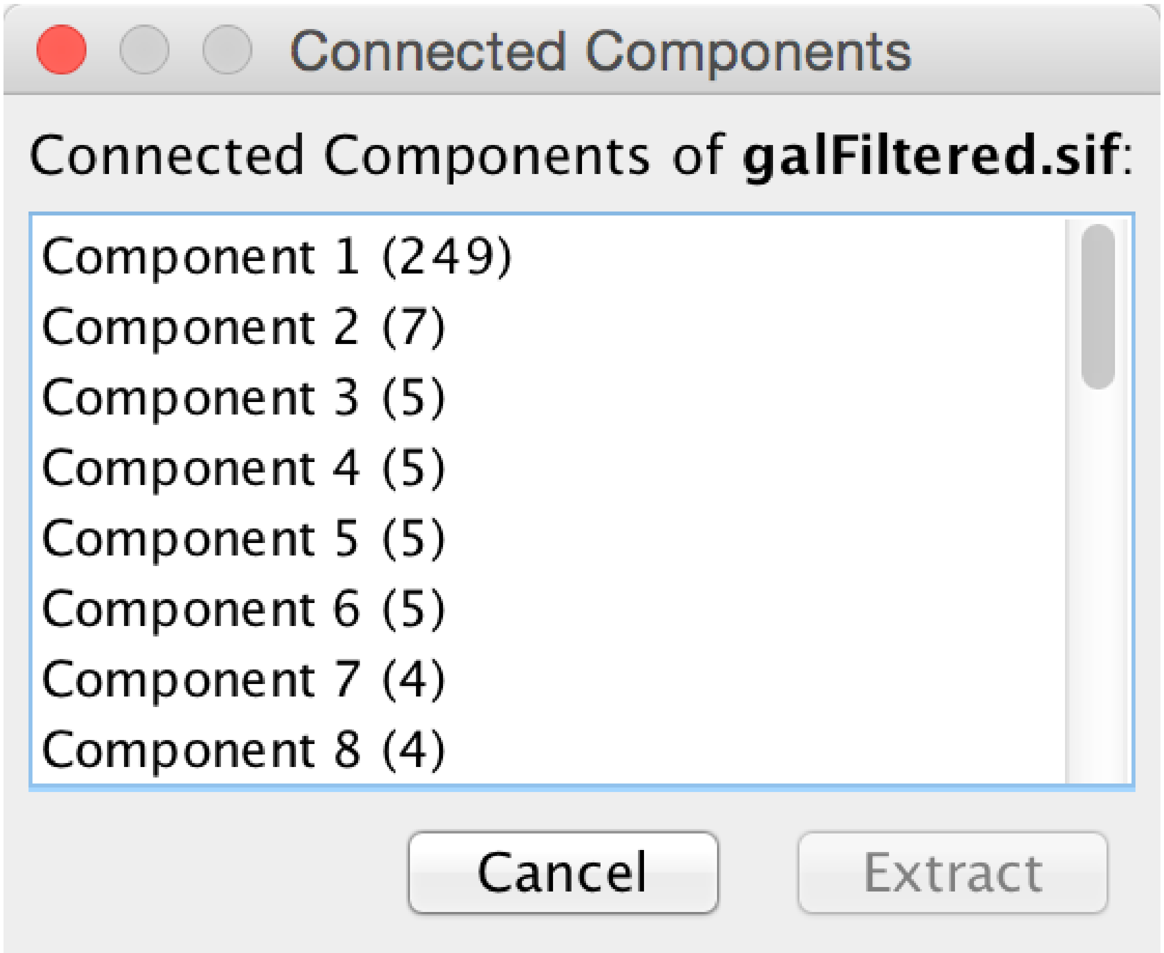
- **Store node / edge parameters in node / edge table**: For every node in a network, NetworkAnalyzer computes its degree (in- and out-degrees for directed networks), its clustering coefficient, the number of self-loops, and a variety of other parameters.

NetworkAnalyzer also computes edge betweenness for each edge in the network. If the respective options are enabled, NetworkAnalyzer can stores the computed values as columns of the corresponding nodes and edges. This enables the users to use the values in Styles or to select nodes or edges based on the values. A complete list of the computed node and edge columns is available here.

- **Use expandable dialog interface for the display of network statistics**: If this option is enabled, analysis results are presented in a window in which all charts are placed below each other in expandable boxes. If this option is disabled, analysis results are presented in a window that contains tabs for the group of simple parameters and for every complex parameter (default). Users who wish to simultaneously view two or more complex parameters of one network, should enable this option.
- NetworkAnalyzer allows the user to change the default colors of parameter visualization.
  - **Background color for parameter visualization**: The color of the background in the network view. It is initially set to the default Cytoscape background color.
  - **Bright color to map parameters**: This color defines the brightest color that parameters can be mapped to. By default its value is green.
  - **Middle color**: This color defines the intermediate color, that parameters can be mapped to. By default its value is yellow.
  - **Dark color**: This color defines the darkest color that parameters can be mapped to. By default its value is red.
- **Location of the help documents**: URL of the original help web page for NetworkAnalyzer. This also enables the local download and storage of this help page.

Subnetwork Creation

NetworkAnalyzer allows for the creation of sub-networks of connected components. The user selects a number of connected components from a list and each selected component is visualized as a sub-network. To create sub-networks from connected components, select **Tools** → **NetworkAnalyzer** → **Subnetwork Creation** → **Extract Connected Components**.



Connected Components

Connected Components of **galFiltered.sif**:

Component 1 (249)
Component 2 (7)
Component 3 (5)
Component 4 (5)
Component 5 (5)
Component 6 (5)
Component 7 (4)
Component 8 (4)

NetworkAnalyzerDemo: Computation and Visualization of Topological Parameters and Centrality Measures for Biological Networks

Yassen Assenov[1], Nadezhda Doncheva[1], Thomas Lengauer[1], and Mario Albrecht[1]

1 Department of Computational Biology and Applied Algorithmics, Max Planck Institute for Informatics, Campus E1.4, 66123 Saarbrücken, Germany

NetworkAnalyzer is a versatile and highly customizable Cytoscape plugin that requires no expert knowledge in graph theory from the user. It computes and displays a comprehensive set of topological parameters and centrality measures for undirected and directed networks, which includes the number of nodes, edges, and connected components, the network diameter, radius, density, centralization, heterogeneity, clustering coefficient, and the characteristic path length. In addition, NetworkAnalyzer shows charts of the distribution of node degrees, neighborhood connectivities, average clustering coefficients, and shortest path lengths. NetworkAnalyzer also contains extra functionality, for instance, for constructing the intersection or union of two networks.

The NetworkAnalyzer plugin and a comprehensive online documentation with a tutorial are available at http://med.bioinf.mpi-inf.mpg.de/networkanalyzer/.

Data keywords: network, graph, topology

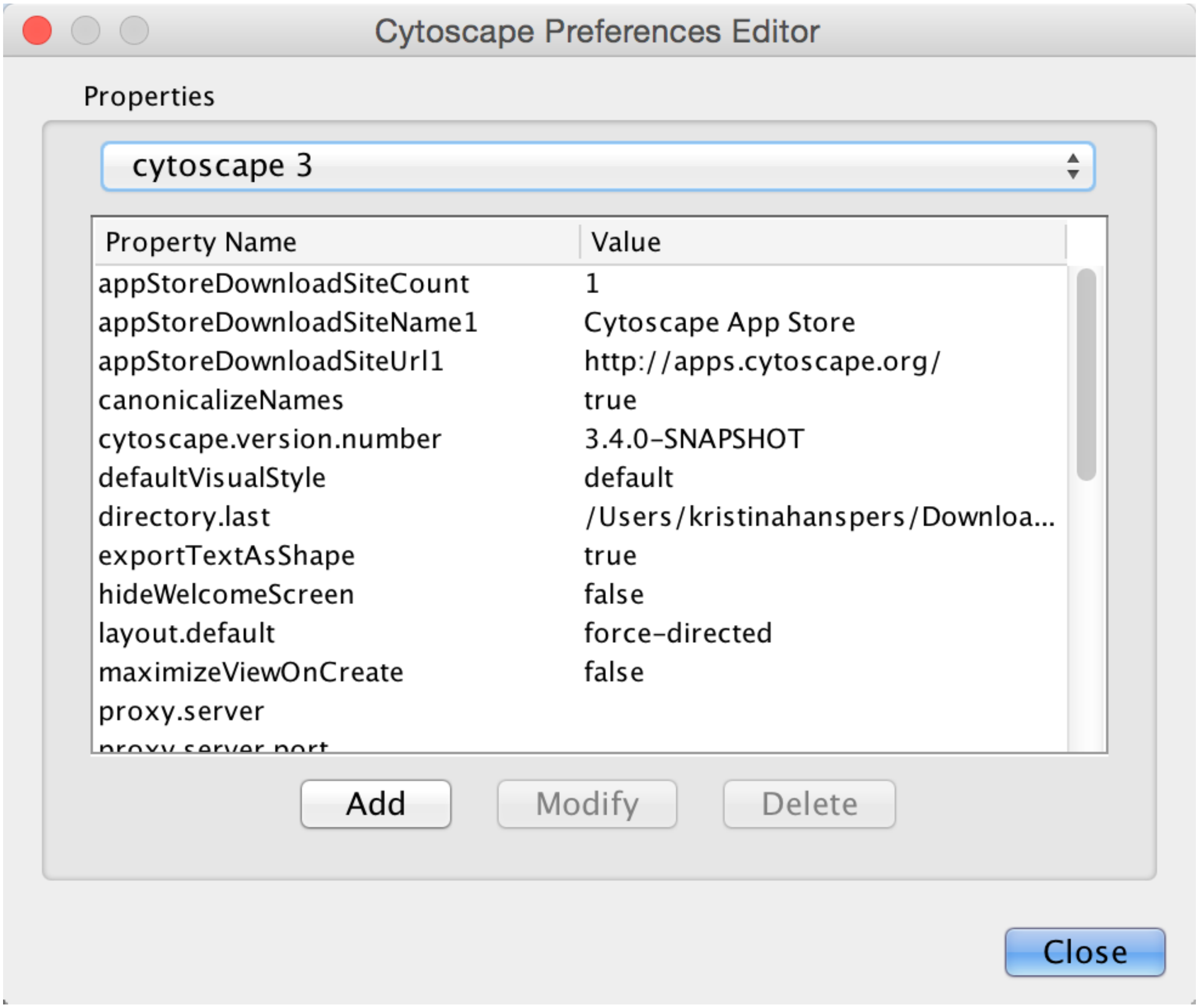
Cytoscape keywords: Network Analysis

Cytoscape Preferences

Managing Properties

The Cytoscape properties editor, accessed via **Edit** → **Preferences** → **Properties...**, is used to specify default properties. Any changes made to these properties will be saved in .props files under the `CytoscapeConfiguration` subdirectory of the user's home directory.

Cytoscape properties are configurable using the Add, Modify and Delete buttons as seen below.



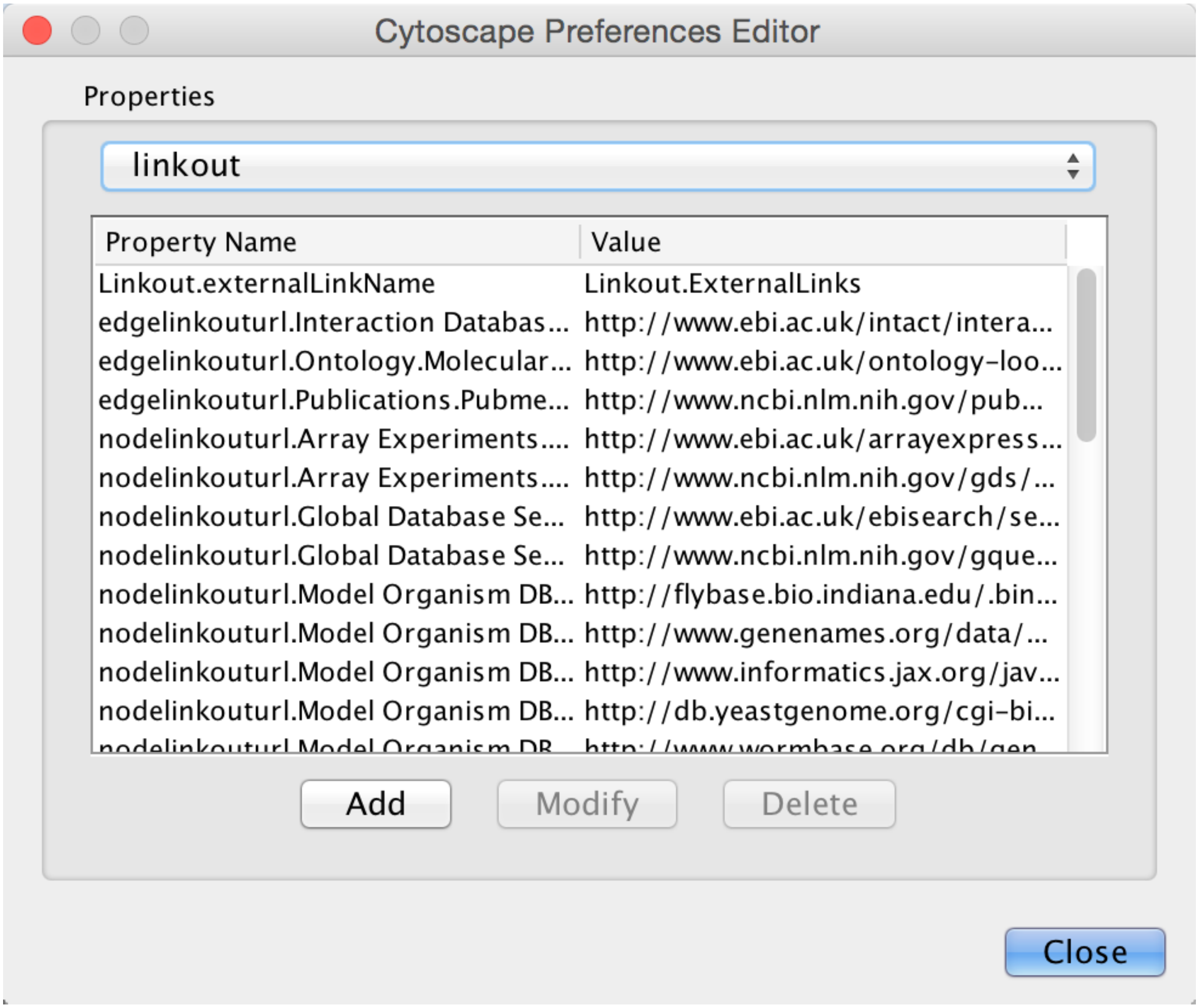
Cytoscape Preferences Editor

Properties

cytoscape 3

Property Name	Value
appStoreDownloadSiteCount	1
appStoreDownloadSiteName1	Cytoscape App Store
appStoreDownloadSiteUrl1	http://apps.cytoscape.org/
canonicalizeNames	true
cytoscape.version.number	3.4.0-SNAPSHOT
defaultVisualStyle	default
directory.last	/Users/kristinahanspers/Downloa...
exportTextAsShape	true
hideWelcomeScreen	false
layout.default	force-directed
maximizeViewOnCreate	false
proxy.server	
proxy.server.port	

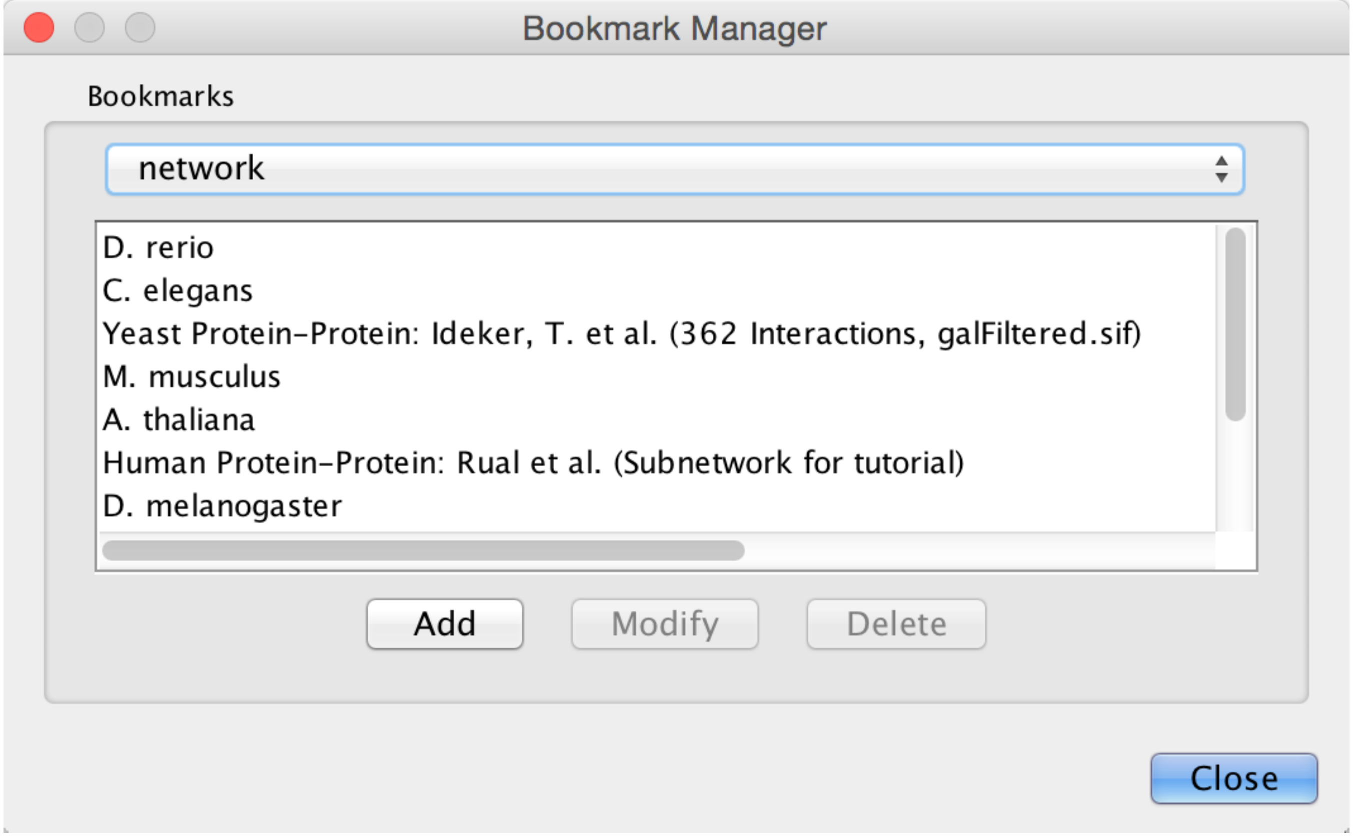
App properties may also be edited in the same way as editing Cytoscape properties. For example, to edit the properties of Linkout, select 'linkout' from the combobox of the Preferences Editor. Some apps may store properties inside session files in addition to (or instead of) storing them in the `CytoscapeConfiguration` directory.



Property Name	Value
Linkout.externalLinkName	Linkout.ExternalLinks
edgelinkouturl.Interaction Databas...	http://www.ebi.ac.uk/intact/intera...
edgelinkouturl.Ontology.Molecular...	http://www.ebi.ac.uk/ontology-loo...
edgelinkouturl.Publications.Pubme...	http://www.ncbi.nlm.nih.gov/pub...
nodelinkouturl.Array Experiments....	http://www.ebi.ac.uk/arrayexpress...
nodelinkouturl.Array Experiments....	http://www.ncbi.nlm.nih.gov/gds/...
nodelinkouturl.Global Database Se...	http://www.ebi.ac.uk/ebisearch/se...
nodelinkouturl.Global Database Se...	http://www.ncbi.nlm.nih.gov/gque...
nodelinkouturl.Model Organism DB...	http://flybase.bio.indiana.edu/.bin...
nodelinkouturl.Model Organism DB...	http://www.genenames.org/data/...
nodelinkouturl.Model Organism DB...	http://www.informatics.jax.org/jav...
nodelinkouturl.Model Organism DB...	http://db.yeastgenome.org/cgi-bi...
nodelinkouturl.Model Organism DB...	http://www.wormbase.org/db/gen...

Managing Bookmarks

Cytoscape contains a pre-defined list of bookmarks, which point to sample network files located on the Cytoscape web server. Users may add, modify, and delete bookmarks through the Bookmark manager, accessed by going to **Edit** → **Preferences** → **Bookmarks...**.

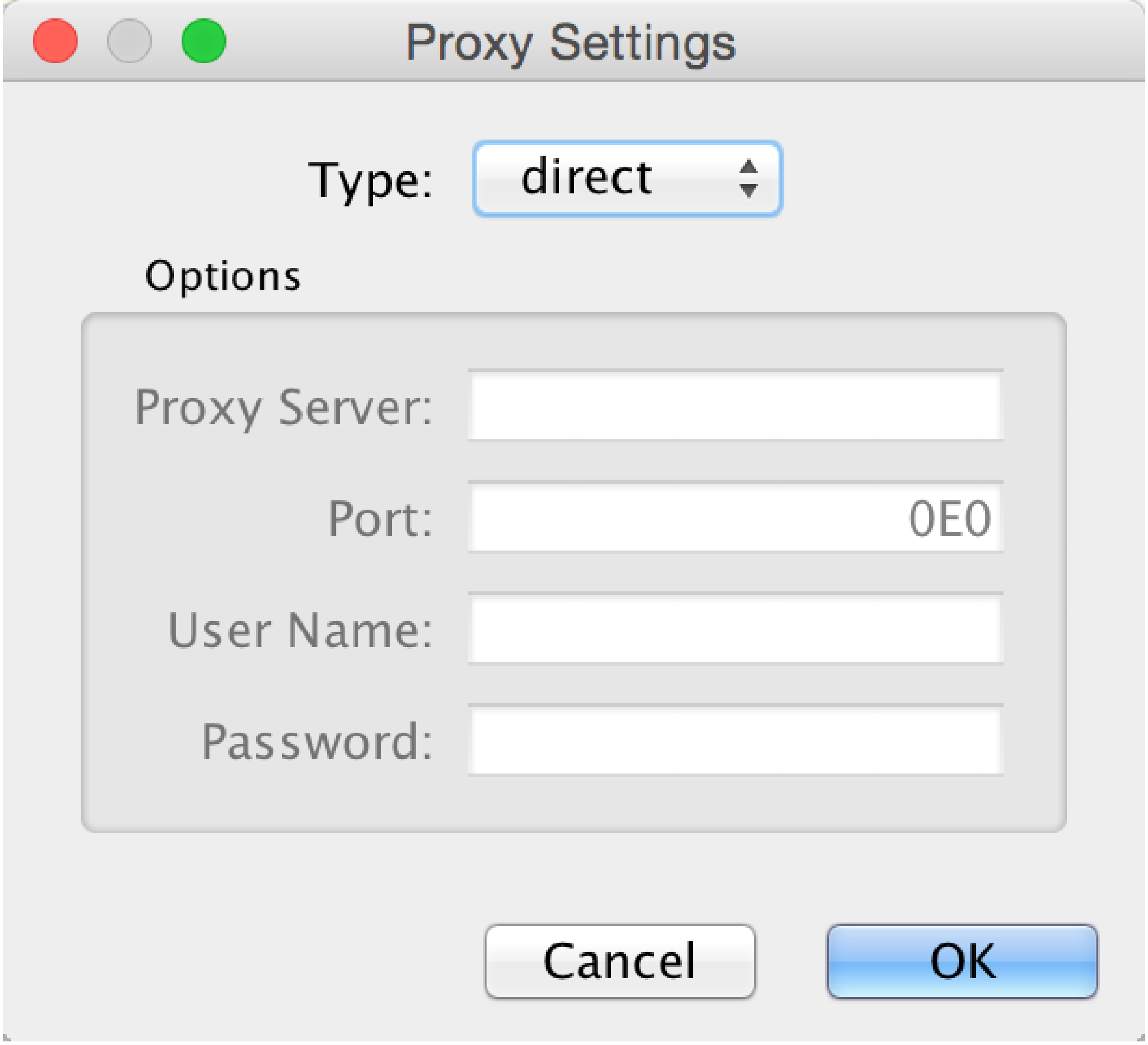


D. rerio
C. elegans
Yeast Protein-Protein: Ideker, T. et al. (362 Interactions, galFiltered.sif)
M. musculus
A. thaliana
Human Protein-Protein: Rual et al. (Subnetwork for tutorial)
D. melanogaster

There are currently several types of bookmarks (based on data categories), including network and table. Network bookmarks are URLs pointing to Cytoscape network files. These are normal networks that can be loaded into Cytoscape. Table bookmarks are URLs pointing to data table files.

Managing Proxy Servers

You can define and configure a proxy server for Cytoscape by going to **Edit** → **Preferences** → **Proxy Settings...**.



Type: direct

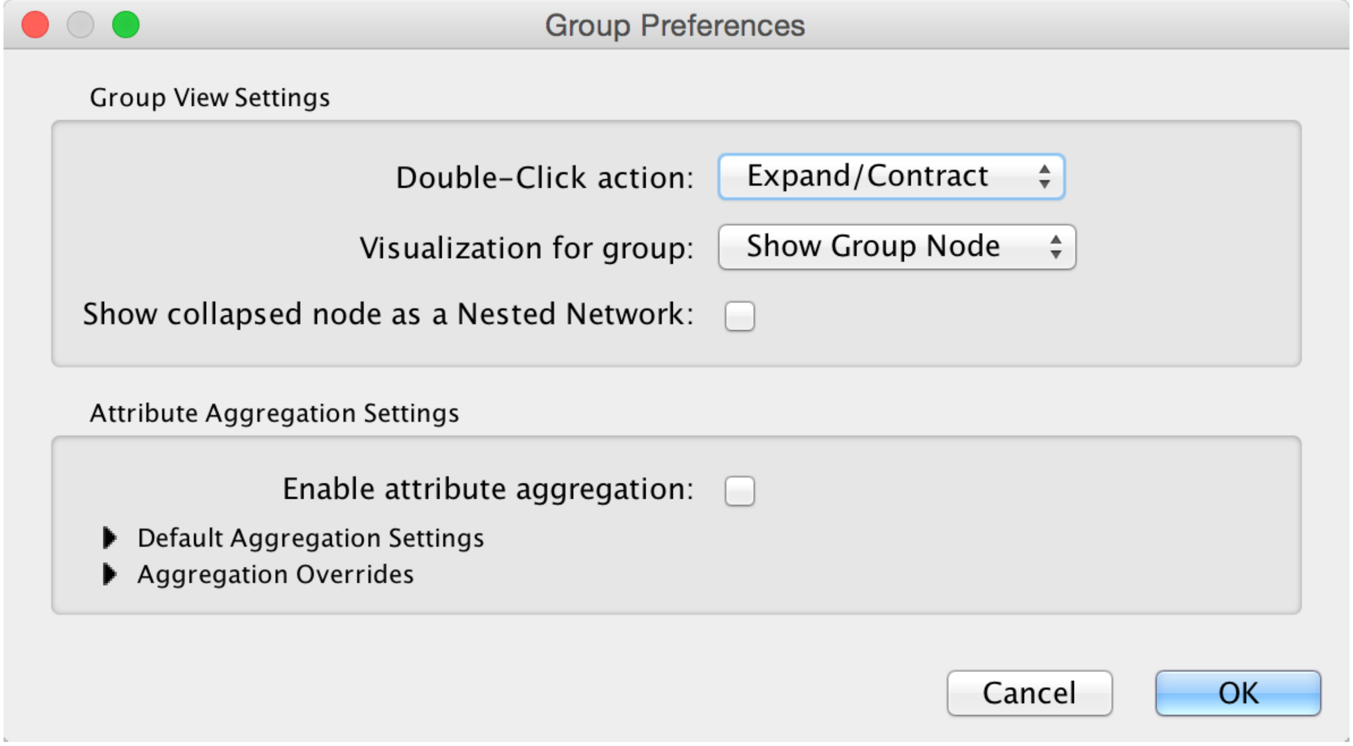
Port: 0E0

After the proxy server is set, all network traffic related to loading a network from URL will pass through the proxy server. Cytoscape apps use this capability as well. The proxy settings are saved in `cytoscape3.props`. Each time you click the OK button after making a change to the proxy settings, an attempt is made to connect to a well known site on the Internet (e.g., google.com) using your settings. For both success and failure you are notified and for failure you are given an opportunity to change your proxy settings.

If you no longer need to use a proxy to connect to the Internet, simply set the Proxy type to “direct” and click the OK button.

Managing Group View

The configuration of Cytoscape group view may also be edited through **Edit** → **Preferences** → **Group Preferences...**.


Group Preferences

Group View Settings

Double-Click action: Expand/Contract

Visualization for group: Show Group Node

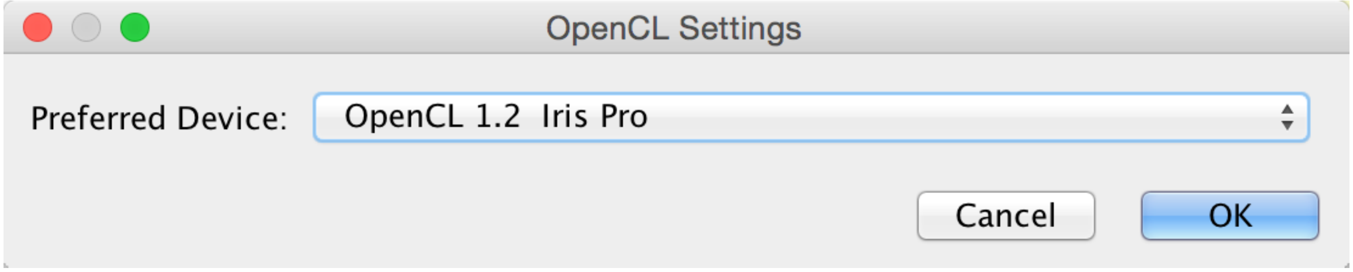
Show collapsed node as a Nested Network: □

Attribute Aggregation Settings

Enable attribute aggregation: □

Managing OpenCL Settings

You can choose between one or more OpenCL drivers installed on your system by going to **Edit → Preferences → OpenCL Settings…**.


OpenCL Settings

Preferred Device: OpenCL 1.2 Iris Pro

OpenCL is a library that enables Cytoscape to use your system's graphics processing unit (GPU) to accelerate certain layouts and other calculations. If no choices are presented, consult the support web page for your system's graphics card.

Linkout

Linkout provides a mechanism to link nodes and edges to external web resources within Cytoscape. Right-clicking on a node or edge in Cytoscape opens a popup menu with a list of web links.

The external links are specified in a `linkout.props` file which is internal to Cytoscape. The defaults include a number of links such as Entrez, SGD, iHOP, and Google, as well as a number of species-specific links. In addition to the default links, users can customize the **External**

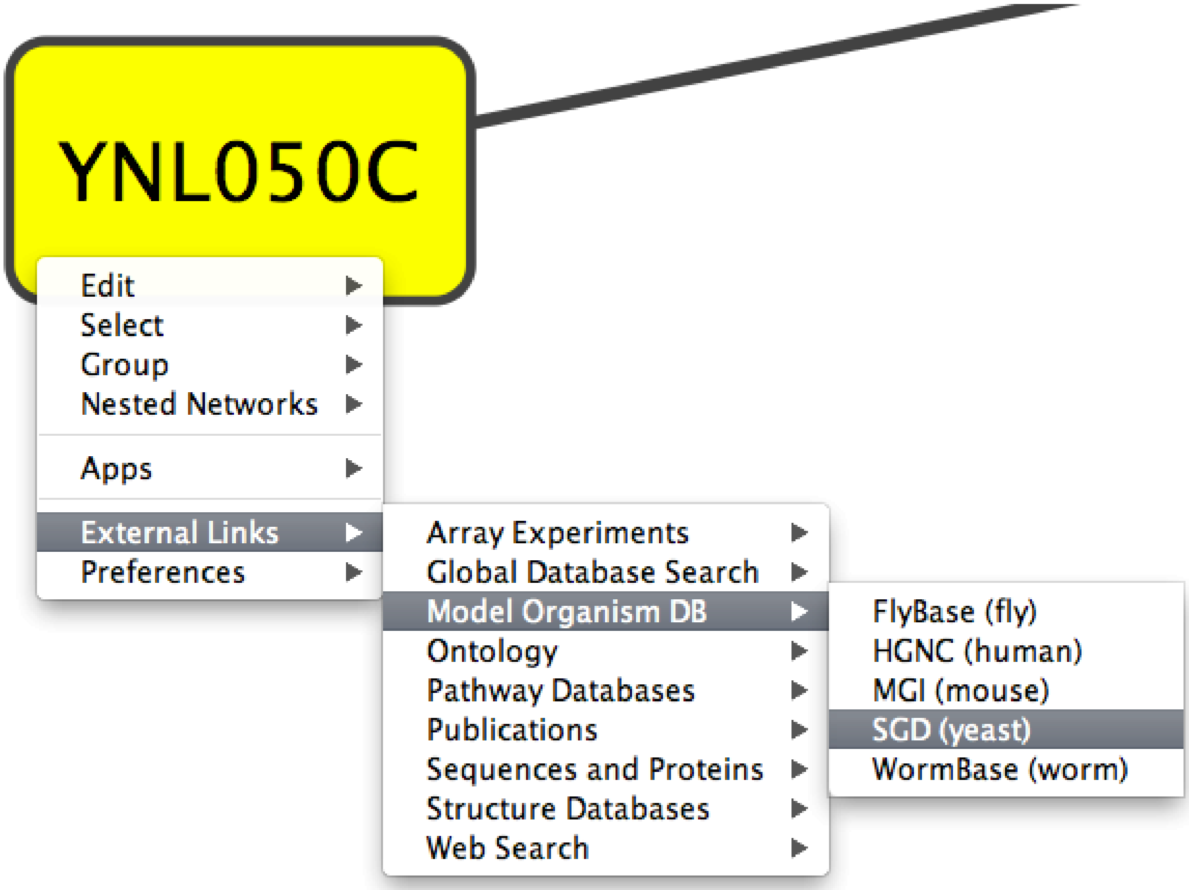
Links menu and add (or remove) links by editing the linkout properties (found under **Edit** → **Preferences** → **Properties...**).

External links are listed as *'key'*-*'value'* pairs in the `linkout.props` file where *key* specifies the name of the link and *value* is the search URL. The LinkOut menus are organized in a hierarchical structure that is specified in the key. Linkout key terms specific for nodes start with the keyword `nodelinkouturl`, for edges this is `edgelinkouturl`.

For example, the following entry:

```
nodelinkouturl.Model Organism DB.SGD (yeast)=http://www.yeastgenome.org/cgi-bin/locus.fpl?
locus=%ID%
```

places the SGD link under the Model Organism DB submenu. This link will appear in Cytoscape as:



In a similar fashion one can add new submenus.

The `%ID%` string in the URL is a place-holder for the node label. When the popup menu is generated this marker is substituted with the node label. In the above example, the generated SGD link for the YNL050C protein is:

```
http://www.yeastgenome.org/cgi-bin/locus.fpl?locus=YNL050C
```

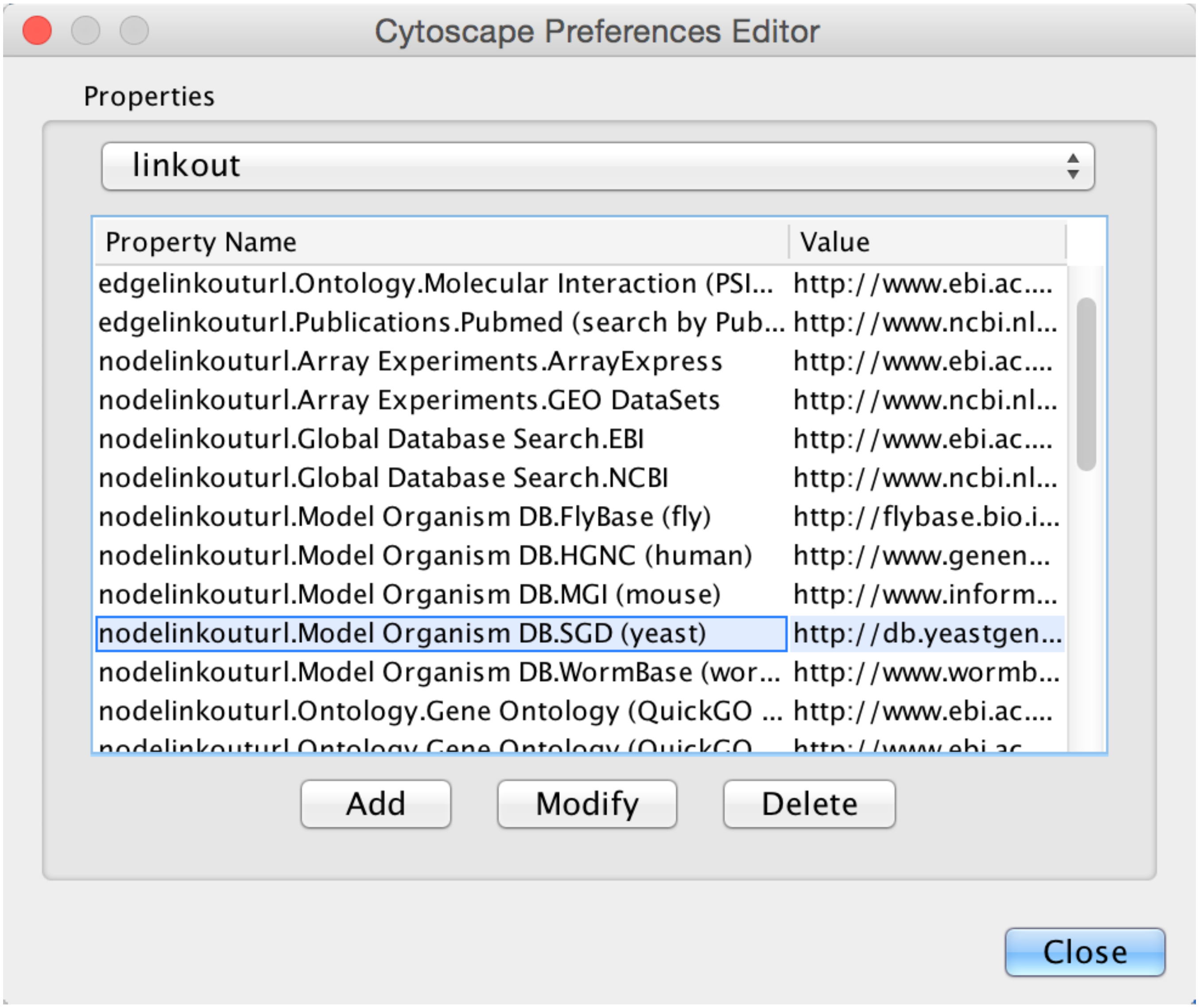
If you want to query based on a different column, you need to specify a different node label using Styles.

For edges the mechanism is much the same; however here the placeholders `%ID1%` and `%ID2%` reflect the source and target node label respectively.

Currently there is no mechanism to check whether the constructed URL query is correct and if the node label is meaningful. Similarly, there is no ID mapping between various identifiers. For example, a link to NCBI Entrez from a network that uses Ensembl gene identifiers as node labels will produce a link to Entrez using the Ensembl ID, which results in an incorrect link. It is the user's responsibility to ensure that the node label that is used as the search term in the URL link will result in a meaningful link.

Adding and Removing Links

The default links are defined in a `linkout.props` file contained inside the Linkout JAR bundle under the framework/system/org/cytoscape/linkout-impl subdirectory of the Cytoscape installation. These links are normal Java properties and can be edited by going to **Edit** → **Preferences** → **Properties...** and selecting linkout from the box (shown below). Linkouts can be modified, added or removed using this dialog; however, note that the modifications would not be stored in the file. To change a URL permanently, you would need to edit the linkout.props file directly.



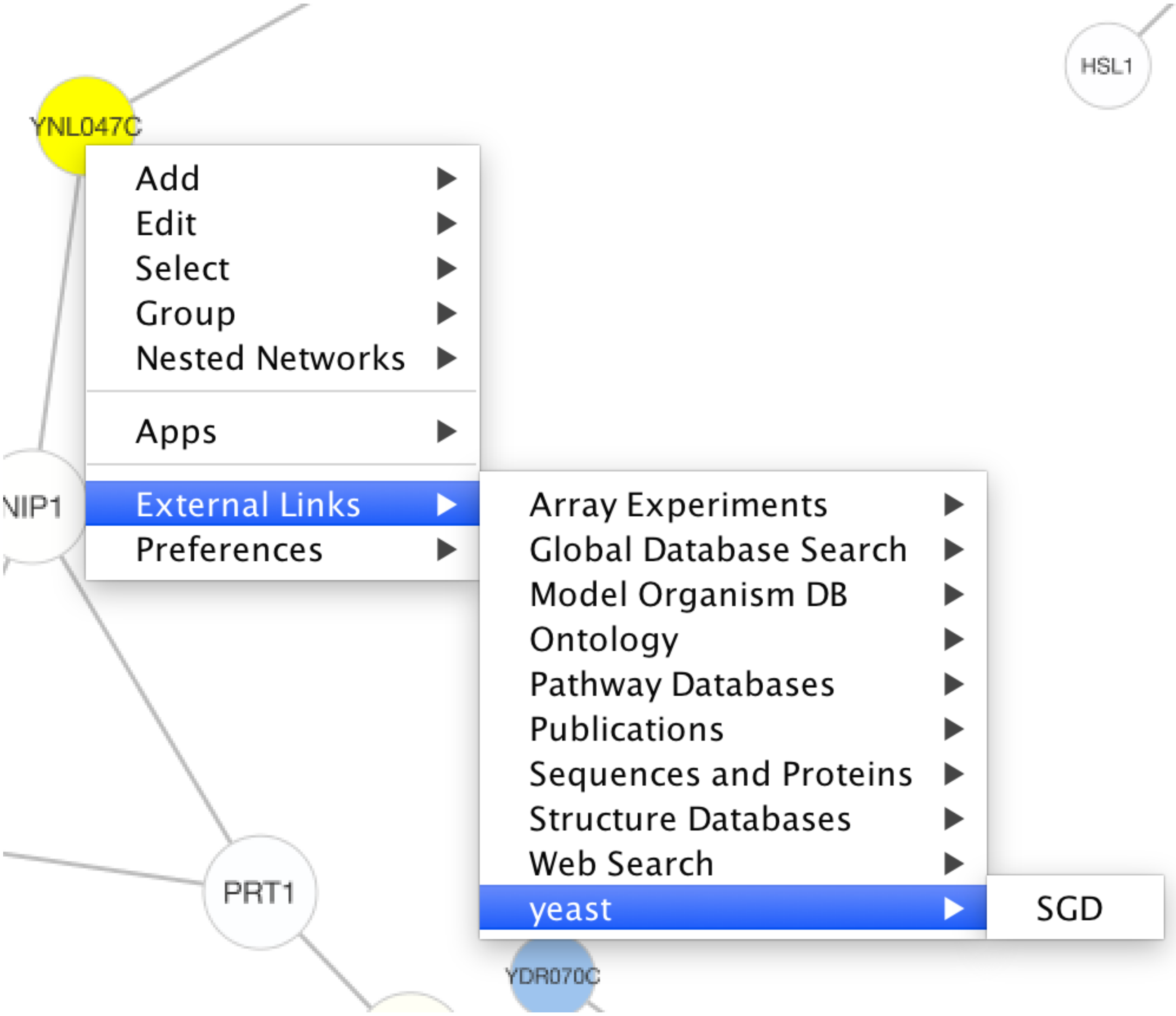
Property Name	Value
edgelinkouturl.Ontology.Molecular Interaction (PSI...	http://www.ebi.ac....
edgelinkouturl.Publications.Pubmed (search by Pub...	http://www.ncbi.nl...
nodelinkouturl.Array Experiments.ArrayExpress	http://www.ebi.ac....
nodelinkouturl.Array Experiments.GEO DataSets	http://www.ncbi.nl...
nodelinkouturl.Global Database Search.EBI	http://www.ebi.ac....
nodelinkouturl.Global Database Search.NCBI	http://www.ncbi.nl...
nodelinkouturl.Model Organism DB.FlyBase (fly)	http://flybase.bio.i...
nodelinkouturl.Model Organism DB.HGNC (human)	http://www.genen...
nodelinkouturl.Model Organism DB.MGI (mouse)	http://www.inform...
nodelinkouturl.Model Organism DB.SGD (yeast)	http://db.yeastgen...
nodelinkouturl.Model Organism DB.WormBase (wor...	http://www.wormb...
nodelinkouturl.Ontology.Gene Ontology (QuickGO ...	http://www.ebi.ac....
nodelinkouturl.Ontology.Gene Ontology (QuickGO ...	http://www.ebi.ac...

In addition, new links can be defined when starting Cytoscape from command line by specifying individual properties. The formatting of the command is `cytoscape.sh -P [context_menu_definition]=[link]`. *context_menu_definition* specifies the context menu for showing the linkout menu item. The structure of this definition is "." separated and the first item needs to be either *nodelinkouturl* or *edgelinkouturl*. The former will add the linkout item as a node context menu and the latter will add it as an edge context menu. The rest of the definition would define the hierarchy of the menu.

For instance this command:

```
cytoscape.sh -P nodelinkouturl.yeast.SGD=http://db.yeastgenome.org/cgi-bin/locus.pl?locus\=%ID%
```

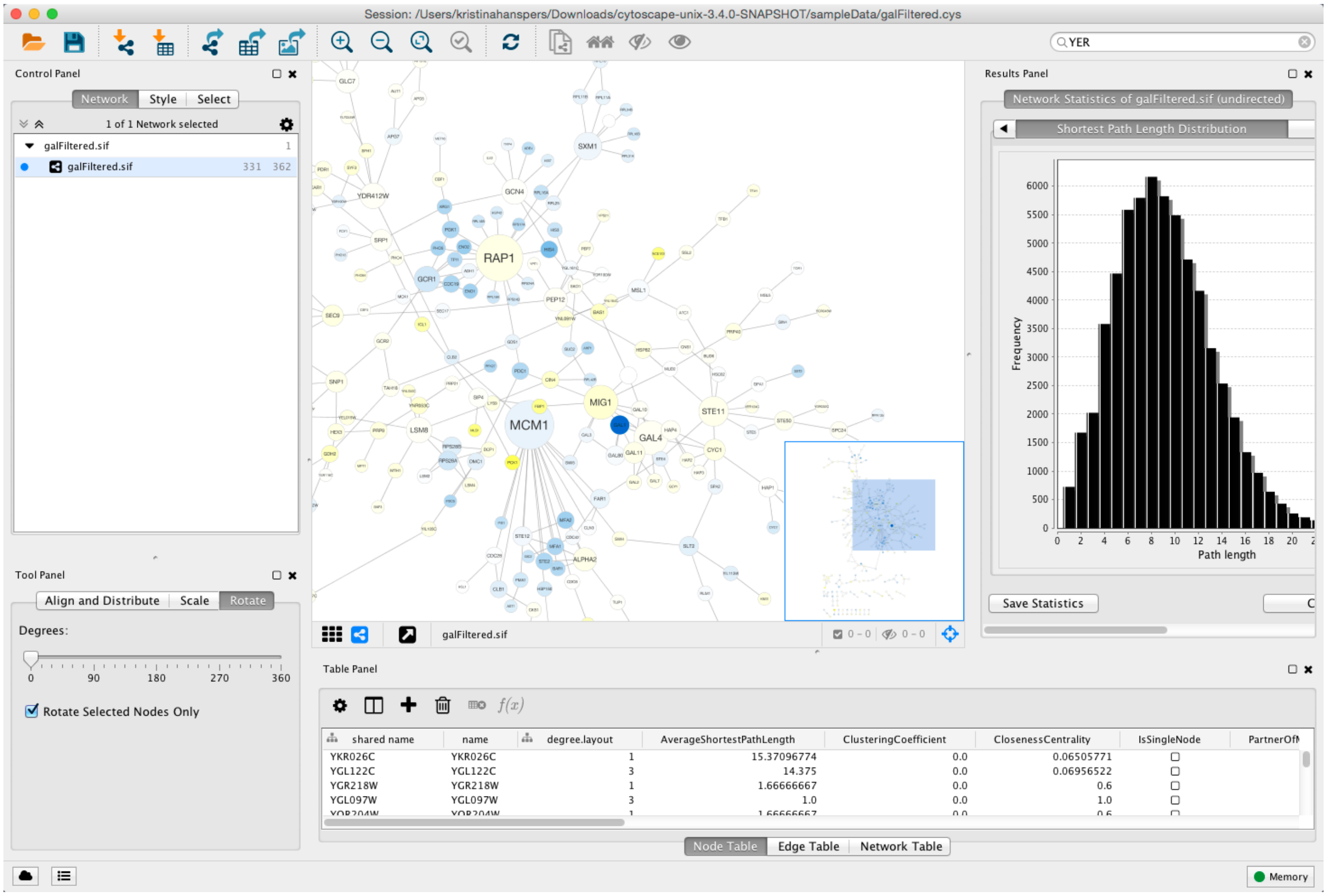
will add this menu item:



To remove a link from the menu, simply delete the property using **Edit** → **Preferences** → **Properties...** and selecting **commandline**. Linkouts added in the command line will be available for the running instance of Cytoscape.

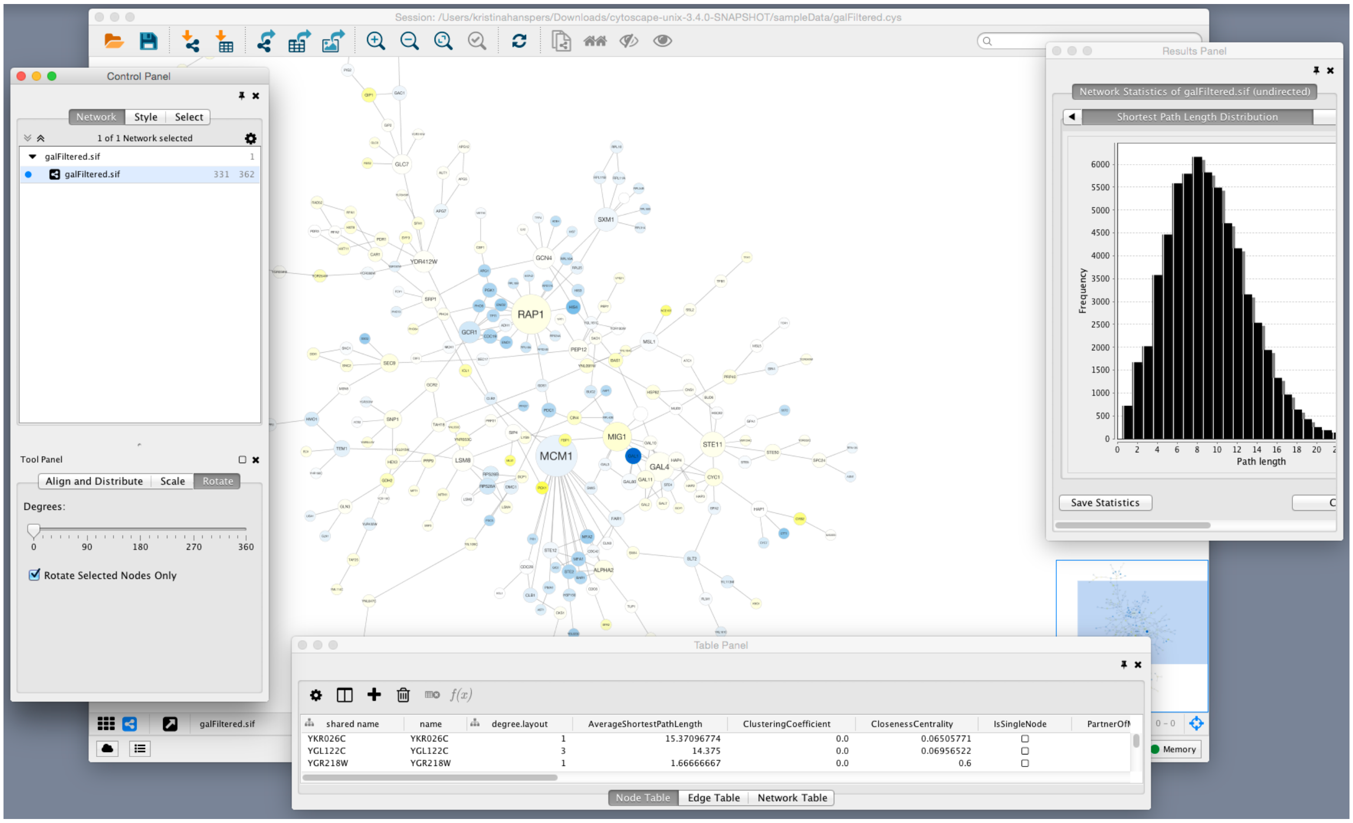
Panels

Panels are floatable/dockable panels designed to cut down on the number of pop-up windows within Cytoscape and to create a more unified user experience. They are named based on their functions – **Control Panel**, **Table Panel**, **Tool Panel** and **Result Panel**. The following screenshot shows the file `galFiltered.sif` loaded into Cytoscape, with a force-directed layout, **Rotate** tools showing in the **Tool Panel**, and with results from Network Analyzer (**Tools** → **Network Analysis** → **Analyze Network**). The **Control Panel** (at the left-hand side of the screen) contains the Network Manager, Network Overview, Styles and Select tabs. On the bottom of the panel, there is another panel called **Tool Panel**. In the **Table Panel**, the **Node Table** is shown. In addition, analysis results from Network Analyzer are shown in **Result Panel** (at the right-hand side).



shared name	name	degree.layout	AverageShortestPathLength	ClusteringCoefficient	ClosenessCentrality	IsSingleNode	PartnerOfN
YKR026C	YKR026C	1	15.37096774	0.0	0.06505771	☐	
YGL122C	YGL122C	3	14.375	0.0	0.06956522	☐	
YGR218W	YGR218W	1	1.66666667	0.0	0.6	☐	
YGL097W	YGL097W	3	1.0	0.0	1.0	☐	
YOR204W	YOR204W	1	1.66666667	0.0	0.6	☐	

The user can then choose to resize, hide or float Panels. For example, in the screenshot below, the Network, Table and Results panels are floating and the Tool Panel is hidden:

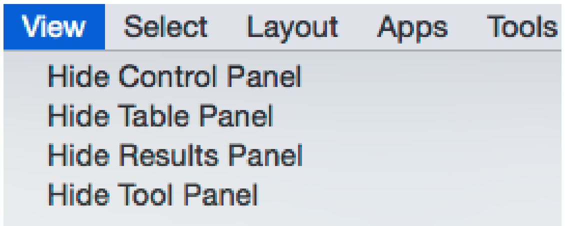


shared name	name	degree.layout	AverageShortestPathLength	ClusteringCoefficient	ClosenessCentrality	IsSingleNode	PartnerOfN
YKR026C	YKR026C	1	15.37096774	0.0	0.06505771	☐	
YGL122C	YGL122C	3	14.375	0.0	0.06956522	☐	
YGR218W	YGR218W	1	1.66666667	0.0	0.6	☐	

Basic Usage

Cytoscape includes four Panels: the **Control Panel** on the left, **Tool Panel** on the bottom of the **Control Panel**, the **Table Panel** on the bottom right, and the **Result Panel** on the right. By default, only the **Control Panel** and the **Data Panel** will appear. The **Result Panel** may appear, depending on the mix of Cytoscape apps that you currently have installed. The **Tool Panel** will appear when you select the following commands under the **Layout** menu: **Rotate**, **Scale**, and **Align and Distribute**.

All panels can be shown or hidden using the **View** → **Show/Hide** functions.



In addition, Panels can be floated or docked using icon buttons at the top right corner of each Panel. The **Float Window** control ▢ will undock any panel which is useful when you want assign the network panel as much screen space as possible. To dock the window again, click the **Dock Window** icon 📌 . Clicking the **Hide Panel** icon ✖ will hide the panel; this can be shown again by choosing **View** → **Show** and selecting the relevant panel.

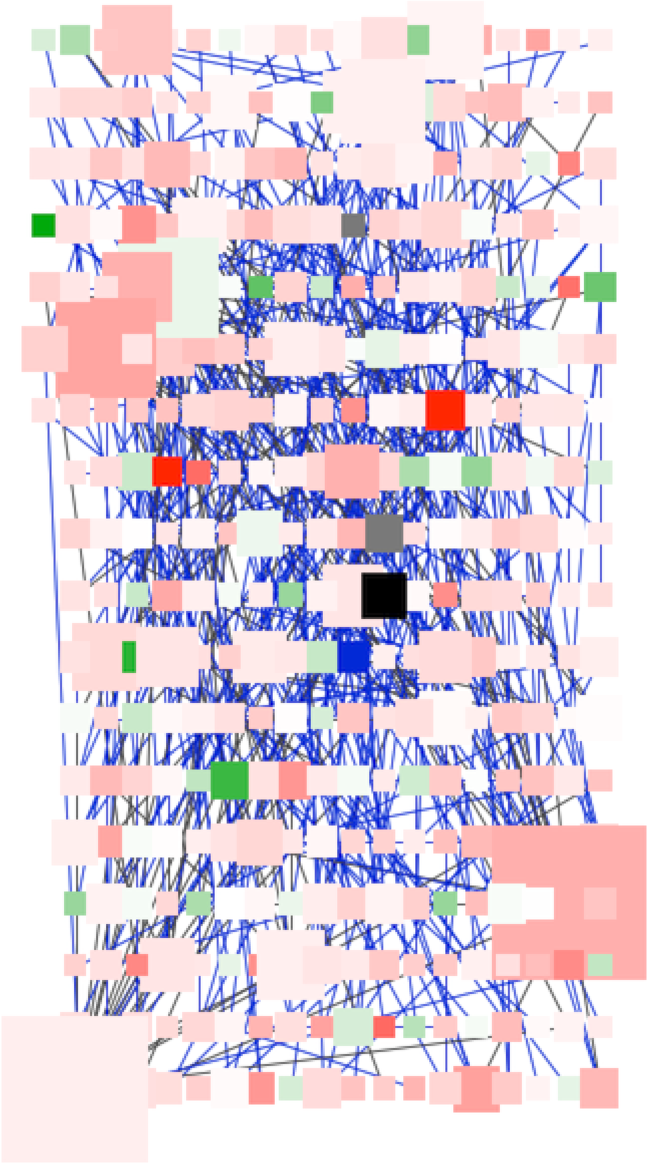
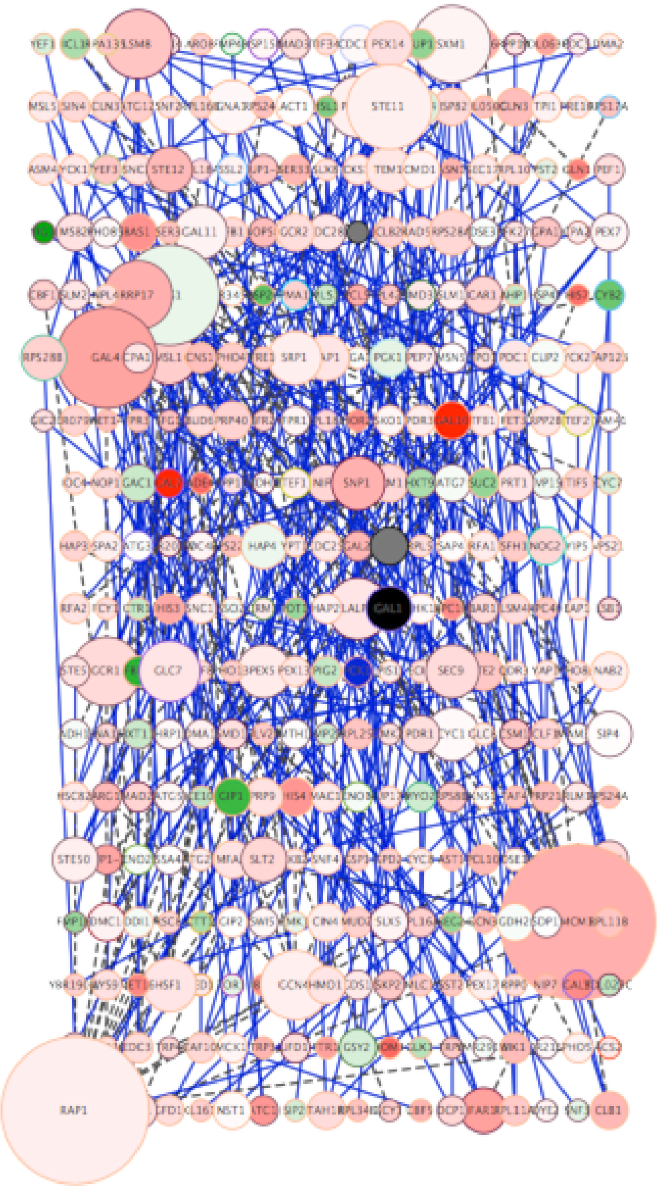
Rendering Engine

What is Level of Detail (LOD)?

Cytoscape 3.0 retains the rendering engine found in version 2.8. It is to be able to display large networks (> 10,000 nodes), yet retain interactive speed. To accomplish this goal, a technique involving **level of detail (LOD)** is being used. Based on the number of objects (nodes and edges) being rendered, an appropriate **level of detail** is chosen. For example, by default, node labels (if present) are only rendered when less than 200 nodes are visible because drawing text is a relatively expensive operation. This can create some unusual behavior. If the screen currently contains 198 nodes, node labels will be displayed. If you pan across the network, such that now 201 nodes are displayed, the node labels will disappear. As another example, if the sum of rendered edges and rendered nodes is greater than or equal to a default value of 4000, a very coarse level of detail is chosen, where edges are always straight lines, nodes are always rectangles, and no anti-aliasing is done. The default values used to determine these thresholds can be changed by setting properties under **Edit | Preferences | Properties...**.

Low LOD vs High LOD

Levels of Detail

NETWORK WITH LOW LOD	NETWORK WITH HIGH LOD
	

With low LOD values, all nodes are displayed as square and anti-aliasing is turned off. With high LOD values, anti-aliasing is turned on and nodes are displayed as actual shape user specified in the Style.

Parameters for Controlling LOD

NOTE: *The greater these thresholds become, the slower performance will become.* If you work with small networks (a few hundred nodes), this shouldn't be a problem, but for large networks it will produce noticeable slowing. The various thresholds are described below.

LOD Thresholds

PARAMETER	DESCRIPTION
	If the sum of *rendered* nodes and *rendered* edges equals to or exceeds this number, a very coarse level of detail will be chosen

RENDER.COARSEDETAILTHRESHOLD	exceeds this number, a very coarse level of detail will be chosen and all other detail parameters will be ignored. If the total number of nodes and edges is below this threshold, anti-alias will be turned on; this value defaults to 4000.
RENDER.NODEBORDERTHRESHOLD	If the number of *rendered* nodes equals to or exceeds this number, node borders will not be rendered; this value defaults to 400.
RENDER.NODELABELTHRESHOLD	If the number of *rendered* nodes equals to or exceeds this number, node labels will not be rendered; this value defaults to 200.
RENDER.EDGEARROWTHRESHOLD	If the number of *rendered* edges equals to or exceeds this number, edge arrows will not be rendered; this value defaults to 600.
RENDER.EDGELABELTHRESHOLD	If the number of *rendered* edges equals to or exceeds this number, edge labels will not be rendered; this value defaults to 200.

When printing networks or exporting to formats such as PostScript, the highest level of detail is always chosen, regardless of what is currently being displayed on the screen.

Force to Display Detail

If you want to display every detail of the network regardless of LOD values, you can toggle to full details mode by **View | Show Graphics Details (or CTR+SHIFT+F on Windows/Linux, Command+SHIFT+F for Mac)**. This option forces the display of all graphics details. If the network is large, this option slows down rendering speed. To hide details, select the menu item again (**View | Hide Graphics Details**).

Publish Your Data

Publish Your Visualizations

When you finish your data analysis and visualization, you need to publish your data to share the results. Cytoscape has several options to do it, with most options suitable for Cytoscape users and other options suitable for programmers wanting to create unusual or complex network viewers. They're further explained below.

- A session file
- A static image
- An interactive web application
  - CyNetShare
  - Full web application

- Simple network view (for web application developers)

As a Session File

The easiest way to share your results with others is simply saving everything as a session file (which is a zipped session archive). You can save your current session by clicking ***Save Session*** icon. You can save to a thumb drive, a shared file system, or even a cloud drive directory such as Dropbox – if you save to a shared drive, beware not to have two people work on the same session file with Cytoscape at the same time, as unpredictable results may occur.

As a Static Image

Cytoscape can generate publication-quality images from network views. By selecting ***File | Export | Network View as Graphics...***, you can export the current network view into the following formats:

- JPG
- PNG
- PS (Post Script)
- SVG
- PDF

We recommend using **PDF** for publications because it is a standard vector graphics format, and it is easy to edit in other applications such as Adobe Illustrator.

Known Issues

For PDF export, there is an option to ***Export Texts as Fonts***. **This option does not work for two-byte characters such as Chinese or Japanese**. To avoid corrupted texts in the exported images, please uncheck this option when you publish networks including those non-English characters.

As an Interactive Web Application (New in 3.2.0)

The Web is an excellent platform for data sharing and collaboration, and Cytoscape provides a number of ways to publish your network on the web, with each choice representing tradeoffs between ease, simplicity, and customization options. All solutions leverage the cytoscape.js drawing library, and so enable not only viewing but also Cytoscape-style interactive browsing of networks and attributes.

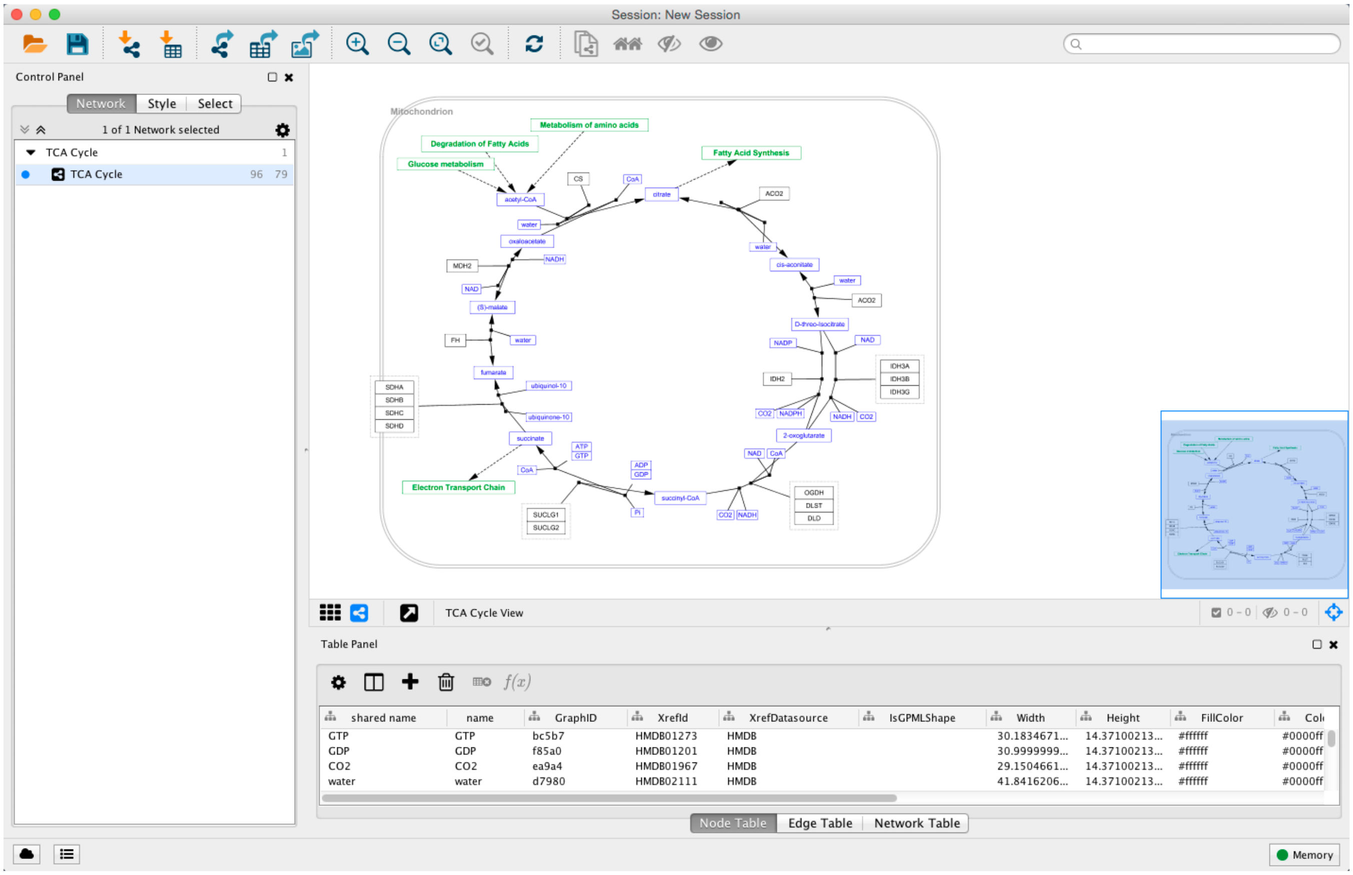
The simplest choice is CyNetShare, where you save your network (and optionally a style) on a public file system, then interactively view the network in a browser. Like Google Maps, you can generate and publish a URL that allows collaborators to also view your network.

Alternatively, Cytoscape can generate an entire web site showing a single page containing the viewer with your network pre-loaded. You can load this directly onto your own web server to become part of your web site.

Finally, Cytoscape can generate the skeleton of a web site for further customization by JavaScript programmers.

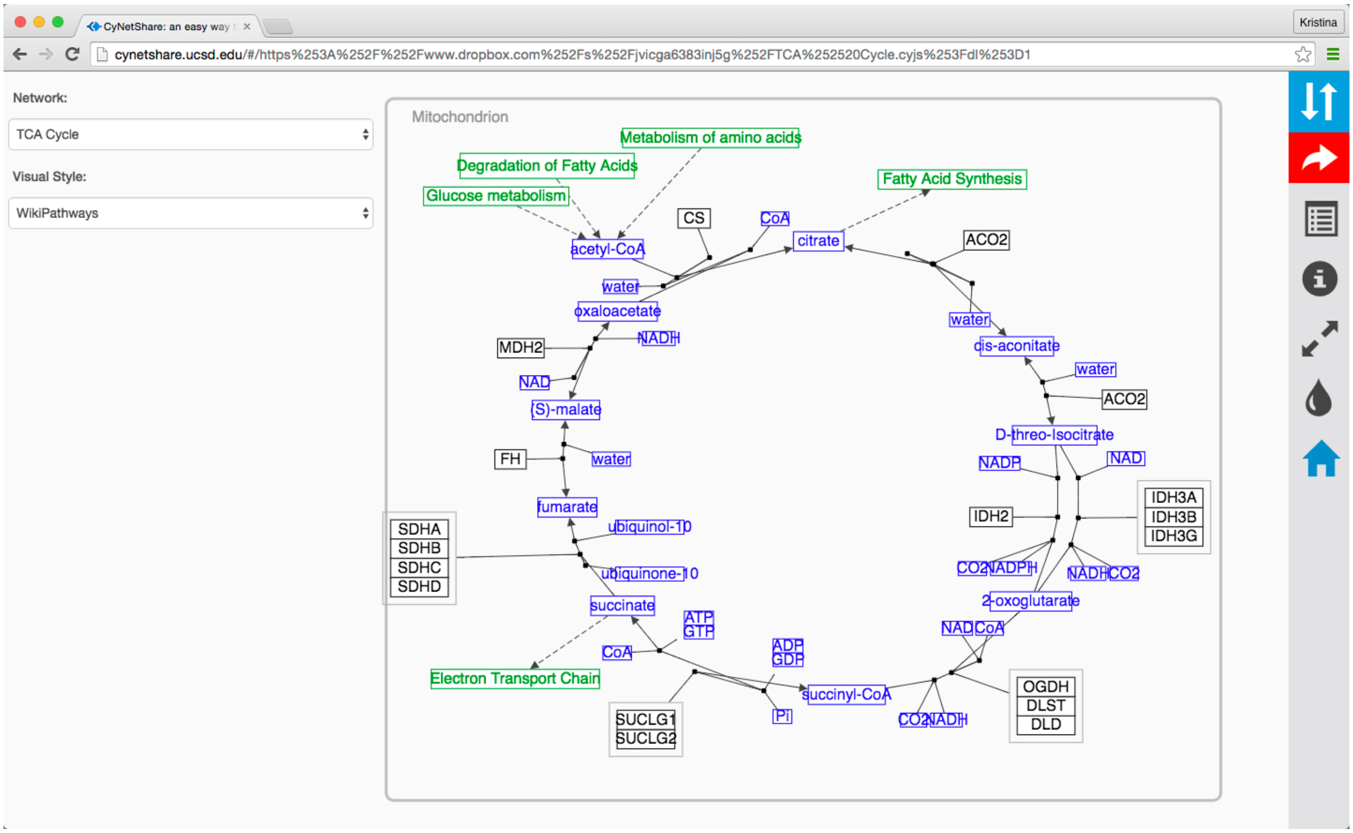
These features are available as Export menu items under the File menu, and are described in sections below.

For example, here is a network in Cytoscape:



shared name	name	GraphID	XrefId	XrefDatasource	IsGPMLShape	Width	Height	FillColor	Col
GTP	GTP	bc5b7	HMDB01273	HMDB		30.1834671...	14.37100213...	#ffffff	#0000ff
GDP	GDP	f85a0	HMDB01201	HMDB		30.9999999...	14.37100213...	#ffffff	#0000ff
CO2	CO2	ea9a4	HMDB01967	HMDB		29.1504661...	14.37100213...	#ffffff	#0000ff
water	water	d7980	HMDB02111	HMDB		41.8416206...	14.37100213...	#ffffff	#0000ff

Here is the same network as an interactive web visualization:



CyNetShare: an easy way

cynetshare.ucsd.edu/#/https%253A%252F%252Fwww.dropbox.com%252Fs%252Fjvicga6383inj5g%252FTCA%252520Cycle.cyjs%253Fdl%253D1

Network: TCA Cycle

Visual Style: WikiPathways

Note that web browsers can render small networks (e.g., 1000 nodes) quickly and effectively, but attempting to render large ones (e.g., 5000 nodes) will take a very long time.

A word about exporting styles styles to interactive web applications: Our web applications are based on the cytoscape.js display library, which renders a subset of Cytoscape styles. For more information, see the **Export Styles to Cytoscape.js** section below.

Sharing via CyNetShare

CyNetShare is a browser-based web application that renders JSON-formatted networks and attributes saved in public directories. Optionally, you can specify visual styles that the web application will use to draw your network as it appears in Cytoscape. CyNetShare is similar to Google Maps in that once you have loaded your network and have tweaked its appearance to suit, you can have CyNetShare generate a new URL that you can e-mail or post as a link on your own web site. That URL will bring up CyNetShare preloaded with your network for anyone to see.

To use CyNetShare:

1. Select ***File | Export | Network and View...*** to export your network to a public directory. Choose the *Cytoscape.js JSON (*.cyjs)* export file format.
2. Optionally, select ***File | Export | Style...*** to export your style settings. Choose the *Style for cytoscape.js (*.json)* export file format.

3. Use your public directory system to determine direct URLs for the files you exported.
4. Start CyNetShare
5. Enter the network's URL as the Graph URL.
6. Optionally, enter the style's URL.
7. Click the ***Visualize*** button.

The CyNetShare User Guide is provided on the CyNetShare web page:

- CyNetShare

Note that if you specify a style URL, the style is added to the list of styles available in CyNetShare's Visual Style dropdown, and you can apply the style by selecting it in the list. CyNetShare's initial display uses the visual style named "default" – use the Visual Style dropdown to choose the style in effect when Cytoscape generated the .cyjs and .json files.

Note that the mechanics of generating a URL for a file in a public directory system is a fast moving topic. Until recently, systems like Dropbox (and others) allowed users to create a URL that resolved directly to a file – a "direct" URL would be appropriate for use with CyNetShare. As of this writing, some public directory systems (e.g., Dropbox) generate "shareable" URLs instead, which resolve to a web page that allows file download – a "shareable" URL makes CyNetShare hang. Systems that offer "shareable" URLs may offer "direct" URLs as part of their premium (or Pro) package. To tell if your public directory system generates a "direct" URL, have it generate a URL for a file, then paste the URL into the address field of a browser and observe whether the browser displays the file itself (good!) or a download page for the file (bad!).

Hint: if Dropbox generates a "shareable" link that looks like `https://www.dropbox.com/s/w5e7towcchuvdeu/cynetworm.cyjs?dl=0`, you may be able to create a "direct" link by changing the `dl=0` to `dl=1`: `https://www.dropbox.com/s/w5e7towcchuvdeu/cynetworm.cyjs?dl=1`.

A simple strategy for always getting a "direct" URL is to store your file in a public directory served up by a web server, if you have access to one – a URL served by a web server might appear as: http://myserver.com/~mypublicdir/netstyle.json.

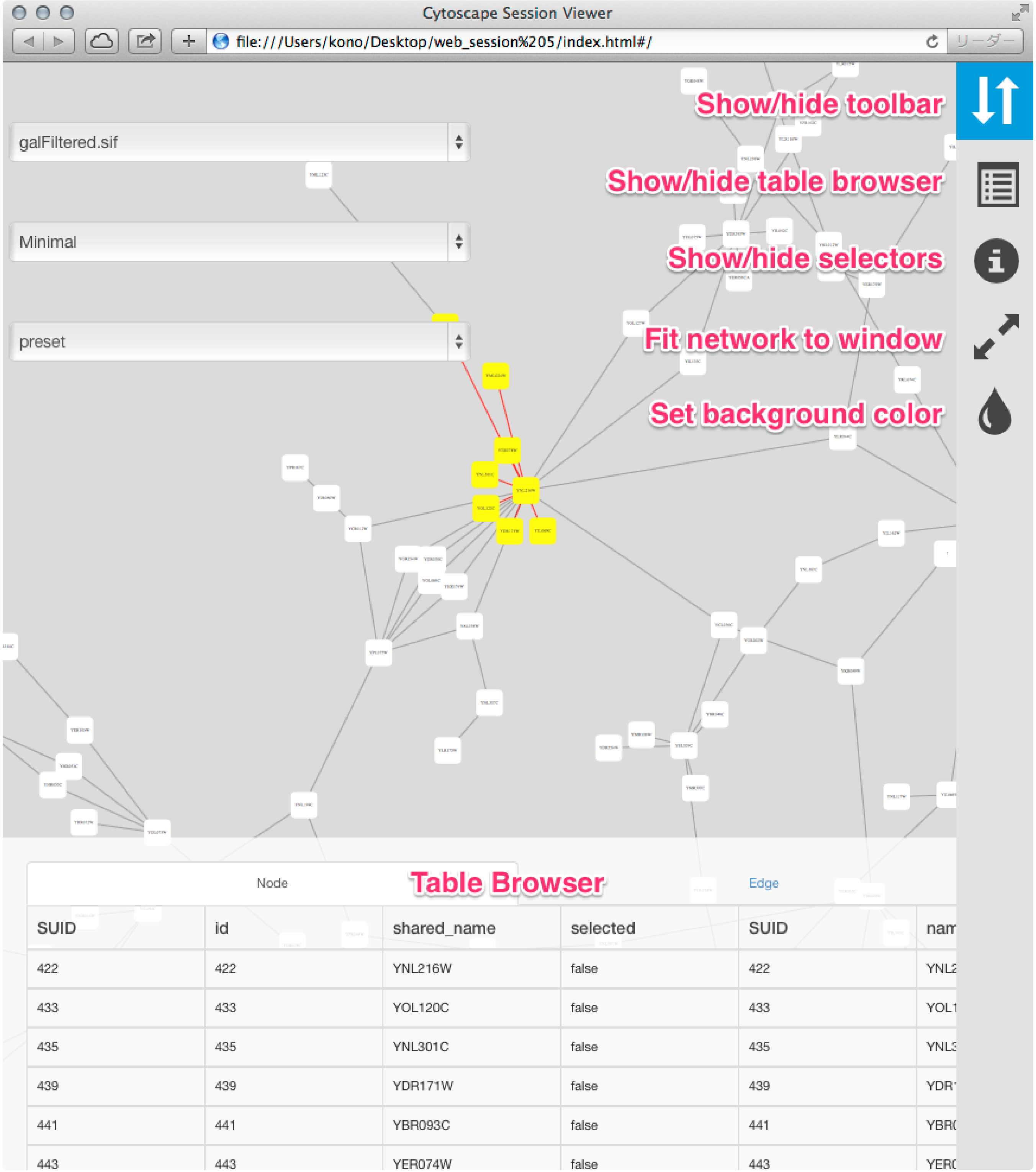
Alternately, you can use Gist to create a shareable document having a "direct" URL. To try this:

1. Use Cytoscape to generate your network as a .cyjs file on your local disk
2. Use an editor to open the file and copy its contents to the clipboard
3. Use a web browser to surf to Gist

4. Paste the contents into a Gist document
5. Select ***Create public Gist***
6. Select ***Raw*** to place the "direct" URL into your browser's address field
7. Use that URL with CyNetShare

Generating a Full Web Application

The full page export option is designed for users who want to publish their network as a complete single-page application. Cytoscape creates a zip archive containing a complete JavaScript-based web application that works as a basic viewer (like CyNetShare) for Cytoscape-generated network visualizations. You can unzip the archive onto a web server (or your PC) and view the network via a web browser on PCs and tablets.



galFiltered.sif

Minimal

preset

SUID	id	shared_name	selected	SUID	nam
422	422	YNL216W	false	422	YNL2
433	433	YOL120C	false	433	YOL1
435	435	YNL301C	false	435	YNL3
439	439	YDR171W	false	439	YDR1
441	441	YBR093C	false	441	YBR0
443	443	YER074W	false	443	YER0

To generate an entire web page as a zip archive, select ***File | Export | Network View(s) as Web Page ...***.

To view the web page, unzip the archive into a folder on your PC or web server. The folder will contain an ***index.html*** file, the network data, and other files. You can open the ***index.html*** file in your browser (usually from your browser's ***File | Open*** menu item.)

Depending on your browser's security settings, you may not be able to open the ***index.html*** file directly if it is stored on your PC – you may need to start a web server on your PC. An easy way to set up a local web server is by running the Python simple HTTP server. If you have Python installed on your machine, just go into the web archive folder and type:

```
python -m SimpleHTTPServer 8000
```

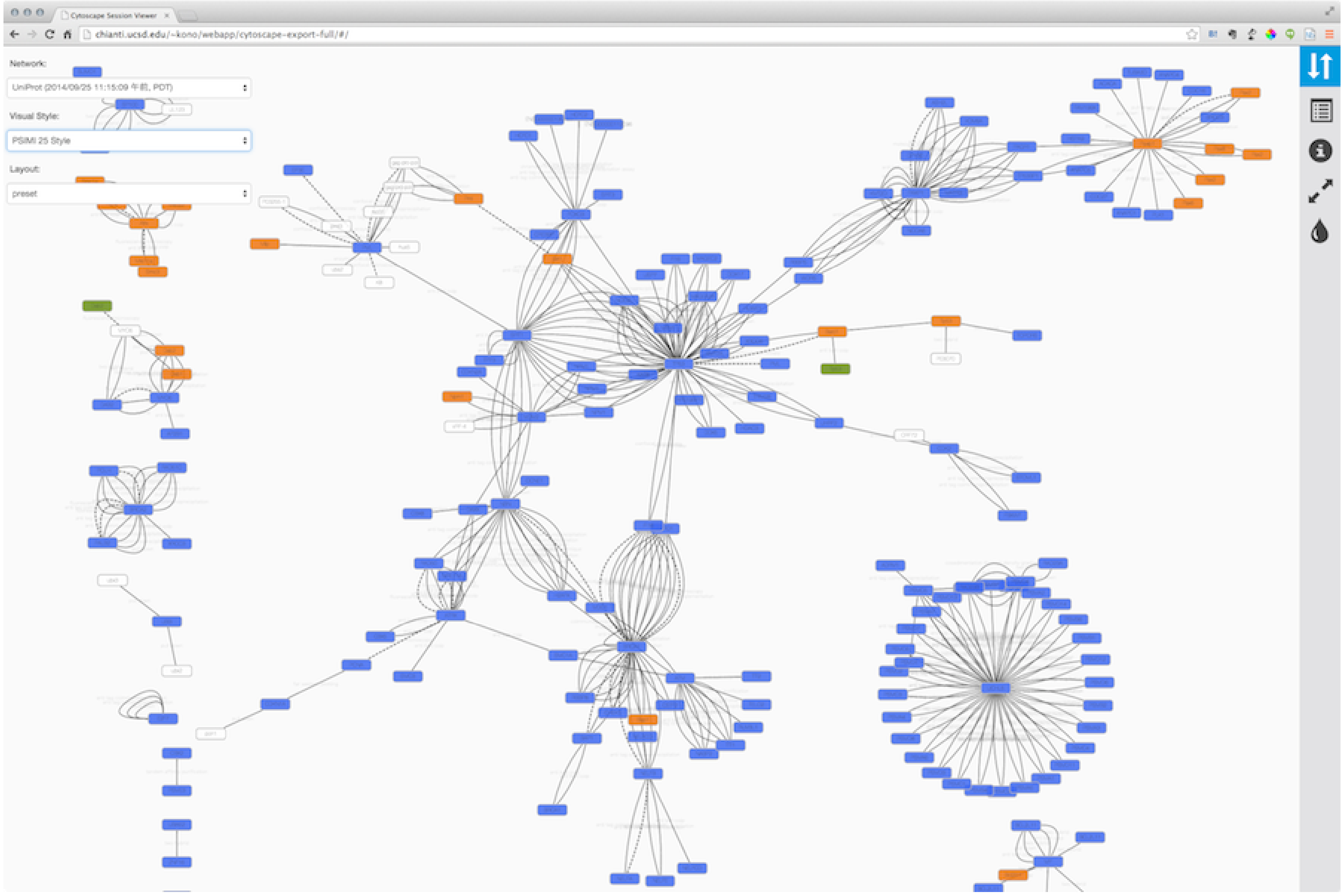
and use your browser to open:

```
http://localhost:8000/
```

Testing the archive on your PC will serve as an easy test of the web page, but to publish it to collaborators, you should unzip your archive onto a web server.

Here is an example exported file from Cytoscape:

- Example full export deployed to web server
- Archive file



Network:
UniProt (2014/09/25 11:15:09 午前, PDT)

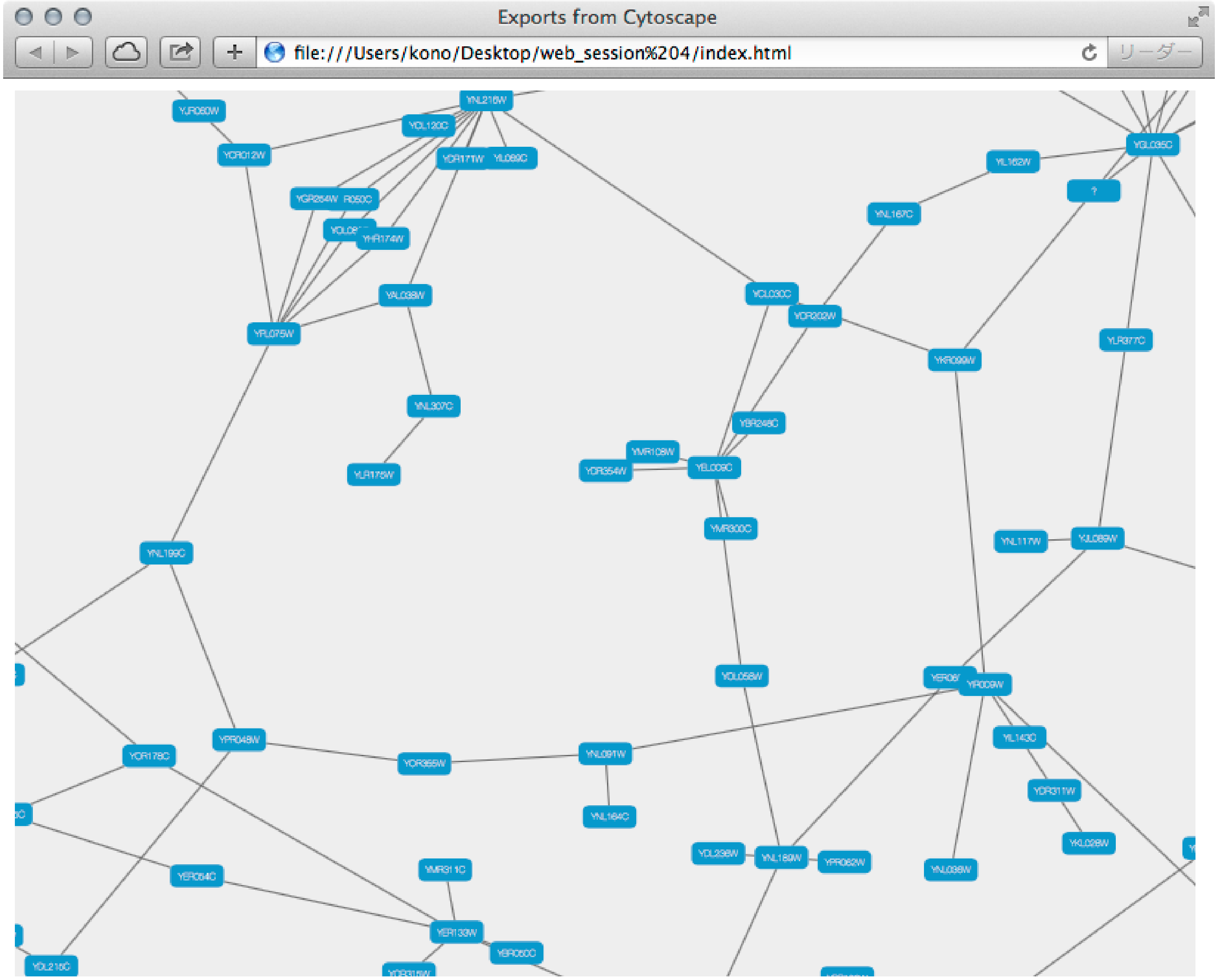
Visual Style:
PSIMI 25 Style

Layout:
preset

Note that because Cytoscape uses the latest HTML5-based web technologies, it cannot support older or non-conformant web browsers such as Internet Explorer. We strongly recommend using the latest version of modern web browsers such as Google Chrome, Mozilla Firefox, or Apple Safari.

Generating a Simple Network View (For Web Application Developers)

The Simple Network View export option is designed for users who want to publish their data as a complete single-page application, but are interested in customizing the web viewer application themselves. Cytoscape creates a zip archive containing a partial JavaScript-based web application based on the cytoscape.js library and including simple “boilerplate” code and the current network view. The user can create a custom viewer by customizing this code.

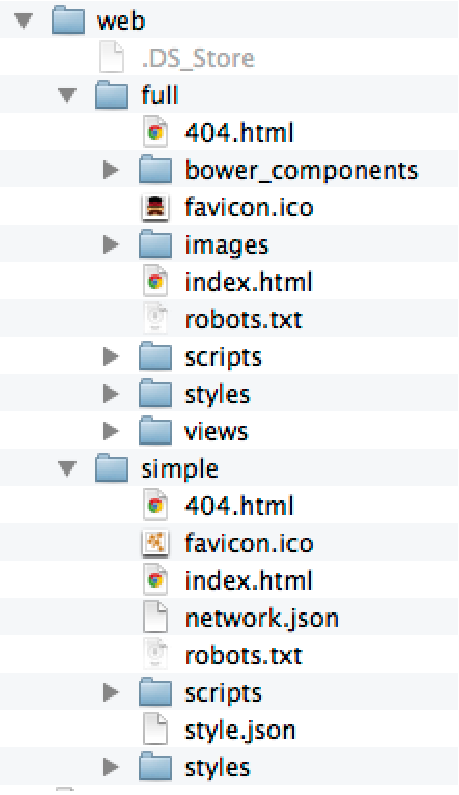


To generate an entire web page as a zip archive, select ***File | Export | Network View(s) as Web Page ...***, and choose the *Simple viewer for current network only* format.

For instructions on testing the customized web application, see *Generating a Full Web Application* above.

Customize Export Template (For Web Application Developers)

The code generated by Cytoscape for the Full Web Application and the Simple Network View web applications is minimalistic. While you can directly modify this code yourself to create your own page design or add new features, the modifications will apply to a single exported network. If you are a web application developer, you can change the application code generated for **all** exports by editing HTML5 template code resource files in your ***~/CytoscapeConfiguration/web*** directory:



```
▼ web
    .DS_Store
  ▼ full
      404.html
    ▶ bower_components
      favicon.ico
    ▶ images
      index.html
      robots.txt
    ▶ scripts
    ▶ styles
    ▶ views
  ▼ simple
      404.html
      favicon.ico
      index.html
      network.json
      robots.txt
    ▶ scripts
      style.json
    ▶ styles
```

In this folder, you can find ***full*** and ***simple*** sub directories corresponding to Full Web Application and the Simple Network View described above.

Requirements

To build these project, you need the following tools installed:

- Node.js
- gulp
- grunt

Full Export Template

This is an AngularJS based web application built with grunt (at least for now – we have plans to migrate to gulp). Source code and more documentations are available here:

* https://github.com/idekerlab/cyjs-full-export

To build the project into *dist* directory, type:

```
grunt
```

Simple Export Template

This template is generated by a simple gulp project. The source code is available here:

- https://github.com/idekerlab/cyjs-export-simple

To build the project into *dist* directory, type:

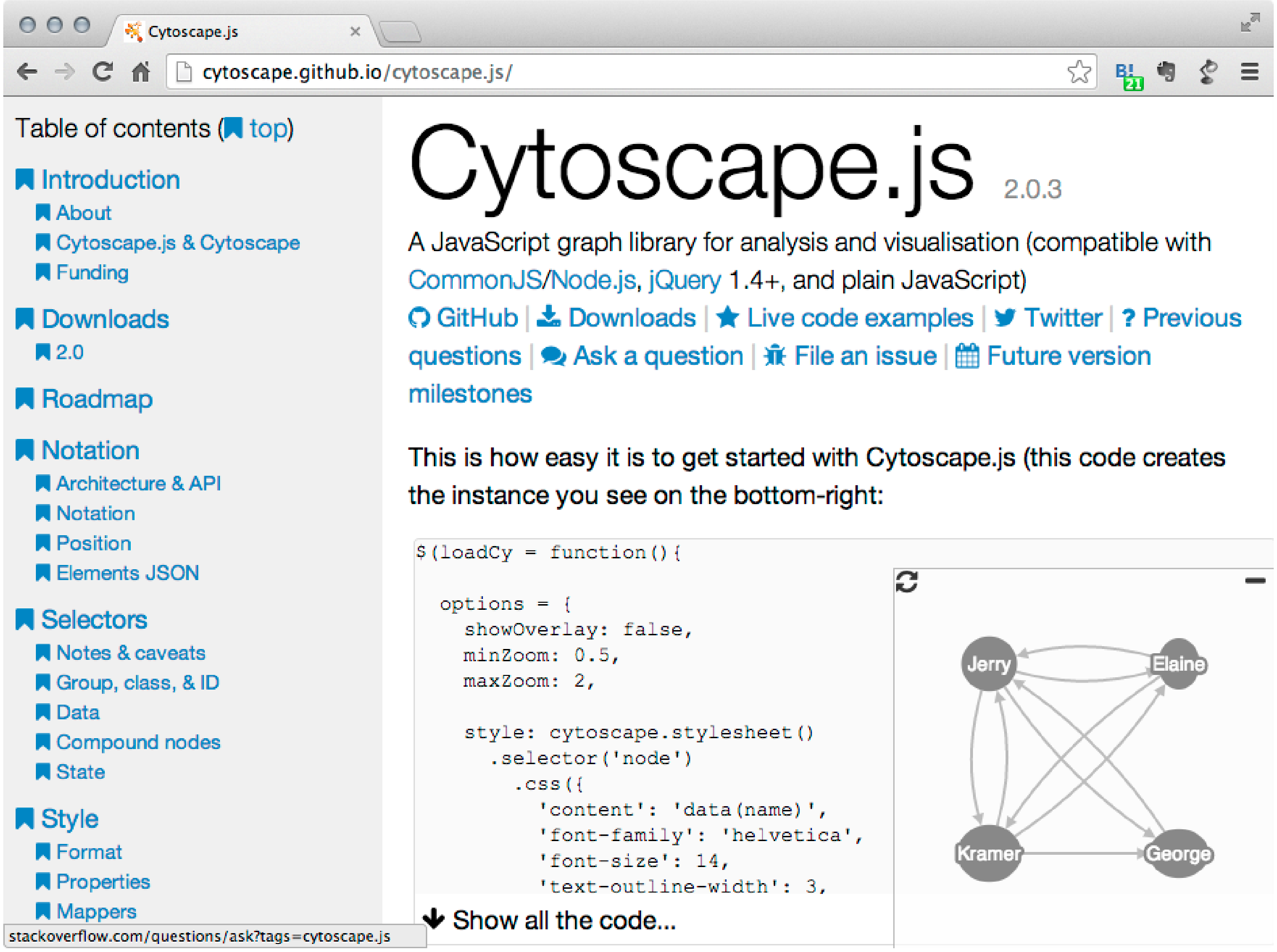
```
gulp
```

Use Your Custom Templates for Export

Once you have your own builds, you can deploy your templates by replacing the contents of *full* and *simple* with your own builds.

Cytoscape.js and Cytoscape

What is Cytoscape.js?



Cytoscape.js 2.0.3

A JavaScript graph library for analysis and visualisation (compatible with CommonJS/Node.js, jQuery 1.4+, and plain JavaScript)

GitHub | Downloads | Live code examples | Twitter | ? Previous questions | Ask a question | File an issue | Future version milestones

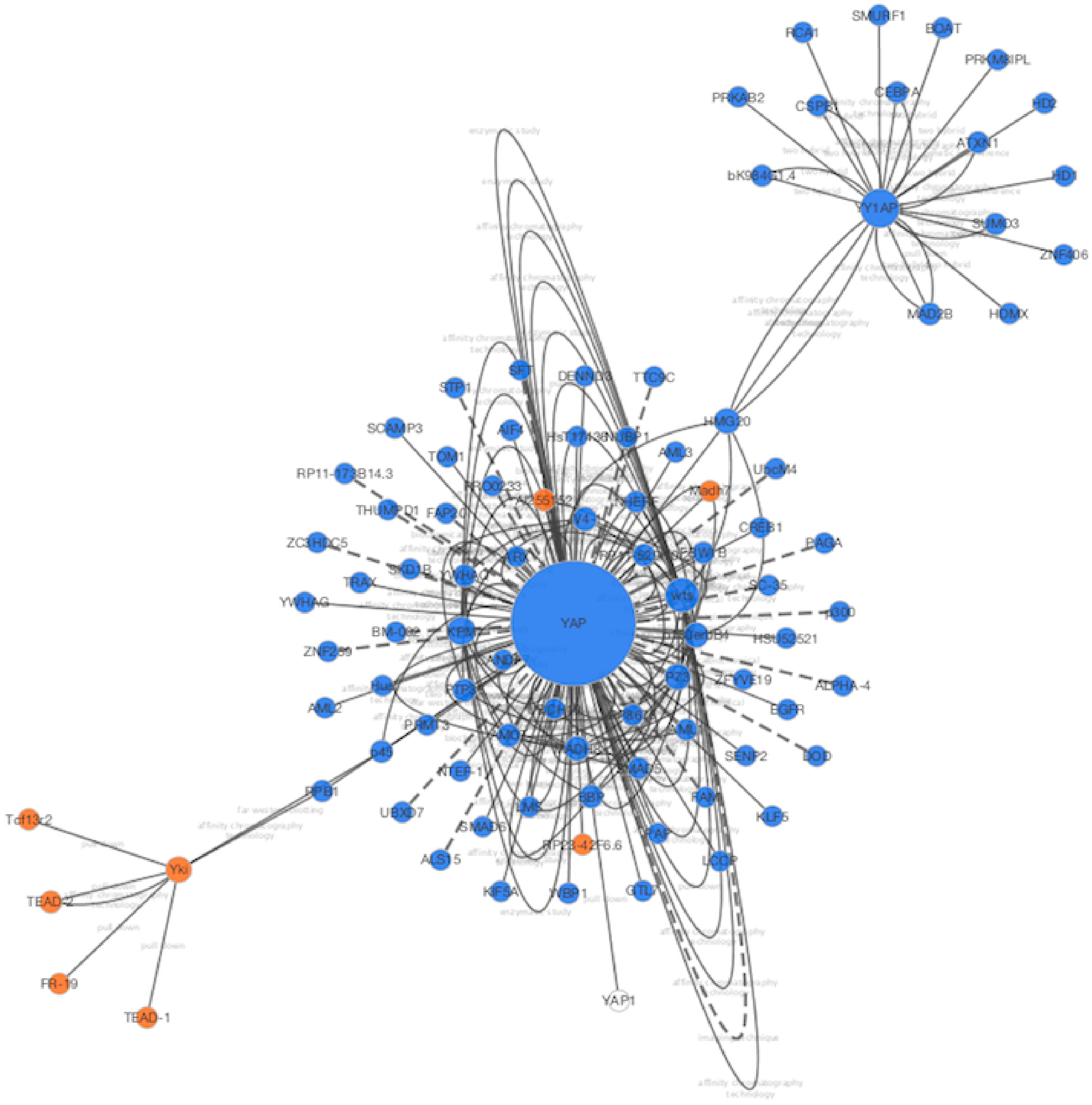
This is how easy it is to get started with Cytoscape.js (this code creates the instance you see on the bottom-right:

```
$(loadCy = function(){

  options = {
    showOverlay: false,
    minZoom: 0.5,
    maxZoom: 2,

    style: cytoscape.stylesheet()
      .selector('node')
        .css({
          'content': 'data(name)',
          'font-family': 'helvetica',
          'font-size': 14,
          'text-outline-width': 3,
```

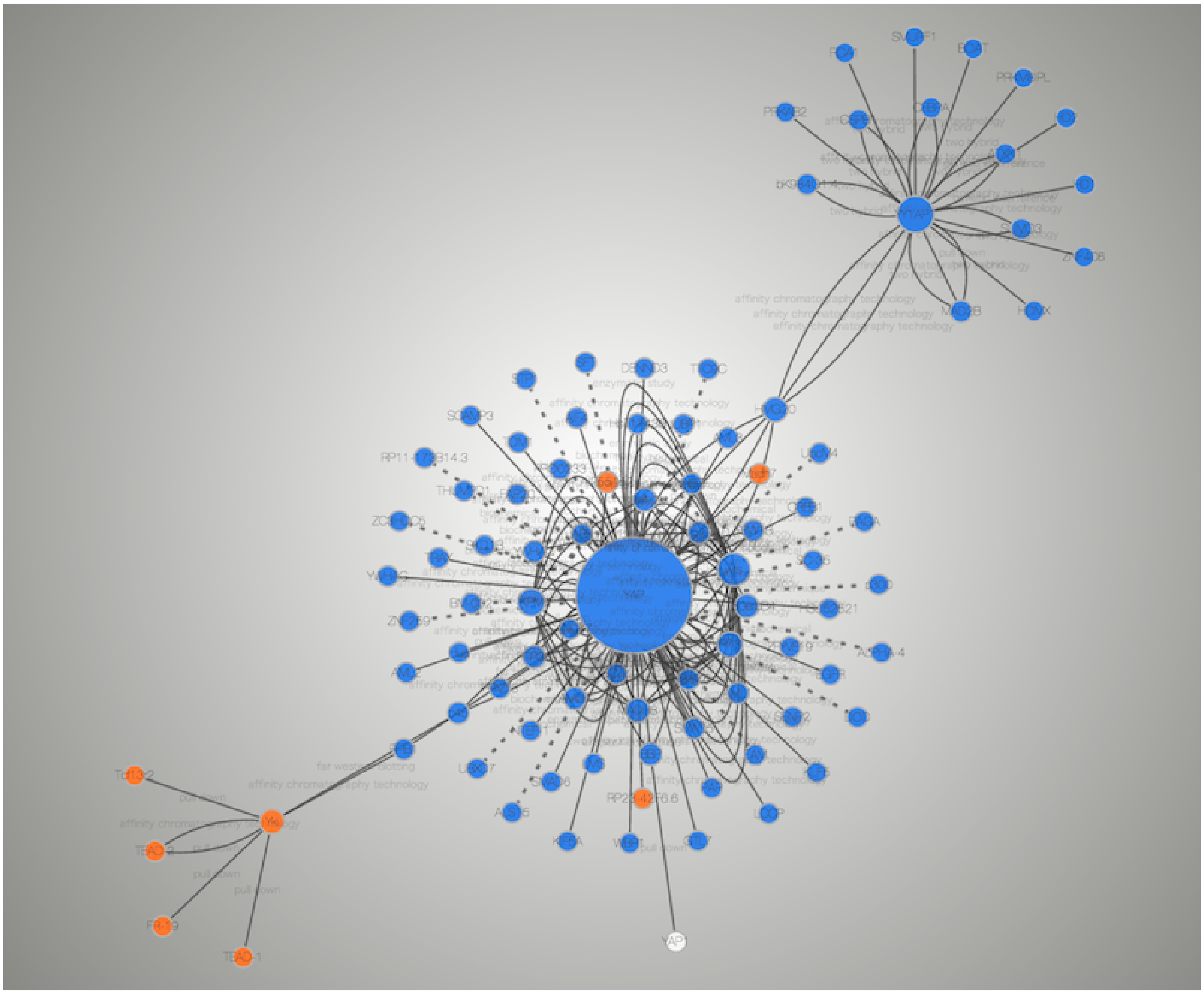
Cytoscape.js is a **JavaScript library** for interactive network visualization. It is a building block for web applications and is **NOT** a complete web application. If you have network data sets and want to share visualizations created with Cytoscape, you can build your own website using Cytoscape.js and this new **Export to Cytoscape.js** feature.

Examples

- A network visualized with Cytoscape 3.1.0



- Same network exported to Cytoscape.js (Rendered with Google Chrome and Cytoscape.js 2.0.3)



- Interactive example (galFiltered.sif rendered with Cytoscape.js 2.0.3)

Differences and Shared Concepts

Although Cytoscape and Cytoscape.js are two completely independent software packages, they are sharing higher level concepts. The following is the list of similarities and differences between the two:

Cytoscape

- **Desktop application** for network visualization written in Java programming language
- Needs desktop or laptop computers to run
- Users have to install Java runtime
- High performance application for large scale network analysis and visualization
- Expandable by Apps
- Use **Styles** to map data to network properties, such as node color, edge width, node shape, etc.

Cytoscape.js

- A **JavaScript library** for network visualization, **NOT** a complete web application nor mobile app
- Runs on most of modern web browsers, including tablets and smart phones
- No plugins are required to run. Modern web browser is the only requirement
- Need to write code to set up your web site or web application
- Expandable by Extensions
- Use **CSS-based Styles** to map data to network properties

In a long term, Cytoscape and Cytoscape.js will be more integrated, and as the first step Cytoscape now supports reading and writing Cytoscape.js network/table JSON files. In addition, Cytoscape can convert *Styles* to Cytoscape.js Style object.

Data Exchange between Cytoscape and Cytoscape.js

Since Cytoscape.js is a JavaScript library, its basic data exchange format is JSON (JavaScript Object Notation). All of these import/export functions are part of standard Cytoscape user interface, and you can read/write Cytoscape.js JSON files just like other standard files such as SIF.

Export Network and Table to Cytoscape.js

Cytoscape.js stores network data (graph) and its data table in the same object. Cytoscape writes such data complex as JSON, i.e., both network and data tables will be exported as a single JSON file. You can select a network and export it from **File | Export | Network**.

Cytoscape only supports one of the Cytoscape.js supported JSON formats, which is:

```
{
    elements:{
        nodes:[],
        edges:[]
    }
}
```

SUID will be used as unique identifier for nodes and edges in the JSON. For more information about this format, please visit Cytoscape.js web site.

Important Note about Data Compatibility

Cytoscape creates JSON file directly from data table and tries to extract as much data as possible from the original table. However, since table column names will be directly converted into JavaScript variable names, invalid characters will be replaced by underscore (_):

- Original Data Table Column Names:

```
Gene Name
KEGG.pathway
```

- The Names above will be replaced to:

```
Gene_Name
KEGG_pathway
```

You should be careful when you plan to use this feature for data roundtrip: from Cytoscape to Cytoscape.js back to Cytoscape. For such use cases, **we strongly recommend to use JavaScript-safe characters only in your table column names**. Naming your columns only with alphanumeric characters and underscore (_) is the best practice. (For actual data entries, all characters are allowed. This restriction is only for table column names.)

Export Styles to Cytoscape.js

Cytoscape and Cytoscape.js are sharing a concept called **Style**. This is a collection of mappings from data point to network property. oCytoscape can export its Styles into CSS-based Cytoscape.js JSON.

You can export *all Styles into one JSON file* from **File | Export | Style** and select Cytoscape.js JSON as its format.

Limitations

Cytoscape.js does not support all of Cytoscape Network Properties. Those properties will be ignored or simplified when you export to JSON Style file.

Currently, the following Visual Properties will not be exported to Cytoscape.js JSON:

- Custom Graphics and its positions
- Edge Bends

- Tooltips
- Node Label Width
- Node Border Line Type
- Arrow Colors (they are always same as edge color)

Cytoscape.js and Cytoscape Compatibility

Cytoscape's network rendering system is designed for desktop use, while the browser-based renderer incorporates web technologies (e.g., cytoscape.js and Cascaded Style Sheets). As a result, most but not all networks will render the same in the browser as in Cytoscape. Cytoscape visual styles not supported in the web browser are ignored. A complete compatibility list is available here.

Import Cytoscape.js data into Cytoscape

Cytoscape.js network JSON files can be loaded from standard Cytoscape file menu: **File | Import | Network ....** Both File and URL are supported.

Build Your Own Web Application with Cytoscape.js

Although Cytoscape can export networks, tables, and Style as Cytoscape.js-compatible JSON, users have to write some JavaScript code to visualize the data files with Cytoscape.js. Details of web application development with Cytoscape.js is beyond the scope of this document. If you need examples and tutorials about web application development with Cytoscape.js, please visit the following web site:

- https://github.com/cytoscape/cyjs-sample/wiki

Questions?

If you have questions and comments about web application development with Cytoscape and Cytoscape.js, please send yours to our mailing list.

Programmatic Access to Cytoscape Features (Scripting)

Programmatic Access to Cytoscape Features

In this chapter, you will learn how to use Cytoscape from the command line and scripts. These features replace the ***Scripting*** module in past versions of Cytoscape.

Topics

- *Commands*
- RESTful API
  - Command REST API
  - cyREST

Background

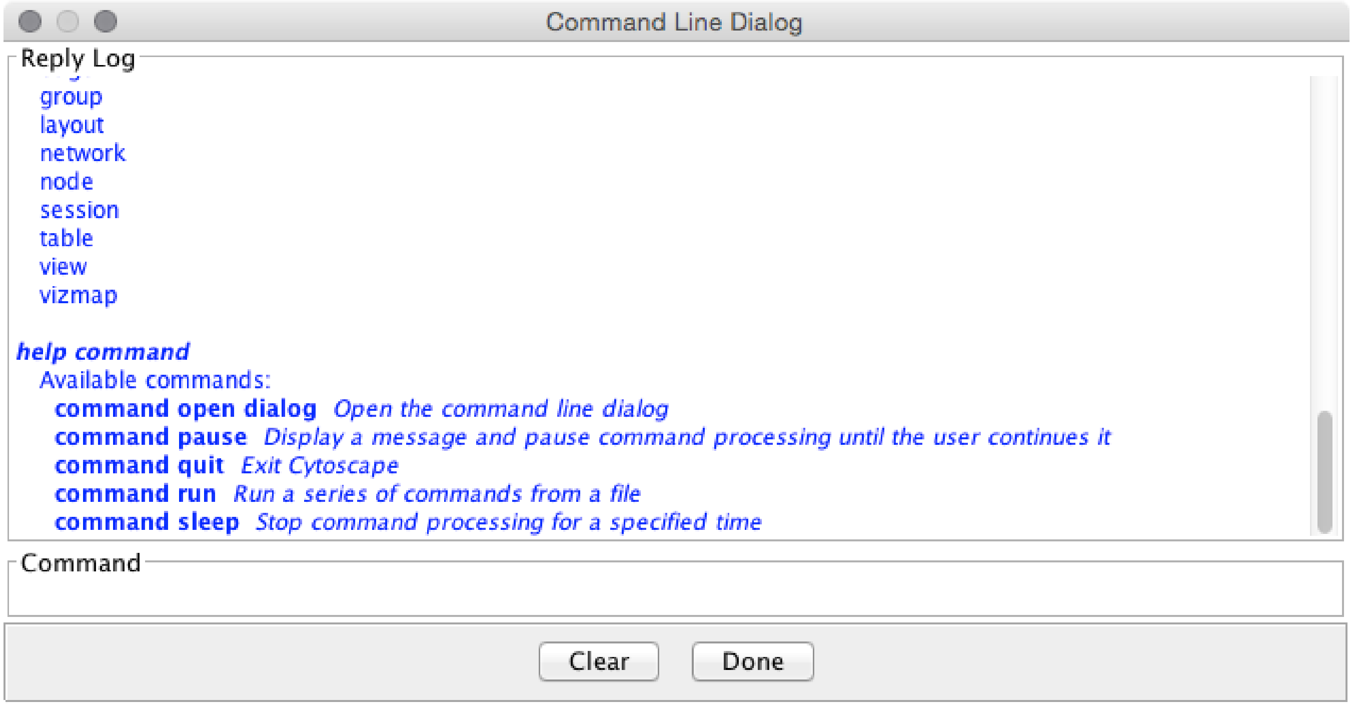
Cytoscape's intuitive graphical user interface is useful for *interactive* network data integration, analysis, and visualization. It provides great features for exploratory data analysis, but what happens if you have hundreds of data files or need to ask someone to execute your data analysis workflows? It is virtually impossible to apply the same operations to hundreds of networks manually using a GUI. More importantly, although you can save your ***results*** as session files, you cannot save your ***workflows*** if you perform your data analysis with point-and-click GUI operations. Cytoscape has several options that support scripting and automating your workflows: Commands and RESTful API.

The Command feature allows you to script a number of Cytoscape commands and menu items, and commands can have parameter values that would normally be provided by a user via Cytoscape dialog box. For example, *session open file="C:\myfile.cys"* loads a session from a file similarly to the **File | Open** menu item. You can create a command script file that Cytoscape can execute via the **Tools | Execute Command File** menu item or on the Cytoscape command line at startup.

The RESTful API feature allows you to access Cytoscape from a separate application, thereby orchestrating Cytoscape operations from that application. The application may be written in a general programming language (e.g., Python) that keeps its own data structures, performs complex flow control, or directly manipulates Cytoscape nodes, edges, attributes, and visual styles.

The Command feature is useful for executing sequences of commands, whereas the RESTful API feature is useful when Cytoscape is to be used as a service relative to an application.

Commands



```
   group
   layout
   network
   node
   session
   table
   view
   vizmap

help command
   Available commands:
     command open dialog  Open the command line dialog
     command pause  Display a message and pause command processing until the user continues it
     command quit  Exit Cytoscape
     command run  Run a series of commands from a file
     command sleep  Stop command processing for a specified time
```

Commands is the built-in Cytoscape feature to automate your workflow as simple script. You can learn more about this feature in this section:

- Command Tool

RESTful API

In some cases, you may need to use fully featured programming languages, such as Python, R, Ruby, or JavaScript to script your workflow. Such languages enable complex control and data structures, and Cytoscape would be called as a service. For such use cases, accessing Cytoscape via REST API is the right option.

Cytoscape offers two flavors of REST-style control: REST Commands and cyREST. REST Commands uses a REST interface to issue script commands. cyREST uses a REST interface to access the Cytoscape data model as a document via a formal API.

1. REST API for Commands

In addition to running Command scripts, Command module has REST API to enable command execution from another program.

By default, this feature is disabled. To enable the REST API server for Commands, please follow these steps:

1. Open a terminal session:

- PowerShell or Command (For windows)
- Terminal or iTerm2 (For Mac)
- Terminal (For Linux)

2. Start Cytoscape from command-line. You must specify a TCP/IP port number as a parameter – in this example, port 8888 will be opened for Command:

   - For Mac/Linux

     ```
     ./cytoscape.sh -R 8888
     ```

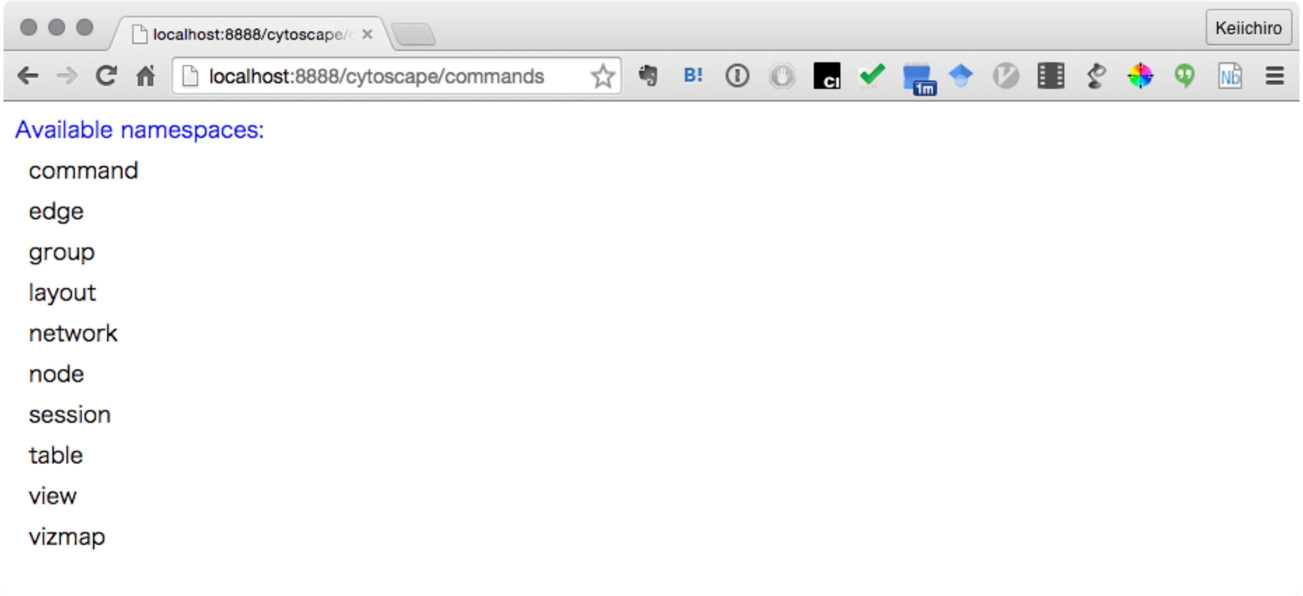
     For Windows

     ```
     ./cytoscape.bat -R 8888
     ```

3. To test the Command interface, open the following URL with your web browser:

   - http://localhost:8888/v1/commands

4. If you see list of available commands, you are ready to use Command API

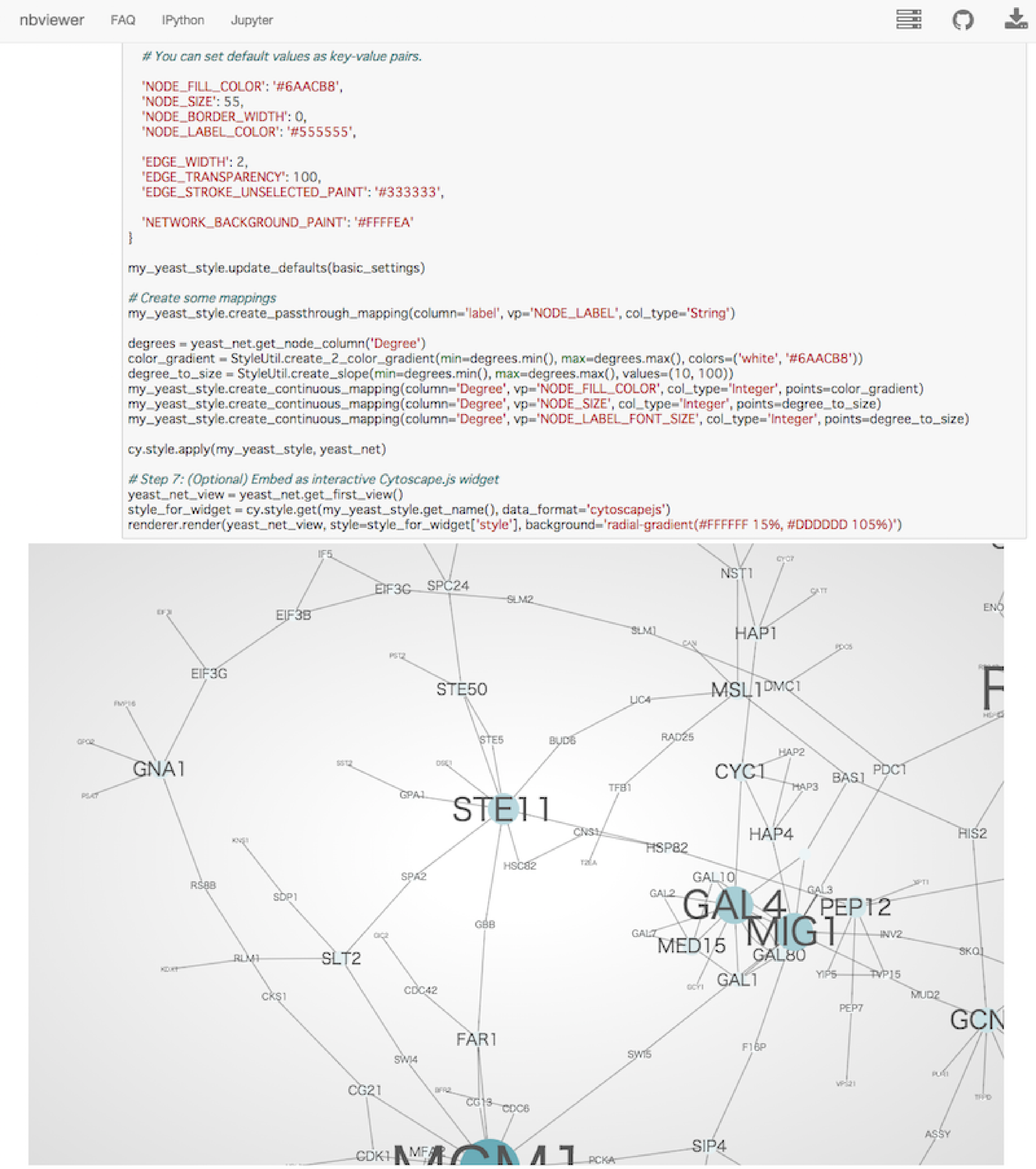

localhost:8888/cytoscape/ ×
Keiichiro
localhost:8888/cytoscape/commands
B!

Available namespaces:

command
edge
group
layout
network
node
session
table
view
vizmap

2. cyREST

cyREST is a language-agnostic, programmer-friendly RESTful API module for Cytoscape. If you want to build your own workflow with R, Python or other programming languages along with Cytoscape, this is the option for you. You can use popular tools, including IPython/Jupyter Notebook and RStudio as your orchestration tool for your data visualization workflow with Cytoscape.



```
    # You can set default values as key-value pairs.

    'NODE_FILL_COLOR': '#6AACB8',
    'NODE_SIZE': 55,
    'NODE_BORDER_WIDTH': 0,
    'NODE_LABEL_COLOR': '#555555',

    'EDGE_WIDTH': 2,
    'EDGE_TRANSPARENCY': 100,
    'EDGE_STROKE_UNSELECTED_PAINT': '#333333',

    'NETWORK_BACKGROUND_PAINT': '#FFFFEA'
}

my_yeast_style.update_defaults(basic_settings)

# Create some mappings
my_yeast_style.create_passthrough_mapping(column='label', vp='NODE_LABEL', col_type='String')

degrees = yeast_net.get_node_column('Degree')
color_gradient = StyleUtil.create_2_color_gradient(min=degrees.min(), max=degrees.max(), colors=('white', '#6AACB8'))
degree_to_size = StyleUtil.create_slope(min=degrees.min(), max=degrees.max(), values=(10, 100))
my_yeast_style.create_continuous_mapping(column='Degree', vp='NODE_FILL_COLOR', col_type='Integer', points=color_gradient)
my_yeast_style.create_continuous_mapping(column='Degree', vp='NODE_SIZE', col_type='Integer', points=degree_to_size)
my_yeast_style.create_continuous_mapping(column='Degree', vp='NODE_LABEL_FONT_SIZE', col_type='Integer', points=degree_to_size)

cy.style.apply(my_yeast_style, yeast_net)

# Step 7: (Optional) Embed as interactive Cytoscape.js widget
yeast_net_view = yeast_net.get_first_view()
style_for_widget = cy.style.get(my_yeast_style.get_name(), data_format='cytoscapejs')
renderer.render(yeast_net_view, style=style_for_widget['style'], background='radial-gradient(#FFFFFF 15%, #DDDDDD 105%)')
```

(Sample Jupyter Notebook written with cyREST and py2cytoscape)

Currently, cyREST is available as an App for Cytoscape 3.2.1 and later, and requires the Java 8 (or later) virtual machine. As of Cytoscape v3.3, cyREST is installed automatically with Cytoscape. Please visit the link below for more information.

- cyREST App Store page

Cytoscape Privacy Policy

We respect the privacy of all Cytoscape users, and we do not collect any information on Cytoscape users except in the situations listed below. In no case do we attempt to tie any of this information back to a user, nor do we give, share, sell, or transfer this information to any third party unless required by law. We use this information only in the aggregate to generate statistics to assist in securing continued funding for Cytoscape.

- On the Cytoscape download web page, we log the date, time, browser signature, and IP address to which we deliver Cytoscape.
- For a news feed fetched for display on the Cytoscape Welcome screen, we log the date and time the news was fetched, the browser signature, and the IP address for the workstation running Cytoscape.

This policy may change from time to time, and if it does, we will notify you via the Cytoscape Welcome screen news feed and via our normal mass notification media. We will also update this section of the user manual.

Note that some internal Cytoscape Apps and Apps available through the Cytoscape App Store connect with third party services via the Internet. Once an App links to such a service, you are subject to the privacy policy of that service.

A Python Book: Beginning Python, Advanced Python, and Python Exercises

Author:

Dave Kuhlman

Contact:

dkuhlman@davekuhlman.org

Address:

http://www.davekuhlman.org

Revision

1.3a

Date

December 15, 2013

Copyright

Abstract

This document is a self-learning document for a course in Python programming. This course contains (1) a part for beginners, (2) a discussion of several advanced topics that are of interest to Python programmers, and (3) a Python workbook with lots of exercises.

Contents

Preface

This book is a collection of materials that I've used when conducting Python training and also materials from my Web site that are intended for self-instruction.

You may prefer a machine readable copy of this book. You can find it in various formats here:

- HTML – http://www.davekuhlman.org/python_book_01.html
- PDF -- http://www.davekuhlman.org /python_book_01.pdf
- ODF/OpenOffice -- http://www.davekuhlman.org /python_book_01.odt

And, let me thank the students in my Python classes. Their questions and suggestions were a great help in the preparation of these materials.

1 Part 1 -- Beginning Python

1.1 Introductions Etc

Introductions

Practical matters: restrooms, breakroom, lunch and break times, etc.

Starting the Python interactive interpreter. Also, IPython and Idle.

Running scripts

Editors -- Choose an editor which you can configure so that it indents with 4 spaces, not tab characters. For a list of editors for Python, see: http://wiki.python.org/moin/PythonEditors. A few possible editors:

- SciTE -- http://www.scintilla.org/SciTE.html.
- MS Windows only -- (1) TextPad -- http://www.textpad.com; (2) UltraEdit -- http://www.ultraedit.com/.
- Jed -- See http://www.jedsoft.org/jed/.
- Emacs -- See http://www.gnu.org/software/emacs/ and http://www.xemacs.org/faq/xemacs-faq.html.
- jEdit -- Requires a bit of customization for Python -- See http://jedit.org.
- Vim -- http://www.vim.org/
- Geany -- http://www.geany.org/
- And many more.

Interactive interpreters:

- `python`
- `ipython`
- Idle

IDEs -- Also see http://en.wikipedia.org/wiki/List_of_integrated_development_environments_for_Python:

- PyWin -- MS Windows only. Available at: http://sourceforge.net/projects/pywin32/.
- WingIDE -- See http://wingware.com/wingide/.
- Eclipse -- http://eclipse.org/. There is a plug-in that supports Python.
- Kdevelop -- Linux/KDE -- See http://www.kdevelop.org/.
- Eric -- Linux KDE? -- See http://eric-ide.python-projects.org/index.html
- Emacs and SciTE will evaluate a Python buffer within the editor.

1.1.1 Resources

Where else to get help:

- Python home page -- http://www.python.org
- Python standard documentation -- http://www.python.org/doc/. You will also find links to tutorials there.
- FAQs -- http://www.python.org/doc/faq/.
- The Python Wiki -- http://wiki.python.org/
- The Python Package Index -- Lots of Python packages -- https://pypi.python.org/pypi
- Special interest groups (SIGs) -- http://www.python.org/sigs/
- Other python related mailing lists and lists for specific applications (for example, Zope, Twisted, etc). Try: http://dir.gmane.org/search.php?match=python.
- http://sourceforge.net -- Lots of projects. Search for "python".
- USENET -- comp.lang.python. Can also be accessed through Gmane: http://dir.gmane.org/gmane.comp.python.general.
- The Python tutor email list -- http://mail.python.org/mailman/listinfo/tutor

Local documentation:

- On MS Windows, the Python documentation is installed with the standard installation.
- Install the standard Python documentation on your machine from http://www.python.org/doc/.
- `pydoc`. Example, on the command line, type: `pydoc re`.
- Import a module, then view its `.__doc__` attribute.
- At the interactive prompt, use `help(obj)`. You might need to import it first. Example:

```
>>> import urllib
>>> help(urllib)
```

- In IPython, the question mark operator gives help. Example:

```
In [13]: open?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in function open>
Namespace:      Python builtin
Docstring:
    open(name[, mode[, buffering]]) -> file object

    Open a file using the file() type, returns a file
object.
Constructor Docstring:
    x.__init__(...) initializes x; see
x.__class__.__doc__ for signature
```

```
Callable:       Yes
Call def:       Calling definition not available.Call
docstring:
    x.__call__(...) <==> x(...)
```

1.1.2 A general description of Python

Python is a high-level general purpose programming language:

- Because code is automatically compiled to byte code and executed, Python is suitable for use as a scripting language, Web application implementation language, etc.
- Because Python can be extended in C and C++, Python can provide the speed needed for even compute intensive tasks.
- Because of its strong structuring constructs (nested code blocks, functions, classes, modules, and packages) and its consistent use of objects and object-oriented programming, Python enables us to write clear, logical applications for small and large tasks.

Important features of Python:

- Built-in high level data types: strings, lists, dictionaries, etc.
- The usual control structures: if, if-else, if-elif-else, while, plus a powerful collection iterator (for).
- Multiple levels of organizational structure: functions, classes, modules, and packages. These assist in organizing code. An excellent and large example is the Python standard library.
- Compile on the fly to byte code -- Source code is compiled to byte code without a separate compile step. Source code modules can also be "pre-compiled" to byte code files.
- Object-oriented -- Python provides a consistent way to use objects: everything is an object. And, in Python it is easy to implement new object types (called classes in object-oriented programming).
- Extensions in C and C++ -- Extension modules and extension types can be written by hand. There are also tools that help with this, for example, SWIG, sip, Pyrex.
- Jython is a version of Python that "plays well with" Java. See: The Jython Project -- http://www.jython.org/Project/.

Some things you will need to know:

- Python uses indentation to show block structure. Indent one level to show the beginning of a block. Out-dent one level to show the end of a block. As an example, the following C-style code:

```
if (x)
{
        if (y)
        {
            f1()
        }
        f2()
    }
```

in Python would be:

```
    if x:
        if y:
            f1()
        f2()
```

And, the convention is to use four spaces (and no hard tabs) for each level of indentation. Actually, it's more than a convention; it's practically a requirement. Following that "convention" will make it so much easier to merge your Python code with code from other sources.

An overview of Python:

- A scripting language -- Python is suitable (1) for embedding, (2) for writing small unstructured scripts, (3) for "quick and dirty" programs.
- *Not* a scripting language -- (1) Python scales. (2) Python encourages us to write code that is clear and well-structured.
- Interpreted, but also compiled to byte-code. Modules are automatically compiled (to .pyc) when imported, but may also be explicitly compiled.
- Provides an interactive command line and interpreter shell. In fact, there are several.
- Dynamic -- For example:
    - Types are bound to values, not to variables.
    - Function and method lookup is done at runtime.
    - Values are inspect-able.
    - There is an interactive interpreter, more than one, in fact.
    - You can list the methods supported by any given object.
- Strongly typed at run-time, not compile-time. Objects (values) have a type, but variables do not.
- Reasonably high level -- High level built-in data types; high level control structures (for walking lists and iterators, for example).
- Object-oriented -- Almost everything is an object. Simple object definition. Data hiding by agreement. Multiple inheritance. Interfaces by convention. Polymorphism.
- Highly structured -- Statements, functions, classes, modules, and packages enable us to write large, well-structured applications. Why structure? Readability, locate-ability, modifiability.
- Explicitness

- First-class objects:
    - Definition: Can (1) pass to function; (2) return from function; (3) stuff into a data structure.
    - Operators can be applied to *values* (not variables). Example: `f(x)[3]`
- Indented block structure -- "Python is pseudo-code that runs."
- Embedding and extending Python -- Python provides a well-documented and supported way (1) to embed the Python interpreter in C/C++ applications and (2) to extend Python with modules and objects implemented in C/C++.
    - In some cases, SWIG can generate wrappers for existing C/C++ code automatically. See http://www.swig.org/
    - Cython enables us to generate C code from Python *and* to "easily" create wrappers for C/C++ functions. See http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/
    - To embed and extend Python with Java, there is Jython. See http://www.jython.org/
- Automatic garbage collection. (But, there is a `gc` module to allow explicit control of garbage collection.)
- Comparison with other languages: compiled languages (e.g. C/C++); Java; Perl, Tcl, and Ruby. Python excells at: development speed, execution speed, clarity and maintainability.
- Varieties of Python:
    - CPython -- Standard Python 2.x implemented in C.
    - Jython -- Python for the Java environment -- http://www.jython.org/
    - PyPy -- Python with a JIT compiler and stackless mode -- http://pypy.org/
    - Stackless -- Python with enhanced thread support and microthreads etc. -- http://www.stackless.com/
    - IronPython -- Python for .NET and the CLR -- http://ironpython.net/
    - Python 3 -- The new, new Python. This is intended as a replacement for Python 2.x. -- http://www.python.org/doc/. A few differences (from Python 2.x):
        - The `print` statement changed to the `print` function.
        - Strings are unicode by default.
        - Classes are all "new style" classes.
        - Changes to syntax for catching exceptions.
        - Changes to integers -- no long integer; integer division with automatic convert to float.
        - More pervasive use of iterables (rather than collections).
        - Etc.

      For a more information about differences between Python 2.x and Python 3.x, see the description of the various fixes that can be applied with the `2to3` tool:

http://docs.python.org/3/library/2to3.html#fixers
The migration tool, `2to3`, eases the conversion of 2.x code to 3.x.

- Also see The Zen of Python -- http://www.python.org/peps/pep-0020.html. Or, at the Python interactive prompt, type:

```
    >>> import this
```

1.1.3 Interactive Python

If you execute Python from the command line with no script (no arguments), Python gives you an interactive prompt. This is an excellent facility for learning Python and for trying small snippets of code. Many of the examples that follow were developed using the Python interactive prompt.

Start the Python interactive interpreter by typing `python` with no arguments at the command line. For example:

```
$ python
Python 2.6.1 (r261:67515, Jan 11 2009, 15:19:23)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print 'hello'
hello
>>>
```

You may also want to consider using IDLE. IDLE is a graphical integrated development environment for Python; it contains a Python shell. It is likely that Idle was installed for you when you installed Python. You will find a script to start up IDLE in the Tools/scripts directory of your Python distribution. IDLE requires Tkinter.

In addition, there are tools that will give you a more powerful and fancy Python interactive interpreter. One example is IPython, which is available at http://ipython.scipy.org/.

1.2 Lexical matters

1.2.1 Lines

- Python does what you want it to do *most* of the time so that you only have to add extra characters *some* of the time.
- Statement separator is a semi-colon, but is only needed when there is more than one statement on a line. And, writing more than one statement on the same line is considered bad form.
- Continuation lines -- A back-slash as last character of the line makes the

following line a continuation of the current line. But, note that an opening "context" (parenthesis, square bracket, or curly bracket) makes the back-slash unnecessary.

1.2.2 Comments

Everything after "#" on a line is ignored. No block comments, but doc strings are a comment in quotes at the beginning of a module, class, method or function. Also, editors with support for Python often provide the ability to comment out selected blocks of code, usually with "##".

1.2.3 Names and tokens

- Allowed characters: a-z A-Z 0-9 underscore, and must begin with a letter or underscore.
- Names and identifiers are case sensitive.
- Identifiers can be of unlimited length.
- Special names, customizing, etc. -- Usually begin and end in double underscores.
- Special name classes -- Single and double underscores.
    - Single leading single underscore -- Suggests a "private" method or variable name. Not imported by "from module import *".
    - Single trailing underscore -- Use it to avoid conflicts with Python keywords.
    - Double leading underscores -- Used in a class definition to cause name mangling (weak hiding). But, not often used.
- Naming conventions -- Not rigid, but:
    - Modules and packages -- all lower case.
    - Globals and constants -- Upper case.
    - Classes -- Bumpy caps with initial upper.
    - Methods and functions -- All lower case with words separated by underscores.
    - Local variables -- Lower case (with underscore between words) or bumpy caps with initial lower or your choice.
    - Good advice -- Follow the conventions used in the code on which you are working.
- Names/variables in Python do not have a type. Values have types.

1.2.4 Blocks and indentation

Python represents block structure and nested block structure with indentation, not with begin and end brackets.

The empty block -- Use the `pass` no-op statement.

Benefits of the use of indentation to indicate structure:

- Reduces the need for a coding standard. Only need to specify that indentation is 4 spaces and no hard tabs.
- Reduces inconsistency. Code from different sources follow the same indentation style. It has to.
- Reduces work. Only need to get the indentation correct, not *both* indentation and brackets.
- Reduces clutter. Eliminates all the curly brackets.
- If it looks correct, it is correct. Indentation cannot fool the reader.

Editor considerations -- The standard is 4 spaces (no hard tabs) for each indentation level. You will need a text editor that helps you respect that.

1.2.5 Doc strings

Doc strings are like comments, but they are carried with executing code. Doc strings can be viewed with several tools, e.g. `help()`, `obj.__doc__`, and, in IPython, a question mark (`?`) after a name will produce help.

A doc string is written as a quoted string that is at the top of a module or the first lines after the header line of a function or class.

We can use triple-quoting to create doc strings that span multiple lines.

There are also tools that extract and format doc strings, for example:

- pydoc -- Documentation generator and online help system -- http://docs.python.org/lib/module-pydoc.html.
- epydoc -- Epydoc: Automatic API Documentation Generation for Python -- http://epydoc.sourceforge.net/index.html
- Sphinx -- Can also extract documentation from Python doc strings. See http://sphinx-doc.org/index.html.

See the following for suggestions and more information on doc strings: Docstring conventions -- http://www.python.org/dev/peps/pep-0257/.

1.2.6 Program structure

- Execution -- def, class, etc are executable statements that add something to the current name-space. Modules can be both executable and import-able.
- Statements, data structures, functions, classes, modules, packages.
- Functions
- Classes
- Modules correspond to files with a "*.py" extension. Packages correspond to a directory (or folder) in the file system; a package contains a file named "__init__.py". Both modules and packages can be imported (see section import

statement).

- Packages -- A directory containing a file named "__init__.py". Can provide additional initialization when the package or a module in it is loaded (imported).

1.2.7 Operators

- See: http://docs.python.org/ref/operators.html. Python defines the following operators:

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=      <>
```

  The comparison operators `<>` and `!=` are alternate spellings of the same operator. `!=` is the preferred spelling; `<>` is obsolescent.

- Logical operators:

```
and     or      is      not     in
```

- There are also (1) the dot operator, (2) the subscript operator `[]`, and the function/method call operator `()`.
- For information on the precedences of operators, see the table at http://docs.python.org/2/reference/expressions.html#operator-precedence, which is reproduced below.
- For information on what the different operators *do*, the section in the "Python Language Reference" titled "Special method names" may be of help: http://docs.python.org/2/reference/datamodel.html#special-method-names
  The following table summarizes the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators on the same line have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators on the same line group left to right (except for comparisons, including tests, which all have the same precedence and chain from left to right -- see section 5.9 -- and exponentiation, which groups from right to left):

```
Operator                   Description
========================   ==================
lambda                     Lambda expression
or                         Boolean OR
and                        Boolean AND
not x                      Boolean NOT
in, not in                 Membership tests
is, is not                 Identity tests
<, <=, >, >=, <>, !=, ==   Comparisons
|                          Bitwise OR
^                          Bitwise XOR
&                          Bitwise AND
<<, >>                     Shifts
```

```
    +, -                          Addition and subtraction
    *, /, %                       Multiplication, division,
    remainder
    +x, -x                        Positive, negative
    ~x                            Bitwise not
    **                            Exponentiation
    x.attribute                   Attribute reference
    x[index]                      Subscription
    x[index:index]                Slicing
    f(arguments...)               Function call
    (expressions...)              Binding or tuple display
    [expressions...]              List display
    {key:datum...}                Dictionary display
    `expressions...`              String conversion
```

- Note that most operators result in calls to methods with special names, for example `__add__`, `__sub__`, `__mul__`, etc. See Special method names http://docs.python.org/2/reference/datamodel.html#special-method-names Later, we will see how these operators can be emulated in classes that you define yourself, through the use of these special names.

1.2.8 Also see

For more on lexical matters and Python styles, see:

- Code Like a Pythonista: Idiomatic Python -- http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html.
- Style Guide for Python Code -- http://www.python.org/dev/peps/pep-0008/
- The flake8 style checking program. See https://pypi.python.org/pypi/flake8. Also see the pylint code checker: https://pypi.python.org/pypi/pylint.

1.2.9 Code evaluation

Understanding the Python execution model -- How Python evaluates and executes your code.

Evaluating expressions.

Creating names/variables -- Binding -- The following all create names (variables) and bind values (objects) to them: (1) assignment, (2) function definition, (3) class definition, (4) function and method call, (5) importing a module, ...

First class objects -- Almost all objects in Python are first class. Definition: An object is first class if: (1) we can put it in a structured object; (2) we can pass it to a function; (3) we can return it from a function.

References -- Objects (or references to them) can be shared. What does this mean?

- The object(s) satisfy the identity test operator `is`.

- The built-in function `id()` returns the same value.
- The consequences for mutable objects are different from those for immutable objects.
- Changing (updating) a mutable object referenced through one variable or container also changes that object referenced through other variables or containers, because *it is the same object*.
- `del()` -- The built-in function `del()` removes a reference, not (necessarily) the object itself.

1.3 Statements and inspection -- preliminaries

`print` -- Example:

```
print obj
print "one", "two", 'three'
```

`for:` -- Example:

```
stuff = ['aa', 'bb', 'cc']
for item in stuff:
    print item
```

Learn what the *type* of an object is -- Example:

```
type(obj)
```

Learn what attributes an object has and what it's capabilities are -- Example:

```
dir(obj)
value = "a message"
dir(value)
```

Get help on a class or an object -- Example:

```
help(str)
help("")
value = "abc"
help(value)
help(value.upper)
```

In IPython (but not standard Python), you can also get help at the interactive prompt by typing "?" and "??" after an object. Example:

```
In [48]: a = ''
In [49]: a.upper?
Type:       builtin_function_or_method
String Form:<built-in method upper of str object at 0x7f1c426e0508>
Docstring:
S.upper() -> string
```

```
Return a copy of the string S converted to uppercase.
```

1.4 Built-in data-types

For information on built-in data types, see section Built-in Types -- http://docs.python.org/lib/types.html in the Python standard documentation.

1.4.1 Numeric types

The numeric types are:

- Plain integers -- Same precision as a C long, usually a 32-bit binary number.
- Long integers -- Define with `100L`. But, plain integers are automatically promoted when needed.
- Floats -- Implemented as a C double. Precision depends on your machine. See `sys.float_info`.
- Complex numbers -- Define with, for example, `3j` or `complex(3.0, 2.0)`.

See 2.3.4 Numeric Types -- int, float, long, complex -- http://docs.python.org/lib/typesnumeric.html.

Python does mixed arithmetic.

Integer division truncates. This is changed in Python 3. Use `float(n)` to force coercion to a float. Example:

```
In [8]: a = 4
In [9]: b = 5
In [10]: a / b
Out[10]: 0                # possibly wrong?
In [11]: float(a) / b
Out[11]: 0.8
```

Applying the function call operator (parentheses) to a type or class creates an instance of that type or class.

Scientific and heavily numeric programming -- High level Python is not very efficient for numerical programming. But, there are libraries that help -- Numpy and SciPy -- See: SciPy: Scientific Tools for Python -- http://scipy.org/

1.4.2 Tuples and lists

List -- A list is a dynamic array/sequence. It is ordered and indexable. A list is mutable.

List constructors: `[]`, `list()`.

`range()` and `xrange()`:

- `range(n)` creates a list of n integers. Optional arguments are the starting integer and a stride.
- `xrange` is like `range`, except that it creates an iterator that produces the items in the list of integers instead of the list itself.

Tuples -- A tuple is a sequence. A tuple is immutable.

Tuple constructors: `()`, but really a comma; also `tuple()`.

Tuples are like lists, but are not mutable.

Python lists are (1) heterogeneous (2) indexable, and (3) dynamic. For example, we can add to a list and make it longer.

Notes on sequence constructors:

- To construct a tuple with a single element, use `(x,)`; a tuple with a single element requires a comma.
- You can spread elements across multiple lines (and no need for backslash continuation character "\").
- A comma can follow the last element.

The length of a tuple or list (or other container): `len(mylist)`.

Operators for lists:

- Try: `list1 + list2`, `list1 * n`, `list1 += list2`, etc.
- Comparison operators: `<`, `==`, `>=`, etc.
- Test for membership with the `in` operator. Example:

```
In [77]: a = [11, 22, 33]
In [78]: a
Out[78]: [11, 22, 33]
In [79]: 22 in a
Out[79]: True
In [80]: 44 in a
Out[80]: False
```

Subscription:

- Indexing into a sequence
- Negative indexes -- Effectively, length of sequence plus (minus) index.
- Slicing -- Example: `data[2:5]`. Default values: beginning and end of list.
- Slicing with strides -- Example: `data[::2]`.

Operations on tuples -- No operations that change the tuple, since tuples are immutable. We can do iteration and subscription. We can do "contains" (the `in` operator) and get the length (the `len()` operator). We can use certain boolean operators.

Operations on lists -- Operations similar to tuples plus:

- Append -- `mylist.append(newitem)`.

- Insert -- `mylist.insert(index, newitem)`. Note on efficiency: The `insert` method is not as fast as the `append` method. If you find that you need to do a large number of `mylist.insert(0, obj)` (that is, inserting at the beginning of the list) consider using a deque instead. See: http://docs.python.org/2/library/collections.html#collections.deque. Or, use `append` and `reverse`.
- Extend -- `mylist.extend(anotherlist)`. Also can use `+` and `+=`.
- Remove -- `mylist.remove(item)` and `mylist.pop()`. Note that `append()` together with `pop()` implements a stack.
- Delete -- `del mylist[index]`.
- Pop -- Get last (right-most) item and remove from list -- `mylist.pop()`.

List operators -- `+`, `*`, etc.

For more operations and operators on sequences, see: http://docs.python.org/2/library/stdtypes.html#sequence-types-str-unicode-list-tuple-byte array-buffer-xrange.

Exercises:

- Create an empty list. Append 4 strings to the list. Then pop one item off the end of the list. Solution:

```
In [25]: a = []
In [26]: a.append('aaa')
In [27]: a.append('bbb')
In [28]: a.append('ccc')
In [29]: a.append('ddd')
In [30]: print a
['aaa', 'bbb', 'ccc', 'ddd']
In [31]: a.pop()
Out[31]: 'ddd'
```

- Use the `for` statement to print the items in the list. Solution:

```
In [32]: for item in a:
   ....:         print item
   ....:
aaa
bbb
ccc
```

- Use the string `join` operation to concatenate the items in the list. Solution:

```
In [33]: '||'.join(a)
Out[33]: 'aaa||bbb||ccc'
```

- Use lists containing three (3) elements to create and show a tree:

```
In [37]: b = ['bb', None, None]
In [38]: c = ['cc', None, None]
In [39]: root = ['aa', b, c]
In [40]:
In [40]:
In [40]: def show_tree(t):
   ....:     if not t:
   ....:         return
   ....:     print t[0]
   ....:     show_tree(t[1])
   ....:     show_tree(t[2])
   ....:
   ....:
In [41]: show_tree(root)
aa
bb
cc
```

Note that we will learn a better way to represent tree structures when we cover implementing classes in Python.

1.4.3 Strings

Strings are sequences. They are immutable. They are indexable. They are iterable.

For operations on strings, see http://docs.python.org/lib/string-methods.html or use:

```
>>> help(str)
```

Or:

```
>>> dir("abc")
```

String operations (methods).

String operators, e.g. `+`, `<`, `<=`, `==`, etc..

Constructors/literals:

- Quotes: single and double. Escaping quotes and other special characters with a back-slash.
- Triple quoting -- Use triple single quotes or double quotes to define multi-line strings.
- `str()` -- The constructor and the name of the type/class.
- `'aSeparator'.join(aList)`
- Many more.

Escape characters in strings -- `\t`, `\n`, `\\`, etc.

String formatting -- See:
http://docs.python.org/2/library/stdtypes.html#string-formatting-operations

Examples:

```
In [18]: name = 'dave'
In [19]: size = 25
In [20]: factor = 3.45
In [21]: print 'Name: %s  Size: %d  Factor: %3.4f' % (name, size,
factor, )
Name: dave  Size: 25  Factor: 3.4500
In [25]: print 'Name: %s  Size: %d  Factor: %08.4f' % (name, size,
factor, )
Name: dave  Size: 25  Factor: 003.4500
```

If the right-hand argument to the formatting operator is a dictionary, then you can (actually, must) use the names of keys in the dictionary in your format strings. Examples:

```
In [115]: values = {'vegetable': 'chard', 'fruit': 'nectarine'}
In [116]: 'I love %(vegetable)s and I love %(fruit)s.' % values
Out[116]: 'I love chard and I love nectarine.'
```

Also consider using the right justify and left justify operations. Examples: `mystring.rjust(20)`, `mystring.ljust(20, ':')`.

In Python 3, the `str.format` method is preferred to the string formatting operator. This method is also available in Python 2.7. It has benefits and advantages over the string formatting operator. You can start learning about it here: http://docs.python.org/2/library/stdtypes.html#string-methods

Exercises:

- Use a literal to create a string containing (1) a single quote, (2) a double quote, (3) both a single and double quote. Solutions:

  ```
  "Some 'quoted' text."
  'Some "quoted" text.'
  'Some "quoted" \'extra\' text.'
  ```

- Write a string literal that spans multiple lines. Solution:

  ```
  """This string
  spans several lines
  because it is a little long.
  """
  ```

- Use the string `join` operation to create a string that contains a colon as a separator. Solution:

  ```
  >>> content = []
  >>> content.append('finch')
  >>> content.append('sparrow')
  >>> content.append('thrush')
  >>> content.append('jay')
  >>> contentstr = ':'.join(content)
  >>> print contentstr
  finch:sparrow:thrush:jay
  ```

- Use string formatting to produce a string containing your last and first names, separated by a comma. Solution:

```
>>> first = 'Dave'
>>> last = 'Kuhlman'
>>> full = '%s, %s' % (last, first, )
>>> print full
Kuhlman, Dave
```

Incrementally building up large strings from lots of small strings -- **the old way** -- Since strings in Python are immutable, appending to a string requires a re-allocation. So, it is faster to append to a list, then use `join`. Example:

```
In [25]: strlist = []
In [26]: strlist.append('Line #1')
In [27]: strlist.append('Line #2')
In [28]: strlist.append('Line #3')
In [29]: str = '\n'.join(strlist)
In [30]: print str
Line #1
Line #2
Line #3
```

Incrementally building up large strings from lots of small strings -- **the new way** -- The `+=` operation on strings has been optimized. So, when you do this `str1 += str2`, even many times, it is efficient.

The `translate` method enables us to map the characters in a string, replacing those in one table by those in another. And, the `maketrans` function in the `string` module, makes it easy to create the mapping table:

```
import string

def test():
    a = 'axbycz'
    t = string.maketrans('abc', '123')
    print a
    print a.translate(t)

test()
```

1.4.3.1 The new string.format method

The new way to do string formatting (which is standard in Python 3 and *perhaps* preferred for new code in Python 2) is to use the `string.format` method. See here:

- http://docs.python.org/2/library/stdtypes.html#str.format
- http://docs.python.org/2/library/string.html#format-string-syntax
- http://docs.python.org/2/library/string.html#format-specification-mini-language

Some examples:

```
In [1]: 'aaa {1} bbb {0} ccc {1} ddd'.format('xx', 'yy', )
Out[1]: 'aaa yy bbb xx ccc yy ddd'
In [2]: 'number: {0:05d} ok'.format(25)
Out[2]: 'number: 00025 ok'
In [4]: 'n1: {num1}  n2: {num2}'.format(num2=25, num1=100)
Out[4]: 'n1: 100  n2: 25'
In [5]: 'n1: {num1}  n2: {num2}  again: {num1}'.format(num2=25,
num1=100)
Out[5]: 'n1: 100  n2: 25  again: 100'
In [6]: 'number: {:05d} ok'.format(25)
Out[6]: 'number: 00025 ok'
In [7]: values = {'name': 'dave', 'hobby': 'birding'}
In [8]: 'user: {name}  activity: {hobby}'.format(**values)
Out[8]: 'user: dave  activity: birding'
```

1.4.3.2 Unicode strings

Representing unicode:

```
In [96]: a = u'abcd'
In [97]: a
Out[97]: u'abcd'
In [98]: b = unicode('efgh')
In [99]: b
Out[99]: u'efgh'
```

Convert to unicode: `a_string.decode(encoding)`. Examples:

```
In [102]: 'abcd'.decode('utf-8')
Out[102]: u'abcd'
In [103]:
In [104]: 'abcd'.decode(sys.getdefaultencoding())
Out[104]: u'abcd'
```

Convert out of unicode: `a_unicode_string.encode(encoding)`. Examples:

```
In [107]: a = u'abcd'
In [108]: a.encode('utf-8')
Out[108]: 'abcd'
In [109]: a.encode(sys.getdefaultencoding())
Out[109]: 'abcd'
In [110]: b = u'Sel\xe7uk'
In [111]: print b.encode('utf-8')
Selçuk
```

Test for unicode type -- Example:

```
In [122]: import types
In [123]: a = u'abcd'
In [124]: type(a) is types.UnicodeType
Out[124]: True
In [125]:
```

```
In [126]: type(a) is type(u'')
Out[126]: True
```

Or better:

```
In [127]: isinstance(a, unicode)
Out[127]: True
```

An example with a character "c" with a hachek:

```
In [135]: name = 'Ivan Krsti\xc4\x87'
In [136]: name.decode('utf-8')
Out[136]: u'Ivan Krsti\u0107'
In [137]:
In [138]: len(name)
Out[138]: 12
In [139]: len(name.decode('utf-8'))
Out[139]: 11
```

You can also create a unicode character by using the `unichr()` built-in function:

```
In [2]: a = 'aa' + unichr(170) + 'bb'
In [3]: a
Out[3]: u'aa\xaabb'
In [6]: b = a.encode('utf-8')
In [7]: b
Out[7]: 'aa\xc2\xaabb'
In [8]: print b
aaªbb
```

Guidance for use of encodings and unicode -- If you are working with a multibyte character set:

1. Convert/decode from an external encoding to unicode *early* (`my_string.decode(encoding)`).
2. Do your work in unicode.
3. Convert/encode to an external encoding *late* (`my_string.encode(encoding)`).

For more information, see:

- Unicode In Python, Completely Demystified -- http://farmdev.com/talks/unicode/
- PEP 100: Python Unicode Integration -- http://www.python.org/dev/peps/pep-0100/
- In the Python standard library:
  - codecs -- Codec registry and base classes -- http://docs.python.org/2/library/codecs.html#module-codecs
  - Standard Encodings -- http://docs.python.org/2/library/codecs.html#standard-encodings

If you are reading and writing multibyte character data from or to a *file*, then look at the `codecs.open()` in the codecs module -- http://docs.python.org/2/library/codecs.html#codecs.open.

Handling multi-byte character sets in Python 3 is easier, I think, but different. One hint is to use the `encoding` keyword parameter to the `open` built-in function. Here is an example:

```
def test():
    infile = open('infile1.txt', 'r', encoding='utf-8')
    outfile = open('outfile1.txt', 'w', encoding='utf-8')
    for line in infile:
        line = line.upper()
        outfile.write(line)
    infile.close()
    outfile.close()

test()
```

1.4.4 Dictionaries

A dictionary is a collection, whose values are accessible by key. It is a collection of name-value pairs.

The order of elements in a dictionary is undefined. But, we can iterate over (1) the keys, (2) the values, and (3) the items (key-value pairs) in a dictionary. We can set the value of a key and we can get the value associated with a key.

Keys must be immutable objects: ints, strings, tuples, ...

Literals for constructing dictionaries:

```
d1 = {}
d2 = {key1: value1, key2: value2, }
```

Constructor for dictionaries -- `dict()` can be used to create instances of the class `dict`. Some examples:

```
dict({'one': 2, 'two': 3})
dict({'one': 2, 'two': 3}.items())
dict({'one': 2, 'two': 3}.iteritems())
dict(zip(('one', 'two'), (2, 3)))
dict([['two', 3], ['one', 2]])
dict(one=2, two=3)
dict([(['one', 'two'][i-2], i) for i in (2, 3)])
```

For operations on dictionaries, see http://docs.python.org/lib/typesmapping.html or use:

```
>>> help({})
```

Or:

```
>>> dir({})
```

Indexing -- Access or add items to a dictionary with the indexing operator `[ ]`. Example:

```
In [102]: dict1 = {}
In [103]: dict1['name'] = 'dave'
In [104]: dict1['category'] = 38
In [105]: dict1
Out[105]: {'category': 38, 'name': 'dave'}
```

Some of the operations produce the keys, the values, and the items (pairs) in a dictionary. Examples:

```
In [43]: d = {'aa': 111, 'bb': 222}
In [44]: d.keys()
Out[44]: ['aa', 'bb']
In [45]: d.values()
Out[45]: [111, 222]
In [46]: d.items()
Out[46]: [('aa', 111), ('bb', 222)]
```

When iterating over large dictionaries, use methods `iterkeys()`, `itervalues()`, and `iteritems()`. Example:

```
In [47]:
In [47]: d = {'aa': 111, 'bb': 222}
In [48]: for key in d.iterkeys():
   ....:     print key
   ....:
   ....:
aa
bb
```

To test for the existence of a key in a dictionary, use the `in` operator or the `mydict.has_key(k)` method. The `in` operator is preferred. Example:

```
>>> d = {'tomato': 101, 'cucumber': 102}
>>> k = 'tomato'
>>> k in d
True
>>> d.has_key(k)
True
```

You can often avoid the need for a test by using method `get`. Example:

```
>>> d = {'tomato': 101, 'cucumber': 102}
>>> d.get('tomato', -1)
101
>>> d.get('chard', -1)
-1
>>> if d.get('eggplant') is None:
...     print 'missing'
```

```
...
missing
```

Dictionary "view" objects provide dynamic (automatically updated) views of the keys or the values or the items in a dictionary. View objects also support set operations. Create views with `mydict.viewkeys()`, `mydict.viewvalues()`, and `mydict.viewitems()`. See: http://docs.python.org/2/library/stdtypes.html#dictionary-view-objects.

The dictionary `setdefault` method provides a way to get the value associated with a key from a dictionary and to set that value if the key is missing. Example:

```
In [106]: a
Out[106]: {}
In [108]: a.setdefault('cc', 33)
Out[108]: 33
In [109]: a
Out[109]: {'cc': 33}
In [110]: a.setdefault('cc', 44)
Out[110]: 33
In [111]: a
Out[111]: {'cc': 33}
```

Exercises:

- Write a literal that defines a dictionary using both string literals and variables containing strings. Solution:

  ```
      >>> first = 'Dave'
      >>> last = 'Kuhlman'
      >>> name_dict = {first: last, 'Elvis': 'Presley'}
      >>> print name_dict
      {'Dave': 'Kuhlman', 'Elvis': 'Presley'}
  ```

- Write statements that iterate over (1) the keys, (2) the values, and (3) the items in a dictionary. (Note: Requires introduction of the `for` statement.) Solutions:

  ```
      >>> d = {'aa': 111, 'bb': 222, 'cc': 333}
      >>> for key in d.keys():
      ...   print key
      ...
      aa
      cc
      bb
      >>> for value in d.values():
      ...   print value
      ...
      111
      333
      222
      >>> for item in d.items():
      ...   print item
  ```

```
...
('aa', 111)
('cc', 333)
('bb', 222)
>>> for key, value in d.items():
...   print key, '::', value
...
aa :: 111
cc :: 333
bb :: 222
```

Additional notes on dictionaries:

- You can use `iterkeys()`, `itervalues()`, `iteritems()` to obtain iterators over keys, values, and items.
- A dictionary itself is iterable: it iterates over its keys. So, the following two lines are equivalent:

```
for k in myDict: print k
for k in myDict.iterkeys(): print k
```

- The `in` operator tests for a key in a dictionary. Example:

```
In [52]: mydict = {'peach': 'sweet', 'lemon': 'tangy'}
In [53]: key = 'peach'
In [54]: if key in mydict:
   ....:     print mydict[key]
   ....:
sweet
```

1.4.5 Files

Open a file with the `open` factory method. Example:

```
In [28]: f = open('mylog.txt', 'w')
In [29]: f.write('message #1\n')
In [30]: f.write('message #2\n')
In [31]: f.write('message #3\n')
In [32]: f.close()
In [33]: f = file('mylog.txt', 'r')
In [34]: for line in f:
   ....:     print line,
   ....:
message #1
message #2
message #3
In [35]: f.close()
```

Notes:

- Use the (built-in) `open(path, mode)` function to open a file and create a file object. You could also use `file()`, but `open()` is recommended.

- A file object that is open for reading a text file supports the iterator protocol and, therefore, can be used in a `for` statement. It iterates over the *lines* in the file. This is most likely only useful for text files.
- `open` is a factory method that creates file objects. Use it to open files for reading, writing, and appending. Examples:

  ```
  infile = open('myfile.txt', 'r')     # open for reading
  outfile = open('myfile.txt', 'w')    # open for (over-)
  writing
  log = open('myfile.txt', 'a')        # open for
  appending to existing content
  ```

- When you have finished with a file, close it. Examples:

  ```
  infile.close()
  outfile.close()
  ```

- You can also use the `with:` statement to automatically close the file. Example:

  ```
  with open('tmp01.txt', 'r') as infile:
      for x in infile:
          print x,
  ```

  The above works because a file is a context manager: it obeys the context manager protocol. A file has methods `__enter__` and `__exit__`, and the `__exit__` method automatically closes the file for us. See the section on the with: statement.
- In order to open multiple files, you can nest `with:` statements, or use a single `with:` statement with multiple "expression as target" clauses. Example:

  ```
  def test():
      #
      # use multiple nested with: statements.
      with open('small_file.txt', 'r') as infile:
          with open('tmp_outfile.txt', 'w') as outfile:
              for line in infile:
                  outfile.write('line: %s' %
  line.upper())
      print infile
      print outfile
      #
      # use a single with: statement.
      with open('small_file.txt', 'r') as infile, \
              open('tmp_outfile.txt', 'w') as outfile:
          for line in infile:
              outfile.write('line: %s' % line.upper())
      print infile
      print outfile

  test()
  ```

- `file` is the file type and can be used as a constructor to create file objects. *But*,

  `open` is preferred.
- Lines read from a text file have a newline. Strip it off with something like: `line.rstrip('\n')`.
- For binary files you should add the binary mode, for example: `rb`, `wb`. For more about modes, see the description of the `open()` function at Built-in Functions -- http://docs.python.org/lib/built-in-funcs.html.
- Learn more about file objects and the methods they provide at: 2.3.9 File Objects -- http://docs.python.org/2/library/stdtypes.html#file-objects.

You can also append to an existing file. Note the "a" mode in the following example:

```
In [39]: f = open('mylog.txt', 'a')
In [40]: f.write('message #4\n')
In [41]: f.close()
In [42]: f = file('mylog.txt', 'r')
In [43]: for line in f:
   ....:     print line,
   ....:
message #1
message #2
message #3
message #4
In [44]: f.close()
```

For binary files, add "b" to the mode. Not strictly necessary on UNIX, but needed on MS Windows. And, you will want to make your code portable across platforms. Example:

```
In [62]: import zipfile
In [63]: outfile = open('tmp1.zip', 'wb')
In [64]: zfile = zipfile.ZipFile(outfile, 'w', zipfile.ZIP_DEFLATED)
In [65]: zfile.writestr('entry1', 'my heroes have always been
cowboys')
In [66]: zfile.writestr('entry2', 'and they still are it seems')
In [67]: zfile.writestr('entry3', 'sadly in search of and')
In [68]: zfile.writestr('entry4', 'on step in back of')
In [69]:
In [70]: zfile.writestr('entry4', 'one step in back of')
In [71]: zfile.writestr('entry5', 'themselves and their slow moving
ways')
In [72]: zfile.close()
In [73]: outfile.close()
In [75]:
$
$ unzip -lv tmp1.zip
Archive:  tmp1.zip
 Length   Method    Size  Ratio   Date    Time   CRC-32    Name
--------  ------  ------- -----   ----    ----   ------    ----
      34  Defl:N       36  -6%  05-29-08 17:04  f6b7d921  entry1
      27  Defl:N       29  -7%  05-29-08 17:07  10da8f3d  entry2
      22  Defl:N       24  -9%  05-29-08 17:07  3fd17fda  entry3
      18  Defl:N       20 -11%  05-29-08 17:08  d55182e6  entry4
      19  Defl:N       21 -11%  05-29-08 17:08  1a892acd  entry4
      37  Defl:N       39  -5%  05-29-08 17:09  e213708c  entry5
--------          -------  ---                            -------
     157              169  -8%                            6 files
```

Exercises:

- Read all of the lines of a file into a list. Print the 3rd and 5th lines in the file/list. Solution:

  ```
  In [55]: f = open('tmp1.txt', 'r')
  In [56]: lines = f.readlines()
  In [57]: f.close()
  In [58]: lines
  Out[58]: ['the\n', 'big\n', 'brown\n', 'dog\n',
  'had\n', 'long\n', 'hair\n']
  In [59]: print lines[2]
  brown

  In [61]: print lines[4]
  had
  ```

More notes:

- Strip newlines (and other whitespace) from a string with methods `strip()`, `lstrip()`, and `rstrip()`.
- Get the current position within a file by using `myfile.tell()`.
- Set the current position within a file by using `myfile.seek()`. It may be helpful to use `os.SEEK_CUR` and `os.SEEK_END`. For example:
  - `f.seek(2, os.SEEK_CUR)` advances the position by two
  - `f.seek(-3, os.SEEK_END)` sets the position to the third to last.
  - `f.seek(25)` sets the position relative to the beginning of the file.

1.4.6 Other built-in types

Other built-in data types are described in section Built-in Types -- http://docs.python.org/lib/types.html in the Python standard documentation.

1.4.6.1 The None value/type

The unique value `None` is used to indicate "no value", "nothing", "non-existence", etc. There is only one `None` value; in other words, it's a singleton.

Use `is` to test for `None`. Example:

```
>>> flag = None
>>>
>>> if flag is None:
...     print 'clear'
```

```
...
clear
>>> if flag is not None:
...     print 'hello'
...
>>>
```

1.4.6.2 Boolean values

`True` and `False` are the boolean values.

The following values also count as false, for example, in an `if:` statement: `False`, numeric zero, `None`, the empty string, an empty list, an empty dictionary, any empty container, etc. All other values, including `True`, act as true values.

1.4.6.3 Sets and frozensets

A set is an unordered collection of immutable objects. A set does not contain duplicates.

Sets support several set operations, for example: union, intersection, difference, ...

A frozenset is like a set, except that a frozenset is immutable. Therefore, a frozenset is hash-able and can be used as a key in a dictionary, and it can be added to a set.

Create a set with the set constructor. Examples:

```
>>> a = set()
>>> a
set([])
>>> a.add('aa')
>>> a.add('bb')
>>> a
set(['aa', 'bb'])
>>> b = set([11, 22])
>>> b
set([11, 22])
>>> c = set([22, 33])
>>> b.union(c)
set([33, 11, 22])
>>> b.intersection(c)
set([22])
```

For more information on sets, see: Set Types -- set, frozenset -- http://docs.python.org/lib/types-set.html

1.5 Functions and Classes -- A Preview

Structured code -- Python programs are made up of expressions, statements, functions, classes, modules, and packages.

Python objects are first-class objects.

Expressions are evaluated.

Statements are executed.

Functions (1) are objects and (2) are callable.

Object-oriented programming in Python. Modeling "real world" objects. (1) Encapsulation; (2) data hiding; (3) inheritance. Polymorphism.

Classes -- (1) encapsulation; (2) data hiding; (3) inheritance.

An overview of the structure of a typical class: (1) methods; (2) the constructor; (3) class (static) variables; (4) super/subclasses.

1.6 Statements

1.6.1 Assignment statement

Form -- `target = expression`.

Possible targets:

- Identifier
- Tuple or list -- Can be nested. Left and right sides must have equivalent structure. Example:

  ```
  >>> x, y, z = 11, 22, 33
  >>> [x, y, z] = 111, 222, 333
  >>> a, (b, c) = 11, (22, 33)
  >>> a, B = 11, (22, 33)
  ```

  This feature can be used to simulate an enum:

  ```
  In [22]: LITTLE, MEDIUM, LARGE = range(1, 4)
  In [23]: LITTLE
  Out[23]: 1
  In [24]: MEDIUM
  Out[24]: 2
  ```

- Subscription of a sequence, dictionary, etc. Example:

  ```
  In [10]: a = range(10)
  In [11]: a
  Out[11]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  In [12]: a[3] = 'abc'
  In [13]: a
  Out[13]: [0, 1, 2, 'abc', 4, 5, 6, 7, 8, 9]
  In [14]:
  In [14]: b = {'aa': 11, 'bb': 22}
  In [15]: b
  ```

```
Out[15]: {'aa': 11, 'bb': 22}
In [16]: b['bb'] = 1000
In [17]: b['cc'] = 2000
In [18]: b
Out[18]: {'aa': 11, 'bb': 1000, 'cc': 2000}
```

- A slice of a sequence -- Note that the sequence must be mutable. Example:

  ```
  In [1]: a = range(10)
  In [2]: a
  Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  In [3]: a[2:5] = [11, 22, 33, 44, 55, 66]
  In [4]: a
  Out[4]: [0, 1, 11, 22, 33, 44, 55, 66, 5, 6, 7, 8, 9]
  ```

- Attribute reference -- Example:

  ```
  >>> class MyClass:
  ...     pass
  ...
  >>> anObj = MyClass()
  >>> anObj.desc = 'pretty'
  >>> print anObj.desc
  pretty
  ```

There is also augmented assignment. Examples:

```
>>> index = 0
>>> index += 1
>>> index += 5
>>> index += f(x)
>>> index -= 1
>>> index *= 3
```

Things to note:

- Assignment to a name creates a new variable (if it does not exist in the namespace) and a binding. Specifically, it binds a value to the new name. Calling a function also does this to the (formal) parameters within the local namespace.
- In Python, a language with dynamic typing, the data type is associated with the value, not the variable, as is the case in statically typed languages.
- Assignment can also cause sharing of an object. Example:

  ```
  obj1 = A()
  obj2 = obj1
  ```

  Check to determine that the same object is shared with `id(obj)` or the `is` operator. Example:

  ```
  In [23]: a = range(10)
  In [24]: a
  Out[24]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  In [25]: b = a
  In [26]: b
  ```

```
Out[26]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [27]: b[3] = 333
In [28]: b
Out[28]: [0, 1, 2, 333, 4, 5, 6, 7, 8, 9]
In [29]: a
Out[29]: [0, 1, 2, 333, 4, 5, 6, 7, 8, 9]
In [30]: a is b
Out[30]: True
In [31]: print id(a), id(b)
31037920 31037920
```

- You can also do multiple assignment in a single statement. Example:

```
In [32]: a = b = 123
In [33]: a
Out[33]: 123
In [34]: b
Out[34]: 123
In [35]:
In [35]:
In [35]: a = b = [11, 22]
In [36]: a is b
Out[36]: True
```

- You can interchange (swap) the value of two variables using assignment and packing/unpacking:

```
>>> a = 111
>>> b = 222
>>> a, b = b, a
>>> a
222
>>> b
111
```

1.6.2 import statement

Make module (or objects in the module) available.

What `import` does:

- Evaluate the content of a module.
- Likely to create variables in the local (module) namespace.
- Evaluation of a specific module only happens once during a given run of the program. Therefore, a module is shared across an application.
- A module is evaluated from top to bottom. Later statements can replace values created earlier. This is true of functions and classes, as well as (other) variables.
- Which statements are evaluated? Assignment, `class`, `def`, ...
- Use the following idiom to make a module both run-able and import-able:

```
if __name__ == '__main__':
        # import pdb; pdb.set_trace()
        main()        # or "test()" or some other function
    defined in module
```

Notes:

- The above condition will be true *only when* the module is run as a script and will *not* be true when the module is imported.
- The line containing `pdb` can be copied any place in your program and un-commented, and then the program will drop into the Python debugger when that location is reached.

Where `import` looks for modules:

- `sys.path` shows where it looks.
- There are some standard places.
- Add additional directories by setting the environment variable `PYTHONPATH`.
- You can also add paths by modifying `sys.path`, for example:

```
import sys
sys.path.insert(0, '/path/to/my/module')
```

- Packages need a file named `__init__.py`.
- Extensions -- To determine what extensions import looks for, do:

```
>>> import imp
>>> imp.get_suffixes()
[('.so', 'rb', 3), ('module.so', 'rb', 3), ('.py', 'U',
1), ('.pyc', 'rb', 2)]
```

Forms of the `import` statement:

- `import A` -- Names in the local (module) namespace are accessible with the dot operator.
- `import A as B` -- Import the module A, but bind the module object to the variable B.
- `import A1, A2` -- Not recommended
- `from A import B`
- `from A import B1, B2`
- `from A import B as C`
- `from A import *` -- Not recommended: clutters and mixes name-spaces.
- `from A.B import C` -- (1) Possibly import object `C` from module `B` in package `A` or (2) possibly import module `C` from sub-package `B` in package `A`.
- `import A.B.C` -- To reference attributes in `C`, must use fully-qualified name, for example use `A.B.C.D` to reference `D` inside of `C`.

More notes on the `import` statement:

- The import statement and packages -- A file named `__init__.py` is required in a package. This file is evaluated the first time either the package is imported or a file in the package is imported. Question: What is made available when you do `import aPackage`? Answer: All variables (names) that are global inside the `__init__.py` module in that package. But, see notes on the use of `__all__`: The import statement -- http://docs.python.org/ref/import.html

- The use of `if __name__ == "__main__":` -- Makes a module both import-able and executable.
- Using dots in the import statement -- From the Python language reference manual:

  > "Hierarchical module names:when the module names contains one or more dots, the module search path is carried out differently. The sequence of identifiers up to the last dot is used to find a `package`; the final identifier is then searched inside the package. A package is generally a subdirectory of a directory on sys.path that has a file __init__.py."

  See: The import statement -- http://docs.python.org/ref/import.html

Exercises:

- Import a module from the standard library, for example `re`.
- Import an element from a module from the standard library, for example import `compile` from the `re` module.
- Create a simple Python package with a single module in it. Solution:
  1. Create a directory named `simplepackage` in the current directory.
  2. Create an (empty) `__init__.py` in the new directory.
  3. Create an `simple.py` in the new directory.
  4. Add a simple function name `test1` in `simple.py`.
  5. Import using any of the following:

```
>>> import simplepackage.simple
>>> from simplepackage import simple
>>> from simplepackage.simple import test1
>>> from simplepackage.simple import test1 as mytest
```

1.6.3 print statement

`print` sends output to `sys.stdout`. It adds a newline, unless an extra comma is added.

Arguments to `print`:

- Multiple items -- Separated by commas.
- End with comma to suppress carriage return.
- Use string formatting for more control over output.
- Also see various "pretty-printing" functions and methods, in particular, `pprint`. See 3.27 pprint -- Data pretty printer --

http://docs.python.org/lib/module-pprint.html.
String formatting -- Arguments are a tuple. Reference: 2.3.6.2 String Formatting Operations -- http://docs.python.org/lib/typesseq-strings.html.

Can also use `sys.stdout`. Note that a carriage return is *not* automatically added. Example:

```
>>> import sys
>>> sys.stdout.write('hello\n')
```

Controlling the destination and format of print -- Replace `sys.stdout` with an instance of any class that implements the method `write` taking one parameter. Example:

```
import sys

class Writer:
    def __init__(self, file_name):
        self.out_file = file(file_name, 'a')
    def write(self, msg):
        self.out_file.write('[[%s]]' % msg)
    def close(self):
        self.out_file.close()

def test():
    writer = Writer('outputfile.txt')
    save_stdout = sys.stdout
    sys.stdout = writer
    print 'hello'
    print 'goodbye'
    writer.close()
    # Show the output.
    tmp_file = file('outputfile.txt')
    sys.stdout = save_stdout
    content = tmp_file.read()
    tmp_file.close()
    print content

test()
```

There is an alternative form of the `print` statement that takes a file-like object, in particular an object that has a `write` method. For example:

```
In [1]: outfile = open('tmp.log', 'w')
In [2]: print >> outfile, 'Message #1'
In [3]: print >> outfile, 'Message #2'
In [4]: print >> outfile, 'Message #3'
In [5]: outfile.close()
In [6]:
In [6]: infile = open('tmp.log', 'r')
In [7]: for line in infile:
   ...:     print 'Line:', line.rstrip('\n')
   ...:
```

```
Line: Message #1
Line: Message #2
Line: Message #3
In [8]: infile.close()
```

Future deprecation warning -- There is no print *statement* in Python 3. There is a print built-in *function*.

1.6.4 if: elif: else: statement

A template for the `if:` statement:

```
if condition1:
    statements
elif condition2:
    statements
elif condition3:
    statements
else:
    statements
```

The `elif` and `else` clauses are optional.

Conditions -- Expressions -- Anything that returns a value. Compare with `eval()` and `exec`.

Truth values:

- False -- `False`, `None`, numeric zero, the empty string, an empty collection (list or tuple or dictionary or ...).
- True -- `True` and everything else.

Operators:

- `and` and `or` -- Note that both `and` and `or` do short circuit evaluation.
- `not`
- `is` and `is not` -- The identical object. Cf. `a is b` and `id(a) == id(b)`. Useful to test for `None`, for example:

  ```
  if x is None:
      ...
  if x is not None:
      ...
  ```

- `in` and `not in` -- Can be used to test for existence of a key in a dictionary or for the presence of a value in a collection.
  The `in` operator tests for equality, not identity.
  Example:

  ```
  >>> d = {'aa': 111, 'bb': 222}
  >>> 'aa' in d
  ```

```
True
>>> 'aa' not in d
False
>>> 'xx' in d
False
```

- Comparison operators, for example `==`, `!=`, `<`, `<=`, ...

There is an `if` expression. Example:

```
>>> a = 'aa'
>>> b = 'bb'
>>> x = 'yes' if a == b else 'no'
>>> x
'no'
```

Notes:

- The `elif:` clauses and the `else:` clause are optional.
- The `if:`, `elif:`, and `else:` clauses are all header lines in the sense that they are each followed by an indented block of code and each of these header lines ends with a colon. (To put an empty block after one of these, or any other, statement header line, use the `pass` statement. It's effectively a no-op.)
- Parentheses around the condition in an `if:` or `elif:` are not required and are considered bad form, unless the condition extends over multiple lines, in which case parentheses are preferred over use of a line continuation character (backslash at the end of the line).

Exercises:

- Write an `if` statement with an `and` operator.
- Write an `if` statement with an `or` operator.
- Write an `if` statement containing both `and` and `or` operators.

1.6.5 for: statement

Iterate over a sequence or an "iterable" object.

Form:

```
for x in y:
    block
```

Iterator -- Some notes on what it means to be iterable:

- An iterable is something that can be used in an iterator context, for example, in a `for:` statement, in a list comprehension, and in a generator expression.
- Sequences and containers are iterable. Examples: tuples, lists, strings, dictionaries.
- Instances of classes that obey the iterator protocol are iterable. See

http://docs.python.org/lib/typeiter.html.

- We can create an iterator object with built-in functions such as `iter()` and `enumerate()`. See Built-in Functions -- http://docs.python.org/lib/built-in-funcs.html in the Python standard library reference.
- Functions that use the `yield` statement, produce an iterator, although it's actually called a generator.
- An iterable implements the iterator interface and satisfies the iterator protocol. The iterator protocol: `__iter__()` and `next()` methods. See 2.3.5 Iterator Types -- (http://docs.python.org/lib/typeiter.html).

Testing for "iterability":

- If you can use an object in a `for:` statement, it's iterable.
- If the expresion `iter(obj)` does not produce a `TypeError` exception, it's iterable.

Some ways to produce iterators:

- `iter()` and `enumerate()` -- See: http://docs.python.org/lib/built-in-funcs.html.
- `some_dict.iterkeys()`, `some_dict.itervalues()`, `some_dict.iteritems()`.
- Use a sequence in an iterator context, for example in a `for` statement. Lists, tuples, dictionaries, and strings can be used in an iterator context to produce an iterator.
- Generator expressions -- Latest Python only. Syntactically like list comprehensions, but (1) surrounded by parentheses instead of square brackets and (2) use lazy evaluation.
- A class that implements the iterator protocol -- Example:

```
class A(object):
    def __init__(self):
        self.data = [11,22,33]
        self.idx = 0
    def __iter__(self):
        return self
    def next(self):
        if self.idx < len(self.data):
            x = self.data[self.idx]
            self.idx +=1
            return x
        else:
            raise StopIteration

def test():
    a = A()
    for x in a:
        print x

test()
```

Note that the iterator protocol changes in Python 3.

- A function containing a yield statement. See:
  - Yield expressions -- http://docs.python.org/2/reference/expressions.html#yield-expressions
  - The yield statement -- http://docs.python.org/2/reference/simple_stmts.html#the-yield-statement
- Also see `itertools` module in the Python standard library for much more help with iterators: itertools — Functions creating iterators for efficient looping -- http://docs.python.org/2/library/itertools.html#module-itertools

The `for:` statement can also do unpacking. Example:

```
In [25]: items = ['apple', 'banana', 'cherry', 'date']
In [26]: for idx, item in enumerate(items):
   ....:     print '%d.  %s' % (idx, item, )
   ....:
0.  apple
1.  banana
2.  cherry
3.  date
```

The `for` statement can also have an optional `else:` clause. The `else:` clause is executed if the `for` statement completes normally, that is if a `break` statement is *not* executed.

Helpful functions with `for`:

- `enumerate(iterable)` -- Returns an iterable that produces pairs (tuples) containing count (index) and value. Example:

  ```
  >>> for idx, value in enumerate([11,22,33]):
  ...     print idx, value
  ...
  0 11
  1 22
  2 33
  ```

- `range([start,] stop[, step])` and `xrange([start,] stop[, step])`.

List comprehensions -- Since list comprehensions create lists, they are useful in `for` statements, although, when the number of elements is large, you should consider using a generator expression instead. A list comprehension looks a bit like a `for:` statement, but is inside square brackets, and it is an expression, not a statement. Two forms (among others):

- `[f(x) for x in iterable]`

- `[f(x) for x in iterable if t(x)]`

Generator expressions -- A generator expression looks similar to a list comprehension, except that it is surrounded by parentheses rather than square brackets. Example:

```
In [28]: items = ['apple', 'banana', 'cherry', 'date']
In [29]: gen1 = (item.upper() for item in items)
In [30]: for x in gen1:
   ....:     print 'x:', x
   ....:
x: APPLE
x: BANANA
x: CHERRY
x: DATE
```

Exercises:

- Write a list comprehension that returns all the keys in a dictionary whose associated values are greater than zero.
  - The dictionary: `{'aa': 11, 'cc': 33, 'dd': -55, 'bb': 22}`
  - Solution: `[x[0] for x in my_dict.iteritems() if x[1] > 0]`
- Write a list comprehension that produces even integers from 0 to 10. Use a `for` statement to iterate over those values. Solution:

```
    for x in [y for y in range(10) if y % 2 == 0]:
        print 'x: %s' % x
```

- Write a list comprehension that iterates over two lists and produces all the combinations of items from the lists. Solution:

```
    In [19]: a = range(4)
    In [20]: b = [11,22,33]
    In [21]: a
    Out[21]: [0, 1, 2, 3]
    In [22]: b
    Out[22]: [11, 22, 33]
    In [23]: c = [(x, y) for x in a for y in b]
    In [24]: print c
    [(0, 11), (0, 22), (0, 33), (1, 11), (1, 22), (1, 33),
    (2, 11), (2, 22), (2, 33), (3, 11), (3, 22), (3, 33)]
```

But, note that in the previous exercise, a generator expression would often be better. A generator expression is like a list comprehension, except that, instead of creating the entire list, it produces a generator that can be used to produce each of the elements.

The `break` and `continue` statements are often useful in a `for` statement. See continue and break statements

The `for` statement can also have an optional `else:` clause. The `else:` clause is executed if the `for` statement completes normally, that is if a `break` statement is *not* executed. Example:

```
for item in data1:
    if item > 100:
        value1 = item
        break
else:
    value1 = 'not found'
print 'value1:', value1
```

When run, it prints:

```
value1: not found
```

1.6.6 while: statement

Form:

```
while condition:
    block
```

The `while:` statement is not often used in Python because the `for:` statement is usually more convenient, more idiomatic, and more Pythonic.

Exercises:

- Write a `while` statement that prints integers from zero to 5. Solution:

  ```
  count = 0
  while count < 5:
      count += 1
      print count
  ```

The `break` and `continue` statements are often useful in a `while` statement. See continue and break statements

The `while` statement can also have an optional `else:` clause. The `else:` clause is executed if the `while` statement completes normally, that is if a `break` statement is *not* executed.

1.6.7 continue and break statements

The `break` statement exits from a loop.

The `continue` statement causes execution to immediately continue at the start of the loop.

Can be used in `for:` and `while:`.

When the `for:` statement or the `while:` statement has an `else:` clause, the block in the `else:` clause is executed only if a `break` statement is *not* executed.

Exercises:

- Using `break`, write a `while` statement that prints integers from zero to 5. Solution:

  ```
  count = 0
  while True:
      count += 1
      if count > 5:
          break
      print count
  ```

  Notes:

  - A `for` statement that uses `range()` or `xrange()` would be better than a `while` statement for this use.

- Using `continue`, write a `while` statement that processes only even integers from 0 to 10. Note: `%` is the modulo operator. Solution:

  ```
  count = 0
  while count < 10:
      count += 1
      if count % 2 == 0:
          continue
      print count
  ```

1.6.8 try: except: statement

Exceptions are a systematic and consistent way of processing errors and "unusual" events in Python.

Caught and un-caught exceptions -- Uncaught exceptions terminate a program.

The `try:` statement catches an exception.

Almost all errors in Python are exceptions.

Evaluation (execution model) of the `try` statement -- When an exception occurs in the `try` block, even if inside a nested function call, execution of the `try` block ends and the `except` clauses are searched for a matching exception.

Tracebacks -- Also see the `traceback` module: http://docs.python.org/lib/module-traceback.html

Exceptions are classes.

Exception classes -- subclassing, args.

An exception class in an `except:` clause catches instances of that exception class and all subclasses, but *not* superclasses.

Built-in exception classes -- See:

- Module `exceptions`.

- Built-in exceptions -- http://docs.python.org/lib/module-exceptions.html.

User defined exception classes -- subclasses of `Exception`.

Example:

```
try:
    raise RuntimeError('this silly error')
except RuntimeError, exp:
    print "[[[%s]]]" % exp
```

Reference: http://docs.python.org/lib/module-exceptions.html

You can also get the arguments passed to the constructor of an exception object. In the above example, these would be:

```
exp.args
```

Why would you define your own exception class? One answer: You want a user of your code to catch your exception and no others.

Catching an exception by exception class catches exceptions of that class and all its subclasses. So:

```
except SomeExceptionClass, exp:
```

matches and catches an exception if SomeExceptionClass is the exception class or a base class (superclass) of the exception class. The exception object (usually an instance of some exception class) is bound to `exp`.

A more "modern" syntax is:

```
except SomeExceptionClass as exp:
```

So:

```
class MyE(ValueError):
    pass

try:
    raise MyE()
except ValueError:
    print 'caught exception'
```

will print "caught exception", because `ValueError` is a base class of `MyE`.

Also see the entries for "EAFP" and "LBYL" in the Python glossary: http://docs.python.org/3/glossary.html.

Exercises:

- Write a *very* simple, empty exception subclass. Solution:

  ```
  class MyE(Exception):
  ```

```
        pass
```

- Write a `try:except:` statement that raises your exception and also catches it. Solution:

```
    try:
        raise MyE('hello there dave')
    except MyE, e:
        print e
```

1.6.9 raise statement

Throw or raise an exception.

Forms:

- `raise instance`
- `raise MyExceptionClass(value)` -- preferred.
- `raise MyExceptionClass, value`

The `raise` statement takes:

- An (instance of) a built-in exception class.
- An instance of class `Exception` or
- An instance of a built-in subclass of class `Exception` or
- An instance of a user-defined subclass of class `Exception` or
- One of the above classes and (optionally) a value (for example, a string or a tuple).

See http://docs.python.org/ref/raise.html.

For a list of built-in exceptions, see http://docs.python.org/lib/module-exceptions.html.

The following example defines an exception subclass and throws an instance of that subclass. It also shows how to pass and catch multiple arguments to the exception:

```
class NotsobadError(Exception):
    pass

def test(x):
    try:
        if x == 0:
            raise NotsobadError('a moderately bad error', 'not too 
bad')
    except NotsobadError, e:
        print 'Error args: %s' % (e.args, )

test(0)
```

Which prints out the following:

```
Error args: ('a moderately bad error', 'not too bad')
```

Notes:

- In order to pass in multiple arguments with the exception, we use a tuple, or we pass multiple arguments to the constructor.

The following example does a small amount of processing of the arguments:

```
class NotsobadError(Exception):
    """An exception class.
    """
    def get_args(self):
        return '::::'.join(self.args)

def test(x):
    try:
        if x == 0:
            raise NotsobadError('a moderately bad error', 'not too
bad')
    except NotsobadError, e:
        print 'Error args: {{{%s}}}' % (e.get_args(), )

test(0)
```

1.6.10 with: statement

The `with` statement enables us to use a context manager (any object that satisfies the context manager protocol) to add code before (on entry to) and after (on exit from) a block of code.

1.6.10.1 Writing a context manager

A context manager is an instance of a class that satisfies this interface:

```
class Context01(object):
    def __enter__(self):
        pass
    def __exit__(self, exc_type, exc_value, traceback):
        pass
```

Here is an example that uses the above context manager:

```
class Context01(object):
    def __enter__(self):
        print 'in __enter__'
        return 'some value or other'    # usually we want to return
self
    def __exit__(self, exc_type, exc_value, traceback):
        print 'in __exit__'
```

Notes:

- The `__enter__` method is called *before* our block of code is entered.
- Usually, but not always, we will want the `__enter__` method to return `self`, that is, the instance of our context manager class. We do this so that we can write:

  ```
  with MyContextManager() as obj:
      pass
  ```

  and then use the instance (`obj` in this case) in the nested block.
- The `__exit__` method is called when our block of code is exited either normally or because of an exception.
- If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.
- If the block exits normally, the value of `exc_type`, `exc_value`, and `traceback` will be `None`.

For more information on the `with:` statement, see Context Manager Types -- http://docs.python.org/2/library/stdtypes.html#context-manager-types.

See module `contextlib` for strange ways of writing context managers: http://docs.python.org/2/library/contextlib.html#module-contextlib

1.6.10.2 Using the with: statement

Here are examples:

```
# example 1
with Context01():
    print 'in body'

# example 2
with Context02() as a_value:
    print 'in body'
    print 'a_value: "%s"' % (a_value, )
    a_value.some_method_in_Context02()

# example 3
with open(infilename, 'r') as infile, open(outfilename, 'w') as
outfile:
    for line in infile:
        line = line.rstrip()
        outfile.write('%s\n' % line.upper())
```

Notes:

- In the form `with ... as val`, the value returned by the `__enter__` method is assigned to the variable (`val` in this case).
- In order to use more than one context manager, you can nest `with:` statements, or separate uses of of the context managers with commas, which is usually

preferred. See example 3 above.

1.6.11 del

The `del` statement can be used to:

- Remove names from namespace.
- Remove items from a collection.

If name is listed in a `global` statement, then `del` removes name from the global namespace.

Names can be a (nested) list. Examples:

```
>>> del a
>>> del a, b, c
```

We can also delete items from a list or dictionary (and perhaps from other objects that we can subscript). Examples:

```
In [9]:d = {'aa': 111, 'bb': 222, 'cc': 333}
In [10]:print d
{'aa': 111, 'cc': 333, 'bb': 222}
In [11]:del d['bb']
In [12]:print d
{'aa': 111, 'cc': 333}
In [13]:
In [13]:a = [111, 222, 333, 444]
In [14]:print a
[111, 222, 333, 444]
In [15]:del a[1]
In [16]:print a
[111, 333, 444]
```

And, we can delete an attribute from an instance. Example:

```
In [17]:class A:
   ....:    pass
   ....:
In [18]:a = A()
In [19]:a.x = 123
In [20]:dir(a)
Out[20]:['__doc__', '__module__', 'x']
In [21]:print a.x
123
In [22]:del a.x
In [23]:dir(a)
Out[23]:['__doc__', '__module__']
In [24]:print a.x
----------------------------------------------
exceptions.AttributeError     Traceback (most recent call last)
```

```
/home/dkuhlman/a1/Python/Test/<console>

AttributeError: A instance has no attribute 'x'
```

1.6.12 case statement

There is no case statement in Python. Use the `if:` statement with a sequence of `elif:` clauses. Or, use a dictionary of functions.

1.7 Functions, Modules, Packages, and Debugging

1.7.1 Functions

1.7.1.1 The def statement

The `def` statement is used to define functions and methods.

The `def` statement is evaluated. It produces a function/method (object) and binds it to a variable in the current name-space.

Although the `def` statement is evaluated, the code in its nested block is not executed. Therefore, many errors may not be detected until each and every path through that code is tested. Recommendations: (1) Use a Python code checker, for example `flake8` or `pylint`; (2) Do thorough testing and use the Python `unittest` framework. Pythonic wisdom: If it's not tested, it's broken.

1.7.1.2 Returning values

The `return` statement is used to return values from a function.

The `return` statement takes zero or more values, separated by commas. Using commas actually returns a single tuple.

The default value is `None`.

To return multiple values, use a tuple or list. Don't forget that (assignment) unpacking can be used to capture multiple values. Returning multiple items separated by commas is equivalent to returning a tuple. Example:

```
In [8]: def test(x, y):
   ...:     return x * 3, y * 4
   ...:
In [9]: a, b = test(3, 4)
In [10]: print a
9
```

```
In [11]: print b
16
```

1.7.1.3 Parameters

Default values -- Example:

```
In [53]: def t(max=5):
   ....:     for val in range(max):
   ....:         print val
   ....:
   ....:
In [54]: t(3)
0
1
2
In [55]: t()
0
1
2
3
4
```

Giving a parameter a default value makes that parameter optional.

Note: If a function has a parameter with a default value, then all "normal" arguments must proceed the parameters with default values. More completely, parameters must be given from left to right in the following order:

1. Normal arguments.
2. Arguments with default values.
3. Argument list (`*args`).
4. Keyword arguments (`**kwargs`).

List parameters -- `*args`. It's a tuple.

Keyword parameters -- `**kwargs`. It's a dictionary.

1.7.1.4 Arguments

When calling a function, values may be passed to a function with positional arguments or keyword arguments.

Positional arguments must placed before (to the left of) keyword arguments.

Passing lists to a function as multiple arguments -- `some_func(*aList)`. Effectively, this syntax causes Python to unroll the arguments. Example:

```
def fn1(*args, **kwargs):
    fn2(*args, **kwargs)
```

1.7.1.5 Local variables

Creating local variables -- Any binding operation creates a local variable. Examples are (1) parameters of a function; (2) assignment to a variable in a function; (3) the `import` statement; (4) etc. Contrast with accessing a variable.

Variable look-up -- The LGB/LEGB rule -- The local, enclosing, global, built-in scopes are searched in that order. See: http://www.python.org/dev/peps/pep-0227/

The `global` statement -- Inside a function, we must use `global` when we want to set the value of a global variable. Example:

```
def fn():
    global Some_global_variable, Another_global_variable
    Some_global_variable = 'hello'
    ...
```

1.7.1.6 Other things to know about functions

- Functions are first-class -- You can store them in a structure, pass them to a function, and return them from a function.
- Function calls can take keyword arguments. Example:

  ```
  >>> test(size=25)
  ```

- Formal parameters to a function can have default values. Example:

  ```
  >>> def test(size=0):
          ...
  ```

  Do *not* use mutable objects as default values.
- You can "capture" remaining arguments with `*args`, and `**kwargs`. (Spelling is not significant.) Example:

  ```
  In [13]: def test(size, *args, **kwargs):
     ....:     print size
     ....:     print args
     ....:     print kwargs
     ....:
     ....:
  In [14]: test(32, 'aa', 'bb', otherparam='xyz')
  32
  ('aa', 'bb')
  {'otherparam': 'xyz'}
  ```

- Normal arguments must come before default arguments which must come before keyword arguments.
- A function that does not explicitly return a value, returns `None`.
- In order to *set* the value of a global variable, declare the variable with `global`.

Exercises:

- Write a function that takes a single argument, prints the value of the argument, and returns the argument as a string. Solution:

```
>>> def t(x):
...     print 'x: %s' % x
...     return '[[%s]]' % x
...
>>> t(3)
x: 3
'[[3]]'
```

- Write a function that takes a variable number of arguments and prints them all. Solution:

```
>>> def t(*args):
...     for arg in args:
...         print 'arg: %s' % arg
...
>>> t('aa', 'bb', 'cc')
arg: aa
arg: bb
arg: cc
```

- Write a function that prints the names and values of keyword arguments passed to it. Solution:

```
>>> def t(**kwargs):
...     for key in kwargs.keys():
...         print 'key: %s  value: %s' % (key,
kwargs[key], )
...
>>> t(arg1=11, arg2=22)
key: arg1  value: 11
key: arg2  value: 22
```

1.7.1.7 Global variables and the global statement

By default, assignment in a function or method creates local variables.

Reference (not assignment) to a variable, accesses a local variable if it has already been created, else accesses a global variable.

In order to assign a value to a global variable, declare the variable as global at the beginning of the function or method.

If in a function or method, you both reference and assign to a variable, then you must either:

1. Assign to the variable first, or
2. Declare the variable as global.

The `global` statement declares one or more variables, separated by commas, to be global.

Some examples:

```
In [1]:
In [1]: X = 3
In [2]: def t():
   ...:       print X
   ...:
In [3]:
In [3]: t()
3
In [4]: def s():
   ...:      X = 4
   ...:
In [5]:
In [5]:
In [5]: s()
In [6]: t()
3
In [7]: X = -1
In [8]: def u():
   ...:      global X
   ...:      X = 5
   ...:
In [9]:
In [9]: u()
In [10]: t()
5
In [16]: def v():
   ....:     x = X
   ....:     X = 6
   ....:     return x
   ....:
In [17]:
In [17]: v()
------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
  File "<ipython console>", line 2, in v
UnboundLocalError: local variable 'X' referenced before assignment
In [18]: def w():
   ....:     global X
   ....:     x = X
   ....:     X = 7
   ....:     return x
   ....:
In [19]:
In [19]: w()
Out[19]: 5
In [20]: X
Out[20]: 7
```

1.7.1.8 Doc strings for functions

Add docstrings as a triple-quoted string beginning with the first line of a function or method. See epydoc for a suggested format.

1.7.1.9 Decorators for functions

A decorator performs a transformation on a function. Examples of decorators that are built-in functions are: `@classmethod`, `@staticmethod`, and `@property`. See: http://docs.python.org/2/library/functions.html#built-in-functions

A decorator is applied using the "@" character on a line immediately preceeding the function definition header. Examples:

```
class SomeClass(object):

    @classmethod
    def HelloClass(cls, arg):
        pass
    ## HelloClass = classmethod(HelloClass)

    @staticmethod
    def HelloStatic(arg):
        pass
    ## HelloStatic = staticmethod(HelloStatic)

#
# Define/implement a decorator.
def wrapper(fn):
    def inner_fn(*args, **kwargs):
        print '>>'
        result = fn(*args, **kwargs)
        print '<<'
        return result
    return inner_fn

#
# Apply a decorator.
@wrapper
def fn1(msg):
    pass
## fn1 = wrapper(fn1)
```

Notes:

- The decorator form (with the "@" character) is equivalent to the form (commented out) that calls the decorator function explicitly.
- The use of `classmethods` and `staticmethod` will be explained later in the section on object-oriented programming.
- A decorator is implemented as a function. Therefore, to learn about some specific

decorator, you should search for the documentation on or the implementation of that function. Remember that in order to use a function, it must be defined in the current module or imported by the current module or be a built-in.

- The form that explicitly calls the decorator function (commented out in the example above) is equivalent to the form using the "@" character.

1.7.2 lambda

Use a lambda, as a convenience, when you need a function that both:

- is anonymous (does not need a name) and
- contains only an expression and no statements.

Example:

```
In [1]: fn = lambda x, y, z: (x ** 2) + (y * 2) + z
In [2]: fn(4, 5, 6)
Out[2]: 32
In [3]:
In [3]: format = lambda obj, category: 'The "%s" is a "%s".' % (obj,
category, )
In [4]: format('pine tree', 'conifer')
Out[4]: 'The "pine tree" is a "conifer".'
```

A lambda can take multiple arguments and can return (like a function) multiple values. Example:

```
In [79]: a = lambda x, y: (x * 3, y * 4, (x, y))
In [80]:
In [81]: a(3, 4)
Out[81]: (9, 16, (3, 4))
```

Suggestion: In some cases, a lambda may be useful as an event handler.

Example:

```
class Test:
    def __init__(self, first='', last=''):
        self.first = first
        self.last = last
    def test(self, formatter):
        """
        Test for lambdas.
        formatter is a function taking 2 arguments, first and last
          names.  It should return the formatted name.
        """
        msg = 'My name is %s' % (formatter(self.first, self.last),)
        print msg

def test():
    t = Test('Dave', 'Kuhlman')
    t.test(lambda first, last: '%s %s' % (first, last, ))
    t.test(lambda first, last: '%s, %s' % (last, first, ))

test()
```

A `lambda` enables us to define "functions" where we do not need names for those functions. Example:

```
In [45]: a = [
   ....: lambda x: x,
   ....: lambda x: x * 2,
   ....: ]
In [46]:
In [46]: a
Out[46]: [<function __main__.<lambda>>, <function __main__.<lambda>>]
In [47]: a[0](3)
Out[47]: 3
In [48]: a[1](3)
Out[48]: 6
```

Reference: http://docs.python.org/2/reference/expressions.html#lambda

1.7.3 Iterators and generators

Concepts:

iterator

And iterator is something that satisfies the iterator protocol. Clue: If it's an iterator, you can use it in a `for:` statement.

generator

A generator is a class or function that implements an iterator, i.e. that implements the iterator protocol.

the iterator protocol

An object satisfies the iterator protocol if it does the following:

- It implements a `__iter__` method, which returns an iterator object.
- It implements a `next` function, which returns the next item from the collection, sequence, stream, etc of items to be iterated over
- It raises the `StopIteration` exception when the items are exhausted and the `next()` method is called.

yield

The `yield` statement enables us to write functions that are generators. Such functions may be similar to coroutines, since they may "yield" multiple times and are resumed.

For more information on iterators, see the section on iterator types in the Python Library Reference -- http://docs.python.org/2/library/stdtypes.html#iterator-types.

For more on the `yield` statement, see: http://docs.python.org/2/reference/simple_stmts.html#the-yield-statement

Actually, `yield` is an expression. For more on yield expressions and on the `next()` and `send()` generator methods, as well as others, see: Yield expression -- http://docs.python.org/2/reference/expressions.html#yield-expressions in the Python language reference manual.

A function or method containing a `yield` statement implements a generator. Adding the `yield` statement to a function or method turns that function or method into one which, when called, returns a generator, i.e. an object that implements the iterator protocol.

A generator (a function containing `yield`) provides a convenient way to implement a filter. But, also consider:

- The `filter()` built-in function
- List comprehensions with an `if` clause

Here are a few examples:

```
def simplegenerator():
    yield 'aaa'                       # Note 1
    yield 'bbb'
    yield 'ccc'

def list_tripler(somelist):
    for item in somelist:
        item *= 3
        yield item

def limit_iterator(somelist, max):
    for item in somelist:
        if item > max:
            return                    # Note 2
        yield item

def test():
    print '1.', '-' * 30
    it = simplegenerator()
    for item in it:
        print item
    print '2.', '-' * 30
    alist = range(5)
    it = list_tripler(alist)
    for item in it:
        print item
    print '3.', '-' * 30
    alist = range(8)
    it = limit_iterator(alist, 4)
    for item in it:
        print item
    print '4.', '-' * 30
    it = simplegenerator()
    try:
        print it.next()                  # Note 3
        print it.next()
        print it.next()
        print it.next()
    except StopIteration, exp:           # Note 4
        print 'reached end of sequence'

if __name__ == '__main__':
    test()
```

Notes:

1. The `yield` statement returns a value. When the next item is requested and the iterator is "resumed", execution continues immediately after the `yield` statement.
2. We can terminate the sequence generated by an iterator by using a `return` statement with no value.
3. To resume a generator, use the generator's `next()` or `send()` methods. `send()` is like `next()`, but provides a value to the yield expression.
4. We can alternatively obtain the items in a sequence by calling the iterator's `next()` method. Since an iterator is a first-class object, we can save it in a data structure and can pass it around for use at different locations and times in our program.
1. When an iterator is exhausted or empty, it throws the `StopIteration` exception, which we can catch.

And here is the output from running the above example:

```
$ python test_iterator.py
1. ------------------------------
aaa
bbb
ccc
2. ------------------------------
0
3
6
9
12
3. ------------------------------
0
1
2
3
```

```
4
4. ------------------------------
aaa
bbb
ccc
reached end of sequence
```

An instance of a class which implements the `__iter__` method, returning an iterator, is iterable. For example, it can be used in a `for` statement or in a list comprehension, or in a generator expression, or as an argument to the `iter()` built-in method. But, notice that the class most likely implements a generator method which can be called directly.

Examples -- The following code implements an iterator that produces all the objects in a tree of objects:

```
class Node:
    def __init__(self, data, children=None):
        self.initlevel = 0
        self.data = data
        if children is None:
            self.children = []
        else:
            self.children = children
    def set_initlevel(self, initlevel): self.initlevel = initlevel
    def get_initlevel(self): return self.initlevel
    def addchild(self, child):
        self.children.append(child)
    def get_data(self):
        return self.data
    def get_children(self):
        return self.children
    def show_tree(self, level):
        self.show_level(level)
        print 'data: %s' % (self.data, )
        for child in self.children:
            child.show_tree(level + 1)
    def show_level(self, level):
        print '   ' * level,
    #
    # Generator method #1
    # This generator turns instances of this class into iterable
objects.
    #
    def walk_tree(self, level):
        yield (level, self, )
        for child in self.get_children():
            for level1, tree1 in child.walk_tree(level+1):
                yield level1, tree1
    def __iter__(self):
        return self.walk_tree(self.initlevel)
```

```
#
# Generator method #2
# This generator uses a support function (walk_list) which calls
#   this function to recursively walk the tree.
# If effect, this iterates over the support function, which
#   iterates over this function.
#
def walk_tree(tree, level):
    yield (level, tree)
    for child in walk_list(tree.get_children(), level+1):
        yield child

def walk_list(trees, level):
    for tree in trees:
        for tree in walk_tree(tree, level):
            yield tree


#
# Generator method #3
# This generator is like method #2, but calls itself (as an
iterator),
#   rather than calling a support function.
#
def walk_tree_recur(tree, level):
    yield (level, tree,)
    for child in tree.get_children():
        for level1, tree1 in walk_tree_recur(child, level+1):
            yield (level1, tree1, )


def show_level(level):
    print '   ' * level,


def test():
    a7 = Node('777')
    a6 = Node('666')
    a5 = Node('555')
    a4 = Node('444')
    a3 = Node('333', [a4, a5])
    a2 = Node('222', [a6, a7])
    a1 = Node('111', [a2, a3])
    initLevel = 2
    a1.show_tree(initLevel)
    print '=' * 40
    for level, item in walk_tree(a1, initLevel):
        show_level(level)
        print 'item:', item.get_data()
    print '=' * 40
    for level, item in walk_tree_recur(a1, initLevel):
        show_level(level)
        print 'item:', item.get_data()
    print '=' * 40
    a1.set_initlevel(initLevel)
    for level, item in a1:
        show_level(level)
        print 'item:', item.get_data()
    iter1 = iter(a1)
    print iter1
    print iter1.next()
    print iter1.next()
    print iter1.next()
    print iter1.next()
    print iter1.next()
    print iter1.next()
    print iter1.next()
##    print iter1.next()
    return a1

if __name__ == '__main__':
    test()
```

Notes:

- An instance of class `Node` is "iterable". It can be used directly in a `for` statement, a list comprehension, etc. So, for example, when an instance of `Node` is used in a `for` statement, it produces an iterator.
- We could also call the `Node.walk_method` directly to obtain an iterator.
- Method `Node.walk_tree` and functions `walk_tree` and `walk_tree_recur` are generators. When called, they return an iterator. They do this because they each contain a `yield` statement.
- These methods/functions are recursive. They call themselves. Since they are generators, they must call themselves in a context that uses an iterator, for example in a `for` statement.

1.7.4 Modules

A module is a Python source code file.

A module can be imported. When imported, the module is evaluated, and a module object is created. The module object has attributes. The following attributes are of special interest:

- `__doc__` -- The doc string of the module.
- `__name__` -- The name of the module when the module is imported, but the string "__main__" when the module is executed.
- Other names that are created (bound) in the module.

A module can be run.

To make a module both import-able and run-able, use the following idiom (at the end of the module):

```
def main():
    o
    o
    o

if __name__ == '__main__':
    main()
```

Where Python looks for modules:

- See `sys.path`.
- Standard places.
- Environment variable `PYTHONPATH`.

Notes about modules and objects:

- A module is an object.
- A module (object) can be shared.
- A specific module is imported only once in a single run. This means that a single module object is shared by all the modules that import it.

1.7.4.1 Doc strings for modules

Add docstrings as a triple-quoted string at or near the top of the file. See epydoc for a suggested format.

1.7.5 Packages

A package is a directory on the file system which contains a file named `__init__.py`.

The `__init__.py` file:

- Why is it there? -- It makes modules in the directory "import-able".
- Can `__init__.py` be empty? -- Yes. Or, just include a comment.
- When is it evaluated? -- It is evaluated the first time that an application imports anything from that directory/package.
- What can you do with it? -- Any code that should be executed exactly once and during import. For example:
    - Perform initialization needed by the package.
    - Make variables, functions, classes, etc available. For example, when the package is imported rather than modules in the package. You can also expose objects defined in modules contained in the package.
- Define a variable named `__all__` to specify the list of names that will be imported by `from my_package import *`. For example, if the following is present in `my_package/__init__.py`:

```
    __all__ = ['func1', 'func2',]
```

Then, `from my_package import *` will import `func1` and `func2`, but not other names defined in `my_package`.

Note that `__all__` can be used at the module level, as well as at the package level.

For more information, see the section on packages in the Python tutorial: http://docs.python.org/2/tutorial/modules.html#packages.

Guidance and suggestions:

- "Flat is better" -- Use the `__init__.py` file to present a "flat" view of the API for your code. Enable your users to do `import mypackage` and then reference:
  - `mypackage.item1`
  - `mypackage.item2`
  - `mypackage.item3`
  - Etc.

  Where `item1`, `item2`, etc compose the API you want your users to use, even though the implementation of these items may be buried deep in your code.
- Use the `__init__.py` module to present a "clean" API. Present only the items that you intend your users to use, and by doing so, "hide" items you do *not* intend them to use.

1.8 Classes

Classes model the behavior of objects in the "real" world. Methods implement the behaviors of these types of objects. Member variables hold (current) state. Classes enable us to implement new data types in Python.

The `class:` statement is used to define a class. The `class:` statement creates a class object and binds it to a name.

1.8.1 A simple class

```
In [104]: class A:
   .....:     pass
   .....:
In [105]: a = A()
```

To define a new style class (recommended), inherit from `object` or from another class that does. Example:

```
In [21]: class A(object):
   ....:     pass
   ....:
```

```
In [22]:
In [22]: a = A()
In [23]: a
Out[23]: <__main__.A object at 0x82fbfcc>
```

1.8.2 Defining methods

A method is a function defined in class scope and with first parameter `self`:

```
In [106]: class B(object):
   .....:     def show(self):
   .....:         print 'hello from B'
   .....:
In [107]: b = B()
In [108]: b.show()
hello from B
```

A method as we describe it here is more properly called an *instance method*, in order to distinguish it from *class methods* and *static methods*.

1.8.3 The constructor

The constructor is a method named `__init__`.

Exercise: Define a class with a member variable `name` and a `show` method. Use `print` to show the name. Solution:

```
In [109]: class A(object):
   .....:       def __init__(self, name):
   .....:           self.name = name
   .....:       def show(self):
   .....:           print 'name: "%s"' % self.name
   .....:
In [111]: a = A('dave')
In [112]: a.show()
name: "dave"
```

Notes:

- The `self` variable is explicit. It references the current object, that is the object whose method is currently executing.
- The spelling ("self") is optional, but *everyone* spells it that way.

1.8.4 Member variables

Defining member variables -- Member variables are created with assignment. Example:

```
class A(object):
    def __init__(self, name):
        self.name = name
```

A small gotcha -- Do this:

```
In [28]: class A(object):
   ....:     def __init__(self, items=None):
   ....:         if items is None:
   ....:             self.items = []
   ....:         else:
   ....:             self.items = items
```

Do *not* do this:

```
In [29]: class B:
   ....:     def __init__(self, items=[]):   # wrong.  list ctor
evaluated only once.
   ....:         self.items = items
```

In the second example, the `def` statement and the list constructor are evaluated only once. Therefore, the item member variable of all instances of class B, will share the same value, which is most likely *not* what you want.

1.8.5 Calling methods

- Use the instance and the dot operator.
- Calling a method defined in the same class or a superclass:
  - From outside the class -- Use the instance:

    ```
    some_object.some_method()
    an_array_of_of_objects[1].another_method()
    ```

  - From within the same class -- Use `self`:

    ```
    self.a_method()
    ```

  - From with a subclass when the method is in the superclass and there is a method with the same name in the current class -- Use the class (name) or use `super`:

    ```
    SomeSuperClass.__init__(self, arg1, arg2)
    super(CurrentClass,
    self).__init__(arg1, arg2)
    ```

- Calling a method defined in a specific superclass -- Use the class (name).

1.8.6 Adding inheritance

Referencing superclasses -- Use the built-in `super` or the explicit name of the superclass. Use of `super` is preferred. For example:

```
In [39]: class B(A):
```

```
   ....:      def __init__(self, name, size):
   ....:          super(B, self).__init__(name)
   ....:          # A.__init__(self, name)    # an older alternative 
form
   ....:          self.size = size
```

The use of `super()` may solve problems searching for the base class when using multiple inheritance. A better solution is to not use multiple inheritance.

You can also use multiple inheritance. But, it can cause confusion. For example, in the following, class C inherits from both A and B:

```
class C(A, B):
    ...
```

Python searches superclasses MRO (method resolution order). If only single inheritance is involved, there is little confusion. If multiple inheritance is being used, the search order of super classes can get complex -- see here: http://www.python.org/download/releases/2.3/mro

For more information on inheritance, see the tutorial in the standard Python documentation set: 9.5 Inheritance and 9.5.1 Multiple Inheritance.

Watch out for problems with inheriting from multiple classes that have a common base class.

1.8.7 Class variables

- Also called static data.
- A class variable is shared by instances of the class.
- Define at class level with assignment. Example:

```
class A(object):
    size = 5
    def get_size(self):
        return A.size
```

- Reference with `classname.variable`.
- Caution: `self.variable = x` creates a new member variable.

1.8.8 Class methods and static methods

Instance (plain) methods:

- An instance method receives the instance as its first argument.

Class methods:

- A class method receives the class as its first argument.
- Define class methods with built-in function `classmethod()` or with decorator

`@classmethod`.

- See the description of `classmethod()` built-in function at "Built-in Functions": http://docs.python.org/2/library/functions.html#classmethod

Static methods:

- A static method receives neither the instance nor the class as its first argument.
- Define static methods with built-in function `staticmethod()` or with decorator `@staticmethod`.
- See the description of `staticmethod()` built-in function at "Built-in Functions": http://docs.python.org/2/library/functions.html#staticmethod

Notes on decorators:

- A decorator of the form `@afunc` is the same as `m = afunc(m)`. So, this:

  ```
  @afunc
  def m(self): pass
  ```

  is the same as:

  ```
  def m(self): pass
  m = afunc(m)
  ```

- You can use decorators `@classmethod` and `@staticmethod` (instead of the `classmethod()` and `staticmethod()` built-in functions) to declare class methods and static methods.

Example:

```
class B(object):

    Count = 0

    def dup_string(x):
        s1 = '%s%s' % (x, x,)
        return s1
    dup_string = staticmethod(dup_string)

    @classmethod
    def show_count(cls, msg):
        print '%s  %d' % (msg, cls.Count, )

def test():
    print B.dup_string('abcd')
    B.show_count('here is the count: ')
```

An alternative way to implement "static methods" -- Use a "plain", module-level function. For example:

```
In [1]: def inc_count():
   ...:         A.count += 1
   ...:
In [2]:
```

```
In [2]: def dec_count():
   ...:         A.count -= 1
   ...:
In [3]:
In [3]: class A:
   ...:         count = 0
   ...:     def get_count(self):
   ...:             return A.count
   ...:
In [4]:
In [4]: a = A()
In [5]: a.get_count()
Out[5]: 0
In [6]:
In [6]:
In [6]: inc_count()
In [7]: inc_count()
In [8]: a.get_count()
Out[8]: 2
In [9]:
In [9]: b = A()
In [10]: b.get_count()
Out[10]: 2
```

1.8.9 Properties

The property built-in function enables us to write classes in a way that does not require a user of the class to use getters and setters. Example:

```
class TestProperty(object):
    def __init__(self, description):
        self._description = description
    def _set_description(self, description):
        print 'setting description'
        self._description = description
    def _get_description(self):
        print 'getting description'
        return self._description
    description = property(_get_description, _set_description)
```

The `property` built-in function is also a decorator. So, the following is equivalent to the above example:

```
class TestProperty(object):
    def __init__(self, description):
        self._description = description

    @property
    def description(self):
        print 'getting description'
        return self._description
    @description.setter
    def description(self, description):
        print 'setting description'
        self._description = description
```

Notes:

- We mark the instance variable as private by prefixing it with and underscore.
- The name of the instance variable and the name of the property must be different. If they are not, we get recursion and an error.

For more information on properties, see Built-in Functions -- properties -- http://docs.python.org/2/library/functions.html#property

1.8.10 Interfaces

In Python, to implement an interface is to implement a method with a specific name and a specific arguments.

"Duck typing" -- If it walks like a duck and quacks like a duck ...

One way to define an "interface" is to define a class containing methods that have a header and a doc string but no implementation.

Additional notes on interfaces:

- Interfaces are not enforced.
- A class does not have to implement *all* of an interface.

1.8.11 New-style classes

A new-style class is one that subclasses `object` or a class that subclasses `object` (that is, another new-style class).

You can subclass Python's built-in data-types.

- A simple example -- the following class extends the list data-type:

```
class C(list):
  def get_len(self):
    return len(self)

c = C((11,22,33))
c.get_len()

c = C((11,22,33,44,55,66,77,88))
print c.get_len()
# Prints "8".
```

- A slightly more complex example -- the following class extends the dictionary data-type:

```
class D(dict):
    def __init__(self, data=None, name='no_name'):
        if data is None:
            data = {}
        dict.__init__(self, data)
        self.name = name
    def get_len(self):
        return len(self)
    def get_keys(self):
        content = []
        for key in self:
            content.append(key)
        contentstr = ', '.join(content)
        return contentstr
    def get_name(self):
        return self.name

def test():
    d = D({'aa': 111, 'bb':222, 'cc':333})
    # Prints "3"
    print d.get_len()
    # Prints "'aa, cc, bb'"
    print d.get_keys()
    # Prints "no_name"
    print d.get_name()
```

Some things to remember about new-style classes:

- In order to be new-style, a class must inherit (directly or indirectly) from `object`. Note that if you inherit from a built-in type, you get this automatically.
- New-style classes unify types and classes.
- You can subclass (built-in) types such as dict, str, list, file, etc.
- The built-in types now provide factory functions: `dict()`, `str()`, `int()`, `file()`, etc.
- The built-in types are introspect-able -- Use `x.__class__`, `dir(x.__class__)`, `isinstance(x, list)`, etc.
- New-style classes give you properties and descriptors.
- New-style classes enable you to define static methods. Actually, all classes enable you to do this.
- A new-style class is a user-defined type. For an instance of a new-style class x, `type(x)` is the same as `x.__class__`.

For more on new-style classes, see: http://www.python.org/doc/newstyle/

Exercises:

- Write a class and a subclass of this class.
    - Give the superclass one member variable, a name, which can be entered when

  an instance is constructed.
  - Give the subclass one member variable, a description; the subclass constructor should allow entry of both name and description.
  - Put a `show()` method in the superclass and override the `show()` method in the subclass.

Solution:

```
class A(object):
    def __init__(self, name):
        self.name = name
    def show(self):
        print 'name: %s' % (self.name, )

class B(A):
    def __init__(self, name, desc):
        A.__init__(self, name)
        self.desc = desc
    def show(self):
        A.show(self)
        print 'desc: %s' % (self.desc, )
```

1.8.12 Doc strings for classes

Add docstrings as a (triple-quoted) string beginning with the first line of a class. See epydoc for a suggested format.

1.8.13 Private members

Add an leading underscore to a member name (method or data variable) to "suggest" that the member is private.

1.9 Special Tasks

1.9.1 Debugging tools

`pdb` -- The Python debugger:

- Start the debugger by running an expression:

```
pdb.run('expression')
```

  Example:

```
if __name__ == '__main__':
    import pdb
    pdb.run('main()')
```

- Start up the debugger at a specific location with the following:

```
import pdb; pdb.set_trace()
```

Example:

```
if __name__ == '__main__':
    import pdb
    pdb.set_trace()
    main()
```

- Get help from within the debugger. For example:

```
(Pdb) help
(Pdb) help next
```

Can also embed IPython into your code. See http://ipython.scipy.org/doc/manual/manual.html.

`ipdb` -- Also consider using `ipdb` (and `IPython`). The `ipdb` debugger interactive prompt has some additional features, for example, it does tab name completion.

Inspecting:

- `import inspect`
- See http://docs.python.org/lib/module-inspect.html.
- Don't forget to try `dir(obj)` and `type(obj)` and `help(obj)`, first.

Miscellaneous tools:

- `id(obj)`
- `globals()` and `locals()`.
- `dir(obj)` -- Returns interesting names, but list is not necessarily complete.
- `obj.__class__`
- `cls.__bases__`
- `obj.__class__.__bases__`
- `obj.__doc__`. But usually, `help(obj)` is better. It produces the doc string.
- Customize the representation of your class. Define the following methods in your class:
    - `__repr__()` -- Called by (1) `repr()`, (2) interactive interpreter when representation is needed.
    - `__str__()` -- Called by (1) `str()`, (2) string formatting.

`pdb` is implemented with the `cmd` module in the Python standard library. You can implement similar command line interfaces by using `cmd`. See: cmd -- Support for line-oriented command interpreters -- http://docs.python.org/lib/module-cmd.html.

1.9.2   File input and output

Create a file object. Use `open()`.

This example reads and prints each line of a file:

```
def test():
    f = file('tmp.py', 'r')
    for line in f:
        print 'line:', line.rstrip()
    f.close()

test()
```

Notes:

- A text file is an iterable. It iterates over the lines in a file. The following is a common idiom:

  ```
  infile = file(filename, 'r')
  for line in infile:
      process_a_line(line)
  infile.close()
  ```

- `string.rstrip()` strips new-line and other whitespace from the right side of each line. To strip new-lines only, but not other whitespace, try `rstrip('\n')`.
- Other ways of reading from a file/stream object: `my_file.read()`, `my_file.readline()`, `my_file.readlines()`,

This example writes lines of text to a file:

```
def test():
    f = file('tmp.txt', 'w')
    for ch in 'abcdefg':
        f.write(ch * 10)
        f.write('\n')
    f.close()

test()
```

Notes:

- The `write` method, unlike the `print` statement, does not automatically add new-line characters.
- Must close file in order to flush output. Or, use `my_file.flush()`.

And, don't forget the `with:` statement. It makes closing files automatic. The following example converts all the vowels in an input file to upper case and writes the converted lines to an output file:

```
import string

def show_file(infilename, outfilename):
    tran_table = string.maketrans('aeiou', 'AEIOU')
    with open(infilename, 'r') as infile, open(outfilename, 'w') as
outfile:
        for line in infile:
            line = line.rstrip()
            outfile.write('%s\n' % line.translate(tran_table))
```

1.9.3 Unit tests

For more documentation on the unit test framework, see unittest -- Unit testing framework -- http://docs.python.org/2/library/unittest.html#module-unittest

For help and more information do the following at the Python interactive prompt:

```
>>> import unittest
>>> help(unittest)
```

And, you can read the source: `Lib/unittest.py` in the Python standard library.

1.9.3.1 A simple example

Here is a very simple example. You can find more information about this primitive way of structuring unit tests in the library documentation for the `unittest` module Basic example -- http://docs.python.org/lib/minimal-example.html

```
import unittest

class UnitTests02(unittest.TestCase):

    def testFoo(self):
        self.failUnless(False)

class UnitTests01(unittest.TestCase):

    def testBar01(self):
        self.failUnless(False)

    def testBar02(self):
        self.failUnless(False)

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

Notes:

- The call to `unittest.main()` runs all tests in all test fixtures in the module. It actually creates an instance of class `TestProgram` in module `Lib/unittest.py`, which automatically runs tests.
- Test fixtures are classes that inherit from `unittest.TestCase`.
- Within a test fixture (a class), the tests are any methods whose names begin with the prefix "test".
- In any test, we check for success or failure with inherited methods such as `failIf()`, `failUnless()`, `assertNotEqual()`, etc. For more on these

methods, see the library documentation for the `unittest` module TestCase Objects -- http://docs.python.org/lib/testcase-objects.html.

- If you want to change (1) the test method prefix or (2) the function used to sort (the order of) execution of tests within a test fixture, then you can create your own instance of class `unittest.TestLoader` and customize it. For example:

  ```
  def main():
      my_test_loader = unittest.TestLoader()
      my_test_loader.testMethodPrefix = 'check'
      my_test_loader.sortTestMethodsUsing = my_cmp_func
      unittest.main(testLoader=my_test_loader)

  if __name__ == '__main__':
      main()
  ```

  But, see the notes in section Additional unittest features for instructions on a (possibly) better way to do this.

1.9.3.2 Unit test suites

Here is another, not quite so simple, example:

```
#!/usr/bin/env python

import sys, popen2
import getopt
import unittest


class GenTest(unittest.TestCase):

    def test_1_generate(self):
        cmd = 'python ../generateDS.py -f -o out2sup.py -s out2sub.py
people.xsd'
        outfile, infile = popen2.popen2(cmd)
        result = outfile.read()
        outfile.close()
        infile.close()
        self.failUnless(len(result) == 0)

    def test_2_compare_superclasses(self):
        cmd = 'diff out1sup.py out2sup.py'
        outfile, infile = popen2.popen2(cmd)
        outfile, infile = popen2.popen2(cmd)
        result = outfile.read()
        outfile.close()
        infile.close()
        #print 'len(result):', len(result)
        # Ignore the differing lines containing the date/time.
        #self.failUnless(len(result) < 130 and
result.find('Generated') > -1)
```

```
        self.failUnless(check_result(result))

    def test_3_compare_subclasses(self):
        cmd = 'diff out1sub.py out2sub.py'
        outfile, infile = popen2.popen2(cmd)
        outfile, infile = popen2.popen2(cmd)
        result = outfile.read()
        outfile.close()
        infile.close()
        # Ignore the differing lines containing the date/time.
        #self.failUnless(len(result) < 130 and
result.find('Generated') > -1)
        self.failUnless(check_result(result))


def check_result(result):
    flag1 = 0
    flag2 = 0
    lines = result.split('\n')
    len1 = len(lines)
    if len1 <= 5:
        flag1 = 1
    s1 = '\n'.join(lines[:4])
    if s1.find('Generated') > -1:
        flag2 = 1
    return flag1 and flag2


# Make the test suite.
def suite():
    # The following is obsolete.  See Lib/unittest.py.
    #return unittest.makeSuite(GenTest)
    loader = unittest.TestLoader()
    # or alternatively
    # loader = unittest.defaultTestLoader
    testsuite = loader.loadTestsFromTestCase(GenTest)
    return testsuite


# Make the test suite and run the tests.
def test():
    testsuite = suite()
    runner = unittest.TextTestRunner(sys.stdout, verbosity=2)
    runner.run(testsuite)


USAGE_TEXT = """
Usage:
    python test.py [options]
Options:
    -h, --help      Display this help message.
Example:
    python test.py
```

```
"""

def usage():
    print USAGE_TEXT
    sys.exit(-1)


def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    relink = 1
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 0:
        usage()
    test()


if __name__ == '__main__':
    main()
    #import pdb
    #pdb.run('main()')
```

Notes:

- `GenTest` is our test suite class. It inherits from `unittest.TestCase`.
- Each method in `GenTest` whose name begins with "test" will be run as a test.
- The tests are run in alphabetic order by method name.
- Defaults in class `TestLoader` for the test name prefix and sort comparison function can be overridden. See 5.3.8 TestLoader Objects -- http://docs.python.org/lib/testloader-objects.html.
- A test case class may also implement methods named `setUp()` and `tearDown()` to be run before and after tests. See 5.3.5 TestCase Objects -- http://docs.python.org/lib/testcase-objects.html. Actually, the first test method in our example should, perhaps, be a `setUp()` method.
- The tests use calls such as `self.failUnless()` to report errors. These are inherited from class `TestCase`. See 5.3.5 TestCase Objects -- http://docs.python.org/lib/testcase-objects.html.
- Function `suite()` creates an instance of the test suite.
- Function `test()` runs the tests.

1.9.3.3 Additional unittest features

And, the following example shows several additional features. See the notes that follow the code:

```
import unittest

class UnitTests02(unittest.TestCase):
    def testFoo(self):
        self.failUnless(False)
    def checkBar01(self):
        self.failUnless(False)


class UnitTests01(unittest.TestCase):
    # Note 1
    def setUp(self):
        print 'setting up UnitTests01'
    def tearDown(self):
        print 'tearing down UnitTests01'
    def testBar01(self):
        print 'testing testBar01'
        self.failUnless(False)
    def testBar02(self):
        print 'testing testBar02'
        self.failUnless(False)

def function_test_1():
    name = 'mona'
    assert not name.startswith('mo')

def compare_names(name1, name2):
    if name1 < name2:
        return 1
    elif name1 > name2:
        return -1
    else:
        return 0

def make_suite():
    suite = unittest.TestSuite()
    # Note 2
    suite.addTest(unittest.makeSuite(UnitTests01,
sortUsing=compare_names))
    # Note 3
    suite.addTest(unittest.makeSuite(UnitTests02, prefix='check'))
    # Note 4
    suite.addTest(unittest.FunctionTestCase(function_test_1))
    return suite

def main():
    suite = make_suite()
    runner = unittest.TextTestRunner()
    runner.run(suite)

if __name__ == '__main__':
    main()
```

Notes:

1. If you run this code, you will notice that the `setUp` and `tearDown` methods in class `UnitTests01` are run before and after each test in that class.
2. We can control the order in which tests are run by passing a compare function to the `makeSuite` function. The default is the `cmp` built-in function.
3. We can control which methods in a test fixture are selected to be run by passing the optional argument `prefix` to the `makeSuite` function.
4. If we have an existing function that we want to "wrap" and run as a unit test, we can create a test case from a function with the `FunctionTestCase` function. If we do that, notice that we use the `assert` statement to test and possibly cause failure.

1.9.3.4 Guidance on Unit Testing

Why should we use unit tests? Many reasons, including:

- Without unit tests, corner cases may not be checked. This is especially important, since Python does relatively little compile time error checking.
- Unit tests facilitate a frequent and short design and implement and release development cycle. See ONLamp.com -- Extreme Python -- http://www.onlamp.com/pub/a/python/2001/03/28/pythonnews.html and What is XP -- http://www.xprogramming.com/what_is_xp.htm.
- Designing the tests before writing the code is "a good idea".

Additional notes:

- In a test class, instance methods `setUp` and `tearDown` are run automatically before each and after each individual test.
- In a test class, class methods `setUpClass` and `tearDownClass` are run automatically *once* before and after *all* the tests in a class.
- Module level functions `setUpModule` and `tearDownModule` are run before and after any tests in a module.
- In some cases you can also run tests directly from the command line. Do the following for help:

```
    $ python -m unittest --help
```

1.9.4 doctest

For simple test harnesses, consider using `doctest`. With `doctest` you can (1) run a test at the Python interactive prompt, then (2) copy and paste that test into a doc string in your module, and then (3) run the tests automatically from within your module under doctest.

There are examples and explanation in the standard Python documentation: 5.2 doctest -- Test interactive Python examples -- http://docs.python.org/lib/module-doctest.html.

A simple way to use doctest in your module:

1. Run several tests in the Python interactive interpreter. Note that because doctest looks for the interpreter's ">>>" prompt, you must use the standard interpreter, and not, for example, IPython. Also, make sure that you include a line with the ">>>" prompt after each set of results; this enables doctest to determine the extent of the test results.
2. Use copy and paste, to insert the tests and their results from your interactive session into the docstrings.
3. Add the following code at the bottom of your module:

```
def _test():
    import doctest
    doctest.testmod()

if __name__ == "__main__":
    _test()
```

Here is an example:

```
def f(n):
    """
    Print something funny.

    >>> f(1)
    10
    >>> f(2)
    -10
    >>> f(3)
    0
    """
    if n == 1:
        return 10
    elif n == 2:
        return -10
    else:
        return 0


def test():
    import doctest, test_doctest
    doctest.testmod(test_doctest)

if __name__ == '__main__':
    test()
```

And, here is the output from running the above test with the -v flag:

```
$ python test_doctest.py -v
Running test_doctest.__doc__
0 of 0 examples failed in test_doctest.__doc__
Running test_doctest.f.__doc__
Trying: f(1)
Expecting: 10
ok
Trying: f(2)
Expecting: -10
ok
Trying: f(3)
Expecting: 0
ok
0 of 3 examples failed in test_doctest.f.__doc__
Running test_doctest.test.__doc__
0 of 0 examples failed in test_doctest.test.__doc__
2 items had no tests:
    test_doctest
    test_doctest.test
1 items passed all tests:
   3 tests in test_doctest.f
3 tests in 3 items.
3 passed and 0 failed.
Test passed.
```

1.9.5 The Python database API

Python database API defines a standard interface for access to a relational database.

In order to use this API you must install the database adapter (interface module) for your particular database, e.g. PostgreSQL, MySQL, Oracle, etc.

You can learn more about the Python DB-API here:
http://www.python.org/dev/peps/pep-0249/

The following simple example uses sqlite3 -- http://docs.python.org/2/library/sqlite3.html

```
#!/usr/bin/env python

"""
Create a relational database and a table in it.
Add some records.
Read and display the records.
"""

import sys
import sqlite3

def create_table(db_name):
    con = sqlite3.connect(db_name)
    cursor = con.cursor()
    cursor.execute('''CREATE TABLE plants
    (name text, desc text, cat int)''')
    cursor.execute(
        '''INSERT INTO plants VALUES ('tomato', 'red and juicy',
1)''')
    cursor.execute(
        '''INSERT INTO plants VALUES ('pepper', 'green and crunchy',
2)''')
    cursor.execute('''INSERT INTO plants VALUES ('pepper', 'purple',
2)''')
    con.commit()
    con.close()

def retrieve(db_name):
    con = sqlite3.connect(db_name)
    cursor = con.cursor()
    cursor.execute('''select * from plants''')
    rows = cursor.fetchall()
    print rows
    print '-' * 40
    cursor.execute('''select * from plants''')
    for row in cursor:
        print row
    con.close()

def test():
    args = sys.argv[1:]
    if len(args) != 1:
        sys.stderr.write('\nusage: test_db.py <db_name>\n\n')
        sys.exit(1)
    db_name = args[0]
    create_table(db_name)
    retrieve(db_name)

test()
```

1.9.6   Installing Python packages

Simple:

```
$ python setup.py build
$ python setup.py install    # as root
```

More complex:

- Look for a `README` or `INSTALL` file at the root of the package.
- Type the following for help:

  ```
  $ python setup.py cmd --help
  $ python setup.py --help-commands
  $ python setup.py --help [cmd1 cmd2 ...]
  ```

- And, for even more details, see Installing Python Modules --

http://docs.python.org/inst/inst.html

`pip` is becoming popular for installing and managing Python packages. See: https://pypi.python.org/pypi/pip

Also, consider using `virtualenv`, especially if you suspect or worry that installing some new package will alter the behavior of a package currently installed on your machine. See: https://pypi.python.org/pypi/virtualenv. `virtualenv` creates a directory and sets up a Python environment into which you can install and use Python packages without changing your usual Python installation.

1.10 More Python Features and Exercises

[As time permits, explain more features and do more exercises as requested by class members.]

Thanks to David Goodger for the following list or references. His "Code Like a Pythonista: Idiomatic Python" (http://python.net/~goodger/projects/pycon/2007/idiomatic/) is worth a careful reading:

- "Python Objects", Fredrik Lundh, http://www.effbot.org/zone/python-objects.htm
- "How to think like a Pythonista", Mark Hammond, http://python.net/crew/mwh/hacks/objectthink.html
- "Python main() functions", Guido van Rossum, http://www.artima.com/weblogs/viewpost.jsp?thread=4829
- "Python Idioms and Efficiency", http://jaynes.colorado.edu/PythonIdioms.html
- "Python track: python idioms", http://www.cs.caltech.edu/courses/cs11/material/python/misc/python_idioms.html
- "Be Pythonic", Shalabh Chaturvedi, http://shalabh.infogami.com/Be_Pythonic2
- "Python Is Not Java", Phillip J. Eby, http://dirtsimple.org/2004/12/python-is-not-java.html
- "What is Pythonic?", Martijn Faassen, http://faassen.n--tree.net/blog/view/weblog/2005/08/06/0
- "Sorting Mini-HOWTO", Andrew Dalke, http://wiki.python.org/moin/HowTo/Sorting
- "Python Idioms", http://www.gungfu.de/facts/wiki/Main/PythonIdioms
- "Python FAQs", http://www.python.org/doc/faq/

2 Part 2 -- Advanced Python

2.1 Introduction -- Python 201 -- (Slightly) Advanced Python Topics

This document is intended as notes for a course on (slightly) advanced Python topics.

2.2 Regular Expressions

For more help on regular expressions, see:

- re - Regular expression operations http://docs.python.org/library/re.html
- Regular Expression HOWTO -- http://docs.python.org/howto/regex.html

2.2.1 Defining regular expressions

A regular expression pattern is a sequence of characters that will match sequences of characters in a target.

The patterns or regular expressions can be defined as follows:

- Literal characters must match exactly. For example, "a" matches "a".
- Concatenated patterns match concatenated targets. For example, "ab" ("a" followed by "b") matches "ab".
- Alternate patterns (separated by a vertical bar) match either of the alternative patterns. For example, "(aaa)|(bbb)" will match either "aaa" or "bbb".
- Repeating and optional items:
    - "abc*" matches "ab" followed by zero or more occurances of "c", for example, "ab", "abc", "abcc", etc.
    - "abc+" matches "ab" followed by one or more occurances of "c", for example, "abc", "abcc", etc, but not "ab".
    - "abc?" matches "ab" followed by zero or one occurances of "c", for example, "ab" or "abc".
- Sets of characters -- Characters and sequences of characters in square brackets form a set; a set matches any character in the set or range. For example, "[abc]" matches "a" or "b" or "c". And, for example, "[_a-z0-9]" matches an underscore or any lower-case letter or any digit.
- Groups -- Parentheses indicate a group with a pattern. For example, "ab(cd)*ef" is a pattern that matches "ab" followed by any number of occurances of "cd" followed by "ef", for example, "abef", "abcdef", "abcdcdef", etc.
- There are special names for some sets of characters, for example "\d" (any digit),

"\w" (any alphanumeric character), "\W" (any non-alphanumeric character), etc. More more information, see Python Library Reference: Regular Expression Syntax -- http://docs.python.org/library/re.html#regular-expression-syntax

Because of the use of backslashes in patterns, you are usually better off defining regular expressions with raw strings, e.g. `r"abc"`.

2.2.2 Compiling regular expressions

When a regular expression is to be used more than once, you should consider compiling it. For example:

```
import sys, re

pat = re.compile('aa[bc]*dd')

while 1:
    line = raw_input('Enter a line ("q" to quit):')
    if line == 'q':
        break
    if pat.search(line):
        print 'matched:', line
    else:
        print 'no match:', line
```

Comments:

- We import module re in order to use regular expresions.
- `re.compile()` compiles a regular expression so that we can reuse the compiled regular expression without compiling it repeatedly.

2.2.3 Using regular expressions

Use `match()` to match at the beginning of a string (or not at all).

Use `search()` to search a string and match the first string from the left.

Here are some examples:

```
>>> import re
>>> pat = re.compile('aa[0-9]*bb')
>>> x = pat.match('aa1234bbccddee')
>>> x
<_sre.SRE_Match object at 0x401e9608>
>>> x = pat.match('xxxxaa1234bbccddee')
>>> x
>>> type(x)
<type 'NoneType'>
>>> x = pat.search('xxxxaa1234bbccddee')
>>> x
```

```
<_sre.SRE_Match object at 0x401e9608>
```

Notes:

- When a match or search is successful, it returns a match object. When it fails, it returns None.
- You can also call the corresponding functions match and search in the re module, e.g.:

  ```
  >>> x = re.search(pat, 'xxxxaa1234bbccddee')
  >>> x
  <_sre.SRE_Match object at 0x401e9560>
  ```

  For a list of functions in the re module, see Module Contents -- http://docs.python.org/library/re.html#module-contents.

2.2.4 Using match objects to extract a value

Match objects enable you to extract matched sub-strings after performing a match. A match object is returned by successful match. The part of the target available in the match object is the portion matched by groups in the pattern, that is the portion of the pattern inside parentheses. For example:

```
In [69]: mo = re.search(r'height: (\d*) width: (\d*)', 'height: 123
width: 456')
In [70]: mo.groups()
Out[70]: ('123', '456')
```

Here is another example:

```
import sys, re

Targets = [
    'There are <<25>> sparrows.',
    'I see <<15>> finches.',
    'There is nothing here.',
    ]

def test():
    pat = re.compile('<<([0-9]*)>>')
    for line in Targets:
        mo = pat.search(line)
        if mo:
            value = mo.group(1)
            print 'value: %s' % value
        else:
            print 'no match'

test()
```

When we run the above, it prints out the following:

```
value: 25
value: 15
no match
```

Explanation:

- In the regular expression, put parentheses around the portion of the regular expression that will match what you want to extract. Each pair of parentheses marks off a group.
- After the search, check to determine if there was a successful match by checking for a matching object. "pat.search(line)" returns None if the search fails.
- If you specify more than one group in your regular expression (more that one pair of parentheses), then you can use "value = mo.group(N)" to extract the value matched by the Nth group from the matching object. "value = mo.group(1)" returns the first extracted value; "value = mo.group(2)" returns the second; etc. An argument of 0 returns the string matched by the entire regular expression.

In addition, you can:

- Use "values = mo.groups()" to get a tuple containing the strings matched by all groups.
- Use "mo.expand()" to interpolate the group values into a string. For example, "mo.expand(r'value1: \1 value2: \2')"inserts the values of the first and second group into a string. If the first group matched "aaa" and the second matched "bbb", then this example would produce "value1: aaa value2: bbb". For example:

```
In [76]: mo = re.search(r'h: (\d*) w: (\d*)', 'h: 123
w: 456')
In [77]: mo.expand(r'Height: \1  Width: \2')
Out[77]: 'Height: 123  Width: 456'
```

2.2.5 Extracting multiple items

You can extract multiple items with a single search. Here is an example:

```
import sys, re

pat = re.compile('aa([0-9]*)bb([0-9]*)cc')

while 1:
    line = raw_input('Enter a line ("q" to quit):')
    if line == 'q':
        break
    mo = pat.search(line)
    if mo:
        value1, value2 = mo.group(1, 2)
        print 'value1: %s  value2: %s' % (value1, value2)
    else:
        print 'no match'
```

Comments:

- Use multiple parenthesized substrings in the regular expression to indicate the portions (groups) to be extracted.
- "mo.group(1, 2)" returns the values of the first and second group in the string matched.
- We could also have used "mo.groups()" to obtain a tuple that contains both values.
- Yet another alternative would have been to use the following: `print mo.expand(r'value1: \1 value2: \2')`.

2.2.6 Replacing multiple items

A simple way to perform multiple replacements using a regular expression is to use the `re.subn()` function. Here is an example:

```
In [81]: re.subn(r'\d+', '***', 'there are 203 birds sitting in 2
trees')
Out[81]: ('there are *** birds sitting in *** trees', 2)
```

For more complex replacements, use a function instead of a constant replacement string:

```
import re

def repl_func(mo):
    s1 = mo.group(1)
    s2 = '*' * len(s1)
    return s2

def test():
    pat = r'(\d+)'
    in_str = 'there are 2034 birds in 21 trees'
    out_str, count = re.subn(pat, repl_func, in_str)
    print 'in:  "%s"' % in_str
    print 'out: "%s"' % out_str
    print 'count: %d' % count

test()
```

And when we run the above, it produces:

```
in:  "there are 2034 birds in 21 trees"
out: "there are **** birds in ** trees"
count: 2
```

Notes:

- The replacement function receives one argument, a match object.
- The `re.subn()` function returns a tuple containing two values: (1) the string after replacements and (2) the number of replacements performed.

Here is an even more complex example -- You can locate sub-strings (slices) of a match and replace them:

```
import sys, re

pat = re.compile('aa([0-9]*)bb([0-9]*)cc')

while 1:
    line = raw_input('Enter a line ("q" to quit): ')
    if line == 'q':
        break
    mo = pat.search(line)
    if mo:
        value1, value2 = mo.group(1, 2)
        start1 = mo.start(1)
        end1 = mo.end(1)
        start2 = mo.start(2)
        end2 = mo.end(2)
        print 'value1: %s  start1: %d  end1: %d' % (value1, start1, 
end1)
        print 'value2: %s  start2: %d  end2: %d' % (value2, start2, 
end2)
        repl1 = raw_input('Enter replacement #1: ')
        repl2 = raw_input('Enter replacement #2: ')
        newline = (line[:start1] + repl1 + line[end1:start2] +
            repl2 + line[end2:])
        print 'newline: %s' % newline
    else:
        print 'no match'
```

Explanation:

- Alternatively, use "mo.span(1)" instead of "mo.start(1)" and "mo.end(1)" in order to get the start and end of a sub-match in a single operation. "mo.span(1)"returns a tuple: (start, end).
- Put together a new string with string concatenation from pieces of the original string and replacement values. You can use string slices to get the sub-strings of the original string. In our case, the following gets the start of the string, adds the first replacement, adds the middle of the original string, adds the second replacement, and finally, adds the last part of the original string:

```
newline = line[:start1] + repl1 + line[end1:start2] + 
repl2 + line[end2:]
```

You can also use the sub function or method to do substitutions. Here is an example:

```
import sys, re

pat = re.compile('[0-9]+')

print 'Replacing decimal digits.'
```

```
while 1:
    target = raw_input('Enter a target line ("q" to quit): ')
    if target == 'q':
        break
    repl = raw_input('Enter a replacement: ')
    result = pat.sub(repl, target)
    print 'result: %s' % result
```

Here is another example of the use of a function to insert calculated replacements.

```
import sys, re, string

pat = re.compile('[a-m]+')

def replacer(mo):
    return string.upper(mo.group(0))

print 'Upper-casing a-m.'
while 1:
    target = raw_input('Enter a target line ("q" to quit): ')
    if target == 'q':
        break
    result = pat.sub(replacer, target)
    print 'result: %s' % result
```

Notes:

- If the replacement argument to sub is a function, that function must take one argument, a match object, and must return the modified (or replacement) value. The matched sub-string will be replaced by the value returned by this function.
- In our case, the function replacer converts the matched value to upper case.

This is also a convenient use for a lambda instead of a named function, for example:

```
import sys, re, string

pat = re.compile('[a-m]+')

print 'Upper-casing a-m.'
while 1:
    target = raw_input('Enter a target line ("q" to quit): ')
    if target == 'q':
        break
    result = pat.sub(
        lambda mo: string.upper(mo.group(0)),
        target)
    print 'result: %s' % result
```

2.3 Iterator Objects

Note 1: You will need a sufficiently recent version of Python in order to use iterators and generators. I believe that they were introduced in Python 2.2.

Note 2: The iterator protocol has changed slightly in Python version 3.0.

Goals for this section:

- Learn how to implement a generator function, that is, a function which, when called, returns an iterator.
- Learn how to implement a class containing a generator method, that is, a method which, when called, returns an iterator.
- Learn the iterator protocol, specifically what methods an iterator must support and what those methods must do.
- Learn how to implement an iterator class, that is, a class whose instances are iterator objects.
- Learn how to implement recursive iterator generators, that is, an iterator generator which recursively produces iterator generators.
- Learn that your implementation of an iterator object (an iterator class) can "refresh" itself and learn at least one way to do this.

Definitions:

- Iterator - And iterator is an object that satisfies (implements) the iterator protocol.
- Iterator protocol - An object implements the iterator protocol if it implements both a `next()` and an `__iter__()` method which satisfy these rules: (1) the `__iter__()` method must return the iterator; (2) the `next()` method should return the next item to be iterated over and when finished (there are no more items) should raise the `StopIteration` exception. The iterator protocol is described at Iterator Types -- http://docs.python.org/library/stdtypes.html#iterator-types.
- Iterator class - A class that implements (satisfies) the iterator protocol. In particular, the class implements `next()` and `__iter__()` methods as described above and in Iterator Types -- http://docs.python.org/library/stdtypes.html#iterator-types.
- (Iterator) generator function - A function (or method) which, when called, returns an iterator object, that is, an object that satisfies the iterator protocol. A function containing a yield statement automatically becomes a generator.
- Generator expression - An expression which produces an iterator object. Generator expressions have a form similar to a list comprehension, but are enclosed in parentheses rather than square brackets. See example below.

A few additional basic points:

- A function that contains a yield statement is a generator function. When called, it returns an iterator, that is, an object that provides `next()` and `__iter__()` methods.
- The iterator protocol is described here: Python Standard Library: Iterator Types -- http://docs.python.org/library/stdtypes.html#iterator-types.

- A class that defines both a `next()` method and a `__iter__()` method satisfies the iterator protocol. So, instances of such a class will be iterators.
- Python provides a variety of ways to produce (implement) iterators. This section describes a few of those ways. You should also look at the `iter()` built-in function, which is described in The Python Standard Library: Built-in Functions: iter() -- http://docs.python.org/library/functions.html#iter.
- An iterator can be used in an iterator context, for example in a for statement, in a list comprehension, and in a generator expression. When an iterator is used in an iterator context, the iterator produces its values.

This section attempts to provide examples that illustrate the generator/iterator pattern.

Why is this important?

- Once mastered, it is a simple, convenient, and powerful programming pattern.
- It has many and pervasive uses.
- It helps to lexically separate the producer code from the consumer code. Doing so makes it easier to locate problems and to modify or fix code in a way that is localized and does not have unwanted side-effects.
- Implementing your own iterators (and generators) enables you to define your own abstract sequences, that is, sequences whose composition are defined by your computations rather than by their presence in a container. In fact, your iterator can calculate or retrieve values as each one is requested.

Examples - The remainder of this section provides a set of examples which implement and use iterators.

2.3.1 Example - A generator function

This function contains a yield statement. Therefore, when we call it, it produces an iterator:

```
def generateItems(seq):
    for item in seq:
        yield 'item: %s' % item

anIter = generateItems([])
print 'dir(anIter):', dir(anIter)
anIter = generateItems([111,222,333])
for x in anIter:
    print x
anIter = generateItems(['aaa', 'bbb', 'ccc'])
print anIter.next()
print anIter.next()
print anIter.next()
print anIter.next()
```

Running this example produces the following output:

```
dir(anIter): ['__class__', '__delattr__', '__doc__',
'__getattribute__',
'__hash__', '__init__', '__iter__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__str__', 'gi_frame',
'gi_running', 'next']
item: 111
item: 222
item: 333
item: aaa
item: bbb
item: ccc
Traceback (most recent call last):
  File "iterator_generator.py", line 14, in ?
    print anIter.next()
StopIteration
```

Notes and explanation:

- The value returned by the call to the generator (function) is an iterator. It obeys the iterator protocol. That is, `dir(anIter)` shows that it has both `__iter__()` and `next()` methods.
- Because this object is an iterator, we can use a for statement to iterate over the values returned by the generator.
- We can also get its values by repeatedly calling the `next()` method, until it raises the StopIteration exception. This ability to call the next method enables us to pass the iterator object around and get values at different locations in our code.
- Once we have obtained all the values from an iterator, it is, in effect, "empty" or "exhausted". The iterator protocol, in fact, specifies that once an iterator raises the StopIteration exception, it should continue to do so. Another way to say this is that there is no "rewind" operation. But, you can call the the generator function again to get a "fresh" iterator.

An alternative and perhaps simpler way to create an interator is to use a generator expression. This can be useful when you already have a collection or iterator to work with.

Then following example implements a function that returns a generator object. The effect is to generate the objects in a collection which excluding items in a separte collection:

```
DATA = [
    'lemon',
    'lime',
    'grape',
    'apple',
    'pear',
    'watermelon',
    'canteloupe',
    'honeydew',
    'orange',
    'grapefruit',
    ]

def make_producer(collection, excludes):
    gen = (item for item in collection if item not in excludes)
    return gen

def test():
    iter1 = make_producer(DATA, ('apple', 'orange', 'honeydew', ))
    print '%s' % iter1
    for fruit in iter1:
        print fruit

test()
```

When run, this example produces the following:

```
$ python workbook063.py
<generator object <genexpr> at 0x7fb3d0f1bc80>
lemon
lime
grape
pear
watermelon
canteloupe
grapefruit
```

Notes:

- A generator expression looks almost like a list comprehension, but is surrounded by parentheses rather than square brackets. For more on list comprehensions see section Example - A list comprehension.
- The `make_producer` function returns the object produced by the generator expression.

2.3.2 Example - A class containing a generator method

Each time this method is called, it produces a (new) iterator object. This method is analogous to the iterkeys and itervalues methods in the dictionary built-in object:

```
#
# A class that provides an iterator generator method.
#
class Node:
    def __init__(self, name='<noname>', value='<novalue>',
children=None):
        self.name = name
        self.value = value
        self.children = children
        if children is None:
            self.children = []
        else:
            self.children = children
    def set_name(self, name): self.name = name
    def get_name(self): return self.name
    def set_value(self, value): self.value = value
    def get_value(self): return self.value
    def iterchildren(self):
        for child in self.children:
            yield child
    #
    # Print information on this node and walk over all children and
    #   grandchildren ...
    def walk(self, level=0):
        print '%sname: %s  value: %s' % (
            get_filler(level), self.get_name(), self.get_value(), )
        for child in self.iterchildren():
            child.walk(level + 1)

#
# An function that is the equivalent of the walk() method in
#   class Node.
#
def walk(node, level=0):
    print '%sname: %s  value: %s' % (
        get_filler(level), node.get_name(), node.get_value(), )
    for child in node.iterchildren():
        walk(child, level + 1)

def get_filler(level):
    return '    ' * level

def test():
    a7 = Node('gilbert', '777')
    a6 = Node('fred', '666')
    a5 = Node('ellie', '555')
    a4 = Node('daniel', '444')
    a3 = Node('carl', '333', [a4, a5])
    a2 = Node('bill', '222', [a6, a7])
    a1 = Node('alice', '111', [a2, a3])
    # Use the walk method to walk the entire tree.
    print 'Using the method:'
    a1.walk()
    print '=' * 30
    # Use the walk function to walk the entire tree.
    print 'Using the function:'
    walk(a1)

test()
```

Running this example produces the following output:

```
Using the method:
name: alice  value: 111
```

```
    name: bill  value: 222
        name: fred  value: 666
        name: gilbert  value: 777
    name: carl  value: 333
        name: daniel  value: 444
        name: ellie  value: 555
==============================
Using the function:
name: alice  value: 111
    name: bill  value: 222
        name: fred  value: 666
        name: gilbert  value: 777
    name: carl  value: 333
        name: daniel  value: 444
        name: ellie  value: 555
```

Notes and explanation:

- This class contains a method iterchildren which, when called, returns an iterator.
- The yield statement in the method iterchildren makes it into a generator.
- The yield statement returns one item each time it is reached. The next time the iterator object is "called" it resumes immediately after the yield statement.
- A function may have any number of yield statements.
- A for statement will iterate over all the items produced by an iterator object.
- This example shows two ways to use the generator, specifically: (1) the walk method in the class Node and (2) the walk function. Both call the generator iterchildren and both do pretty much the same thing.

2.3.3 Example - An iterator class

This class implements the iterator protocol. Therefore, instances of this class are iterators. The presence of the `next()` and `__iter__()` methods means that this class implements the iterator protocol and makes instances of this class iterators.

Note that when an iterator is "exhausted" it, normally, cannot be reused to iterate over the sequence. However, in this example, we provide a refresh method which enables us to "rewind" and reuse the iterator instance:

```
#
# An iterator class that does *not* use ``yield``.
#   This iterator produces every other item in a sequence.
#
class IteratorExample:
    def __init__(self, seq):
        self.seq = seq
        self.idx = 0
    def next(self):
        self.idx += 1
        if self.idx >= len(self.seq):
            raise StopIteration
        value = self.seq[self.idx]
        self.idx += 1
        return value
    def __iter__(self):
        return self
    def refresh(self):
        self.idx = 0

def test_iteratorexample():
    a = IteratorExample('edcba')
    for x in a:
        print x
    print '----------'
    a.refresh()
    for x in a:
        print x
    print '=' * 30
    a = IteratorExample('abcde')
    try:
        print a.next()
        print a.next()
        print a.next()
        print a.next()
        print a.next()
        print a.next()
    except StopIteration, e:
        print 'stopping', e

test_iteratorexample()
```

Running this example produces the following output:

```
d
b
----------
d
b
==============================
b
d
stopping
```

Notes and explanation:

- The next method must keep track of where it is and what item it should produce next.
- **Alert:** The iterator protocol has changed slightly in Python 3.0. In particular, the `next()` method has been renamed to `__next__()`. See: Python Standard Library: Iterator Types -- http://docs.python.org/3.0/library/stdtypes.html#iterator-types.

2.3.4 Example - An iterator class that uses yield

There may be times when the next method is easier and more straight-forward to implement using yield. If so, then this class might serve as an model. If you do not feel the need to do this, then you should ignore this example:

```
#
# An iterator class that uses ``yield``.
#   This iterator produces every other item in a sequence.
#
class YieldIteratorExample:
    def __init__(self, seq):
        self.seq = seq
        self.iterator = self._next()
        self.next = self.iterator.next
    def _next(self):
        flag = 0
        for x in self.seq:
            if flag:
                flag = 0
                yield x
            else:
                flag = 1
    def __iter__(self):
        return self.iterator
    def refresh(self):
        self.iterator = self._next()
        self.next = self.iterator.next

def test_yielditeratorexample():
    a = YieldIteratorExample('edcba')
    for x in a:
        print x
    print '----------'
    a.refresh()
    for x in a:
        print x
    print '=' * 30
    a = YieldIteratorExample('abcde')
    try:
        print a.next()
        print a.next()
        print a.next()
        print a.next()
        print a.next()
        print a.next()
    except StopIteration, e:
        print 'stopping', e

test_yielditeratorexample()
```

Running this example produces the following output:

```
d
b
----------
d
b
==============================
b
d
stopping
```

Notes and explanation:

- Because the _next method uses yield, calling it (actually, calling the iterator object it produces) in an iterator context causes it to be "resumed" immediately after the yield statement. This reduces bookkeeping a bit.
- However, with this style, we must explicitly produce an iterator. We do this by calling the _next method, which contains a yield statement, and is therefore a generator. The following code in our constructor (`__init__`) completes the set-up of our class as an iterator class:

  ```
  self.iterator = self._next()
  self.next = self.iterator.next
  ```

  Remember that we need both `__iter__()` and `next()` methods in `YieldIteratorExample` to satisfy the iterator protocol. The `__iter__()` method is already there and the above code in the constructor creates the `next()` method.

2.3.5 Example - A list comprehension

A list comprehension looks a bit like an iterator, but it produces a list. See: The Python Language Reference: List displays -- http://docs.python.org/reference/expressions.html#list-displays for more on list comprehensions.

Here is an example:

```
In [4]: def f(x):
   ...:     return x * 3
   ...:
In [5]: list1 = [11, 22, 33]
In [6]: list2 = [f(x) for x in list1]
In [7]: print list2
[33, 66, 99]
```

2.3.6 Example - A generator expression

A generator expression looks quite similar to a list comprehension, but is enclosed in parentheses rather than square brackets. Unlike a list comprehension, a generator expression does not produce a list; it produces an generator object. A generator object is an iterator.

For more on generator expressions, see The Python Language Reference: Generator expressions -- http://docs.python.org/reference/expressions.html#generator-expressions.

The following example uses a generator expression to produce an iterator:

```
mylist = range(10)

def f(x):
    return x*3

genexpr = (f(x) for x in mylist)

for x in genexpr:
    print x
```

Notes and explanation:

- The generator expression (f(x) for x in mylist) produces an iterator object.
- Notice that we can use the iterator object later in our code, can save it in a data structure, and can pass it to a function.

2.4 Unit Tests

Unit test and the Python unit test framework provide a convenient way to define and run tests that ensure that a Python application produces specified results.

This section, while it will not attempt to explain everything about the unit test framework, will provide examples of several straight-forward ways to construct and run tests.

Some assumptions:

- We are going to develop a software project incrementally. We will not implement and release all at once. Therefore, each time we add to our existing code base, we need a way to verify that our additions (and fixes) have not caused new problems in old code.
- Adding new code to existing code will cause problems. We need to be able to check/test for those problems at each step.
- As we add code, we need to be able to add tests for that new code, too.

2.4.1 Defining unit tests

2.4.1.1 Create a test class.

In the test class, implement a number of methods to perform your tests. Name your test methods with the prefix "test". Here is an example:

```
import unittest

class MyTest(unittest.TestCase):
    def test_one(self):
        # some test code
        pass
    def test_two(self):
        # some test code
        pass
```

Create a test harness. Here is an example:

```
import unittest

# make the test suite.
def suite():
    loader = unittest.TestLoader()
    testsuite = loader.loadTestsFromTestCase(MyTest)
    return testsuite

# Make the test suite; run the tests.
def test():
    testsuite = suite()
    runner = unittest.TextTestRunner(sys.stdout, verbosity=2)
    result = runner.run(testsuite)
```

Here is a more complete example:

```
import sys, StringIO, string
import unittest
import webserv_example_heavy_sub

# A comparison function for case-insenstive sorting.
def mycmpfunc(arg1, arg2):
    return cmp(string.lower(arg1), string.lower(arg2))

class XmlTest(unittest.TestCase):
    def test_import_export1(self):
        inFile = file('test1_in.xml', 'r')
        inContent = inFile.read()
        inFile.close()
        doc = webserv_example_heavy_sub.parseString(inContent)
        outFile = StringIO.StringIO()
        outFile.write('<?xml version="1.0" ?>\n')
        doc.export(outFile, 0)
        outContent = outFile.getvalue()
        outFile.close()
        self.failUnless(inContent == outContent)

# make the test suite.
def suite():
    loader = unittest.TestLoader()
    # Change the test method prefix: test --> trial.
    #loader.testMethodPrefix = 'trial'
    # Change the comparison function that determines the order of
tests.
    #loader.sortTestMethodsUsing = mycmpfunc
    testsuite = loader.loadTestsFromTestCase(XmlTest)
    return testsuite

# Make the test suite; run the tests.
def test_main():
    testsuite = suite()
    runner = unittest.TextTestRunner(sys.stdout, verbosity=2)
    result = runner.run(testsuite)

if __name__ == "__main__":
    test_main()
```

Running the above script produces the following output:

```
test_import_export (__main__.XmlTest) ... ok

---------------------------------------------------------------------
-
Ran 1 test in 0.035s

OK
```

A few notes on this example:

- This example tests the ability to parse an xml document test1_in.xml and export that document back to XML. The test succeeds if the input XML document and the exported XML document are the same.
- The code which is being tested parses an XML document returned by a request to Amazon Web services. You can learn more about Amazon Web services at: http://www.amazon.com/webservices. This code was generated from an XML Schema document by generateDS.py. So we are in effect, testing generateDS.py. You can find generateDS.py at: http://http://www.davekuhlman.org/#generateds-py.
- Testing for success/failure and reporting failures -- Use the methods listed at http://www.python.org/doc/current/lib/testcase-objects.html to test for and report success and failure. In our example, we used "self.failUnless(inContent == outContent)" to ensure that the content we parsed and the content that we exported were the same.
- Add additional tests by adding methods whose names have the prefix "test". If you prefer a different prefix for tests names, add something like the following to the above script:

  ```
      loader.testMethodPrefix = 'trial'
  ```

- By default, the tests are run in the order of their names sorted by the cmp function. So, if needed, you can control the order of execution of tests by selecting their names, for example, using names like test_1_checkderef, test_2_checkcalc, etc. Or, you can change the comparison function by adding something like the following to the above script:

```
    loader.sortTestMethodsUsing = mycmpfunc
```

As a bit of motivation for creating and using unit tests, while developing this example, I discovered several errors (or maybe "special features") in `generateDS.py`.

2.5 Extending and embedding Python

2.5.1 Introduction and concepts

Extending vs. embedding -- They are different but related:

- Extending Python means to implement an extension module or an extension type. An extension module creates a new Python module which is implemented in C/C++. From Python code, an extension module appears to be just like a module implemented in Python code. An extension type creates a new Python (built-in) type which is implemented in C/C++. From Python code, an extension type appears to be just like a built-in type.
- Embedding Python, by contrast, is to put the Python interpreter within an application (i.e. link it in) so that the application can run Python scripts. The scripts can be executed or triggered in a variety of ways, e.g. they can be bound to keys on the keyboard or to menu items, they can be triggered by external events, etc. Usually, in order to make the embedded Python interpreter useful, Python is also extended with functions from the embedding application, so that the scripts can call functions that are implemented by the embedding C/C++ application.

Documentation -- The two important sources for information about extending and embedding are the following:

- Extending and Embedding the Python Interpreter -- http://www.python.org/doc/current/ext/ext.html
- Python/C API Reference Manual -- http://www.python.org/doc/current/api/api.html

Types of extensions:

- Extension modules -- From the Python side, it appears to be a Python module. Usually it exports functions.
- Extension types -- Used to implement a new Python data type.
- Extension classes -- From the Python side, it appears to be a class.

Tools -- There are several tools that support the development of Python extensions:

- SWIG -- Learn about SWIG at: http://www.swig.org
- Pyrex -- Learn about Pyrex at: http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/
- There is also Cython, which seems to be an advanced version of, or at least an alternative to Pyrex. See: Cython - C Extensions for Python -- http://www.cython.org/

2.5.2 Extension modules

Writing an extension module by hand -- What to do:

- Create the "init" function -- The name of this function must be "init" followed by the name of the module. Every extension module must have such a function.
- Create the function table -- This table maps function names (referenced from Python code) to function pointers (implemented in C/C++).
- Implement each wrapper function.

Implementing a wrapper function -- What to do:

1. Capture the arguments with PyArg_ParseTuple. The format string specifies how arguments are to be converted and captured. See 1.7 Extracting Parameters in Extension Functions. Here are some of the most commonly used types:
    - Use "i", "s", "f", etc to convert and capture simple types such as integers, strings, floats, etc.
    - Use "O" to get a pointer to Python "complex" types such as lists, tuples, dictionaries, etc.
    - Use items in parentheses to capture and unpack sequences (e.g. lists and tuples) of fixed length. Example:

      ```
      if (!PyArg_ParseTuple(args, "(ii)(ii)", &x, &y,
      &width, &height))
      {
          return NULL;
      } /* if */
      ```

      A sample call might be:

      ```
      lowerLeft = (x1, y1)
      extent = (width1, height1)
      scan(lowerLeft, extent)
      ```

    - Use ":aName" (colon) at the end of the format string to provide a function name for error messages. Example:

      ```
      if (!PyArg_ParseTuple(args, "O:setContentHandler",
      &pythonInstance))
      {
      ```

```
    return NULL;
} /* if */
```

   - Use ";an error message" (semicolon) at the end of the format string to provide a string that replaces the default error message.
   - Docs are available at: http://www.python.org/doc/current/ext/parseTuple.html.
2. Write the logic.
3. Handle errors and exceptions -- You will need to understand how to (1) clearing errors and exceptions and (2) Raise errors (exceptions).
   - Many functions in the Python C API raise exceptions. You will need to check for and clear these exceptions. Here is an example:

```
char * message;
int messageNo;

message = NULL;
messageNo = -1;
/* Is the argument a string?
*/
if (! PyArg_ParseTuple(args, "s", &message))
{
    /* It's not a string.  Clear the error.
    *  Then try to get a message number (an
integer).
    */
    PyErr_Clear();
    if (! PyArg_ParseTuple(args, "i", &messageNo))
    {
        o
        o
        o
```

   - You can also raise exceptions in your C code that can be caught (in a "try:except:" block) back in the calling Python code. Here is an example:

```
if (n == 0)
{
    PyErr_SetString(PyExc_ValueError, "Value must
not be zero");
    return NULL;
}
```

     See Include/pyerrors.h in the Python source distribution for more exception/error types.
   - And, you can test whether a function in the Python C API that you have called has raised an exception. For example:

```
if (PyErr_Occurred())
{
    /* An exception was raised.
    *  Do something about it.
```

```
        */
        o
        o
        o
```

   For more documentation on errors and exceptions, see: http://www.python.org/doc/current/api/exceptionHandling.html.

4. Create and return a value:
   - For each built-in Python type there is a set of API functions to create and manipulate it. See the "Python/C API Reference Manual" for a description of these functions. For example, see:
     - http://www.python.org/doc/current/api/intObjects.html
     - http://www.python.org/doc/current/api/stringObjects.html
     - http://www.python.org/doc/current/api/tupleObjects.html
     - http://www.python.org/doc/current/api/listObjects.html
     - http://www.python.org/doc/current/api/dictObjects.html
     - Etc.
   - The reference count -- You will need to follow Python's rules for reference counting that Python uses to garbage collect objects. You can learn about these rules at http://www.python.org/doc/current/ext/refcounts.html. You will not want Python to garbage collect objects that you create too early or too late. With respect to Python objects created with the above functions, these new objects are owned and may be passed back to Python code. However, there are situations where your C/C++ code will not automatically own a reference, for example when you extract an object from a container (a list, tuple, dictionary, etc). In these cases you should increment the reference count with Py_INCREF.

2.5.3 SWIG

Note: Our discussion and examples are for SWIG version 1.3

SWIG will often enable you to generate wrappers for functions in an existing C function library. SWIG does not understand everything in C header files. But it does a fairly impressive job. You should try it first before resorting to the hard work of writing wrappers by hand.

More information on SWIG is at http://www.swig.org.

Here are some steps that you can follow:

1. Create an interface file -- Even when you are wrapping functions defined in an existing header file, creating an interface file is a good idea. Include your existing header file into it, then add whatever else you need. Here is an extremely simple example of a SWIG interface file:

```
%module MyLibrary

%{
#include "MyLibrary.h"
%}

%include "MyLibrary.h"
```

Comments:

- The "%{" and "%}" brackets are directives to SWIG. They say: "Add the code between these brackets to the generated wrapper file without processing it.
- The "%include" statement says: "Copy the file into the interface file here. In effect, you are asking SWIG to generate wrappers for all the functions in this header file. If you want wrappers for only some of the functions in a header file, then copy or reproduce function declarations for the desired functions here. An example:

  ```
  %module MyLibrary

  %{
  #include "MyLibrary.h"
  %}

  int calcArea(int width, int height);
  int calcVolume(int radius);
  ```

  This example will generate wrappers for only two functions.
- You can find more information about the directives that are used in SWIG interface files in the SWIG User Manual, in particular at:
  - http://www.swig.org/Doc1.3/Preprocessor.html
  - http://www.swig.org/Doc1.3/Python.html

2. Generate the wrappers:

   ```
   swig -python MyLibrary.i
   ```

3. Compile and link the library. On Linux, you can use something like the following:

   ```
   gcc -c MyLibrary.c
   gcc -c -I/usr/local/include/python2.3 MyLibrary_wrap.c
   gcc -shared MyLibrary.o MyLibrary_wrap.o -o
   _MyLibrary.so
   ```

   Note that we produce a shared library whose name is the module name prefixed with an underscore. SWIG also generates a .py file, without the leading underscore, which we will import from our Python code and which, in turn, imports the shared library.

4. Use the extension module in your python code:

   ```
   Python 2.3b1 (#1, Apr 25 2003, 20:36:09)
   [GCC 2.95.4 20011002 (Debian prerelease)] on linux2
   ```

```
Type "help", "copyright", "credits" or "license" for
more information.
>>> import MyLibrary
>>> MyLibrary.calcArea(4.0, 5.0)
20.0
```

Here is a makefile that will execute swig to generate wrappers, then compile and link the extension.

```
    CFLAGS = -I/usr/local/include/python2.3

    all: _MyLibrary.so

_MyLibrary.so: MyLibrary.o MyLibrary_wrap.o

   gcc -shared MyLibrary.o MyLibrary_wrap.o -o _MyLibrary.so

MyLibrary.o: MyLibrary.c

   gcc -c MyLibrary.c -o MyLibrary.o

MyLibrary_wrap.o: MyLibrary_wrap.c

   gcc -c ${CFLAGS} MyLibrary_wrap.c -o MyLibrary_wrap.o

MyLibrary_wrap.c: MyLibrary.i

   swig -python MyLibrary.i

clean:

   rm -f MyLibrary.py MyLibrary.o MyLibrary_wrap.c

      MyLibrary_wrap.o _MyLibrary.so
```

Here is an example of running this makefile:

```
$ make -f MyLibrary_makefile clean
rm -f MyLibrary.py MyLibrary.o MyLibrary_wrap.c \
        MyLibrary_wrap.o _MyLibrary.so
$ make -f MyLibrary_makefile
gcc -c MyLibrary.c -o MyLibrary.o
swig -python MyLibrary.i
gcc -c -I/usr/local/include/python2.3 MyLibrary_wrap.c -o
MyLibrary_wrap.o
gcc -shared MyLibrary.o MyLibrary_wrap.o -o _MyLibrary.so
```

And, here are C source files that can be used in our example.

MyLibrary.h:

```
/* MyLibrary.h
*/
```

```
float calcArea(float width, float height);
float calcVolume(float radius);

int getVersion();

int getMode();
```

MyLibrary.c:

```
/* MyLibrary.c
*/

float calcArea(float width, float height)
{
   return (width * height);
}

float calcVolume(float radius)
{
   return (3.14 * radius * radius);
}

int getVersion()
{
    return 123;
}

int getMode()
{
    return 1;
}
```

2.5.4 Pyrex

Pyrex is a useful tool for writing Python extensions. Because the Pyrex language is similar to Python, writing extensions in Pyrex is easier than doing so in C. Cython appears to be the a newer version of Pyrex.

More information is on Pyrex and Cython is at:

- Pyrex -- http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/
- Cython - C Extensions for Python -- http://www.cython.org/

Here is a simple function definition in Pyrex:

```
# python_201_pyrex_string.pyx

import string

def formatString(object s1, object s2):
    s1 = string.strip(s1)
    s2 = string.strip(s2)
    s3 = '<<%s||%s>>' % (s1, s2)
    s4 = s3 * 4
    return s4
```

And, here is a make file:

```
CFLAGS = -DNDEBUG -O3 -Wall -Wstrict-prototypes -fPIC \
    -I/usr/local/include/python2.3

all: python_201_pyrex_string.so

python_201_pyrex_string.so: python_201_pyrex_string.o
    gcc -shared python_201_pyrex_string.o -o 
python_201_pyrex_string.so

python_201_pyrex_string.o: python_201_pyrex_string.c
    gcc -c ${CFLAGS} python_201_pyrex_string.c -o 
python_201_pyrex_string.o

python_201_pyrex_string.c: python_201_pyrex_string.pyx
    pyrexc python_201_pyrex_string.pyx

clean:
    rm -f python_201_pyrex_string.so python_201_pyrex_string.o \
            python_201_pyrex_string.c
```

Here is another example. In this one, one function in the .pyx file calls another. Here is the implementation file:

```
# python_201_pyrex_primes.pyx

def showPrimes(int kmax):
    plist = primes(kmax)
    for p in plist:
        print 'prime: %d' % p

cdef primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] <> 0:
            i = i + 1
        if i == k:
           p[k] = n
           k = k + 1
           result.append(n)
        n = n + 1
    return result
```

And, here is a make file:

> #CFLAGS = -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC # -I/usr/local/include/python2.3 CFLAGS = -DNDEBUG -I/usr/local/include/python2.3
>
> all: python_201_pyrex_primes.so

python_201_pyrex_primes.so: python_201_pyrex_primes.o

gcc -shared python_201_pyrex_primes.o -o python_201_pyrex_primes.so

python_201_pyrex_primes.o: python_201_pyrex_primes.c

gcc -c ${CFLAGS} python_201_pyrex_primes.c -o python_201_pyrex_primes.o

python_201_pyrex_primes.c: python_201_pyrex_primes.pyx

pyrexc python_201_pyrex_primes.pyx

clean:

rm -f python_201_pyrex_primes.so python_201_pyrex_primes.o

python_201_pyrex_primes.c

Here is the output from running the makefile:

```
$ make -f python_201_pyrex_makeprimes clean
rm -f python_201_pyrex_primes.so python_201_pyrex_primes.o \
        python_201_pyrex_primes.c
$ make -f python_201_pyrex_makeprimes
pyrexc python_201_pyrex_primes.pyx
gcc -c -DNDEBUG -I/usr/local/include/python2.3
python_201_pyrex_primes.c -o python_201_pyrex_primes.o
gcc -shared python_201_pyrex_primes.o -o python_201_pyrex_primes.so
```

Here is an interactive example of its use:

```
$ python
Python 2.3b1 (#1, Apr 25 2003, 20:36:09)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import python_201_pyrex_primes
>>> dir(python_201_pyrex_primes)
['__builtins__', '__doc__', '__file__', '__name__', 'showPrimes']
>>> python_201_pyrex_primes.showPrimes(5)
prime: 2
prime: 3
prime: 5
prime: 7
```

```
prime: 11
```

This next example shows how to use Pyrex to implement a new extension type, that is a new Python built-in type. Notice that the class is declared with the cdef keyword, which tells Pyrex to generate the C implementation of a type instead of a class.

Here is the implementation file:

```
# python_201_pyrex_clsprimes.pyx

"""An implementation of primes handling class
for a demonstration of Pyrex.
"""

cdef class Primes:
    """A class containing functions for
    handling primes.
    """

    def showPrimes(self, int kmax):
        """Show a range of primes.
        Use the method primes() to generate the primes.
        """
        plist = self.primes(kmax)
        for p in plist:
            print 'prime: %d' % p

    def primes(self, int kmax):
        """Generate the primes in the range 0 - kmax.
        """
        cdef int n, k, i
        cdef int p[1000]
        result = []
        if kmax > 1000:
            kmax = 1000
        k = 0
        n = 2
        while k < kmax:
            i = 0
            while i < k and n % p[i] <> 0:
                i = i + 1
            if i == k:
               p[k] = n
               k = k + 1
               result.append(n)
            n = n + 1
        return result
```

And, here is a make file:

```
CFLAGS = -DNDEBUG -I/usr/local/include/python2.3

all: python_201_pyrex_clsprimes.so
```

```

python_201_pyrex_clsprimes.so: python_201_pyrex_clsprimes.o
    gcc -shared python_201_pyrex_clsprimes.o -o
python_201_pyrex_clsprimes.so

python_201_pyrex_clsprimes.o: python_201_pyrex_clsprimes.c
    gcc -c ${CFLAGS} python_201_pyrex_clsprimes.c -o
python_201_pyrex_clsprimes.o

python_201_pyrex_clsprimes.c: python_201_pyrex_clsprimes.pyx
    pyrexc python_201_pyrex_clsprimes.pyx

clean:
    rm -f python_201_pyrex_clsprimes.so
python_201_pyrex_clsprimes.o \
            python_201_pyrex_clsprimes.c
```

Here is output from running the makefile:

```
$ make -f python_201_pyrex_makeclsprimes clean
rm -f python_201_pyrex_clsprimes.so python_201_pyrex_clsprimes.o \
        python_201_pyrex_clsprimes.c
$ make -f python_201_pyrex_makeclsprimes
pyrexc python_201_pyrex_clsprimes.pyx
gcc -c -DNDEBUG -I/usr/local/include/python2.3
python_201_pyrex_clsprimes.c -o python_201_pyrex_clsprimes.o
gcc -shared python_201_pyrex_clsprimes.o -o
python_201_pyrex_clsprimes.so
```

And here is an interactive example of its use:

```
$ python
Python 2.3b1 (#1, Apr 25 2003, 20:36:09)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import python_201_pyrex_clsprimes
>>> dir(python_201_pyrex_clsprimes)
['Primes', '__builtins__', '__doc__', '__file__', '__name__']
>>> primes = python_201_pyrex_clsprimes.Primes()
>>> dir(primes)
['__class__', '__delattr__', '__doc__', '__getattribute__',
'__hash__',
'__init__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__', 'primes', 'showPrimes']
>>> primes.showPrimes(4)
prime: 2
prime: 3
prime: 5
prime: 7
```

Documentation -- Also notice that Pyrex preserves the documentation for the module, the class, and the methods in the class. You can show this documentation with pydoc, as follows:

```
$ pydoc python_201_pyrex_clsprimes
```

Or, in Python interactive mode, use:

```
$ python
Python 2.3b1 (#1, Apr 25 2003, 20:36:09)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import python_201_pyrex_clsprimes
>>> help(python_201_pyrex_clsprimes)
```

2.5.5 SWIG vs. Pyrex

Choose SWIG when:

- You already have an existing C or C++ implementation of the code you want to call from Python. In this case you want SWIG to generate the wrappers. But note that Cython promises to enable you to quickly wrap and call functions implemented in C.
- You want to write the implementation in C or C++ by hand. Perhaps, because you think you can do so quickly, for example, or because you believe that you can make it highly optimized. Then, you want to be able to generate the Python (extension) wrappers for it quickly.

Choose Pyrex when:

- You do not have a C/C++ implementation and you want an easier way to write that C implementation. Writing Pyrex code, which is a lot like Python, is easier than writing C or C++ code by hand).
- You start to write the implementation in C, then find that it requires lots of calls to the Python C API, and you want to avoid having to learn how to do that.

2.5.6 Cython

Here is a simple example that uses Cython to wrap a function implemented in C.

First the C header file:

```
/* test_c_lib.h */

int calculate(int width, int height);
```

And, the C implementation file:

```
/* test_c_lib.c */

```

```
#include "test_c_lib.h"

int calculate(int width, int height)
{
    int result;
    result = width * height * 3;
    return result;
}
```

Here is a Cython file that calls our C function:

```
# test_c.pyx

# Declare the external C function.
cdef extern from "test_c_lib.h":
    int calculate(int width, int height)

def test(w, h):
    # Call the external C function.
    result = calculate(w, h)
    print 'result from calculate: %d' % result
```

We can compile our code using this script (on Linux):

```
#!/bin/bash -x
cython test_c.pyx
gcc -c -fPIC -I/usr/local/include/python2.6  -o test_c.o test_c.c
gcc -c -fPIC -I/usr/local/include/python2.6  -o test_c_lib.o
test_c_lib.c
gcc -shared -fPIC -I/usr/local/include/python2.6  -o test_c.so
test_c.o test_c_lib.o
```

Here is a small Python file that uses the wrapper that we wrote in Cython:

```
# run_test_c.py

import test_c

def test():
    test_c.test(4, 5)
    test_c.test(12, 15)

if __name__ == '__main__':
    test()
```

And, when we run it, we see the following:

```
$ python run_test_c.py
result from calculate: 60
result from calculate: 540
```

2.5.7 Extension types

The goal -- A new built-in data type for Python.

Existing examples -- Objects/listobject.c, Objects/stringobject.c, Objects/dictobject.c, etc in the Python source code distribution.

In older versions of the Python source code distribution, a template for the C code was provided in Objects/xxobject.c. Objects/xxobject.c is no longer included in the Python source code distribution. However:

- The discussion and examples for creating extension types have been expanded. See: Extending and Embedding the Python Interpreter, 2. Defining New Types -- http://docs.python.org/extending/newtypes.html.
- In the Tools/framer directory of the Python source code distribution there is an application that will generate a skeleton for an extension type from a specification object written in Python. Run Tools/framer/example.py to see it in action.

And, you can use Pyrex to generate a new built-in type. To do so, implement a Python/Pyrex class and declare the class with the Pyrex keyword cdef. In fact, you may want to use Pyrex to generate a minimal extension type, and then edit that generated code to insert and add functionality by hand. See the Pyrex section for an example.

Pyrex also goes some way toward giving you access to (existing) C structs and functions from Python.

2.5.8 Extension classes

Extension classes the easy way -- SWIG shadow classes.

Start with an implementation of a C++ class and its header file.

Use the following SWIG flags:

```
swig -c++ -python mymodule.i
```

More information is available with the SWIG documentation at: http://www.swig.org/Doc1.3/Python.html.

Extension classes the Pyrex way -- An alternatie is to use Pyrex to compile a class definition that does not have the cdef keyword. Using cdef on the class tells Pyrex to generate an extension type instead of a class. You will have to determine whether you want an extension class or an extension type.

2.6 Parsing

Python is an excellent language for text analysis.

In some cases, simply splitting lines of text into words will be enough. In these cases use string.split().

In other cases, regular expressions may be able to do the parsing you need. If so, see the section on regular expressions in this document.

However, in some cases, more complex analysis of input text is required. This section describes some of the ways that Python can help you with this complex parsing and analysis.

2.6.1 Special purpose parsers

There are a number of special purpose parsers which you will find in the Python standard library:

- ConfigParser parser - Configuration file parser -- http://docs.python.org/library/configparser.html
- getopt -- Parser for command line options -- http://docs.python.org/library/getopt.html
- optparse -- More powerful command line option parser -- http://docs.python.org/library/optparse.html
- urlparse -- Parse URLs into components -- http://docs.python.org/library/urlparse.html
- csv -- CSV (comma separated values) File Reading and Writing -- http://docs.python.org/library/csv.html#module-csv
- os.path - Common pathname manipulations -- http://docs.python.org/library/os.path.html

XML parsers and XML tools -- There is lots of support for parsing and processing XML in Python. Here are a few places to look for support:

- The Python standard library -- Structured Markup Processing Tools -- http://docs.python.org/library/markup.html.
- In particular, you may be interested in xml.dom.minidom - Lightweight DOM implementation -- http://docs.python.org/library/xml.dom.minidom.html.
- ElementTree -- You can think of ElementTree as an enhanced DOM (document object model). Many find it easier to use than minidom. ElementTree is in the Python standard library, and documentation is here: ElementTree Overview -- http://effbot.org/zone/element-index.htm.
- Lxml mimics the ElementTree API, but has additional capabilities. Find out about Lxml at lxml -- http://codespeak.net/lxml/index.html -- Note that lxml also has support for XPath and XSLT.
- Dave's support for Python and XML -- http://www.rexx.com/~dkuhlman.

2.6.2 Writing a recursive descent parser by hand

For simple grammars, this is not so hard.

You will need to implement:

- A recognizer method or function for each production rule in your grammar. Each recognizer method begins looking at the current token, then consumes as many tokens as needed to recognize it's own production rule. It calls the recognizer functions for any non-terminals on its right-hand side.
- A tokenizer -- Something that will enable each recognizer function to get tokens, one by one. There are a variety of ways to do this, e.g. (1) a function that produces a list of tokens from which recognizers can pop tokens; (2) a generator whose next method returns the next token; etc.

As an example, we'll implement a recursive descent parser written in Python for the following grammer:

```
Prog ::= Command | Command Prog
Command ::= Func_call
Func_call ::= Term '(' Func_call_list ')'
Func_call_list ::= Func_call | Func_call ',' Func_call_list
Term = <word>
```

Here is an implementation of a recursive descent parser for the above grammar:

```
#!/usr/bin/env python

"""
A recursive descent parser example.

Usage:
    python rparser.py [options] <inputfile>
Options:
    -h, --help      Display this help message.
Example:
    python rparser.py myfile.txt

The grammar:
    Prog ::= Command | Command Prog
    Command ::= Func_call
    Func_call ::= Term '(' Func_call_list ')'
    Func_call_list ::= Func_call | Func_call ',' Func_call_list
    Term = <word>
"""

import sys
import string
import types
import getopt
```

```
#
# To use the IPython interactive shell to inspect your running
#   application, uncomment the following lines:
#
## from IPython.Shell import IPShellEmbed
## ipshell = IPShellEmbed((),
##     banner = '>>>>>>>> Into IPython >>>>>>>>',
##     exit_msg = '<<<<<<<< Out of IPython <<<<<<<<')
#
# Then add the following line at the point in your code where
#   you want to inspect run-time values:
#
#       ipshell('some message to identify where we are')
#
# For more information see: http://ipython.scipy.org/moin/
#

#
# Constants
#

# AST node types
NoneNodeType = 0
ProgNodeType = 1
CommandNodeType = 2
FuncCallNodeType = 3
FuncCallListNodeType = 4
TermNodeType = 5

# Token types
NoneTokType = 0
LParTokType = 1
RParTokType = 2
WordTokType = 3
CommaTokType = 4
EOFTokType = 5

# Dictionary to map node type values to node type names
NodeTypeDict = {
    NoneNodeType: 'NoneNodeType',
    ProgNodeType: 'ProgNodeType',
    CommandNodeType: 'CommandNodeType',
    FuncCallNodeType: 'FuncCallNodeType',
    FuncCallListNodeType: 'FuncCallListNodeType',
    TermNodeType: 'TermNodeType',
    }

#
# Representation of a node in the AST (abstract syntax tree).
#
class ASTNode:
    def __init__(self, nodeType, *args):
        self.nodeType = nodeType
        self.children = []
        for item in args:
            self.children.append(item)
    def show(self, level):
        self.showLevel(level)
        print 'Node -- Type %s' % NodeTypeDict[self.nodeType]
        level += 1
        for child in self.children:
            if isinstance(child, ASTNode):
                child.show(level)
            elif type(child) == types.ListType:
                for item in child:
                    item.show(level)
            else:
                self.showLevel(level)
                print 'Child:', child
    def showLevel(self, level):
        for idx in range(level):
            print '   ',

#
# The recursive descent parser class.
#   Contains the "recognizer" methods, which implement the grammar
#   rules (above), one recognizer method for each production rule.
#
class ProgParser:
    def __init__(self):
        pass

    def parseFile(self, infileName):
        self.infileName = infileName
        self.tokens = None
        self.tokenType = NoneTokType
        self.token = ''
        self.lineNo = -1
        self.infile = file(self.infileName, 'r')
        self.tokens = genTokens(self.infile)
        try:
            self.tokenType, self.token, self.lineNo = 
self.tokens.next()
        except StopIteration:
            raise RuntimeError, 'Empty file'
        result = self.prog_reco()
        self.infile.close()
        self.infile = None
        return result

    def parseStream(self, instream):
        self.tokens = genTokens(instream, '<instream>')
        try:
            self.tokenType, self.token, self.lineNo = 
self.tokens.next()
        except StopIteration:
            raise RuntimeError, 'Empty file'
        result = self.prog_reco()
        return result

    def prog_reco(self):
        commandList = []
        while 1:
            result = self.command_reco()
            if not result:
                break
            commandList.append(result)
        return ASTNode(ProgNodeType, commandList)

    def command_reco(self):
        if self.tokenType == EOFTokType:
            return None
        result = self.func_call_reco()
        return ASTNode(CommandNodeType, result)

    def func_call_reco(self):
        if self.tokenType == WordTokType:
            term = ASTNode(TermNodeType, self.token)
            self.tokenType, self.token, self.lineNo = 
self.tokens.next()
            if self.tokenType == LParTokType:
                self.tokenType, self.token, self.lineNo = 
self.tokens.next()
                result = self.func_call_list_reco()
                if result:
                    if self.tokenType == RParTokType:
                        self.tokenType, self.token, self.lineNo = \
                            self.tokens.next()
                        return ASTNode(FuncCallNodeType, term, 
result)
                    else:
                        raise ParseError(self.lineNo, 'missing right 
paren')
                else:
                    raise ParseError(self.lineNo, 'bad func call 
list')
            else:
                raise ParseError(self.lineNo, 'missing left paren')
        else:
            return None

    def func_call_list_reco(self):
        terms = []
        while 1:
            result = self.func_call_reco()
            if not result:
                break
            terms.append(result)
            if self.tokenType != CommaTokType:
                break
            self.tokenType, self.token, self.lineNo =
self.tokens.next()
        return ASTNode(FuncCallListNodeType, terms)

#
# The parse error exception class.
#
class ParseError(Exception):
    def __init__(self, lineNo, msg):
        RuntimeError.__init__(self, msg)
        self.lineNo = lineNo
        self.msg = msg
    def getLineNo(self):
        return self.lineNo
    def getMsg(self):
        return self.msg

def is_word(token):
    for letter in token:
        if letter not in string.ascii_letters:
            return None
    return 1

#
# Generate the tokens.
# Usage:
#    gen = genTokens(infile)
#    tokType, tok, lineNo = gen.next()
#    ...
def genTokens(infile):
    lineNo = 0
    while 1:
        lineNo += 1
        try:
            line = infile.next()
        except:
            yield (EOFTokType, None, lineNo)
        toks = line.split()
        for tok in toks:
            if is_word(tok):
                tokType = WordTokType
            elif tok == '(':
                tokType = LParTokType
            elif tok == ')':
                tokType = RParTokType
            elif tok == ',':
                tokType = CommaTokType
            yield (tokType, tok, lineNo)

def test(infileName):
    parser = ProgParser()
    #ipshell('(test) #1\nCtrl-D to exit')
    result = None
    try:
        result = parser.parseFile(infileName)
    except ParseError, exp:
        sys.stderr.write('ParseError: (%d) %s\n' % \
            (exp.getLineNo(), exp.getMsg()))
    if result:
        result.show(0)

def usage():
    print __doc__
    sys.exit(1)

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    relink = 1
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 1:
        usage()
    inputfile = args[0]
    test(inputfile)

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    main()
```

Comments and explanation:

- The tokenizer is a Python generator. It returns a Python generator that can produce "(tokType, tok, lineNo)" tuples. Our tokenizer is so simple-minded that we have to separate all of our tokens with whitespace. (A little later, we'll see how to use Plex to overcome this limitation.)
- The parser class (ProgParser) contains the recognizer methods that implement the production rules. Each of these methods recognizes a syntactic construct defined by a rule. In our example, these methods have names that end with "_reco".
- We could have, alternatively, implemented our recognizers as global functions, instead of as methods in a class. However, using a class gives us a place to "hang" the variables that are needed across methods and saves us from having to use ("evil") global variables.
- A recognizer method recognizes terminals (syntactic elements on the right-hand side of the grammar rule for which there is no grammar rule) by (1) checking the token type and the token value, and then (2) calling the tokenizer to get the next token (because it has consumed a token).

- A recognizer method checks for and processes a non-terminal (syntactic elements on the right-hand side for which there is a grammar rule) by calling the recognizer method that implements that non-terminal.
- If a recognizer method finds a syntax error, it raises an exception of class ParserError.
- Since our example recursive descent parser creates an AST (an abstract syntax tree), whenever a recognizer method successfully recognizes a syntactic construct, it creates an instance of class ASTNode to represent it and returns that instance to its caller. The instance of ASTNode has a node type and contains child nodes which were constructed by recognizer methods called by this one (i.e. that represent non-terminals on the right-hand side of a grammar rule).
- Each time a recognizer method "consumes a token", it calls the tokenizer to get the next token (and token type and line number).
- The tokenizer returns a token type in addition to the token value. It also returns a line number for error reporting.
- The syntax tree is constructed from instances of class ASTNode.
- The ASTNode class has a show method, which walks the AST and produces output. You can imagine that a similar method could do code generation. And, you should consider the possibility of writing analogous tree walk methods that perform tasks such as optimization, annotation of the AST, etc.

And, here is a sample of the data we can apply this parser to:

```
aaa ( )
bbb ( ccc ( ) )
ddd ( eee ( ) , fff ( ggg ( ) , hhh ( ) , iii ( ) ) )
```

And, if we run the parser on the this input data, we see:

```
$ python workbook045.py workbook045.data
Node -- Type ProgNodeType
    Node -- Type CommandNodeType
        Node -- Type FuncCallNodeType
            Node -- Type TermNodeType
                Child: aaa
            Node -- Type FuncCallListNodeType
    Node -- Type CommandNodeType
        Node -- Type FuncCallNodeType
            Node -- Type TermNodeType
                Child: bbb
            Node -- Type FuncCallListNodeType
                Node -- Type FuncCallNodeType
                    Node -- Type TermNodeType
                        Child: ccc
                    Node -- Type FuncCallListNodeType
    Node -- Type CommandNodeType
        Node -- Type FuncCallNodeType
            Node -- Type TermNodeType
```

```
                Child: ddd
            Node -- Type FuncCallListNodeType
                Node -- Type FuncCallNodeType
                    Node -- Type TermNodeType
                        Child: eee
                    Node -- Type FuncCallListNodeType
                Node -- Type FuncCallNodeType
                    Node -- Type TermNodeType
                        Child: fff
                    Node -- Type FuncCallListNodeType
                        Node -- Type FuncCallNodeType
                            Node -- Type TermNodeType
                                Child: ggg
                            Node -- Type FuncCallListNodeType
                        Node -- Type FuncCallNodeType
                            Node -- Type TermNodeType
                                Child: hhh
                            Node -- Type FuncCallListNodeType
                        Node -- Type FuncCallNodeType
                            Node -- Type TermNodeType
                                Child: iii
                            Node -- Type FuncCallListNodeType
```

2.6.3 Creating a lexer/tokenizer with Plex

Lexical analysis -- The tokenizer in our recursive descent parser example was (for demonstration purposes) overly simple. You can always write more complex tokenizers by hand. However, for more complex (and real) tokenizers, you may want to use a tool to build your tokenizer.

In this section we'll describe Plex and use it to produce a tokenizer for our recursive descent parser.

You can obtain Plex at http://www.cosc.canterbury.ac.nz/~greg/python/Plex/.

In order to use it, you may want to add Plex-1.1.4/Plex to your PYTHONPATH.

Here is a simple example from the Plex tutorial:

```
#!/usr/bin/env python

"""
Sample Plex lexer

Usage:
    python plex_example.py inputfile
"""

import sys
import Plex
```

```
def count_lines(scanner, text):
    scanner.line_count += 1
    print '-' * 60

def test(infileName):
    letter = Plex.Range("AZaz")
    digit =  Plex.Range("09")
    name = letter +  Plex.Rep(letter | digit)
    number =  Plex.Rep1(digit)
    space =  Plex.Any(" \t")
    endline = Plex.Str('\n')
    #comment =  Plex.Str('"') +  Plex.Rep( Plex.AnyBut('"')) +
Plex.Str('"')
    resword =  Plex.Str("if", "then", "else", "end")
    lexicon =  Plex.Lexicon([
        (endline,               count_lines),
        (resword,               'keyword'),
        (name,                  'ident'),
        (number,                'int'),
        ( Plex.Any("+-*/=<>"),  'operator'),
        (space,                 Plex.IGNORE),
        #(comment,               'comment'),
        (Plex.Str('('),         'lpar'),
        (Plex.Str(')'),         'rpar'),
        # comments surrounded by (* and *)
        (Plex.Str("(*"),        Plex.Begin('comment')),
        Plex.State('comment', [
            (Plex.Str("*)"), Plex.Begin('')),
            (Plex.AnyChar,   Plex.IGNORE),
            ]),
    ])
    infile = open(infileName, "r")
    scanner =  Plex.Scanner(lexicon, infile, infileName)
    scanner.line_count = 0
    while True:
        token = scanner.read()
        if token[0] is None:
            break
        position = scanner.position()
        posstr = ('(%d, %d)' % (position[1],
position[2], )).ljust(10)
        tokstr = '"%s"' % token[1]
        tokstr = tokstr.ljust(20)
        print '%s tok: %s tokType: %s' % (posstr, tokstr, token[0],)
    print 'line_count: %d' % scanner.line_count


def usage():
    print __doc__
    sys.exit(1)

def main():
    args = sys.argv[1:]
    if len(args) != 1:
        usage()
    infileName = args[0]
    test(infileName)

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    main()
```

Here is a bit of data on which we can use the above lexer:

```
mass = (height * (* some comment *) width * depth) / density
totalmass = totalmass + mass
```

And, when we apply the above test program to this data, here is what we see:

```
$ python plex_example.py plex_example.data
(1,  0)    tok: "mass"               tokType: ident
(1,  5)    tok: "="                  tokType: operator
(1,  7)    tok: "("                  tokType: lpar
(1,  8)    tok: "height"             tokType: ident
(1, 15)    tok: "*"                  tokType: operator
(1, 36)    tok: "width"              tokType: ident
(1, 42)    tok: "*"                  tokType: operator
(1, 44)    tok: "depth"              tokType: ident
(1, 49)    tok: ")"                  tokType: rpar
(1, 51)    tok: "/"                  tokType: operator
(1, 53)    tok: "density"            tokType: ident
------------------------------------------------------------
(2,  0)    tok: "totalmass"          tokType: ident
(2, 10)    tok: "="                  tokType: operator
(2, 12)    tok: "totalmass"          tokType: ident
(2, 22)    tok: "+"                  tokType: operator
(2, 24)    tok: "mass"               tokType: ident
------------------------------------------------------------
line_count: 2
```

Comments and explanation:

- Create a lexicon from scanning patterns.
- See the Plex tutorial and reference (and below) for more information on how to construct the patterns that match various tokens.
- Create a scanner with a lexicon, an input file, and an input file name.
- The call "scanner.read()" gets the next token. It returns a tuple containing (1) the token value and (2) the token type.
- The call "scanner.position()" gets the position of the current token. It returns a tuple containing (1) the input file name, (2) the line number, and (3) the column number.
- We can execute a method when a given token is found by specifying the function as the token action. In our example, the function is count_lines. Maintaining a line

count is actually unneeded, since the position gives us this information. However, notice how we are able to maintain a value (in our case `line_count`) as an attribute of the scanner.

And, here are some comments on constructing the patterns used in a lexicon:

- `Plex.Range` constructs a pattern that matches any character in the range.
- `Plex.Rep` constructs a pattern that matches a sequence of zero or more items.
- `Plex.Rep1` constructs a pattern that matches a sequence of one or more items.
- `pat1 + pat2` constructs a pattern that matches a sequence containing `pat1` followed by `pat2`.
- `pat1 | pat2` constructs a pattern that matches either `pat1` or `pat2`.
- `Plex.Any` constructs a pattern that matches any one character in its argument.

Now let's revisit our recursive descent parser, this time with a tokenizer built with Plex. The tokenizer is trivial, but will serve as an example of how to hook it into a parser:

```
#!/usr/bin/env python

"""
A recursive descent parser example using Plex.
This example uses Plex to implement a tokenizer.

Usage:
    python python_201_rparser_plex.py [options] <inputfile>
Options:
    -h, --help      Display this help message.
Example:
    python python_201_rparser_plex.py myfile.txt

The grammar:

    Prog ::= Command | Command Prog
    Command ::= Func_call
    Func_call ::= Term '(' Func_call_list ')'
    Func_call_list ::= Func_call | Func_call ',' Func_call_list
    Term = <word>

"""

import sys, string, types
import getopt
import Plex

## from IPython.Shell import IPShellEmbed
## ipshell = IPShellEmbed((),
##     banner = '>>>>>>>> Into IPython >>>>>>>>',
##     exit_msg = '<<<<<<<< Out of IPython <<<<<<<<')

#
# Constants
#
```

```
# AST node types
NoneNodeType =         0
ProgNodeType =         1
CommandNodeType =      2
FuncCallNodeType =     3
FuncCallListNodeType = 4
TermNodeType =         5

# Token types
NoneTokType =  0
LParTokType =  1
RParTokType =  2
WordTokType =  3
CommaTokType = 4
EOFTokType =   5

# Dictionary to map node type values to node type names
NodeTypeDict = {
    NoneNodeType:         'NoneNodeType',
    ProgNodeType:         'ProgNodeType',
    CommandNodeType:      'CommandNodeType',
    FuncCallNodeType:     'FuncCallNodeType',
    FuncCallListNodeType: 'FuncCallListNodeType',
    TermNodeType:         'TermNodeType',
    }

#
# Representation of a node in the AST (abstract syntax tree).
#
class ASTNode:
    def __init__(self, nodeType, *args):
        self.nodeType = nodeType
        self.children = []
        for item in args:
            self.children.append(item)
    def show(self, level):
        self.showLevel(level)
        print 'Node -- Type %s' % NodeTypeDict[self.nodeType]
        level += 1
        for child in self.children:
            if isinstance(child, ASTNode):
                child.show(level)
            elif type(child) == types.ListType:
                for item in child:
                    item.show(level)
            else:
                self.showLevel(level)
                print 'Child:', child
    def showLevel(self, level):
        for idx in range(level):
            print '   ',
#
# The recursive descent parser class.
#   Contains the "recognizer" methods, which implement the grammar
#   rules (above), one recognizer method for each production rule.
#
class ProgParser:
    def __init__(self):
        self.tokens = None
        self.tokenType = NoneTokType
        self.token = ''
        self.lineNo = -1
        self.infile = None
        self.tokens = None

    def parseFile(self, infileName):
        self.tokens = None
        self.tokenType = NoneTokType
        self.token = ''
        self.lineNo = -1
        self.infile = file(infileName, 'r')
        self.tokens = genTokens(self.infile, infileName)
        try:
            self.tokenType, self.token, self.lineNo =
self.tokens.next()
        except StopIteration:
            raise RuntimeError, 'Empty file'
        result = self.prog_reco()
        self.infile.close()
        self.infile = None
        return result

    def parseStream(self, instream):
        self.tokens = None
        self.tokenType = NoneTokType
        self.token = ''
        self.lineNo = -1
        self.tokens = genTokens(self.instream, '<stream>')
        try:
            self.tokenType, self.token, self.lineNo =
self.tokens.next()
        except StopIteration:
            raise RuntimeError, 'Empty stream'
        result = self.prog_reco()
        self.infile.close()
        self.infile = None
        return result

    def prog_reco(self):
        commandList = []
        while 1:
            result = self.command_reco()
            if not result:
                break
            commandList.append(result)
        return ASTNode(ProgNodeType, commandList)

    def command_reco(self):
        if self.tokenType == EOFTokType:
            return None
        result = self.func_call_reco()
        return ASTNode(CommandNodeType, result)

    def func_call_reco(self):
        if self.tokenType == WordTokType:
            term = ASTNode(TermNodeType, self.token)
            self.tokenType, self.token, self.lineNo = 
self.tokens.next()
            if self.tokenType == LParTokType:
                self.tokenType, self.token, self.lineNo = 
self.tokens.next()
                result = self.func_call_list_reco()
                if result:
                    if self.tokenType == RParTokType:
                        self.tokenType, self.token, self.lineNo = \
                            self.tokens.next()
                        return ASTNode(FuncCallNodeType, term, 
result)
                    else:
                        raise ParseError(self.lineNo, 'missing right 
paren')
                else:
                    raise ParseError(self.lineNo, 'bad func call 
list')
            else:
                raise ParseError(self.lineNo, 'missing left paren')
        else:
            return None

    def func_call_list_reco(self):
        terms = []
        while 1:
            result = self.func_call_reco()
            if not result:
                break
            terms.append(result)
            if self.tokenType != CommaTokType:
                break
            self.tokenType, self.token, self.lineNo = 
self.tokens.next()
        return ASTNode(FuncCallListNodeType, terms)

#
# The parse error exception class.
#
class ParseError(Exception):
    def __init__(self, lineNo, msg):
        RuntimeError.__init__(self, msg)
        self.lineNo = lineNo
        self.msg = msg
    def getLineNo(self):
        return self.lineNo
    def getMsg(self):
        return self.msg

#
# Generate the tokens.
# Usage - example
#    gen = genTokens(infile)
#    tokType, tok, lineNo = gen.next()
#    ...
def genTokens(infile, infileName):
    letter = Plex.Range("AZaz")
    digit =  Plex.Range("09")
    name = letter +  Plex.Rep(letter | digit)
    lpar = Plex.Str('(')
    rpar = Plex.Str(')')
    comma = Plex.Str(',')
    comment = Plex.Str("#") + Plex.Rep(Plex.AnyBut("\n"))
    space = Plex.Any(" \t\n")
    lexicon = Plex.Lexicon([
        (name,      'word'),
        (lpar,      'lpar'),
        (rpar,      'rpar'),
        (comma,     'comma'),
        (comment,   Plex.IGNORE),
        (space,     Plex.IGNORE),
    ])
    scanner = Plex.Scanner(lexicon, infile, infileName)
    while 1:
        tokenType, token = scanner.read()
        name, lineNo, columnNo = scanner.position()
        if tokenType == None:
            tokType = EOFTokType
            token = None
        elif tokenType == 'word':
            tokType = WordTokType
        elif tokenType == 'lpar':
            tokType = LParTokType
        elif tokenType == 'rpar':
            tokType = RParTokType
        elif tokenType == 'comma':
            tokType = CommaTokType
        else:
            tokType = NoneTokType
        tok = token
        yield (tokType, tok, lineNo)

def test(infileName):
    parser = ProgParser()
    #ipshell('(test) #1\nCtrl-D to exit')
    result = None
    try:
        result = parser.parseFile(infileName)
    except ParseError, exp:
        sys.stderr.write('ParseError: (%d) %s\n' % \
            (exp.getLineNo(), exp.getMsg()))
    if result:
        result.show(0)

def usage():
    print __doc__
    sys.exit(-1)

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 1:
        usage()
    infileName = args[0]
    test(infileName)

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    main()
```

And, here is a sample of the data we can apply this parser to:

```
# Test for recursive descent parser and Plex.
# Command #1
aaa()
# Command #2
bbb (ccc())    # An end of line comment.
# Command #3
ddd(eee(), fff(ggg(), hhh(), iii()))
# End of test
```

And, when we run our parser, it produces the following:

```
$ python plex_recusive.py plex_recusive.data
Node -- Type ProgNodeType
    Node -- Type CommandNodeType
        Node -- Type FuncCallNodeType
            Node -- Type TermNodeType
                Child: aaa
            Node -- Type FuncCallListNodeType
    Node -- Type CommandNodeType
```

```
        Node -- Type FuncCallNodeType
            Node -- Type TermNodeType
                Child: bbb
            Node -- Type FuncCallListNodeType
                Node -- Type FuncCallNodeType
                    Node -- Type TermNodeType
                        Child: ccc
                    Node -- Type FuncCallListNodeType
    Node -- Type CommandNodeType
        Node -- Type FuncCallNodeType
            Node -- Type TermNodeType
                Child: ddd
            Node -- Type FuncCallListNodeType
                Node -- Type FuncCallNodeType
                    Node -- Type TermNodeType
                        Child: eee
                    Node -- Type FuncCallListNodeType
                Node -- Type FuncCallNodeType
                    Node -- Type TermNodeType
                        Child: fff
                    Node -- Type FuncCallListNodeType
                        Node -- Type FuncCallNodeType
                            Node -- Type TermNodeType
                                Child: ggg
                            Node -- Type FuncCallListNodeType
                        Node -- Type FuncCallNodeType
                            Node -- Type TermNodeType
                                Child: hhh
                            Node -- Type FuncCallListNodeType
                        Node -- Type FuncCallNodeType
                            Node -- Type TermNodeType
                                Child: iii
                            Node -- Type FuncCallListNodeType
```

Comments:

- We can now put comments in our input, and they will be ignored. Comments begin with a "#" and continue to the end of line. See the definition of comment in function genTokens.
- This tokenizer does not require us to separate tokens with whitespace as did the simple tokenizer in the earlier version of our recursive descent parser.
- The changes we made over the earlier version were to:
  1. Import Plex.
  2. Replace the definition of the tokenizer function genTokens.
  3. Change the call to genTokens so that the call passes in the file name, which is needed to create the scanner.
- Our new version of genTokens does the following:
  1. Create patterns for scanning.
  2. Create a lexicon (an instance of Plex.Lexicon), which uses the patterns.

3. Create a scanner (an instance of Plex.Scanner), which uses the lexicon.
4. Execute a loop that reads tokens (from the scanner) and "yields" each one.

2.6.4 A survey of existing tools

For complex parsing tasks, you may want to consider the following tools:

- kwParsing -- A parser generator in Python -- http://gadfly.sourceforge.net/kwParsing.html
- PLY -- Python Lex-Yacc -- http://systems.cs.uchicago.edu/ply/
- PyLR -- Fast LR parsing in python -- http://starship.python.net/crew/scott/PyLR.html
- Yapps -- The Yapps Parser Generator System -- http://theory.stanford.edu/~amitp/Yapps/

And, for lexical analysis, you may also want to look here:

- Using Regular Expressions for Lexical Analysis -- http://effbot.org/zone/xml-scanner.htm
- Plex -- http://www.cosc.canterbury.ac.nz/~greg/python/Plex/.

In the sections below, we give examples and notes about the use of PLY and pyparsing.

2.6.5 Creating a parser with PLY

In this section we will show how to implement our parser example with PLY.

First down-load PLY. It is available here: PLY (Python Lex-Yacc) -- http://www.dabeaz.com/ply/

Then add the PLY directory to your PYTHONPATH.

Learn how to construct lexers and parsers with PLY by reading `doc/ply.html` in the distribution of PLY and by looking at the examples in the distribution.

For those of you who want a more complex example, see A Python Parser for the RELAX NG Compact Syntax, which is implemented with PLY.

Now, here is our example parser. Comments and explanations are below:

```
#!/usr/bin/env python

"""
A parser example.
This example uses PLY to implement a lexer and parser.

The grammar:

    Prog ::= Command*
    Command ::= Func_call
    Func_call ::= Term '(' Func_call_list ')'
    Func_call_list ::= Func_call*
    Term = <word>

Here is a sample "program" to use as input:

    # Test for recursive descent parser and Plex.
    # Command #1
    aaa()
    # Command #2
    bbb (ccc())    # An end of line comment.
    # Command #3
    ddd(eee(), fff(ggg(), hhh(), iii()))
    # End of test
"""

import sys
import types
import getopt
import ply.lex as lex
import ply.yacc as yacc

#
# Globals
#

startlinepos = 0

#
# Constants
#

# AST node types
NoneNodeType =         0
ProgNodeType =         1
CommandNodeType =      2
CommandListNodeType =  3
FuncCallNodeType =     4
FuncCallListNodeType = 5
TermNodeType =         6

# Dictionary to map node type values to node type names
NodeTypeDict = {
    NoneNodeType:         'NoneNodeType',
    ProgNodeType:         'ProgNodeType',
    CommandNodeType:      'CommandNodeType',
    CommandListNodeType:  'CommandListNodeType',
    FuncCallNodeType:     'FuncCallNodeType',
    FuncCallListNodeType: 'FuncCallListNodeType',
    TermNodeType:         'TermNodeType',
    }

#
```

```
# Representation of a node in the AST (abstract syntax tree).
#
class ASTNode:
    def __init__(self, nodeType, *args):
        self.nodeType = nodeType
        self.children = []
        for item in args:
            self.children.append(item)
    def append(self, item):
        self.children.append(item)
    def show(self, level):
        self.showLevel(level)
        print 'Node -- Type: %s' % NodeTypeDict[self.nodeType]
        level += 1
        for child in self.children:
            if isinstance(child, ASTNode):
                child.show(level)
            elif type(child) == types.ListType:
                for item in child:
                    item.show(level)
            else:
                self.showLevel(level)
                print 'Value:', child
    def showLevel(self, level):
        for idx in range(level):
            print '   ',

#
# Exception classes
#
class LexerError(Exception):
    def __init__(self, msg, lineno, columnno):
        self.msg = msg
        self.lineno = lineno
        self.columnno = columnno
    def show(self):
        sys.stderr.write('Lexer error (%d, %d) %s\n' % \
            (self.lineno, self.columnno, self.msg))

class ParserError(Exception):
    def __init__(self, msg, lineno, columnno):
        self.msg = msg
        self.lineno = lineno
        self.columnno = columnno
    def show(self):
        sys.stderr.write('Parser error (%d, %d) %s\n' % \
            (self.lineno, self.columnno, self.msg))

#
# Lexer specification
#
tokens = (
    'NAME',
    'LPAR','RPAR',
    'COMMA',
    )

# Tokens

t_LPAR =   r'\('
t_RPAR =   r'\)'
t_COMMA =  r'\,'
t_NAME =   r'[a-zA-Z_][a-zA-Z0-9_]*'

# Ignore whitespace
t_ignore = ' \t'

# Ignore comments ('#' to end of line)
def t_COMMENT(t):
    r'\#[^\n]*'
    pass

def t_newline(t):
    r'\n+'
    global startlinepos
    startlinepos = t.lexer.lexpos - 1
    t.lineno += t.value.count("\n")

def t_error(t):
    global startlinepos
    msg = "Illegal character '%s'" % (t.value[0])
    columnno = t.lexer.lexpos - startlinepos
    raise LexerError(msg, t.lineno, columnno)

#
# Parser specification
#
def p_prog(t):
    'prog : command_list'
    t[0] = ASTNode(ProgNodeType, t[1])

def p_command_list_1(t):
    'command_list : command'
    t[0] = ASTNode(CommandListNodeType, t[1])

def p_command_list_2(t):
    'command_list : command_list command'
    t[1].append(t[2])
    t[0] = t[1]

def p_command(t):
    'command : func_call'
    t[0] = ASTNode(CommandNodeType, t[1])

def p_func_call_1(t):
    'func_call : term LPAR RPAR'
    t[0] = ASTNode(FuncCallNodeType, t[1])

def p_func_call_2(t):
    'func_call : term LPAR func_call_list RPAR'
    t[0] = ASTNode(FuncCallNodeType, t[1],  t[3])

def p_func_call_list_1(t):
    'func_call_list : func_call'
    t[0] = ASTNode(FuncCallListNodeType, t[1])

def p_func_call_list_2(t):
    'func_call_list : func_call_list COMMA func_call'
    t[1].append(t[3])
    t[0] = t[1]

def p_term(t):
    'term : NAME'
    t[0] = ASTNode(TermNodeType, t[1])

def p_error(t):
    global startlinepos
    msg = "Syntax error at '%s'" % t.value
    columnno = t.lexer.lexpos - startlinepos
    raise ParserError(msg, t.lineno, columnno)

#
# Parse the input and display the AST (abstract syntax tree)
#
def parse(infileName):
    startlinepos = 0
    # Build the lexer
    lex.lex(debug=1)
    # Build the parser
    yacc.yacc()
    # Read the input
    infile = file(infileName, 'r')
    content = infile.read()
    infile.close()
    try:
        # Do the parse
        result = yacc.parse(content)
        # Display the AST
        result.show(0)
    except LexerError, exp:
        exp.show()
    except ParserError, exp:
        exp.show()

USAGE_TEXT = __doc__

def usage():
    print USAGE_TEXT
    sys.exit(-1)
```

```

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    relink = 1
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 1:
        usage()
    infileName = args[0]
    parse(infileName)

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    main()
```

Applying this parser to the following input:

```
# Test for recursive descent parser and Plex.
# Command #1
aaa()
# Command #2
bbb (ccc())    # An end of line comment.
# Command #3
ddd(eee(), fff(ggg(), hhh(), iii()))
# End of test
```

produces the following output:

```
Node -- Type: ProgNodeType
    Node -- Type: CommandListNodeType
        Node -- Type: CommandNodeType
            Node -- Type: FuncCallNodeType
                Node -- Type: TermNodeType
                    Value: aaa
        Node -- Type: CommandNodeType
            Node -- Type: FuncCallNodeType
                Node -- Type: TermNodeType
                    Value: bbb
                Node -- Type: FuncCallListNodeType
                    Node -- Type: FuncCallNodeType
                        Node -- Type: TermNodeType
                            Value: ccc
        Node -- Type: CommandNodeType
            Node -- Type: FuncCallNodeType
                Node -- Type: TermNodeType
                    Value: ddd
                Node -- Type: FuncCallListNodeType
                    Node -- Type: FuncCallNodeType
```

```
                        Node -- Type: TermNodeType
                            Value: eee
                    Node -- Type: FuncCallNodeType
                        Node -- Type: TermNodeType
                            Value: fff
                        Node -- Type: FuncCallListNodeType
                            Node -- Type: FuncCallNodeType
                                Node -- Type: TermNodeType
                                    Value: ggg
                            Node -- Type: FuncCallNodeType
                                Node -- Type: TermNodeType
                                    Value: hhh
                            Node -- Type: FuncCallNodeType
                                Node -- Type: TermNodeType
                                    Value: iii
```

Comments and explanation:

- Creating the syntax tree -- Basically, each rule (1) recognizes a non-terminal, (2) creates a node (possibly using the values from the right-hand side of the rule), and (3) returns the node by setting the value of t[0]. A deviation from this is the processing of sequences, discussed below.
- Sequences -- p_command_list_1 and p_command_list_1 show how to handle sequences of items. In this case:
    - p_command_list_1 recognizes a command and creates an instance of ASTNode with type CommandListNodeType and adds the command to it as a child, and
    - p_command_list_2 recognizes an additional command and adds it (as a child) to the instance of ASTNode that represents the list.
- Distinguishing between different forms of the same rule -- In order to process alternatives to the same production rule differently, we use different functions with different implementations. For example, we use:
    - p_func_call_1 to recognize and process "func_call : term LPAR RPAR" (a function call without arguments), and
    - p_func_call_2 to recognize and process "func_call : term LPAR func_call_list RPAR" (a function call with arguments).
- Reporting errors -- Our parser reports the first error and quits. We've done this by raising an exception when we find an error. We implement two exception classes: LexerError and ParserError. Implementing more than one exception class enables us to distinguish between different classes of errors (note the multiple except: clauses on the try: statement in function parse). And, we use an instance of the exception class as a container in order to "bubble up" information about the error (e.g. a message, a line number, and a column number).

2.6.6 Creating a parser with pyparsing

pyparsing is a relatively new parsing package for Python. It was implemented and is supported by Paul McGuire and it shows promise. It appears especially easy to use and seems especially appropriate in particular for quick parsing tasks, although it has features that make some complex parsing tasks easy. It follows a very natural Python style for constructing parsers.

Good documentation comes with the pyparsing distribution. See file HowToUseParsing.html. So, I won't try to repeat that here. What follows is an attempt to provide several quick examples to help you solve simple parsing tasks as quickly as possible.

You will also want to look at the samples in the examples directory, which are very helpful. My examples below are fairly simple. You can see more of the ability of pyparsing to handle complex tasks in the examples.

Where to get it - You can find pyparsing at: Pyparsing Wiki Home -- http://pyparsing.wikispaces.com/

How to install it - Put the pyparsing module somewhere on your PYTHONPATH.

And now, here are a few examples.

2.6.6.1 Parsing comma-delimited lines

Note: This example is for demonstration purposes only. If you really to need to parse comma delimited fields, you can probably do so much more easily with the `CSV` (comma separated values) module in the Python standard library.

Here is a simple grammar for lines containing fields separated by commas:

```
import sys
from pyparsing import alphanums, ZeroOrMore, Word

fieldDef = Word(alphanums)
lineDef = fieldDef + ZeroOrMore("," + fieldDef)

def test():
    args = sys.argv[1:]
    if len(args) != 1:
        print 'usage: python pyparsing_test1.py <datafile.txt>'
        sys.exit(-1)
    infilename = sys.argv[1]
    infile = file(infilename, 'r')
    for line in infile:
        fields = lineDef.parseString(line)
        print fields
```

```
test()
```

Here is some sample data:

```
abcd,defg
11111,22222,33333
```

And, when we run our parser on this data file, here is what we see:

```
$ python comma_parser.py sample1.data
['abcd', ',', 'defg']
['11111', ',', '22222', ',', '33333']
```

Notes and explanation:

- Note how the grammar is constructed from normal Python calls to function and object/class constructors. I've constructed the parser in-line because my example is simple, but constructing the parser in a function or even a module might make sense for more complex grammars. pyparsing makes it easy to use these these different styles.
- Use "+" to specify a sequence. In our example, a `lineDef` is a `fieldDef` followed by ....
- Use `ZeroOrMore` to specify repetition. In our example, a `lineDef` is a `fieldDef` followed by zero or more occurances of comma and `fieldDef`. There is also `OneOrMore` when you want to require at least one occurance.
- Parsing comma delimited text happens so frequently that pyparsing provides a shortcut. Replace:

  ```
      lineDef = fieldDef + ZeroOrMore("," + fieldDef)
  ```

  with:

  ```
      lineDef = delimitedList(fieldDef)
  ```

  And note that delimitedList takes an optional argument `delim` used to specify the delimiter. The default is a comma.

2.6.6.2 Parsing functors

This example parses expressions of the form `func(arg1, arg2, arg3)`:

```
from pyparsing import Word, alphas, alphanums, nums, ZeroOrMore,
Literal

lparen = Literal("(")
rparen = Literal(")")
identifier = Word(alphas, alphanums + "_")
integer  = Word( nums )
functor = identifier
arg = identifier | integer
```

```
args = arg + ZeroOrMore("," + arg)
expression = functor + lparen + args + rparen

def test():
    content = raw_input("Enter an expression: ")
    parsedContent = expression.parseString(content)
    print parsedContent

test()
```

Explanation:

- Use Literal to specify a fixed string that is to be matched exactly. In our example, a lparen is a `(`.
- Word takes an optional second argument. With a single (string) argument, it matches any contiguous word made up of characters in the string. With two (string) arguments it matches a word whose first character is in the first string and whose remaining characters are in the second string. So, our definition of identifier matches a word whose first character is an alpha and whose remaining characters are alpha-numerics or underscore. As another example, you can think of Word("0123456789") as analogous to a regexp containing the pattern "[0-9]+".
- Use a vertical bar for alternation. In our example, an arg can be either an identifier or an integer.

2.6.6.3 Parsing names, phone numbers, etc.

This example parses expressions having the following form:

```
Input format:
[name]         [phone]       [city, state zip]
Last, first    111-222-3333  city, ca 99999
```

Here is the parser:

```
import sys
from pyparsing import alphas, nums, ZeroOrMore, Word, Group,
Suppress, Combine

lastname = Word(alphas)
firstname = Word(alphas)
city = Group(Word(alphas) + ZeroOrMore(Word(alphas)))
state = Word(alphas, exact=2)
zip = Word(nums, exact=5)

name = Group(lastname + Suppress(",") + firstname)
phone = Combine(Word(nums, exact=3) + "-" + Word(nums, exact=3) + "-"
+ Word(nums, exact=4))
location = Group(city + Suppress(",") + state + zip)

record = name + phone + location
```

```
def test():
    args = sys.argv[1:]
    if len(args) != 1:
        print 'usage: python pyparsing_test3.py <datafile.txt>'
        sys.exit(-1)
    infilename = sys.argv[1]
    infile = file(infilename, 'r')
    for line in infile:
        line = line.strip()
        if line and line[0] != "#":
            fields = record.parseString(line)
            print fields

test()
```

And, here is some sample input:

```
Jabberer, Jerry           111-222-3333   Bakersfield, CA 95111
Kackler, Kerry            111-222-3334   Fresno, CA 95112
Louderdale, Larry         111-222-3335   Los Angeles, CA 94001
```

Here is output from parsing the above input:

```
[['Jabberer', 'Jerry'], '111-222-3333', [['Bakersfield'], 'CA',
'95111']]
[['Kackler', 'Kerry'], '111-222-3334', [['Fresno'], 'CA', '95112']]
[['Louderdale', 'Larry'], '111-222-3335', [['Los', 'Angeles'], 'CA',
'94001']]
```

Comments:

- We use the `len=n` argument to the Word constructor to restict the parser to accepting a specific number of characters, for example in the zip code and phone number. Word also accepts ```min=n'' and ``max=n``` to enable you to restrict the length of a word to within a range.
- We use Group to group the parsed results into sub-lists, for example in the definition of city and name. Group enables us to organize the parse results into simple parse trees.
- We use Combine to join parsed results back into a single string. For example, in the phone number, we can require dashes and yet join the results back into a single string.
- We use Suppress to remove unneeded sub-elements from parsed results. For example, we do not need the comma between last and first name.

2.6.6.4 A more complex example

This example (thanks to Paul McGuire) parses a more complex structure and produces a dictionary.

Here is the code:

```
from pyparsing import Literal, Word, Group, Dict, ZeroOrMore, alphas,
nums,\
    delimitedList
import pprint

testData = """
+-------+------+------+------+------+------+------+------+------+
|       |  A1  |  B1  |  C1  |  D1  |  A2  |  B2  |  C2  |  D2  |
+=======+======+======+======+======+======+======+======+======+
| min   |   7  |  43  |   7  |  15  |  82  |  98  |   1  |  37  |
| max   |  11  |  52  |  10  |  17  |  85  | 112  |   4  |  39  |
| ave   |   9  |  47  |   8  |  16  |  84  | 106  |   3  |  38  |
| sdev  |   1  |   3  |   1  |   1  |   1  |   3  |   1  |   1  |
+-------+------+------+------+------+------+------+------+------+
"""

# Define grammar for datatable
heading = (Literal(
"+-------+------+------+------+------+------+------+------+------+")
+
"|       |  A1  |  B1  |  C1  |  D1  |  A2  |  B2  |  C2  |  D2  |" +
"+=======+======+======+======+======+======+======+======+======+").
suppress()

vert = Literal("|").suppress()
number = Word(nums)
rowData = Group( vert + Word(alphas) + vert +
delimitedList(number,"|") +
vert )
trailing = Literal(
"+-------+------+------+------+------+------+------+------+------+").
suppress()

datatable = heading + Dict( ZeroOrMore(rowData) ) + trailing

def main():
    # Now parse data and print results
    data = datatable.parseString(testData)
    print "data:", data
    print "data.asList():",
    pprint.pprint(data.asList())
    print "data keys:", data.keys()
    print "data['min']:", data['min']
    print "data.max:", data.max

if __name__ == '__main__':
    main()
```

When we run this, it produces the following:

```
data: [['min', '7', '43', '7', '15', '82', '98', '1', '37'],
 ['max', '11', '52', '10', '17', '85', '112', '4', '39'],
 ['ave', '9', '47', '8', '16', '84', '106', '3', '38'],
 ['sdev', '1', '3', '1', '1', '1', '3', '1', '1']]
data.asList():[['min', '7', '43', '7', '15', '82', '98', '1', '37'],
 ['max', '11', '52', '10', '17', '85', '112', '4', '39'],
 ['ave', '9', '47', '8', '16', '84', '106', '3', '38'],
 ['sdev', '1', '3', '1', '1', '1', '3', '1', '1']]
data keys: ['ave', 'min', 'sdev', 'max']
data['min']: ['7', '43', '7', '15', '82', '98', '1', '37']
data.max: ['11', '52', '10', '17', '85', '112', '4', '39']
```

Notes:

- Note the use of Dict to create a dictionary. The print statements show how to get at the items in the dictionary.
- Note how we can also get the parse results as a list by using method asList.
- Again, we use suppress to remove unneeded items from the parse results.

2.7 GUI Applications

2.7.1 Introduction

This section will help you to put a GUI (graphical user interface) in your Python program.

We will use a particular GUI library: PyGTK. We've chosen this because it is reasonably light-weight and our goal is to embed light-weight GUI interfaces in an (possibly) existing application.

For simpler GUI needs, consider EasyGUI, which is also described below.

For more heavy-weight GUI needs (for example, complete GUI applications), you may want to explore WxPython. See the WxPython home page at: http://www.wxpython.org/

2.7.2 PyGtk

Information about PyGTK is here: The PyGTK home page -- http://www.pygtk.org//.

2.7.2.1 A simple message dialog box

In this section we explain how to pop up a simple dialog box from your Python application.

To do this, do the following:

1. Import gtk into your Python module.
2. Define the dialog and its behavior.

3. Create an instance of the dialog.
4. Run the event loop.

Here is a sample that displays a message box:

```
#!/usr/bin/env python

import sys
import getopt
import gtk

class MessageBox(gtk.Dialog):
    def __init__(self, message="", buttons=(), pixmap=None,
            modal= True):
        gtk.Dialog.__init__(self)
        self.connect("destroy", self.quit)
        self.connect("delete_event", self.quit)
        if modal:
            self.set_modal(True)
        hbox = gtk.HBox(spacing=5)
        hbox.set_border_width(5)
        self.vbox.pack_start(hbox)
        hbox.show()
        if pixmap:
            self.realize()
            pixmap = Pixmap(self, pixmap)
            hbox.pack_start(pixmap, expand=False)
            pixmap.show()
        label = gtk.Label(message)
        hbox.pack_start(label)
        label.show()
        for text in buttons:
            b = gtk.Button(text)
            b.set_flags(gtk.CAN_DEFAULT)
            b.set_data("user_data", text)
            b.connect("clicked", self.click)
            self.action_area.pack_start(b)
            b.show()
        self.ret = None
    def quit(self, *args):
        self.hide()
        self.destroy()
        gtk.main_quit()
    def click(self, button):
        self.ret = button.get_data("user_data")
        self.quit()

# create a message box, and return which button was pressed
def message_box(title="Message Box", message="", buttons=(),
pixmap=None,
        modal= True):
    win = MessageBox(message, buttons, pixmap=pixmap, modal=modal)
    win.set_title(title)
    win.show()
    gtk.main()
    return win.ret

def test():
    result = message_box(title='Test #1',
        message='Here is your message',
        buttons=('Ok', 'Cancel'))
    print 'result:', result

USAGE_TEXT = """
Usage:
    python simple_dialog.py [options]
Options:
    -h, --help      Display this help message.
Example:
    python simple_dialog.py
"""

def usage():
    print USAGE_TEXT
    sys.exit(-1)

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    relink = 1
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 0:
        usage()
    test()

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    main()
```

Some explanation:

- First, we import gtk
- Next we define a class MessageBox that implements a message box. Here are a few important things to know about that class:
    - It is a subclass of gtk.Dialog.
    - It creates a label and packs it into the dialog's client area. Note that a Dialog is a Window that contains a vbox at the top of and an action_area at the bottom of its client area. The intension is for us to pack miscellaneous widgets into the vbox and to put buttons such as "Ok", "Cancel", etc into the action_area.

  - It creates one button for each button label passed to its constructor. The buttons are all connected to the click method.
  - The click method saves the value of the user_data for the button that was clicked. In our example, this value will be either "Ok" or "Cancel".
- And, we define a function (message_box) that (1) creates an instance of the MessageBox class, (2) sets its title, (3) shows it, (4) starts its event loop so that it can get and process events from the user, and (5) returns the result to the caller (in this case "Ok" or "Cancel").
- Our testing function (test) calls function message_box and prints the result.
- This looks like quite a bit of code, until you notice that the class MessageBox and the function message_box could be put it a utility module and reused.

2.7.2.2 A simple text input dialog box

And, here is an example that displays an text input dialog:

```
#!/usr/bin/env python

import sys
import getopt
import gtk

class EntryDialog( gtk.Dialog):
    def __init__(self, message="", default_text='', modal=True):
        gtk.Dialog.__init__(self)
        self.connect("destroy", self.quit)
        self.connect("delete_event", self.quit)
        if modal:
            self.set_modal(True)
        box = gtk.VBox(spacing=10)
        box.set_border_width(10)
        self.vbox.pack_start(box)
        box.show()
        if message:
            label = gtk.Label(message)
            box.pack_start(label)
            label.show()
        self.entry = gtk.Entry()
        self.entry.set_text(default_text)
        box.pack_start(self.entry)
        self.entry.show()
        self.entry.grab_focus()
        button = gtk.Button("OK")
        button.connect("clicked", self.click)
        button.set_flags(gtk.CAN_DEFAULT)
        self.action_area.pack_start(button)
        button.show()
        button.grab_default()
        button = gtk.Button("Cancel")
        button.connect("clicked", self.quit)
        button.set_flags(gtk.CAN_DEFAULT)
        self.action_area.pack_start(button)
        button.show()
        self.ret = None
    def quit(self, w=None, event=None):
        self.hide()
        self.destroy()
        gtk.main_quit()
    def click(self, button):
        self.ret = self.entry.get_text()
        self.quit()

def input_box(title="Input Box", message="", default_text='',
        modal=True):
    win = EntryDialog(message, default_text, modal=modal)
    win.set_title(title)
    win.show()
    gtk.main()
    return win.ret

def test():
    result = input_box(title='Test #2',
        message='Enter a valuexxx:',
        default_text='a default value')
    if result is None:
        print 'Canceled'
    else:
        print 'result: "%s"' % result

USAGE_TEXT = """
Usage:
    python simple_dialog.py [options]
Options:
    -h, --help      Display this help message.
Example:
    python simple_dialog.py
"""

def usage():
    print USAGE_TEXT
    sys.exit(-1)

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    relink = 1
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 0:
        usage()
    test()

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    main()
```

Most of the explanation for the message box example is relevant to this example, too. Here are some differences:

- Our EntryDialog class constructor creates instance of gtk.Entry, sets its default value, and packs it into the client area.
- The constructor also automatically creates two buttons: "OK" and "Cancel". The "OK" button is connect to the click method, which saves the value of the entry field. The "Cancel" button is connect to the quit method, which does not save the value.
- And, if class EntryDialog and function input_box look usable and useful, add them to your utility gui module.

2.7.2.3 A file selection dialog box

This example shows a file selection dialog box:

```
#!/usr/bin/env python

import sys
import getopt
import gtk

class FileChooser(gtk.FileSelection):
    def __init__(self, modal=True, multiple=True):
        gtk.FileSelection.__init__(self)
        self.multiple = multiple
        self.connect("destroy", self.quit)
        self.connect("delete_event", self.quit)
        if modal:
            self.set_modal(True)
        self.cancel_button.connect('clicked', self.quit)
        self.ok_button.connect('clicked', self.ok_cb)
        if multiple:
            self.set_select_multiple(True)
        self.ret = None
    def quit(self, *args):
        self.hide()
        self.destroy()
        gtk.main_quit()
    def ok_cb(self, b):
        if self.multiple:
            self.ret = self.get_selections()
        else:
            self.ret = self.get_filename()
        self.quit()

def file_sel_box(title="Browse", modal=False, multiple=True):
    win = FileChooser(modal=modal, multiple=multiple)
    win.set_title(title)
    win.show()
    gtk.main()
    return win.ret

def file_open_box(modal=True):
    return file_sel_box("Open", modal=modal, multiple=True)
def file_save_box(modal=True):
    return file_sel_box("Save As", modal=modal, multiple=False)

def test():
    result = file_open_box()
    print 'open result:', result
    result = file_save_box()
    print 'save result:', result

USAGE_TEXT = """
Usage:
    python simple_dialog.py [options]
Options:
    -h, --help      Display this help message.
Example:
    python simple_dialog.py
"""

def usage():
    print USAGE_TEXT
    sys.exit(-1)

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    relink = 1
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 0:
        usage()
    test()

if __name__ == '__main__':
    main()
    #import pdb
    #pdb.run('main()')
```

A little guidance:

- There is a pre-defined file selection dialog. We sub-class it.
- This example displays the file selection dialog twice: once with a title "Open" and once with a title "Save As".
- Note how we can control whether the dialog allows multiple file selections. And, if we select the multiple selection mode, then we use get_selections instead of get_filename in order to get the selected file names.
- The dialog contains buttons that enable the user to (1) create a new folder, (2) delete a file, and (3) rename a file. If you do not want the user to perform these operations, then call hide_fileop_buttons. This call is commented out in our sample code.

Note that there are also predefined dialogs for font selection (FontSelectionDialog) and color selection (ColorSelectionDialog)

2.7.3 EasyGUI

If your GUI needs are minimalist (maybe a pop-up dialog or two) and your application is imperative rather than event driven, then you may want to consider EasyGUI. As the name suggests, it is extremely easy to use.

How to know when you might be able to use EasyGUI:

- Your application does not need to run in a window containing menus and a menu bar.
- Your GUI needs amount to little more than displaying a dialog now and then to get responses from the user.
- You do *not* want to write an event driven application, that is, one in which your code sits and waits for the the user to initiate operation, for example, with menu items.

EasyGUI plus documentation and examples are available at EasyGUI home page at SourceForge -- http://easygui.sourceforge.net/

EasyGUI provides functions for a variety of commonly needed dialog boxes, including:

- A message box displays a message.
- A yes/no message box displays "Yes" and "No" buttons.
- A continue/cancel message box displays "Continue" and "Cancel" buttons.
- A choice box displays a selection list.
- An enter box allows entry of a line of text.
- An integer box allows entry of an interger.
- A multiple entry box allows entry into multiple fields.
- Code and text boxes support the display of text in monospaced or porportional

fonts.
- File and directory boxes enable the user to select a file or a directory.

See the documentation at the EasyGUI Web site for more features.

For a demonstration of EasyGUI's capabilities, run the `easygui.py` as a Python script:

```
$ python easygui.py
```

2.7.3.1 A simple EasyGUI example

Here is a simple example that prompts the user for an entry, then shows the response in a message box:

```
import easygui

def testeasygui():
    response = easygui.enterbox(msg='Enter your name:', title='Name
Entry')
    easygui.msgbox(msg=response, title='Your Response')

testeasygui()
```

2.7.3.2 An EasyGUI file open dialog example

This example presents a dialog to allow the user to select a file:

```
import easygui

def test():
    response = easygui.fileopenbox(msg='Select a file')
    print 'file name: %s' % response

test()
```

2.8 Guidance on Packages and Modules

2.8.1 Introduction

Python has an excellent range of implementation organization structures. These range from statements and control structures (at a low level) through functions, methods, and classes (at an intermediate level) and modules and packages at an upper level.

This section provides some guidance with the use of packages. In particular:

- How to construct and implement them.
- How to use them.
- How to distribute and install them.

2.8.2 Implementing Packages

A Python package is a collection of Python modules in a disk directory.

In order to be able to import individual modules from a directory, the directory must contain a file named __init__.py. (Note that requirement does not apply to directories that are listed in PYTHONPATH.) The __init__.py serves several purposes:

- The presence of the file __init__.py in a directory marks the directory as a Python package, which enables importing modules from the directory.
- The first time an application imports any module from the directory/package, the code in the module __init__ is evaluated.
- If the package itself is imported (as opposed to an individual module within the directory/package), then it is the __init__ that is imported (and evaluated).

2.8.3 Using Packages

One simple way to enable the user to import and use a package is to instruct the use to import individual modules from the package.

A second, slightly more advanced way to enable the user to import the package is to expose those features of the package in the __init__ module. Suppose that module mod1 contains functions fun1a and fun1b and suppose that module mod2 contains functions fun2a and fun2b. Then file `__init__.py` might contain the following:

```
from mod1 import fun1a, fun1b
from mod2 import fun2a, fun2b
```

Then, if the following is evaluated in the user's code:

```
import testpackages
```

Then `testpackages` will contain `fun1a`, `fun1b`, `fun2a`, and `fun2b`.

For example, here is an interactive session that demostrates importing the package:

```
>>> import testpackages
>>> print dir(testpackages)
[`__builtins__', `__doc__', `__file__', `__name__',
`__path__',
`fun1a', `fun1b', `fun2a', `fun2b', `mod1', `mod2']
```

2.8.4 Distributing and Installing Packages

Distutils (Python Distribution Utilities) has special support for distrubuting and installing packages. Learn more here: Distributing Python Modules -- http://docs.python.org/distutils/index.html.

As our example, imagine that we have a directory containing the following:

```
Testpackages
Testpackages/README
Testpackages/MANIFEST.in
Testpackages/setup.py
Testpackages/testpackages/__init__.py
Testpackages/testpackages/mod1.py
Testpackages/testpackages/mod2.py
```

Notice the sub-directory Testpackages/testpackages containing the file `__init__.py`. This is the Python package that we will install.

We'll describe how to configure the above files so that they can be packaged as a single distribution file and so that the Python package they contain can be installed as a package by Distutils.

The `MANIFEST.in` file lists the files that we want included in our distribution. Here is the contents of our MANIFEST.in file:

```
include README MANIFEST MANIFEST.in
include setup.py
include testpackages/*.py
```

The `setup.py` file describes to Distutils (1) how to package the distribution file and (2) how to install the distribution. Here is the contents of our sample setup.py:

```
#!/usr/bin/env python

from distutils.core import setup                    # [1]

long_description = 'Tests for installing and distributing Python
packages'

setup(name = 'testpackages',                        # [2]
    version = '1.0a',
    description = 'Tests for Python packages',
    maintainer = 'Dave Kuhlman',
    maintainer_email = 'dkuhlman@rexx.com',
    url = 'http://www.rexx.com/~dkuhlman',
    long_description = long_description,
    packages = ['testpackages']                     # [3]
    )
```

Explanation:

1. We import the necessary component from Distutils.
2. We describe the package and its developer/maintainer.
3. We specify the directory that is to be installed as a package. When the user installs our distribution, this directory and all the modules in it will be installed as a package.

Now, to create a distribution file, we run the following:

```
python setup.py sdist --formats=gztar
```

which will create a file `testpackages-1.0a.tar.gz` under the directory `dist`.

Then, you can give this distribution file to a potential user, who can install it by doing the following:

```
$ tar xvzf testpackages-1.0a.tar.gz
$ cd testpackages-1.0a
$ python setup.py build
$ python setup.py install        # as root
```

2.9 End Matter

2.9.1 Acknowledgements and Thanks

- Thanks to the implementors of Python for producing an exceptionally usable and enjoyable programming language.
- Thanks to Dave Beazley and others for `SWIG` and `PLY`.
- Thanks to Greg Ewing for `Pyrex` and `Plex`.
- Thanks to James Henstridge for `PyGTK`.

2.9.2 See Also

- The main Python Web Site -- http://www.python.org for more information on Python.
- Python Documentation -- http://www.python.org/doc/ for lots of documentation on Python
- Dave's Web Site -- http://http://www.davekuhlman.org for more software and information on using Python for XML and the Web.
- The SWIG home page -- http://www.swig.org for more information on SWIG (Simplified Wrapper and Interface Generator).
- The Pyrex home page -- http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/ for more information on Pyrex.
- PLY (Python Lex-Yacc) home page -- http://www.dabeaz.com/ply/ for more information on PLY.
- The Plex home page -- http://www.cosc.canterbury.ac.nz/greg.ewing/python/Plex/ for more information on Plex.
- Distributing Python Modules -- http://docs.python.org/distutils/index.html for information on the Python Distribution Utilities (`Distutils`).

3 Part 3 -- Python Workbook

3.1 Introduction

This document takes a workbook and exercise-with-solutions approach to Python training. It is hoped that those who feel a need for less explanation and more practical exercises will find this useful.

A few notes about the exercises:

- I've tried to include solutions for most of the exercises. Hopefully, you will be able to copy and paste these solutions into your text editor, then extend and experiment with them.
- I use two interactive Python interpreters (although they are the same Python underneath). When you see this prompt `>>>`, it's the standard Python interpreter. And, when you see this prompt `In [1]:`, it's IPython - http://ipython.scipy.org/moin/.

The latest version of this document is at my Web site (URL above).

If you have comments or suggestions, please send them my way.

3.2 Lexical Structures

3.2.1 Variables and names

A name is any combination of letters, digits, and the underscore, but the first character must be a letter or an underscore. Names may be of any length.

Case is significant.

Exercises:

1. Which of the following are valid names?
   1. `total`
   2. `total_of_all_vegetables`
   3. `big-title-1`
   4. `_inner_func`
   5. `1bigtitle`
   6. `bigtitle1`
2. Which or the following pairs are the same name:
   1. `the_last_item` and `the_last_item`

   2. `the_last_item` and `The_Last_Item`
   3. `itemi` and `itemj`
   4. `item1` and `iteml`

Solutions:

1. Items 1, 2, 4, and 6 are valid. Item 3 is not a single name, but is three items separated by the minus operator. Item 5 is not valid because it begins with a digit.
2. Python names are case-sensitive, which means:
   1. `the_last_item` and `the_last_item` are the same.
   2. `the_last_item` and `The_Last_Item` are different -- The second name has an upper-case characters.
   3. `itemi` and `itemj` are different.
   4. `item1` and `iteml` are different -- This one may be difficult to see, depending on the font you are viewing. One name ends with the digit one; the other ends with the alpha character "el". And this example provides a good reason to use "1" and "l" judiciously in names.

The following are keywords in Python and should **not** be used as variable names:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Exercises:

1. Which of the following are valid names in Python?
   1. _global
   2. global
   3. file

Solutions:

1. Do *not* use keywords for variable names:
   1. Valid
   2. Not a valid name. "global" is a keyword.
   3. Valid, however, "file" is the name of a built-in type, as you will learn later, so you are advised not to redefine it. Here are a few of the names of built-in types: "file", "int", "str", "float", "list", "dict", etc. See Built-in Types -- http://docs.python.org/lib/types.html for more built-in types..

The following are operators in Python and will separate names:

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
```

```
<       >       <=      >=      ==      !=      <>

and     or      is      not     in

Also:   ()      []      . (dot)
```

But, note that the Python style guide suggests that you place blanks around binary operators. One exception to this rule is function arguments and parameters for functions: it is suggested that you *not* put blanks around the equal sign (`=`) used to specify keyword arguments and default parameters.

Exercises:

1. Which of the following are single names and which are names separated by operators?
   1. `fruit_collection`
   2. `fruit-collection`

Solutions:

1. Do not use a dash, or other operator, in the middle of a name:
   1. `fruit_collection` is a single name
   2. `fruit-collection` is two names separated by a dash.

3.2.2 Line structure

In Python, normally we write one statement per line. In fact, Python assumes this. Therefore:

- Statement separators are not normally needed.
- But, if we want more than one statement on a line, we use a statement separator, specifically a semi-colon.
- And, if we want to extend a statement to a second or third line and so on, we sometimes need to do a bit extra.

Extending a Python statement to a subsequent line -- Follow these two rules:

1. If there is an open context, nothing special need be done to extend a statement across multiple lines. An open context is an open parenthesis, an open square bracket, or an open curly bracket.
2. We can always extend a statement on a following line by placing a back slash as the last character of the line.

Exercises:

1. Extend the following statement to a second line using parentheses:

   ```
   total_count = tree_count + vegetable_count +
   fruit_count
   ```

2. Extend the following statement to a second line using the backslash line

continuation character:

```
total_count = tree_count + vegetable_count +
fruit_count
```

Solutions:

1. Parentheses create an open context that tells Python that a statement extends to the next line:

   ```
   total_count = (tree_count +
       vegetable_count + fruit_count)
   ```

2. A backslash as the last character on line tells Python that the current statement extends to the next line:

   ```
   total_count = tree_count + \
       vegetable_count + fruit_count
   ```

For extending a line on a subsequent line, which is better, parentheses or a backslash? Here is a quote:

> "The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression, but sometimes using a backslash looks better."
>
> -- PEP 8: Style Guide for Python Code -- http://www.python.org/dev/peps/pep-0008/

3.2.3 Indentation and program structure

Python uses indentation to indicate program structure. That is to say, in order to nest a block of code inside a compound statement, you indent that nested code. This is different from many programming languages which use some sort of begin and end markers, for example curly brackets.

The standard coding practice for Python is to use four spaces per indentation level and to **not** use hard tabs. (See the Style Guide for Python Code.) Because of this, you will want to use a text editor that you can configure so that it will use four spaces for indentation. See here for a list of Python-friendly text editors: PythonEditors.

Exercises:

1. Given the following, nest the `print` statement inside the `if` statement:

   ```
   if x > 0:

   print x
   ```

2. Nest these two lines:

```
z = x + y
print z
```

inside the following function definition statement:

```
def show_sum(x, y):
```

Solutions:

1. Indentation indicates that one statement is nested inside another statement:

```
if x > 0:
    print x
```

2. Indentation indicates that a block of statements is nested inside another statement:

```
def show_sum(x, y):
    z = x + y
    print z
```

3.3 Execution Model

Here are a few rules:

1. Python evaluates Python code from the top of a module down to the bottom of a module.
2. Binding statements at top level create names (and bind values to those names) as Python evaluates code. Further more, a name is not created until it is bound to a value/object.
3. A nested reference to a name (for example, inside a function definition or in the nested block of an `if` statement) is not used until that nested code is evaluated.

Exercises:

1. Will the following code produce an error?

```
show_version()
def show_version():
    print 'Version 1.0a'
```

2. Will the following code produce an error?

```
def test():
    show_version()

def show_version():
    print 'Version 1.0a'

test()
```

3. Will the following code produce an error? Assume that `show_config` is not defined:

```
x = 3
```

```
if x > 5:
    show_config()
```

Solutions:

1. Answer: Yes, it generates an error. The name `show_version` would not be created and bound to a value until the `def` function definition statement binds a function object to it. That is done after the attempt to use (call) that object.
2. Answer: No. The function `test()` does call the function `show_version()`, but since `test()` is not called until after `show_version()` is defined, that is OK.
3. Answer: No. It's bad code, but in this case will *not* generate an error. Since x is less than 5, the body of the if statement is not evaluated.
   N.B. This example shows why it is important during testing that every line of code in your Python program be evaluated. Here is good Pythonic advice: "If it's not tested, it's broken."

3.4 Built-in Data Types

Each of the subsections in this section on built-in data types will have a similar structure:

1. A brief description of the data type and its uses.
2. Representation and construction -- How to represent an instance of the data type. How to code a literal representation that creates and defines an instance. How to create an instance of the built-in type.
3. Operators that are applicable to the data type.
4. Methods implemented and supported by the data type.

3.4.1 Numbers

The numbers you will use most commonly are likely to be integers and floats. Python also has long integers and complex numbers.

A few facts about numbers (in Python):

- Python will convert to using a long integer automatically when needed. You do not need to worry about exceeding the size of a (standard) integer.
- The size of the largest integer in your version of Python is in `sys.maxint`. To learn what it is, do:

  ```
  >>> import sys
  >>> print sys.maxint
  9223372036854775807
  ```

  The above show the maximum size of an integer on a 64-bit version of Python.
- You can convert from integer to float by using the `float` constructor. Example:

```
>>> x = 25
>>> y = float(x)
>>> print y
25.0
```

- Python does "mixed arithmetic". You can add, multiply, and divide integers and floats. When you do, Python "promotes" the result to a float.

3.4.1.1 Literal representations of numbers

An integer is constructed with a series of digits or the integer constructor (`int(x)`). Be aware that a sequence of digits beginning with zero represents an octal value. Examples:

```
>>> x1 = 1234
>>> x2 = int('1234')
>>> x3 = -25
>>> x1
1234
>>> x2
1234
>>> x3
-25
```

A float is constructed either with digits and a dot (example, 12.345) or with engineering/scientific notation or with the float constructor (`float(x)`). Examples:

```
>>> x1 = 2.0e3
>>> x1 = 1.234
>>> x2 = -1.234
>>> x3 = float('1.234')
>>> x4 = 2.0e3
>>> x5 = 2.0e-3
>>> print x1, x2, x3, x4, x5
1.234 -1.234 1.234 2000.0 0.002
```

Exercises:

Construct these numeric values:

1. Integer zero
2. Floating point zero
3. Integer one hundred and one
4. Floating point one thousand
5. Floating point one thousand using scientific notation
6. Create a positive integer, a negative integer, and zero. Assign them to variables
7. Write several arithmetic expressions. Bind the values to variables. Use a variety of operators, e.g. `+`, `-`, `/`, `*`, etc. Use parentheses to control operator scope.
8. Create several floats and assign them to variables.
9. Write several arithmetic expressions containing your float variables.

10. Write several expressions using mixed arithmetic (integers and floats). Obtain a float as a result of division of one integer by another; do so by explicitly converting one integer to a float.

Solutions:

1. `0`
2. `0.0`, `0.`, or `.0`
3. `101`
4. `1000.0`
5. `1e3` or `1.0e3`
6. Asigning integer values to variables:

   ```
   In [7]: value1 = 23
   In [8]: value2 = -14
   In [9]: value3 = 0
   In [10]: value1
   Out[10]: 23
   In [11]: value2
   Out[11]: -14
   In [12]: value3
   Out[12]: 0
   ```

7. Assigning expression values to variables:

   ```
   value1 = 4 * (3 + 5)
   value2 = (value1 / 3.0) - 2
   ```

8. Assigning floats to variables:

   ```
   value1 = 0.01
   value2 = -3.0
   value3 = 3e-4
   ```

9. Assigning expressions containing varialbes:

   ```
   value4 = value1 * (value2 - value3)
   value4 = value1 + value2 + value3 - value4
   ```

10. Mixed arithmetic:

    ```
    x = 5
    y = 8
    z = float(x) / y
    ```

You can also construct integers and floats using the class. Calling a class (using parentheses after a class name, for example) produces an instance of the class.

Exercises:

1. Construct an integer from the string "123".
2. Construct a float from the integer 123.
3. Construct an integer from the float 12.345.

Solutions:

1. Use the `int` data type to construct an integer instance from a string:

   ```
       int("123")
   ```

2. Use the `float` data type to construct a float instance from an integer:

   ```
       float(123)
   ```

3. Use the `int` data type to construct an integer instance from a float:

   ```
       int(12.345)     # --> 12
   ```

   Notice that the result is truncated to the integer part.

3.4.1.2 Operators for numbers

You can use most of the familiar operators with numbers, for example:

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=      <>
```

Look here for an explanation of these operators when applied to numbers: Numeric Types -- int, float, long, complex -- http://docs.python.org/lib/typesnumeric.html.

Some operators take precedence over others. The table in the Web page just referenced above also shows that order of priority.

Here is a bit of that table:

```
All numeric types (except complex) support the following operations,
sorted by ascending priority (operations in the same box have the
same
priority; all numeric operations have a higher priority than
comparison
operations):

Operation      Result
---------      ------
x + y          sum of x and y
x - y          difference of x and y
x * y          product of x and y
x / y          quotient of x and y
x // y         (floored) quotient of x and y
x % y          remainder of x / y
-x             x negated
+x             x unchanged
abs(x)         absolute value or magnitude of x
int(x)         x converted to integer
long(x)        x converted to long integer
float(x)       x converted to floating point
complex(re,im) a complex number with real part re, imaginary part
               im. im defaults to zero.
c.conjugate()  conjugate of the complex number c
```

```
divmod(x, y)   the pair (x // y, x % y)
pow(x, y)      x to the power y
x ** y         x to the power y
```

Notice also that the same operator may perform a different function depending on the data type of the value to which it is applied.

Exercises:

1. Add the numbers 3, 4, and 5.
2. Add 2 to the result of multiplying 3 by 4.
3. Add 2 plus 3 and multiply the result by 4.

Solutions:

1. Arithmetic expressions are follow standard infix algebraic syntax:

   ```
   3 + 4 + 5
   ```

2. Use another infix expression:

   ```
   2 + 3 * 4
   ```

   Or:

   ```
   2 + (3 * 4)
   ```

   But, in this case the parentheses are not necessary because the `*` operator binds more tightly than the `+` operator.
3. Use parentheses to control order of evaluation:

   ```
   (2 + 3) * 4
   ```

   Note that the `*` operator has precedence over (binds tighter than) the `+` operator, so the parentheses are needed.

Python does mixed arithemetic. When you apply an operation to an integer and a float, it promotes the result to the "higher" data type, a float.

If you need to perform an operation on several integers, but want use a floating point operation, first convert one of the integers to a float using `float(x)`, which effectively creates an instance of class `float`.

Try the following at your Python interactive prompt:

1. `1.0 + 2`
2. `2 / 3` -- Notice that the result is truncated.
3. `float(2) / 3` -- Notice that the result is *not* truncated.

Exercises:

1. Given the following assignments:

   ```
   x = 20
   y = 50
   ```

Divide `x` by `y` giving a float result.

Solutions:

1. Promote one of the integers to float *before* performing the division:

```
    z = float(x) / y
```

3.4.1.3 Methods on numbers

Most of the methods implemented by the data types (classes) `int` and `float` are special methods that are called through the use of operators. Special methods often have names that begin and end with a double underscore. To see a list of the special names and a bit of an indication of when each is called, do any of the following at the Python interactive prompt:

```
>>> help(int)
>>> help(32)
>>> help(float)
>>> help(1.23)
>>> dir(1)
>>> dir(1.2)
```

3.4.2 Lists

Lists are a container data type that acts as a dynamic array. That is to say, a list is a sequence that can be indexed into and that can grow and shrink.

A tuple is an index-able container, like a list, except that a tuple is immutable.

A few characteristics of lists and tuples:

- A list has a (current) length -- Get the length of a list with `len(mylist)`.
- A list has an order -- The items in a list are ordered, and you can think of that order as going from left to right.
- A list is heterogeneous -- You can insert different *types* of objects into the same list.
- Lists are mutable, but tuples are *not*. Thus, the following are true of lists, but *not* of tuples:
  - You can extended or add to a list.
  - You can shrink a list by deleting items from it.
  - You can insert items into the middle of a list or at the beginning of a list. You can add items to the end of a list.
  - You can change which item is at a given position in a list.

3.4.2.1 Literal representation of lists

The literal representation of a list is square brackets containing zero or more items separated by commas.

Examples:

1. Try these at the Python interactive prompt:

   ```
   >>> [11, 22, 33]
   >>> ['aa', 'bb', 'cc', ]
   >>> [100, 'apple', 200, 'banana', ]    # The last comma
   is
   >>> optional.
   ```

2. A list can contain lists. In fact a list can contain any kind of object:

   ```
   >>> [1, [2, 3], 4, [5, 6, 7, ], 8]
   ```

3. Lists are heterogenous, that is, different kinds of objects can be in the same list. Here is a list that contains a number, a string, and another list:

   ```
   >>> [123, 'abc', [456, 789]]
   ```

Exercises:

1. Create (define) the following tuples and lists using a literal:
   1. A tuple of integers
   2. A tuple of strings
   3. A list of integers
   4. A list of strings
   5. A list of tuples or tuple of lists
   6. A list of integers and strings and tuples
   7. A tuple containing exactly one item
   8. An empty tuple
2. Do each of the following:
   1. Print the length of a list.
   2. Print each item in the list -- Iterate over the items in one of your lists. Print each item.
   3. Append an item to a list.
   4. Insert an item at the beginning of a list. Insert an item in the middle of a list.
   5. Add two lists together. Do so by using both the extend method and the plus (+) operator. What is the difference between extending a list and adding two lists?
   6. Retrieve the 2nd item from one of your tuples or lists.
   7. Retrieve the 2nd, 3rd, and 4th items (a slice) from one of your tuples or lists.
   8. Retrieve the last (right-most) item in one of your lists.
   9. Replace an item in a list with a new item.

10. Pop one item off the end of your list.
11. Delete an item from a list.
12. Do the following list manipulations:
    1. Write a function that takes two arguments, a list and an item, and that appends the item to the list.
    2. Create an empty list,
    3. Call your function several times to append items to the list.
    4. Then, print out each item in the list.

Solutions:

1. We can define list literals at the Python or IPython interactive prompt:
    1. Create a tuple using commas, optionally with parentheses:

       ```
       In [1]: a1 = (11, 22, 33, )
       In [2]: a1
       Out[2]: (11, 22, 33)
       ```

    2. Quoted characters separated by commas create a tuple of strings:

       ```
       In [3]: a2 = ('aaa', 'bbb', 'ccc')
       In [4]: a2
       Out[4]: ('aaa', 'bbb', 'ccc')
       ```

    3. Items separated by commas inside square brackets create a list:

       ```
       In [26]: a3 = [100, 200, 300, ]
       In [27]: a3
       Out[27]: [100, 200, 300]
       ```

    4. Strings separated by commas inside square brackets create a list of strings:

       ```
       In [5]: a3 = ['basil', 'parsley', 'coriander']
       In [6]: a3
       Out[6]: ['basil', 'parsley', 'coriander']
       In [7]:
       ```

    5. A tuple or a list can contain tuples and lists:

       ```
       In [8]: a5 = [(11, 22), (33, 44), (55,)]
       In [9]: a5
       Out[9]: [(11, 22), (33, 44), (55,)]
       ```

    6. A list or tuple can contain items of different types:

       ```
       In [10]: a6 = [101, 102, 'abc', "def", (201, 202),
       ('ghi', 'jkl')]
       In [11]: a6
       Out[11]: [101, 102, 'abc', 'def', (201, 202),
       ('ghi', 'jkl')]
       ```

    7. In order to create a tuple containing exactly one item, we must use a comma:

       ```
       In [13]: a7 = (6,)
       In [14]: a7
       ```

```
Out[14]: (6,)
```

8. In order to create an empty tuple, use the tuple class/type to create an instance of a empty tuple:

```
In [21]: a = tuple()
In [22]: a
Out[22]: ()
In [23]: type(a)
Out[23]: <type 'tuple'>
```

3.4.2.2 Operators on lists

There are several operators that are applicable to lists. Here is how to find out about them:

- Do `dir([])` or `dir(any_list_instance)`. Some of the items with special names (leading and training double underscores) will give you clues about operators implemented by the list type.
- Do `help([])` or `help(list)` at the Python interactive prompt.
- Do `help(any_list_instance.some_method)`, where `some_method` is one of the items listed using `dir(any_list_instance)`.
- See Sequence Types -- str, unicode, list, tuple, buffer, xrange -- http://docs.python.org/lib/typesseq.html

Exercises:

1. Concatenate (add) two lists together.
2. Create a single list that contains the items in an initial list repeated 3 times.
3. Compare two lists.

Solutions:

1. The plus operator, applied to two lists produces a new list that is a concatenation of two lists:

```
>>> [11, 22] + ['aa', 'bb']
```

2. Multiplying a list by an integer `n` creates a new list that repeats the original list `n` times:

```
>>> [11, 'abc', 4.5] * 3
```

3. The comparison operators can be used to compare lists:

```
>>> [11, 22] == [11, 22]
>>> [11, 22] < [11, 33]
```

3.4.2.3 Methods on lists

Again, use `dir()` and `help()` to learn about the methods supported by lists.

Examples:

1. Create two (small) lists. Extend the first list with the items in the second.
2. Append several individual items to the end of a list.
3. (a) Insert a item at the beginning of a list. (b) Insert an item somewhere in the middle of a list.
4. Pop an item off the end of a list.

Solutions:

1. The `extend` method adds elements from another list, or other iterable:

   ```
   >>> a = [11, 22, 33, 44, ]
   >>> b = [55, 66]
   >>> a.extend(b)
   >>> a
   [11, 22, 33, 44, 55, 66]
   ```

2. Use the `append` method on a list to add/append an item to the end of a list:

   ```
   >>> a = ['aa', 11]
   >>> a.append('bb')
   >>> a.append(22)
   >>> a
   ['aa', 11, 'bb', 22]
   ```

3. The `insert` method on a list enables us to insert items at a given position in a list:

   ```
   >>> a = [11, 22, 33, 44, ]
   >>> a.insert(0, 'aa')
   >>> a
   ['aa', 11, 22, 33, 44]
   >>> a.insert(2, 'bb')
   >>> a
   ['aa', 11, 'bb', 22, 33, 44]
   ```

   But, note that we use `append` to add items at the end of a list.

4. The `pop` method on a list returns the "right-most" item from a list and removes that item from the list:

   ```
   >>> a = [11, 22, 33, 44, ]
   >>>
   >>> b = a.pop()
   >>> a
   [11, 22, 33]
   >>> b
   44
   >>> b = a.pop()
   >>> a
   [11, 22]
   >>> b
   33
   ```

Note that the `append` and `pop` methods taken together can be used to implement a stack, that is a LIFO (last in first out) data structure.

3.4.2.4 List comprehensions

A list comprehension is a convenient way to produce a list from an iterable (a sequence or other object that can be iterated over).

In its simplest form, a list comprehension resembles the header line of a `for` statement inside square brackets. However, in a list comprehension, the `for` statement header is prefixed with an expression and surrounded by square brackets. Here is a template:

```
[expr(x) for x in iterable]
```

where:

- `expr(x)` is an expression, usually, but not always, containing `x`.
- `iterable` is some iterable. An iterable may be a sequence (for example, a list, a string, a tuple) or an unordered collection or an iterator (something over which we can iterate or apply a `for` statement to).

Here is an example:

```
>>> a = [11, 22, 33, 44]
>>> b = [x * 2 for x in a]
>>> b
[22, 44, 66, 88]
```

Exercises:

1. Given the following list of strings:

   ```
   names = ['alice', 'bertrand', 'charlene']
   ```

   produce the following lists: (1) a list of all upper case names; (2) a list of capitalized (first letter upper case);
2. Given the following function which calculates the factorial of a number:

   ```
   def t(n):
       if n <= 1:
           return n
       else:
           return n * t(n - 1)
   ```

   and the following list of numbers:

   ```
   numbers = [2, 3, 4, 5]
   ```

   create a list of the factorials of each of the numbers in the list.

Solutions:

1. For our expression in a list comprehension, use the `upper` and `capitalize`

methods:

```
>>> names = ['alice', 'bertrand', 'charlene']
>>> [name.upper() for name in names]
['ALICE', 'BERTRAND', 'CHARLENE']
>>> [name.capitalize() for name in names]
['Alice', 'Bertrand', 'Charlene']
```

2. The expression in our list comprehension calls the factorial function:

```
def t(n):
    if n <= 1:
        return n
    else:
        return n * t(n - 1)

def test():
    numbers = [2, 3, 4, 5]
    factorials = [t(n) for n in numbers]
    print 'factorials:', factorials

if __name__ == '__main__':
    test()
```

A list comprehension can also contain an `if` clause. Here is a template:

```
[expr(x) for x in iterable if pred(x)]
```

where:

- `pred(x)` is an expression that evaluates to a true/false value. Values that count as false are numeric zero, `False`, `None`, and any empty collection. All other values count as true.

Only values for which the if clause evaluates to true are included in creating the resulting list.

Examples:

```
>>> a = [11, 22, 33, 44]
>>> b = [x * 3 for x in a if x % 2 == 0]
>>> b
[66, 132]
```

Exercises:

1. Given two lists, generate a list of all the strings in the first list that are not in the second list. Here are two sample lists:

```
names1 = ['alice', 'bertrand', 'charlene', 'daniel']
names2 = ['bertrand', 'charlene']
```

Solutions:

1. The if clause of our list comprehension checks for containment in the list names2:

```
def test():
    names1 = ['alice', 'bertrand', 'charlene',
'daniel']
    names2 = ['bertrand', 'charlene']
    names3 = [name for name in names1 if name not in
names2]
    print 'names3:', names3

if __name__ == '__main__':
    test()
```

When run, this script prints out the following:

```
names3: ['alice', 'daniel']
```

3.4.3 Strings

A string is an ordered sequence of characters. Here are a few characteristics of strings:

- A string has a length. Get the length with the `len()` built-in function.
- A string is indexable. Get a single character at a position in a string with the square bracket operator, for example `mystring[5]`.
- You can retrieve a slice (sub-string) of a string with a slice operation, for example `mystring[5:8]`.

Create strings with single quotes or double quotes. You can put single quotes inside double quotes and you can put double quotes inside single quotes. You can also escape characters with a backslash.

Exercises:

1. Create a string containing a single quote.
2. Create a string containing a double quote.
3. Create a string containing both a single quote a double quote.

Solutions:

1. Create a string with double quotes to include single quotes inside the string:

```
>>> str1 = "that is jerry's ball"
```

2. Create a string enclosed with single quotes in order to include double quotes inside the string:

```
>>> str1 = 'say "goodbye", bullwinkle'
```

3. Take your choice. Escape either the single quotes or the double quotes with a backslash:

```
>>> str1 = 'say "hello" to jerry\'s mom'
>>> str2 = "say \"hello\" to jerry's mom"
>>> str1
'say "hello" to jerry\'s mom'
```

```
>>> str2
'say "hello" to jerry\'s mom'
```

Triple quotes enable you to create a string that spans multiple lines. Use three single quotes or three double quotes to create a single quoted string.

Examples:

1. Create a triple quoted string that contains single and double quotes.

Solutions:

1. Use triple single quotes or triple double quotes to create multi-line strings:

```
String1 = '''This string extends
across several lines.  And, so it has
end-of-line characters in it.
'''

String2 = """
This string begins and ends with an end-of-line
character.  It can have both 'single'
quotes and "double" quotes in it.
"""

def test():
    print String1
    print String2

if __name__ == '__main__':
    test()
```

3.4.3.1 Characters

Python does not have a distinct character type. In Python, a character is a string of length 1. You can use the `ord()` and `chr()` built-in functions to convert from character to integer and back.

Exercises:

1. Create a character "a".
2. Create a character, then obtain its integer representation.

Solutions:

1. The character "a" is a plain string of length 1:

```
>>> x = 'a'
```

2. The integer equivalent of the letter "A":

```
>>> x = "A"
>>> ord(x)
65
```

3.4.3.2 Operators on strings

You can concatenate strings with the "+" operator.

You can create multiple concatenated copies of a string with the "*" operator.

And, augmented assignment (`+=` and `*=`) also work.

Examples:

```
>>> 'cat' + ' and ' + 'dog'
'cat and dog'
>>> '#' * 40
'########################################'
>>>
>>> s1 = 'flower'
>>> s1 += 's'
>>> s1
'flowers'
```

Exercises:

1. Given these strings:

   ```
   >>> s1 = 'abcd'
   >>> s2 = 'efgh'
   ```

   create a new string composed of the first string followed by (concatenated with) the second.
2. Create a single string containing 5 copies of the string 'abc'.
3. Use the multiplication operator to create a "line" of 50 dashes.
4. Here are the components of a path to a file on the file system: "home", "myusername", "Workdir", "notes.txt". Concatenate these together separating them with the path separator to form a complete path to that file. (Note that if you use the backslash to separate components of the path, you will need to use a double backslash, because the backslash is the escape character in strings.

Solutions:

1. The plus (`+`) operator applied to a string can be used to concatenate strings:

   ```
   >>> s3 = s1 + s2
   >>> s3
   'abcdefgh'
   ```

2. The multiplication operator (`*`) applied to a string creates a new string that concatenates a string with itself some number of times:

   ```
   >>> s1 = 'abc' * 5
   >>> s1
   'abcabcabcabcabc'
   ```

3. The multiplication operator (`*`) applied to a string can be used to create a

"horizontal divider line":

```
>>> s1 = '-' * 50
>>> print s1
--------------------------------------------------
```

4. The `sep` member of the `os` module gives us a platform independent way to construct paths:

```
>>> import os
>>>
>>> a = ["home", "myusername", "Workdir", "notes.txt"]
>>> path = a[0] + os.sep + a[1] + os.sep + a[2] +
os.sep + a[3]
>>> path
'home/myusername/Workdir/notes.txt'
```

And, a more concise solution:

```
>>> import os
>>> a = ["home", "myusername", "Workdir", "notes.txt"]
>>> os.sep.join(a)
'home/myusername/Workdir/notes.txt'
```

Notes:

- Note that importing the `os` module and then using `os.sep` from that module gives us a platform independent solution.
- If you do decide to code the path separator character explicitly and if you are on MS Windows where the path separator is the backslash, then you will need to use a double backslash, because that character is the escape character.

3.4.3.3 Methods on strings

String support a variety of operations. You can obtain a list of these methods by using the `dir()` built-in function on any string:

```
>>> dir("")
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__str__', 'capitalize', 'center',
'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find',
'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

And, you can get help on any specific method by using the `help()` built-in function. Here is an example:

```
>>> help("".strip)
Help on built-in function strip:

strip(...)
    S.strip([chars]) -> string or unicode

    Return a copy of the string S with leading and trailing
    whitespace removed.
    If chars is given and not None, remove characters in chars
instead.
    If chars is unicode, S will be converted to unicode before
stripping
```

Exercises:

1. Strip all the whitespace characters off the right end of a string.
2. Center a short string within a longer string, that is, pad a short string with blank characters on both right and left to center it.
3. Convert a string to all upper case.
4. Split a string into a list of "words".
5. (a) Join the strings in a list of strings to form a single string. (b) Ditto, but put a newline character between each original string.

Solutions:

1. The `rstrip()` method strips whitespace off the right side of a string:

   ```
   >>> s1 = 'some text   \n'
   >>> s1
   'some text   \n'
   >>> s2 = s1.rstrip()
   >>> s2
   'some text'
   ```

2. The `center(n)` method centers a string within a padded string of width n:

   ```
   >>> s1 = 'Dave'
   >>> s2 = s1.center(20)
   >>> s2
   '        Dave        '
   ```

3. The `upper()` method produces a new string that converts all alpha characters in the original to upper case:

   ```
   >>> s1 = 'Banana'
   >>> s1
   'Banana'
   >>> s2 = s1.upper()
   >>> s2
   'BANANA'
   ```

4. The `split(sep)` method produces a list of strings that are separated by `sep` in the original string. If `sep` is omitted, whitespace is treated as the separator:

```
>>> s1 = """how does it feel
... to be on your own
... no directions known
... like a rolling stone
... """
>>> words = s1.split()
>>> words
['how', 'does', 'it', 'feel', 'to', 'be', 'on', 'your',
'own', 'no',
'directions', 'known', 'like', 'a', 'rolling', 'stone']
```

Note that the `split()` function in the `re` (regular expression) module is useful when the separator is more complex than whitespace or a single character.

5. The `join()` method concatenates strings from a list of strings to form a single string:

```
>>> lines = []
>>> lines.append('how does it feel')
>>> lines.append('to be on your own')
>>> lines.append('no directions known')
>>> lines.append('like a rolling stone')
>>> lines
['how does it feel', 'to be on your own', 'no
directions known',
 'like a rolling stone']
>>> s1 = ''.join(lines)
>>> s2 = ' '.join(lines)
>>> s3 = '\n'.join(lines)
>>> s1
'how does it feelto be on your ownno directions
knownlike a rolling stone'
>>> s2
'how does it feel to be on your own no directions known
like a rolling stone'
>>> s3
'how does it feel\nto be on your own\nno directions
known\nlike a rolling stone'
>>> print s3
how does it feel
to be on your own
no directions known
like a rolling stone
```

3.4.3.4 Raw strings

Raw strings give us a convenient way to include the backslash character in a string without escaping (with an additional backslash). Raw strings look like plain literal strings, but are prefixed with an "r" or "R". See String literals http://docs.python.org/reference/lexical_analysis.html#string-literals

Excercises:

1. Create a string that contains a backslash character using both plain literal string and a raw string.

Solutions:

1. We use an "r" prefix to define a raw string:

```
>>> print 'abc \\ def'
abc \ def
>>> print r'abc \ def'
abc \ def
```

3.4.3.5 Unicode strings

Unicode strings give us a consistent way to process character data from a variety of character encodings.

Excercises:

1. Create several unicode strings. Use both the unicode prefix character ("u") and the unicode type (`unicode(some_string)`).
2. Convert a string (possibly from another non-ascii encoding) to unicode.
3. Convert a unicode string to another encoding, for example, utf-8.
4. Test a string to determine if it is unicode.
5. Create a string that contains a unicode character, that is, a character outside the ascii character set.

Solutions:

1. We can represent unicode string with either the "u" prefix or with a call to the unicode type:

```
def exercise1():
    a = u'abcd'
    print a
    b = unicode('efgh')
    print b
```

2. We convert a string from another character encoding into unicode with the `decode()` string method:

```
import sys

def exercise2():
    a = 'abcd'.decode('utf-8')
    print a
    b = 'abcd'.decode(sys.getdefaultencoding())
    print b
```

3. We can convert a unicode string to another character encoding with the `encode()` string method:

```
import sys
```

```
def exercise3():
    a = u'abcd'
    print a.encode('utf-8')
    print a.encode(sys.getdefaultencoding())
```

4. Here are two ways to check the type of a string:

```
import types

def exercise4():
    a = u'abcd'
    print type(a) is types.UnicodeType
    print type(a) is type(u'')
```

5. We can encode unicode characters in a string in several ways, for example, (1) by defining a utf-8 string and converting it to unicode or (2) defining a string with an embedded unicode character or (3) concatenating a unicode characher into a string:

```
def exercise5():
    utf8_string = 'Ivan Krsti\xc4\x87'
    unicode_string = utf8_string.decode('utf-8')
    print unicode_string.encode('utf-8')
    print len(utf8_string)
    print len(unicode_string)
    unicode_string = u'aa\u0107bb'
    print unicode_string.encode('utf-8')
    unicode_string = 'aa' + unichr(263) + 'bb'
    print unicode_string.encode('utf-8')
```

Guidance for use of encodings and unicode:

1. Convert/decode from an external encoding to unicode early:

```
my_source_string.decode(encoding)
```

2. Do your work (Python processing) in unicode.
3. Convert/encode to an external encoding late (for example, just before saving to an external file):

```
my_unicode_string.encode(encoding)
```

For more information, see:

- Unicode In Python, Completely Demystified -- http://farmdev.com/talks/unicode/
- Unicode How-to -- http://www.amk.ca/python/howto/unicode.
- PEP 100: Python Unicode Integration -- http://www.python.org/dev/peps/pep-0100/
- 4.8 codecs -- Codec registry and base classes -- http://docs.python.org/lib/module-codecs.html
- 4.8.2 Encodings and Unicode --

http://docs.python.org/lib/encodings-overview.html

- 4.8.3 Standard Encodings -- http://docs.python.org/lib/standard-encodings.html
- Converting Unicode Strings to 8-bit Strings -- http://effbot.org/zone/unicode-convert.htm

3.4.4 Dictionaries

A dictionary is an un-ordered collection of key-value pairs.

A dictionary has a length, specifically the number of key-value pairs.

A dictionary provides fast look up by key.

The keys must be immutable object types.

3.4.4.1 Literal representation of dictionaries

Curley brackets are used to represent a dictionary. Each pair in the dictionary is represented by a key and value separated by a colon. Multiple pairs are separated by comas. For example, here is an empty dictionary and several dictionaries containing key/value pairs:

```
In [4]: d1 = {}
In [5]: d2 = {'width': 8.5, 'height': 11}
In [6]: d3 = {1: 'RED', 2: 'GREEN', 3: 'BLUE', }
In [7]: d1
Out[7]: {}
In [8]: d2
Out[8]: {'height': 11, 'width': 8.5}
In [9]: d3
Out[9]: {1: 'RED', 2: 'GREEN', 3: 'BLUE'}
```

Notes:

- A comma after the last pair is optional. See the RED-GREEN-BLUE example above.
- Strings and integers work as keys, since they are immutable. You might also want to think about the use of tuples of integers as keys in a dictionary used to represent a sparse array.

Exercises:

1. Define a dictionary that has the following key-value pairs:
2. Define a dictionary to represent the "enum" days of the week: Sunday, Monday, Tuesday, ...

Solutions:

1. A dictionary whose keys and values are strings can be used to represent this table:

```
vegetables = {
    'Eggplant': 'Purple',
    'Tomato': 'Red',
    'Parsley': 'Green',
    'Lemon': 'Yellow',
    'Pepper': 'Green',
    }
```

Note that the open curly bracket enables us to continue this statement across multiple lines without using a backslash.

2. We might use strings for the names of the days of the week as keys:

```
DAYS = {
    'Sunday':    1,
    'Monday':    2,
    'Tuesday':   3,
    'Wednesday': 4,
    'Thrusday':  5,
    'Friday':    6,
    'Saturday':  7,
    }
```

3.4.4.2 Operators on dictionaries

Dictionaries support the following "operators":

- Length -- `len(d)` returns the number of pairs in a dictionary.
- Indexing -- You can both set and get the value associated with a key by using the indexing operator `[ ]`. Examples:

```
In [12]: d3[2]
Out[12]: 'GREEN'
In [13]: d3[0] = 'WHITE'
In [14]: d3[0]
Out[14]: 'WHITE'
```

- Test for key -- The `in` operator tests for the existence of a key in a dictionary. Example:

```
In [6]: trees = {'poplar': 'deciduous', 'cedar':
'evergreen'}
In [7]: if 'cedar' in trees:
   ...:     print 'The cedar is %s' %
(trees['cedar'], )
   ...:
The cedar is evergreen
```

Exercises:

1. Create an empty dictionary, then use the indexing operator `[ ]` to in sert the following name-value pairs:

```
"red" --   "255:0:0"
```

```
"green" -- "0:255:0"
"blue" --  "0:0:255"
```

2. Print out the number of items in your dictionary.

Solutions:

1. We can use "[ ]" to set the value of a key in a dictionary:

```
def test():
    colors = {}
    colors["red"] = "255:0:0"
    colors["green"] = "0:255:0"
    colors["blue"] = "0:0:255"
    print 'The value of red is "%s"' %
(colors['red'], )
    print 'The colors dictionary contains %d items.' %
(len(colors), )

test()
```

When we run this, we see:

```
The value of red is "255:0:0"
The colors dictionary contains 3 items.
```

2. The `len()` built-in function gives us the number of items in a dictionary. See the previous solution for an example of this.

3.4.4.3 Methods on dictionaries

Here is a table that describes the methods applicable to dictionarys:

Operation	***Result***
len(a)	the number of items in a
a[k]	the item of a with key k
a[k] = v	set a[k] to v
del a[k]	remove a[k] from a
a.clear()	remove all items from a
a.copy()	a (shallow) copy of a
k in a	True if a has a key k, else False
k not in a	equivalent to not k in a
a.has_key(k)	equivalent to k in a, use that form in new code
a.items()	a copy of a's list of (key, value) pair

Operation	***Result***
a.keys()	a copy of a's list of keys
a.update([b])	updates a with key/value pairs from b, overwriting existing keys, returns None
a.fromkeys(seq[, value])	creates a new dictionary with keys from seq and values set to value
a.values()	a copy of a's list of values
a.get(k[, x])	a[k] if k in a, else x)
a.setdefault(k[, x])	a[k] if k in a, else x (also setting it)
a.pop(k[, x])	a[k] if k in a, else x (and remove k) (8)
a.popitem()	remove and return an arbitrary (key, value) pair
a.iteritems()	return an iterator over (key, value) pairs
a.iterkeys()	return an iterator over the mapping's keys
a.itervalues()	return an iterator over the mapping's values

You can also find this table at the standard documentation Web site in the "Python Library Reference": Mapping Types -- dict http://docs.python.org/lib/typesmapping.html

Exercises:

1. Print the keys and values in the above "vegetable" dictionary.
2. Print the keys and values in the above "vegetable" dictionary with the keys in alphabetical order.
3. Test for the occurance of a key in a dictionary.

Solutions:

1. We can use the `d.items()` method to retrieve a list of tuples containing key-value pairs, then use unpacking to capture the key and value:

```
Vegetables = {
    'Eggplant': 'Purple',
    'Tomato': 'Red',
    'Parsley': 'Green',
    'Lemon': 'Yellow',
    'Pepper': 'Green',
    }
```

```
def test():
    for key, value in Vegetables.items():
        print 'key:', key, ' value:', value

test()
```

2. We retrieve a list of keys with the `keys()` method, the sort it with the list `sort()` method:

   ```
   Vegetables = {
       'Eggplant': 'Purple',
       'Tomato': 'Red',
       'Parsley': 'Green',
       'Lemon': 'Yellow',
       'Pepper': 'Green',
       }

   def test():
       keys = Vegetables.keys()
       keys.sort()
       for key in keys:
           print 'key:', key, ' value:', Vegetables[key]

   test()
   ```

3. To test for the existence of a key in a dictionary, we can use either the `in` operator (preferred) or the `d.has_key()` method (old style):

   ```
   Vegetables = {
       'Eggplant': 'Purple',
       'Tomato': 'Red',
       'Parsley': 'Green',
       'Lemon': 'Yellow',
       'Pepper': 'Green',
       }

   def test():
       if 'Eggplant' in Vegetables:
           print 'we have %s egplants' %
   Vegetables['Eggplant']
       if 'Banana' not in Vegetables:
           print 'yes we have no bananas'
       if Vegetables.has_key('Parsley'):
           print 'we have leafy, %s parsley' %
   Vegetables['Parsley']

   test()
   ```

   Which will print out:

   ```
   we have Purple egplants
   yes we have no bananas
   we have leafy, Green parsley
   ```

3.4.5 Files

A Python file object represents a file on a file system.

A file object open for reading a text file is iterable. When we iterate over it, it produces the lines in the file.

A file may be opened in these modes:

- 'r' -- read mode. The file must exist.
- 'w' -- write mode. The file is created; an existing file is overwritten.
- 'a' -- append mode. An existing file is opened for writing (at the end of the file). A file is created if it does not exist.

The `open()` built-in function is used to create a file object. For example, the following code (1) opens a file for writing, then (2) for reading, then (3) for appending, and finally (4) for reading again:

```
def test(infilename):
    # 1. Open the file in write mode, which creates the file.
    outfile = open(infilename, 'w')
    outfile.write('line 1\n')
    outfile.write('line 2\n')
    outfile.write('line 3\n')
    outfile.close()
    # 2. Open the file for reading.
    infile = open(infilename, 'r')
    for line in infile:
        print 'Line:', line.rstrip()
    infile.close()
    # 3. Open the file in append mode, and add a line to the end of
    #    the file.
    outfile = open(infilename, 'a')
    outfile.write('line 4\n')
    outfile.close()
    print '-' * 40
    # 4. Open the file in read mode once more.
    infile = open(infilename, 'r')
    for line in infile:
        print 'Line:', line.rstrip()
    infile.close()

test('tmp.txt')
```

Exercises:

1. Open a text file for reading, then read the entire file as a single string, and then split the content on newline characters.
2. Open a text file for reading, then read the entire file as a list of strings, where each string is one line in the file.
3. Open a text file for reading, then iterate of each line in the file and print it out.

Solutions:

1. Use the `open()` built-in function to open the file and create a file object. Use the `read()` method on the file object to read the entire file. Use the `split()` or `splitlines()` methods to split the file into lines:

   ```
   >>> infile = open('tmp.txt', 'r')
   >>> content = infile.read()
   >>> infile.close()
   >>> lines = content.splitlines()
   >>> print lines
   ['line 1', 'line 2', 'line 3', '']
   ```

2. The `f.readlines()` method returns a list of lines in a file:

   ```
   >>> infile = open('tmp.txt', 'r')
   >>> lines = infile.readlines()
   >>> infile.close()
   >>> print lines
   ['line 1\n', 'line 2\n', 'line 3\n']
   ```

3. Since a file object (open for reading) is itself an iterator, we can iterate over it in a `for` statement:

   ```
   """
   Test iteration over a text file.
   Usage:
       python test.py in_file_name
   """

   import sys

   def test(infilename):
       infile = open(infilename, 'r')
       for line in infile:
           # Strip off the new-line character and any
   whitespace on
           # the right.
           line = line.rstrip()
           # Print only non-blank lines.
           if line:
               print line
       infile.close()

   def main():
       args = sys.argv[1:]
       if len(args) != 1:
           print __doc__
           sys.exit(1)
       infilename = args[0]
       test(infilename)

   if __name__ == '__main__':
       main()
   ```

Notes:

- The last two lines of this solution check the `__name__` attribute of the module itself so that the module will run as a script but will *not* run when the module is imported by another module.
- The `__doc__` attribute of the module gives us the module's doc-string, which is the string defined at the top of the module.
- `sys.argv` gives us the command line. And, `sys.argv[1:]` chops off the program name, leaving us with the comman line arguments.

3.4.6 A few miscellaneous data types

3.4.6.1 None

`None` is a singleton. There is only one instance of `None`. Use this value to indicate the absence of any other "real" value.

Test for `None` with the identity operator `is`.

Exercises:

1. Create a list, some of whose elements are `None`. Then write a `for` loop that counts the number of occurances of None in the list.

Solutions:

1. The identity operators `is` and `is not` can be used to test for `None`:

```
>>> a = [11, None, 'abc', None, {}]
>>> a
[11, None, 'abc', None, {}]
>>> count = 0
>>> for item in a:
...     if item is None:
...         count += 1
...
>>>
>>> print count
2
```

3.4.6.2 The booleans True and False

Python has the two boolean values `True` and `False`. Many comparison operators return `True` and `False`.

Examples:

1. What value is returned by `3 > 2`?
   Answer: The boolean value `True`.
2. Given these variable definitions:

```
x = 3
y = 4
z = 5
```

What does the following print out:

```
print y > x and z > y
```

Answer -- Prints out "True"

3.5 Statements

3.5.1 Assignment statement

The assignment statement uses the assignment operator `=`.

The assignment statement is a binding statement: it binds a value to a name within a namespace.

Exercises:

1. Bind the value "eggplant" to the variable `vegetable`.

Solutions:

1. The `=` operator is an assignment statement that binds a value to a variable:

```
>>> vegetable = "eggplant"
```

There is also augmented assignment using the operators `+=`, `-=`, `*=`, `/=`, etc.

Exercises:

1. Use augmented assignment to increment the value of an integer.
2. Use augmented assignment to append characters to the end of a string.
3. Use augmented assignment to append the items in one list to another.
4. Use augmented assignment to decrement a variable containing an integer by 1.

Solutions:

1. The `+=` operator increments the value of an integer:

```
>>> count = 0
>>> count += 1
>>> count
1
>>> count += 1
>>> count
2
```

2. The `+=` operator appends characters to the end of a string:

```
>>> buffer = 'abcde'
>>> buffer += 'fgh'
>>> buffer
'abcdefgh'
```

3. The `+=` operator appends items in one list to another:

```
In [20]: a = [11, 22, 33]
In [21]: b = [44, 55]
In [22]: a += b
In [23]: a
Out[23]: [11, 22, 33, 44, 55]
```

1. The `-=` operator decrements the value of an integer:

```
>>> count = 5
>>> count
5
>>> count -= 1
>>> count
4
```

You can also assign a value to (1) an element of a list, (2) an item in a dictionary, (3) an attribute of an object, etc.

Exercises:

1. Create a list of three items, then assign a new value to the 2nd element in the list.
2. Create a dictionary, then assign values to the keys "vegetable" and "fruit" in that dictionary.
3. Use the following code to create an instance of a class:

```
class A(object):
    pass
a = A()
```

Then assign values to an attribue named `category` in that instance.

Solutions:

1. Assignment with the indexing operator `[]` assigns a value to an element in a list:

```
>>> trees = ['pine', 'oak', 'elm']
>>> trees
['pine', 'oak', 'elm']
>>> trees[1] = 'cedar'
>>> trees
['pine', 'cedar', 'elm']
```

2. Assignment with the indexing operator `[]` assigns a value to an item (a key-value pair) in a dictionary:

```
>>> foods = {}
>>> foods
{}
>>> foods['vegetable'] = 'green beans'
>>> foods['fruit'] = 'nectarine'
>>> foods
```

```
{'vegetable': 'green beans', 'fruit': 'nectarine'}
```

3. Assignment along with the dereferencing operator `.` (dot) enables us to assign a value to an attribute of an object:

```
>>> class A(object):
...     pass
...
>>> a = A()
>>> a.category = 25
>>> a.__dict__
{'category': 25}
>>> a.category
25
```

3.5.2 print statement

Warning: Be aware that the `print` statement will go away in Python version 3.0. It will be replaced by the built-in `print()` function.

The `print` statement sends output to standard output. It provides a somewhat more convenient way of producing output than using `sys.stdout.write()`.

The `print` statement takes a series of zero or more objects separated by commas. Zero objects produces a blank line.

The print statement normally adds a newline at the end of its output. To eliminate that, add a comma at the end.

Exercises:

1. Print a single string.
2. Print three strings using a single `print` statement.
3. Given a variable `name` containing a string, print out the string `My name is "xxxx".`, where xxxx is replace by the value of `name`. Use the string formatting operator.

Solutions:

1. We can print a literal string:

```
>>> print 'Hello, there'
Hello, there
```

2. We can print literals and the value of variables:

```
>>> description = 'cute'
>>> print 'I am a', description, 'kid.'
I am a cute kid.
```

3. The string formatting operator gives more control over formatting output:

```
>>> name = 'Alice'
>>> print 'My name is "%s".' % (name, )
My name is "Alice".
```

3.5.3 if: statement exercises

The `if` statement is a compound statement that enables us to conditionally execute blocks of code.

The `if` statement also has optional `elif:` and `else:` clauses.

The condition in an `if:` or `elif:` clause can be any Python expression, in other words, something that returns a value (even if that value is `None`).

In the condition in an `if:` or `elif:` clause, the following values count as "false":

- `False`
- `None`
- Numeric zero
- An empty collection, for example an empty list or dictionary
- An empty string (a string of length zero)

All other values count as true.

Exercises:

1. Given the following list:

   ```
   >>> bananas = ['banana1', 'banana2', 'banana3',]
   ```

   Print one message if it is an empty list and another messge if it is not.
2. Here is one way of defining a Python equivalent of an "enum":

   ```
   NO_COLOR, RED, GREEN, BLUE = range(4)
   ```

   Write an `if:` statement which implements the effect of a "switch" statement in Python. Print out a unique message for each color.

Solutions:

1. We can test for an empty or non-empty list:

   ```
   >>> bananas = ['banana1', 'banana2', 'banana3',]
   >>> if not bananas:
   ...     print 'yes, we have no bananas'
   ... else:
   ...     print 'yes, we have bananas'
   ...
   yes, we have bananas
   ```

2. We can simulate a "switch" statement using `if:elif: ...`:

   ```
   NO_COLOR, RED, GREEN, BLUE = range(4)

   def test(color):
   ```

```
    if color == RED:
        print "It's red."
    elif color == GREEN:
        print "It's green."
    elif color == BLUE:
        print "It's blue."

def main():
    color = BLUE
    test(color)

if __name__ == '__main__':
    main()
```

Which, when run prints out the following:

```
It's blue.
```

3.5.4 for: statement exercises

The `for:` statement is the Python way to iterate over and process the elements of a collection or other iterable.

The basic form of the `for:` statement is the following:

```
for X in Y:
    statement
    o
    o
    o
```

where:

- `X` is something that can be assigned to. It is something to which Python can bind a value.
- `Y` is some collection or other iterable.

Exercises:

1. Create a list of integers. Use a `for:` statement to print out each integer in the list.
2. Create a string. print out each character in the string.

Solutions:

1. The `for:` statement can iterate over the items in a list:

```
In [13]: a = [11, 22, 33, ]
In [14]: for value in a:
   ....:     print 'value: %d' % value
   ....:
   ....:
value: 11
value: 22
value: 33
```

2. The `for:` statement can iterate over the characters in a string:

   ```
   In [16]: b = 'chocolate'
   In [17]: for chr1 in b:
      ....:     print 'character: %s' % chr1
      ....:
      ....:
   character: c
   character: h
   character: o
   character: c
   character: o
   character: l
   character: a
   character: t
   character: e
   ```

   Notes:

   ○ In the solution, I used the variable name `chr1` rather than `chr` so as not to over-write the name of the built-in function `chr()`.

When we need a sequential index, we can use the `range()` built-in function to create a list of integers. And, the `xrange()` built-in function produces an interator that produces a sequence of integers without creating the entire list. To iterate over a large sequence of integers, use `xrange()` instead of `range()`.

Exercises:

1. Print out the integers from 0 to 5 in sequence.
2. Compute the sum of all the integers from 0 to 99999.
3. Given the following generator function:

   ```
   import urllib

   Urls = [
       'http://yahoo.com',
       'http://python.org',
       'http://gimp.org',    # The GNU image manipulation
   program
       ]

   def walk(url_list):
       for url in url_list:
           f = urllib.urlopen(url)
           stuff = f.read()
           f.close()
           yield stuff
   ```

   Write a `for:` statement that uses this iterator generator to print the lengths of the content at each of the Web pages in that list.

Solutions:

1. The `range()` built-in function gives us a sequence to iterate over:

```
In [5]: for idx in range(6):
   ...:     print 'idx: %d' % idx
   ...:
   ...:
idx: 0
idx: 1
idx: 2
idx: 3
idx: 4
idx: 5
```

2. Since that sequence is a bit large, we'll use `xrange()` instead of `range()`:

```
In [8]: count = 0
In [9]: for n in xrange(100000):
   ...:     count += n
   ...:
   ...:
In [10]: count
Out[10]: 4999950000
```

3. The `for:` statement enables us to iterate over iterables as well as collections:

```
import urllib

Urls = [
    'http://yahoo.com',
    'http://python.org',
    'http://gimp.org',    # The GNU image manipulation
program
    ]

def walk(url_list):
    for url in url_list:
        f = urllib.urlopen(url)
        stuff = f.read()
        f.close()
        yield stuff

def test():
    for url in walk(Urls):
        print 'length: %d' % (len(url), )

if __name__ == '__main__':
    test()
```

When I ran this script, it prints the following:

```
length: 9562
length: 16341
length: 12343
```

If you need an index while iterating over a sequence, consider using the `enumerate()` built-in function.

Exercises:

1. Given the following two lists of integers of the same length:

   ```
   a = [1, 2, 3, 4, 5]
   b = [100, 200, 300, 400, 500]
   ```

   Add the values in the first list to the corresponding values in the second list.

Solutions:

1. The `enumerate()` built-in function gives us an index and values from a sequence. Since `enumerate()` gives us an interator that produces a sequence of two-tuples, we can unpack those tuples into index and value variables in the header line of the `for` statement:

   ```
   In [13]: a = [1, 2, 3, 4, 5]
   In [14]: b = [100, 200, 300, 400, 500]
   In [15]:
   In [16]: for idx, value in enumerate(a):
      ....:     b[idx] += value
      ....:
      ....:
   In [17]: b
   Out[17]: [101, 202, 303, 404, 505]
   ```

3.5.5 while: statement exercises

A `while:` statement executes a block of code repeatedly as long as a condition is true.

Here is a template for the `while:` statement:

```
while condition:
    statement
    o
    o
    o
```

Where:

- `condition` is an expression. The expression is something that returns a value which can be interpreted as true or false.

Exercises:

1. Write a `while:` loop that doubles all the values in a list of integers.

Solutions:

1. A `while:` loop with an index variable can be used to modify each element of a list:

   ```
   def test_while():
       numbers = [11, 22, 33, 44, ]
       print 'before: %s' % (numbers, )
   ```

```
    idx = 0
    while idx < len(numbers):
        numbers[idx] *= 2
        idx += 1
    print 'after: %s' % (numbers, )
```

But, notice that this task is easier using the `for:` statement and the built-in `enumerate()` function:

```
def test_for():
    numbers = [11, 22, 33, 44, ]
    print 'before: %s' % (numbers, )
    for idx, item in enumerate(numbers):
        numbers[idx] *= 2
    print 'after: %s' % (numbers, )
```

3.5.6 break and continue statements

The `continue` statement skips the remainder of the statements in the body of a loop and starts immediately at the top of the loop again.

A `break` statement in the body of a loop terminates the loop. It exits from the immediately containing loop.

`break` and `continue` can be used in both `for:` and `while:` statements.

Exercises:

1. Write a `for:` loop that takes a list of integers and triples each integer that is even. Use the `continue` statement.
2. Write a loop that takes a list of integers and computes the sum of all the integers up until a zero is found in the list. Use the `break` statement.

Solutions:

1. The `continue` statement enables us to "skip" items that satisfy a condition or test:

```
def test():
    numbers = [11, 22, 33, 44, 55, 66, ]
    print 'before: %s' % (numbers, )
    for idx, item in enumerate(numbers):
        if item % 2 != 0:
            continue
        numbers[idx] *= 3
    print 'after: %s' % (numbers, )

test()
```

2. The `break` statement enables us to exit from a loop when we find a zero:

```
def test():
    numbers = [11, 22, 33, 0, 44, 55, 66, ]
    print 'numbers: %s' % (numbers, )
    sum = 0
    for item in numbers:
        if item == 0:
            break
        sum += item
    print 'sum: %d' % (sum, )

test()
```

3.5.7 Exceptions and the try:except: and raise statements

The `try:except:` statement enables us to catch an exception that is thrown from within a block of code, or from code called from any depth withing that block.

The `raise` statement enables us to throw an exception.

An exception is a class or an instance of an exception class. If an exception is not caught, it results in a traceback and termination of the program.

There is a set of standard exceptions. You can learn about them here: Built-in Exceptions -- http://docs.python.org/lib/module-exceptions.html.

You can define your own exception classes. To do so, create an empty subclass of the class `Exception`. Defining your own exception will enable you (or others) to throw and then catch that specific exception type while ignore others exceptions.

Exercises:

1. Write a `try:except:` statement that attempts to open a file for reading and catches the exception thrown when the file does not exist.
   Question: How do you find out the name of the exception that is thrown for an input/output error such as the failure to open a file?
2. Define an exception class. Then write a `try:except:` statement in which you throw and catch that specific exception.
3. Define an exception class and use it to implement a multi-level break from an inner loop, by-passing an outer loop.

Solutions:

1. Use the Python interactive interpreter to learn the exception type thrown when a I/O error occurs. Example:

```
>>> infile = open('xx_nothing__yy.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory:
'xx_nothing__yy.txt'
>>>
```

In this case, the exception type is `IOError`.
Now, write a `try:except:` block which catches that exception:

```
def test():
    infilename = 'nothing_noplace.txt'
    try:
        infile = open(infilename, 'r')
        for line in infile:
            print line
    except IOError, exp:
        print 'cannot open file "%s"' % infilename

test()
```

2. We define a exception class as a sub-class of class `Exception`, then throw it (with the `raise` statement) and catch it (with a `try:except:` statement):

```
class SizeError(Exception):
    pass

def test_exception(size):
    try:
        if size <= 0:
            raise SizeError, 'size must be greater than
zero'
        # Produce a different error to show that it
will not be caught.
        x = y
    except SizeError, exp:
        print '%s' % (exp, )
        print 'goodbye'

def test():
    test_exception(-1)
    print '-' * 40
    test_exception(1)

test()
```

When we run this script, it produces the following output:

```
$ python workbook027.py
size must be greater than zero
goodbye
----------------------------------------
Traceback (most recent call last):
  File "workbook027.py", line 20, in <module>
    test()
  File "workbook027.py", line 18, in test
    test_exception(1)
  File "workbook027.py", line 10, in test_exception
    x = y
NameError: global name 'y' is not defined
```

Notes:

- Our `except:` clause caught the `SizeError`, but allowed the `NameError` to be uncaught.

3. We define a sub-class of of class `Exception`, then raise it in an inner loop and catch it outside of an outer loop:

```
class BreakException1(Exception):
    pass

def test():
    a = [11, 22, 33, 44, 55, 66, ]
    b = [111, 222, 333, 444, 555, 666, ]
    try:
        for x in a:
            print 'outer -- x: %d' % x
            for y in b:
                if x > 22 and y > 444:
                    raise BreakException1('leaving
inner loop')
                print 'inner -- y: %d' % y
            print 'outer -- after'
            print '-' * 40
    except BreakException1, exp:
        print 'out of loop -- exp: %s' % exp

test()
```

Here is what this prints out when run:

```
outer -- x: 11
inner -- y: 111
inner -- y: 222
inner -- y: 333
inner -- y: 444
inner -- y: 555
inner -- y: 666
outer -- after
----------------------------------------
outer -- x: 22
inner -- y: 111
inner -- y: 222
inner -- y: 333
inner -- y: 444
inner -- y: 555
inner -- y: 666
outer -- after
----------------------------------------
outer -- x: 33
inner -- y: 111
inner -- y: 222
inner -- y: 333
inner -- y: 444
out of loop -- exp: leaving inner loop
```

3.6 Functions

A function has these characteristics:

- It groups a block of code together so that we can call it by name.
- It enables us to pass values into the the function when we call it.
- It can returns a value (even if None).
- When a function is called, it has its own namespace. Variables in the function are local to the function (and disappear when the function exits).

A function is defined with the `def:` statement. Here is a simple example/template:

```
def function_name(arg1, arg2):
    local_var1 = arg1 + 1
    local_var2 = arg2 * 2
    return local_var1 + local_var2
```

And, here is an example of calling this function:

```
result = function_name(1, 2)
```

Here are a few notes of explanation:

- The above defines a function whose name is `function_name`.
- The function `function_name` has two arguments. That means that we can and must pass in exactly two values when we call it.
- This function has two local variables, `local_var1` and `local_var2`. These variables are local in the sense that after we call this function, these two variables are **not** available in the location of the caller.
- When we call this function, it returns one value, specifically the sum of `local_var1` and `local_var2`.

Exercises:

1. Write a function that takes a list of integers as an argument, and returns the sum of the integers in that list.

Solutions:

1. The `return` statement enables us to return a value from a function:

```
def list_sum(values):
    sum = 0
    for value in values:
        sum += value
    return sum

def test():
    a = [11, 22, 33, 44, ]
    print list_sum(a)

if __name__ == '__main__':
    test()
```

3.6.1 Optional arguments and default values

You can provide a default value for an argument to a function.

If you do, that argument is optional (when the function is called).

Here are a few things to learn about optional arguments:

- Provide a default value with an equal sign and a value. Example:

  ```
  def sample_func(arg1, arg2, arg3='empty', arg4=0):
  ```

- All parameters with default values must be after (to the right of) normal parameters.
- Do not use a mutable object as a default value. Because the `def:` statement is evaluated only once and **not** each time the function is called, the mutable object might be shared across multiple calls to the function. Do not do this:

  ```
  def sample_func(arg1, arg2=[]):
  ```

  Instead, do this:

  ```
  def sample_func(arg1, arg2=None):
      if arg2 is None:
          arg2 = []
  ```

  Here is an example that illustrates how this might go wrong:

  ```
  def adder(a, b=[]):
      b.append(a)
      return b

  def test():
      print adder('aaa')
      print adder('bbb')
      print adder('ccc')

  test()
  ```

  Which, when executed, displays the following:

  ```
  ['aaa']
  ['aaa', 'bbb']
  ['aaa', 'bbb', 'ccc']
  ```

Exercises:

1. Write a function that writes a string to a file. The function takes two arguments: (1) a file that is open for output and (2) a string. Give the second argument (the string) a default value so that when the second argument is omitted, an empty, blank line is written to the file.

2. Write a function that takes the following arguments: (1) a name, (2) a value, and (3) and optional dictionary. The function adds the value to the dictionary using the name as a key in the dictionary.

Solutions:

1. We can pass a file as we would any other object. And, we can use a newline character as a default parameter value:

```
import sys

def writer(outfile, msg='\n'):
    outfile.write(msg)

def test():
    writer(sys.stdout, 'aaaaa\n')
    writer(sys.stdout)
    writer(sys.stdout, 'bbbbb\n')

test()
```

   When run from the command line, this prints out the following:

```
aaaaa

bbbbb
```

2. In this solution we are careful **not** to use a mutable object as a default value:

```
def add_to_dict(name, value, dic=None):
    if dic is None:
        dic = {}
    dic[name] = value
    return dic

def test():
    dic1 = {'albert': 'cute', }
    print add_to_dict('barry', 'funny', dic1)
    print add_to_dict('charlene', 'smart', dic1)
    print add_to_dict('darryl', 'outrageous')
    print add_to_dict('eddie', 'friendly')

test()
```

   If we run this script, we see:

```
{'barry': 'funny', 'albert': 'cute'}
{'barry': 'funny', 'albert': 'cute', 'charlene':
'smart'}
{'darryl': 'outrageous'}
{'eddie': 'friendly'}
```

   Notes:

   ○ It's important that the default value for the dictionary is `None` rather than an empty dictionary, for example (`{}`). Remember that the `def:` statement is evaluated only once, which results in a *single* dictionary, which would be shared by all callers that do not provide a dictionary as an argument.

3.6.2 Passing functions as arguments

A function, like any other object, can be passed as an argument to a function. This is due the the fact that almost all (maybe all) objects in Python are "first class objects". A first class object is one which we can:

1. Store in a data structure (e.g. a list, a dictionary, ...).
2. Pass to a function.
3. Return from a function.

Exercises:

1. Write a function that takes three arguments: (1) an input file, (2) an output file, and (3) a filter function:
   - Argument 1 is a file opened for reading.
   - Argument 2 is a file opened for writing.
   - Argument 3 is a function that takes a single argument (a string), performs a transformation on that string, and returns the transformed string.

   The above function should read each line in the input text file, pass that line through the filter function, then write that (possibly) transformed line to the output file.

   Now, write one or more "filter functions" that can be passed to the function described above.

Solutions:

1. This script adds or removes comment characters to the lines of a file:

```
import sys

def filter(infile, outfile, filterfunc):
    for line in infile:
        line = filterfunc(line)
        outfile.write(line)

def add_comment(line):
    line = '## %s' % (line, )
    return line

def remove_comment(line):
    if line.startswith('## '):
        line = line[3:]
    return line

def main():
    filter(sys.stdin, sys.stdout, add_comment)
```

```
if __name__ == '__main__':
    main()
```

Running this might produce something like the following (note for MS Windows users: use `type` instead of `cat`):

```
$ cat tmp.txt
line 1
line 2
line 3
$ cat tmp.txt | python workbook005.py
## line 1
## line 2
## line 3
```

3.6.3 Extra args and keyword args

Additional positional arguments passed to a function that are not specified in the function definition (the `def:` statement``), are collected in an argument preceded by a single asterisk. Keyword arguments passed to a function that are not specified in the function definition can be collected in a dictionary and passed to an argument preceded by a double asterisk.

Examples:

1. Write a function that takes one positional argument, one argument with a default value, and also extra args and keyword args.
2. Write a function that passes all its arguments, no matter how many, to a call to another function.

Solutions:

1. We use `*args` and `**kwargs` to collect extra arguments and extra keyword arguments:

```
def show_args(x, y=-1, *args, **kwargs):
    print '-' * 40
    print 'x:', x
    print 'y:', y
    print 'args:', args
    print 'kwargs:', kwargs

def test():
    show_args(1)
    show_args(x=2, y=3)
    show_args(y=5, x=4)
    show_args(4, 5, 6, 7, 8)
    show_args(11, y=44, a=55, b=66)

test()
```

Running this script produces the following:

```
$ python workbook006.py
----------------------------------------
x: 1
y: -1
args: ()
kwargs: {}
----------------------------------------
x: 2
y: 3
args: ()
kwargs: {}
----------------------------------------
x: 4
y: 5
args: ()
kwargs: {}
----------------------------------------
x: 4
y: 5
args: (6, 7, 8)
kwargs: {}
----------------------------------------
x: 11
y: 44
args: ()
kwargs: {'a': 55, 'b': 66}
```

Notes:

- The spelling of `args` and `kwargs` is not fixed, but the

2. We use `args` and `kwargs` to catch and pass on all arguments:

```
def func1(*args, **kwargs):
    print 'args: %s' % (args, )
    print 'kwargs: %s' % (kwargs, )

def func2(*args, **kwargs):
    print 'before'
    func1(*args, **kwargs)
    print 'after'

def test():
    func2('aaa', 'bbb', 'ccc', arg1='ddd', arg2='eee')

test()
```

When we run this, it prints the following:

```
before
args: ('aaa', 'bbb', 'ccc')
kwargs: {'arg1': 'ddd', 'arg2': 'eee'}
after
```

Notes:

- In a function *call*, the `*` operator unrolls a list into individual positional arguments, and the `**` operator unrolls a dictionary into individual keyword arguments.

3.6.3.1 Order of arguments (positional, extra, and keyword args)

In a function *definition*, arguments must appear in the following order, from left to right:

1. Positional (normal, plain) arguments
2. Arguments with default values, if any
3. Extra arguments parameter (proceded by single asterisk), if present
4. Keyword arguments parameter (proceded by double asterisk), if present

In a function *call*, arguments must appear in the following order, from left to right:

1. Positional (plain) arguments
2. Extra arguments, if present
3. Keyword arguments, if present

3.6.4 Functions and duck-typing and polymorphism

If the arguments and return value of a function satisfy some description, then we can say that the function is polymorphic with respect to that description.

If the some of the methods of an object satisfy some description, then we can say that the object is polymorphic with respect to that description.

Basically, what this does is to enable us to use a function or an object anywhere that function satisfies the requirements given by a description.

Exercises:

1. Implement a function that takes two arguments: a function and an object. It applies the function argument to the object.
2. Implement a function that takes two arguments: a list of functions and an object. It applies each function in the list to the argument.

Solutions:

1. We can pass a function as an argument to a function:

```
def fancy(obj):
    print 'fancy fancy -- %s -- fancy fancy' % (obj, )

def plain(obj):
    print 'plain -- %s -- plain' % (obj, )

def show(func, obj):
    func(obj)
```

```
def main():
    a = {'aa': 11, 'bb': 22, }
    show(fancy, a)
    show(plain, a)

if __name__ == '__main__':
    main()
```

2. We can also put functions (function objects) in a data structure (for example, a list), and then pass that data structure to a function:

```
def fancy(obj):
    print 'fancy fancy -- %s -- fancy fancy' % (obj, )

def plain(obj):
    print 'plain -- %s -- plain' % (obj, )

Func_list = [fancy, plain, ]

def show(funcs, obj):
    for func in funcs:
        func(obj)

def main():
    a = {'aa': 11, 'bb': 22, }
    show(Func_list, a)

if __name__ == '__main__':
    main()
```

Notice that Python supports polymorphism (with or) without inheritance. This type of polymorphism is enabled by what is called duck-typing. For more on this see: Duck typing -- http://en.wikipedia.org/wiki/Duck_typing at Wikipedia.

3.6.5 Recursive functions

A recursive function is a function that calls itself.

A recursive function must have a limiting condition, or else it will loop endlessly.

Each recursive call consumes space on the function call stack. Therefore, the number of recursions must have some reasonable upper bound.

Exercises:

1. Write a recursive function that prints information about each node in the following tree-structure data structure:

```
Tree = {
    'name': 'animals',
    'left_branch': {
        'name': 'birds',
        'left_branch': {
            'name': 'seed eaters',
            'left_branch': {
                'name': 'house finch',
                'left_branch': None,
                'right_branch': None,
            },
            'right_branch': {
                'name': 'white crowned sparrow',
                'left_branch': None,
                'right_branch': None,
            },
        },
        'right_branch': {
            'name': 'insect eaters',
            'left_branch': {
                'name': 'hermit thrush',
                'left_branch': None,
                'right_branch': None,
            },
            'right_branch': {
                'name': 'black headed phoebe',
                'left_branch': None,
                'right_branch': None,
            },
        },
    },
    'right_branch': None,
}
```

Solutions:

1. We write a recursive function to walk the whole tree. The recursive function calls itself to process each child of a node in the tree:

```
Tree = {
    'name': 'animals',
    'left_branch': {
        'name': 'birds',
        'left_branch': {
            'name': 'seed eaters',
            'left_branch': {
                'name': 'house finch',
                'left_branch': None,
                'right_branch': None,
            },
            'right_branch': {
                'name': 'white crowned sparrow',
                'left_branch': None,
                'right_branch': None,
            },
        },
        'right_branch': {
            'name': 'insect eaters',
            'left_branch': {
                'name': 'hermit thrush',
                'left_branch': None,
                'right_branch': None,
            },
            'right_branch': {
                'name': 'black headed phoebe',
                'left_branch': None,
                'right_branch': None,
            },
        },
    },
    'right_branch': None,
}

Indents = ['    ' * idx for idx in range(10)]

def walk_and_show(node, level=0):
    if node is None:
        return
    print '%sname: %s' % (Indents[level], node['name'],
)
    level += 1
    walk_and_show(node['left_branch'], level)
    walk_and_show(node['right_branch'], level)

def test():
    walk_and_show(Tree)

if __name__ == '__main__':
    test()
```

Notes:

- Later, you will learn how to create equivalent data structures using classes and OOP (object-oriented programming). For more on that see Recursive calls to methods in this document.

3.6.6 Generators and iterators

The "iterator protocol" defines what an iterator object must do in order to be usable in an "iterator context" such as a `for` statement. The iterator protocol is described in the standard library reference: Iterator Types -- http://docs.python.org/lib/typeiter.html

An easy way to define an object that obeys the iterator protocol is to write a generator function. A generator function is a function that contains one or more `yield` statements. If a function contains at least one `yield` statement, then that function when called, returns generator iterator, which is an object that obeys the iterator protocol, i.e. it's an iterator object.

Note that in recent versions of Python, yield is an expression. This enables the consumer to communicate back with the producer (the generator iterator). For more on this, see PEP: 342 Coroutines via Enhanced Generators - http://www.python.org/dev/peps/pep-0342/.

Exercises:

1. Implement a generator function -- The generator produced should `yield` all values from a list/iterable that satisfy a predicate. It should apply the transforms before return each value. The function takes these arguments:
   1. `values` -- A list of values. Actually, it could be any iterable.
   2. `predicate` -- A function that takes a single argument, performs a test on that value, and returns True or False.
   3. `transforms` -- (optional) A list of functions. Apply each function in this list and returns the resulting value. So, for example, if the function is called like this:

      ```
      result = transforms([11, 22], p, [f, g])
      ```

      then the resulting generator might return:

      ```
      g(f(11))
      ```
2. Implement a generator function that takes a list of URLs as its argument and generates the contents of each Web page, one by one (that is, it produces a sequence of strings, the HTML page contents).

Solutions:

1. Here is the implementation of a function which contains `yield`, and, therefore, produces a generator:

   ```
   #!/usr/bin/env python
   """
   filter_and_transform

   filter_and_transform(content, test_func,
   transforms=None)

   Return a generator that returns items from content
   after applying
   the functions in transforms if the item satisfies
   test_func .

   Arguments:

      1. ``values`` -- A list of values

      2. ``predicate`` -- A function that takes a single
   argument,
          performs a test on that value, and returns True
   ```

```
or False.

   3. ``transforms`` -- (optional) A list of functions.
Apply each
      function in this list and returns the resulting
value.  So,
      for example, if the function is called like
this::

       result = filter_and_transforms([11, 22], p, [f,
g])

      then the resulting generator might return::

          g(f(11))
"""

def filter_and_transform(content, test_func,
transforms=None):
    for x in content:
        if test_func(x):
            if transforms is None:
                yield x
            elif isiterable(transforms):
                for func in transforms:
                    x = func(x)
                yield x
            else:
                yield transforms(x)

def isiterable(x):
    flag = True
    try:
        x = iter(x)
    except TypeError, exp:
        flag = False
    return flag

def iseven(n):
    return n % 2 == 0

def f(n):
    return n * 2

def g(n):
    return n ** 2

def test():
    data1 = [11, 22, 33, 44, 55, 66, 77, ]
    for val in filter_and_transform(data1, iseven, f):
        print 'val: %d' % (val, )
    print '-' * 40
    for val in filter_and_transform(data1, iseven, [f,
g]):
        print 'val: %d' % (val, )
    print '-' * 40
    for val in filter_and_transform(data1, iseven):
        print 'val: %d' % (val, )

if __name__ == '__main__':
    test()
```

Notes:

- Because function `filter_and_transform` contains `yield`, when called, it returns an iterator object, which we can use in a `for` statement.
- The second parameter of function `filter_and_transform` takes any function which takes a single argument and returns True or False. This is an example of polymorphism and "duck typing" (see Duck Typing -- http://en.wikipedia.org/wiki/Duck_typing). An analogous claim can be made about the third parameter.

2. The following function uses the `urllib` module and the `yield` function to generate the contents of a sequence of Web pages:

```
import urllib

Urls = [
    'http://yahoo.com',
    'http://python.org',
    'http://gimp.org',    # The GNU image manipulation
program
    ]

def walk(url_list):
    for url in url_list:
        f = urllib.urlopen(url)
        stuff = f.read()
        f.close()
        yield stuff

def test():
    for x in walk(Urls):
        print 'length: %d' % (len(x), )

if __name__ == '__main__':
    test()
```

When I run this, I see:

```
$ python generator_example.py
length: 9554
length: 16748
length: 11487
```

3.7 Object-oriented programming and classes

Classes provide Python's way to define new data types and to do OOP (object-oriented programming).

If you have made it this far, you have already *used* lots of objects. You have been a "consumer" of objects and their services. Now, you will learn how to define and implement new *kinds* of objects. You will become a "producer" of objects. You will define new classes and you will implement the capabilities (methods) of each new class.

A class is defined with the `class` statement. The first line of a `class` statement is a header (it has a colon at the end), and it specifies the name of the class being defined and an (optional) superclass. And that header introduces a compound statement: specifically, the body of the `class` statement which contains indented, nested statements, importantly, `def` statements that define the methods that can be called on instances of the objects implemented by this class.

Exercises:

1. Define a class with one method `show`. That method should print out "Hello". Then, create an instance of your class, and call the `show` method.

Solutions:

1. A simple instance method can have the `self` parameter and no others:

   ```
   class Demo(object):
       def show(self):
           print 'hello'

   def test():
       a = Demo()
       a.show()

   test()
   ```

   Notes:

   - Notice that we use `object` as a superclass, because we want to define an "new-style" class and because there is no other class that we want as a superclass. See the following for more information on new-style classes: New-style Classes -- http://www.python.org/doc/newstyle/.
   - In Python, we create an instance of a class by calling the class, that is, we apply the function call operator (parentheses) to the class.

3.7.1 The constructor

A class can define methods with special names. You have seem some of these before. These names begin and end with a double underscore.

One important special name is `__init__`. It's the constructor for a class. It is called each time an instance of the class is created. Implementing this method in a class gives us a chance to initialize each instance of our class.

Exercises:

1. Implement a class named `Plant` that has a constructor which initializes two instance variables: `name` and `size`. Also, in this class, implement a method named `show` that prints out the values of these instance variables. Create several instances of your class and "show" them.
2. Implement a class name `Node` that has two instance variables: `data` and `children`, where `data` is any, arbitrary object and `children` is a list of child Nodes. Also implement a method named `show` that recursively displays the nodes in a "tree". Create an instance of your class that contains several child instances of your class. Call the show method on the root (top most) object to show the tree.

Solutions:

1. The constructor for a class is a method with the special name `__init__`:

```
class Plant(object):
    def __init__(self, name, size):
        self.name = name
        self.size = size
    def show(self):
        print 'name: "%s"  size: %d' % (self.name,
self.size, )

def test():
    p1 = Plant('Eggplant', 25)
    p2 = Plant('Tomato', 36)
    plants = [p1, p2, ]
    for plant in plants:
        plant.show()

test()
```

   Notes:

   - Our constructor takes two arguments: `name` and `size`. It saves those two values as instance variables, that is in attributes of the instance.
   - The `show()` method prints out the value of those two instance variables.

2. It is a good idea to initialize all instance variables in the constructor. That enables someone reading our code to learn about all the instance variables of a class by looking in a single location:

```
# simple_node.py

Indents = ['    ' * n for n in range(10)]
```

```
class Node(object):
    def __init__(self, name=None, children=None):
        self.name = name
        if children is None:
            self.children = []
        else:
            self.children = children
    def show_name(self, indent):
        print '%sname: "%s"' % (Indents[indent],
self.name, )
    def show(self, indent=0):
        self.show_name(indent)
        indent += 1
        for child in self.children:
            child.show(indent)

def test():
    n1 = Node('N1')
    n2 = Node('N2')
    n3 = Node('N3')
    n4 = Node('N4')
    n5 = Node('N5', [n1, n2,])
    n6 = Node('N6', [n3, n4,])
    n7 = Node('N7', [n5, n6,])
    n7.show()

if __name__ == '__main__':
    test()
```

Notes:

- Notice that we do **not** use the constructor for a list (`[]`) as a default value for the `children` parameter of the constructor. A list is mutable and would be created only once (when the class statement is executed) and would be shared.

3.7.2 Inheritance -- Implementing a subclass

A subclass extends or specializes a superclass by adding additional methods to the superclass and by overriding methods (with the same name) that already exist in the superclass.

Exercises:

1. Extend your `Node` exercise above by adding two additional subclasses of the Node class, one named `Plant` and the other named `Animal`. The `Plant` class also has a `height` instance variable and the `Animal` class also has a `color` instance variable.

Solutions:

1. We can `import` our previous `Node` script, then implement classes that have the `Node` class as a superclass:

```
from simple_node import Node, Indents

class Plant(Node):
    def __init__(self, name, height=-1, children=None):
        Node.__init__(self, name, children)
        self.height = height
    def show(self, indent=0):
        self.show_name(indent)
        print '%sheight: %s' % (Indents[indent],
self.height, )
        indent += 1
        for child in self.children:
            child.show(indent)

class Animal(Node):
    def __init__(self, name, color='no color',
children=None):
        Node.__init__(self, name, children)
        self.color = color
    def show(self, indent=0):
        self.show_name(indent)
        print '%scolor: "%s"' % (Indents[indent],
self.color, )
        indent += 1
        for child in self.children:
            child.show(indent)

def test():
    n1 = Animal('scrubjay', 'gray blue')
    n2 = Animal('raven', 'black')
    n3 = Animal('american kestrel', 'brown')
    n4 = Animal('red-shouldered hawk', 'brown and
gray')
    n5 = Animal('corvid', 'none', [n1, n2,])
    n6 = Animal('raptor', children=[n3, n4,])
    n7a = Animal('bird', children=[n5, n6,])
    n1 = Plant('valley oak', 50)
    n2 = Plant('canyon live oak', 40)
    n3 = Plant('jeffery pine', 120)
    n4 = Plant('ponderosa pine', 140)
    n5 = Plant('oak', children=[n1, n2,])
    n6 = Plant('conifer', children=[n3, n4,])
    n7b = Plant('tree', children=[n5, n6,])
    n8 = Node('birds and trees', [n7a, n7b,])
    n8.show()

if __name__ == '__main__':
    test()
```

Notes:

- The `show` method in class `Plant` calls the `show_name` method in its superclass using `self.show_name(...)`. Python searches up the inheritance tree to find the `show_name` method in class Node.

- The constructor (`__init__`) in classes `Plant` and `Animal` each call the constructor in the superclass by using the *name* of the superclass. Why the difference? Because, if (in the Plant class, for example) it used `self.__init__(...)` it would be calling the `__init__` in the `Plant` class, itself. So, it bypasses itself by referencing the constructor in the superclass directly.
- This exercise also demonstrates "polymorphism" -- The `show` method is called a number of times, but which implementation executes depends on which instance it is called on. Calling on the show method on an instance of class `Plant` results in a call to `Plant.show`. Calling the show method on an instance of class `Animal` results in a call to `Animal.show`. And so on. It is important that each show method takes the correct number of arguments.

3.7.3 Classes and polymorphism

Python also supports class-based polymorphism, which was, by the way, demonstrated in the previous example.

Exercises:

1. Write three classes, each of which implement a `show()` method that takes one argument, a string. The show method should print out the name of the class and the message. Then create a list of instances and call the `show()` method on each object in the list.

Solution:

1. We implement three simple classes and then create a list of instances of these classes:

```
class A(object):
    def show(self, msg):
        print 'class A -- msg: "%s"' % (msg, )

class B(object):
    def show(self, msg):
        print 'class B -- msg: "%s"' % (msg, )

class C(object):
    def show(self, msg):
        print 'class C -- msg: "%s"' % (msg, )

def test():
    objs = [A(), B(), C(), A(), ]
    for idx, obj in enumerate(objs):
        msg = 'message # %d' % (idx + 1, )
        obj.show(msg)
```

```
if __name__ == '__main__':
    test()
```

Notes:

- We can call the `show()` method in any object in the list `objs` as long as we pass in a single parameter, that is, as long as we obey the requirements of duck-typing. We can do this because all objects in that list implement a `show()` method.
- In a statically typed language, that is a language where the type is (also) present in the variable, all the instances in example would have to descend from a common superclass and that superclass would have to implement a `show()` method. Python does not impose this restriction. And, because variables are not not typed in Python, perhaps that would not even possible.
- Notice that this example of polymorphism works even though these three classes (`A`, `B`, and `C`) are not related (for example, in a class hierarchy). All that is required for polymorphism to work in Python is for the method names to be the same and the arguments to be compatible.

3.7.4 Recursive calls to methods

A method in a class can recusively call itself. This is very similar to the way in which we implemented recursive functions -- see: Recursive functions.

Exercises:

1. Re-implement the binary tree of animals and birds described in Recursive functions, but this time, use a class to represent each node in the tree.
2. Solve the same problem, but this time implement a tree in which each node can have any number of children (rather than exactly 2 children).

Solutions:

1. We implement a class with three instance variables: (1) name, (2) left branch, and (3) right branch. Then, we implement a `show()` method that displays the name and calls itself to show the children in each sub-tree:

```
Indents = ['    ' * idx for idx in range(10)]

class AnimalNode(object):

    def __init__(self, name, left_branch=None,
right_branch=None):
        self.name = name
        self.left_branch = left_branch
        self.right_branch = right_branch

    def show(self, level=0):
        print '%sname: %s' % (Indents[level],
self.name, )
        level += 1
        if self.left_branch is not None:
            self.left_branch.show(level)
        if self.right_branch is not None:
            self.right_branch.show(level)

Tree = AnimalNode('animals',
    AnimalNode('birds',
        AnimalNode('seed eaters',
            AnimalNode('house finch'),
            AnimalNode('white crowned sparrow'),
        ),
        AnimalNode('insect eaters',
            AnimalNode('hermit thrush'),
            AnimalNode('black headed phoebe'),
        ),
    ),
    None,
)

def test():
    Tree.show()

if __name__ == '__main__':
    test()
```

2. Instead of using a left branch and a right branch, in this solution we use a list to represent the children of a node:

```
class AnimalNode(object):
    def __init__(self, data, children=None):
        self.data = data
        if children is None:
            self.children = []
        else:
            self.children = children

    def show(self, level=''):
        print '%sdata: %s' % (level, self.data, )
        level += '    '
        for child in self.children:
            child.show(level)

Tree = AnimalNode('animals', [
    AnimalNode('birds', [
        AnimalNode('seed eaters', [
            AnimalNode('house finch'),
            AnimalNode('white crowned sparrow'),
            AnimalNode('lesser gold finch'),
        ]),
        AnimalNode('insect eaters', [
            AnimalNode('hermit thrush'),
            AnimalNode('black headed phoebe'),
        ]),
    ])
])

def test():
    Tree.show()

if __name__ == '__main__':
    test()
```

Notes:

- We represent the children of a node as a list. Each node "has-a" list of children.
- Notice that because a list is mutable, we do not use a list constructor (`[]`) in the initializer of the method header. Instead, we use None, then construct an empty list in the body of the method if necessary. See section Optional arguments and default values for more on this.
- We (recursively) call the show method for each node in the `children` list. Since a node which has no children (a leaf node) will have an empty `children` list, this provides a limit condition for our recursion.

3.7.5 Class variables, class methods, and static methods

A class variable is one whose single value is shared by all instances of the class and, in fact, is shared by all who have access to the class (object).

"Normal" methods are instance methods. An instance method receives the instance as its first argument. A instance method is defined by using the `def` statement in the body of a `class` statement.

A class method receives the class as its first argument. A class method is defined by defining a normal/instance method, then using the `classmethod` built-in function. For example:

```
class ASimpleClass(object):
    description = 'a simple class'
    def show_class(cls, msg):
        print '%s: %s' % (cls.description , msg, )
        show_class = classmethod(show_class)
```

A static method does *not* receive anything special as its first argument. A static method is defined by defining a normal/instance method, then using the `staticmethod` built-in function. For example:

```
class ASimpleClass(object):
    description = 'a simple class'
    def show_class(msg):
        print '%s: %s' % (ASimpleClass.description , msg, )
        show_class = staticmethod(show_class)
```

In effect, both class methods and static methods are defined by creating a normal (instance) method, then creating a wrapper object (a class method or static method) using the `classmethod` or `staticmethod` built-in function.

Exercises:

1. Implement a class that keeps a running total of the number of instances created.
2. Implement another solution to the same problem (a class that keeps a running total of the number of instances), but this time use a static method instead of a class method.

Solutions:

1. We use a class variable named `instance_count`, rather than an instance variable, to keep a running total of instances. Then, we increment that variable each time an instance is created:

```
class CountInstances(object):

    instance_count = 0

    def __init__(self, name='-no name-'):
        self.name = name
        CountInstances.instance_count += 1

    def show(self):
        print 'name: "%s"' % (self.name, )

    def show_instance_count(cls):
        print 'instance count: %d' %
(cls.instance_count, )
    show_instance_count =
classmethod(show_instance_count)


def test():
    instances = []
    instances.append(CountInstances('apple'))
    instances.append(CountInstances('banana'))
    instances.append(CountInstances('cherry'))
    instances.append(CountInstances())
    for instance in instances:
        instance.show()
    CountInstances.show_instance_count()


if __name__ == '__main__':
    test()
```

Notes:

- When we run this script, it prints out the following:

  ```
  name: "apple"
  name: "banana"
  name: "cherry"
  name: "-no name-"
  instance count: 4
  ```

- The call to the `classmethod` built-in function effectively wraps the `show_instance_count` method in a class method, that is, in a method that takes a class object as its first argument rather than an instance object. To read more about `classmethod`, go to Built-in Functions -- http://docs.python.org/lib/built-in-funcs.html and search for "classmethod".

2. A static method takes neither an instance (`self`) nor a class as its first paramenter. And, static method is created with the `staticmethod()` built-in function (rather than with the `classmethod()` built-in):

   ```
   class CountInstances(object):

       instance_count = 0

       def __init__(self, name='-no name-'):
           self.name = name
           CountInstances.instance_count += 1

       def show(self):
           print 'name: "%s"' % (self.name, )

       def show_instance_count():
           print 'instance count: %d' % (
               CountInstances.instance_count, )
       show_instance_count =
   staticmethod(show_instance_count)

   def test():
       instances = []
       instances.append(CountInstances('apple'))
       instances.append(CountInstances('banana'))
       instances.append(CountInstances('cherry'))
       instances.append(CountInstances())
       for instance in instances:
           instance.show()
       CountInstances.show_instance_count()

   if __name__ == '__main__':
       test()
   ```

3.7.5.1 Decorators for classmethod and staticmethod

A decorator enables us to do what we did in the previous example with a somewhat simpler syntax.

For simple cases, the decorator syntax enables us to do this:

```
@functionwrapper
def method1(self):
    o
    o
    o
```

instead of this:

```
def method1(self):
    o
    o
    o
method1 = functionwrapper(method1)
```

So, we can write this:

```
@classmethod
def method1(self):
    o
    o
    o
```

instead of this:

```
def method1(self):
    o
    o
    o
method1 = classmethod(method1)
```

Exercises:

1. Implement the `CountInstances` example above, but use a decorator rather than the explicit call to `classmethod`.

Solutions:

1. A decorator is an easier and cleaner way to define a class method (or a static method):

```
class CountInstances(object):

    instance_count = 0

    def __init__(self, name='-no name-'):
        self.name = name
        CountInstances.instance_count += 1

    def show(self):
        print 'name: "%s"' % (self.name, )

    @classmethod
    def show_instance_count(cls):
        print 'instance count: %d' %
(cls.instance_count, )
    # Note that the following line has been replaced by
    #   the classmethod decorator, above.
    # show_instance_count =
classmethod(show_instance_count)

def test():
    instances = []
    instances.append(CountInstances('apple'))
    instances.append(CountInstances('banana'))
    instances.append(CountInstances('cherry'))
    instances.append(CountInstances())
    for instance in instances:
        instance.show()
    CountInstances.show_instance_count()

if __name__ == '__main__':
    test()
```

3.8 Additional and Advanced Topics

3.8.1 Decorators and how to implement them

Decorators can be used to "wrap" a function with another function.

When implementing a decorator, it is helpful to remember that the following decorator application:

```
@dec
def func(arg1, arg2):
    pass
```

is equivalent to:

```
def func(arg1, arg2):
    pass
func = dec(func)
```

Therefore, to implement a decorator, we write a function that returns a function object, since we replace the value originally bound to the function with this new function object. It may be helpful to take the view that we are creating a function that is a *wrapper* for the original function.

Exercises:

1. Write a decorator that writes a message before and after executing a function.

Solutions:

1. A function that contains and returns an inner function can be used to wrap a function:

   ```
   def trace(func):
       def inner(*args, **kwargs):
           print '>>'
           func(*args, **kwargs)
           print '<<'
       return inner

   @trace
   def func1(x, y):
       print 'x:', x, 'y:', y
       func2((x, y))

   @trace
   def func2(content):
       print 'content:', content

   def test():
       func1('aa', 'bb')

   test()
   ```

   Notes:

   - Your inner function can use `*args` and `**kwargs` to enable it to call functions with any number of arguments.

3.8.1.1 Decorators with arguments

Decorators can also take arguments.

The following decorator with arguments:

```
@dec(argA, argB)
def func(arg1, arg2):
    pass
```

is equivalent to:

```
def func(arg1, arg2):
    pass
func = dec(argA, argB)(func)
```

Because the decorator's arguments are passed to the result of calling the decorator on the decorated function, you may find it useful to implement a decorator with arguments using a function inside a function inside a function.

Exercises:

1. Write and test a decorator that takes one argument. The decorator prints a message along with the value of the argument before and after entering the

decorated function.

Solutions:

1. Implement this decorator that takes arguments with a function containing a nested function which in turn contains a nested function:

   ```
   def trace(msg):
       def inner1(func):
           def inner2(*args, **kwargs):
               print '>> [%s]' % (msg, )
               retval = func(*args, **kwargs)
               print '<< [%s]' % (msg, )
               return retval
           return inner2
       return inner1

   @trace('tracing func1')
   def func1(x, y):
       print 'x:', x, 'y:', y
       result = func2((x, y))
       return result

   @trace('tracing func2')
   def func2(content):
       print 'content:', content
       return content * 3

   def test():
       result = func1('aa', 'bb')
       print 'result:', result

   test()
   ```

3.8.1.2 Stacked decorators

Decorators can be "stacked".

The following stacked decorators:

```
@dec2
@dec1
def func(arg1, arg2, ...):
    pass
```

are equivalent to:

```
def func(arg1, arg2, ...):
    pass
func = dec2(dec1(func))
```

Exercises:

1. Implement a decorator (as above) that traces calls to a decorated function. Then

"stack" that with another decorator that prints a horizontal line of dashes before and after calling the function.

2. Modify your solution to the above exercise so that the decorator that prints the horizontal line takes one argument: a character (or characters) that can be repeated to produce a horizontal line/separator.

Solutions:

1. Reuse your tracing function from the previous exercise, then write a simple decorator that prints a row of dashes:

```
def trace(msg):
    def inner1(func):
        def inner2(*args, **kwargs):
            print '>> [%s]' % (msg, )
            retval = func(*args, **kwargs)
            print '<< [%s]' % (msg, )
            return retval
        return inner2
    return inner1

def horizontal_line(func):
    def inner(*args, **kwargs):
        print '-' * 50
        retval = func(*args, **kwargs)
        print '-' * 50
        return retval
    return inner


@trace('tracing func1')
def func1(x, y):
    print 'x:', x, 'y:', y
    result = func2((x, y))
    return result

@horizontal_line
@trace('tracing func2')
def func2(content):
    print 'content:', content
    return content * 3

def test():
    result = func1('aa', 'bb')
    print 'result:', result

test()
```

2. Once again, a decorator with arguments can be implemented with a function nested inside a function which is nested inside a function. This remains the same whether the decorator is used as a *stacked* decorator or not. Here is a solution:

```
def trace(msg):
    def inner1(func):
        def inner2(*args, **kwargs):
            print '>> [%s]' % (msg, )
            retval = func(*args, **kwargs)
            print '<< [%s]' % (msg, )
            return retval
        return inner2
    return inner1

def horizontal_line(line_chr):
    def inner1(func):
        def inner2(*args, **kwargs):
            print line_chr * 15
            retval = func(*args, **kwargs)
            print line_chr * 15
            return retval
        return inner2
    return inner1

@trace('tracing func1')
def func1(x, y):
    print 'x:', x, 'y:', y
    result = func2((x, y))
    return result

@horizontal_line('<**>')
@trace('tracing func2')
def func2(content):
    print 'content:', content
    return content * 3

def test():
    result = func1('aa', 'bb')
    print 'result:', result

test()
```

3.8.1.3 More help with decorators

There is more about decorators here:

- Python syntax and semantics -- http://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators at Wikipedia.
- PythonDecoratorLibrary -- http://wiki.python.org/moin/PythonDecoratorLibrary at the Python Wiki has lots of sample code.
- PEP 318 -- Decorators for Functions and Methods -- http://www.python.org/dev/peps/pep-0318/ is the formal proposal and specification for Python decorators.

3.8.2 Iterables

3.8.2.1 A few preliminaries on Iterables

Definition: iterable (adjective) -- that which can be iterated over.

A good test of whether something is iterable is whether it can be used in a `for:` statement. For example, if we can write `for item in X:`, then `X` is iterable. Here is another simple test:

```
def isiterable(x):
    try:
        y = iter(x)
    except TypeError, exp:
        return False
    return True
```

Some kinds of iterables:

- Containers -- We can iterate over lists, tuples, dictionaries, sets, strings, and other containers.
- Some built-in (non-container) types -- Examples:
  - A text file open in read mode is iterable: it iterates over the lines in the file.
  - The xrange type -- See XRange Type http://docs.python.org/lib/typesseq-xrange.html. It's useful when you want a large sequence of integers to iterate over.
- Instances of classes that obey the iterator protocol. For a description of the iterator protocol, see Iterator Types -- http://docs.python.org/lib/typeiter.html. Hint: Type `dir(obj)` and look for "__iter__" and "next".
- Generators -- An object returned by any function or method that contains `yield`.

Exercises:

1. Implement a class whose instances are interable. The constructor takes a list of URLs as its argument. An instance of this class, when iterated over, generates the content of the Web page at that address.

Solutions:

1. We implement a class that has `__iter__()` and `next()` methods:

```
import urllib

class WebPages(object):
    def __init__(self, urls):
        self.urls = urls
        self.current_index = 0
    def __iter__(self):
        self.current_index = 0
        return self
    def next(self):
        if self.current_index >= len(self.urls):
            raise StopIteration
        url = self.urls[self.current_index]
        self.current_index += 1
        f = urllib.urlopen(url)
        content = f.read()
        f.close()
        return content

def test():
    urls = [
        'http://www.python.org',
        'http://en.wikipedia.org/',

'http://en.wikipedia.org/wiki/Python_(programming_langu
age)',
        ]
    pages = WebPages(urls)
    for page in pages:
        print 'length: %d' % (len(page), )
    pages = WebPages(urls)
    print '-' * 50
    page = pages.next()
    print 'length: %d' % (len(page), )
    page = pages.next()
    print 'length: %d' % (len(page), )
    page = pages.next()
    print 'length: %d' % (len(page), )
    page = pages.next()
    print 'length: %d' % (len(page), )

test()
```

3.8.2.2 More help with iterables

The `itertools` module in the Python standard library has helpers for iterators: http://docs.python.org/library/itertools.html#module-itertools

3.9 Applications and Recipes

3.9.1 XML -- SAX, minidom, ElementTree, Lxml

Exercises:

1. SAX -- Parse an XML document with SAX, then show some information (tag, attributes, character data) for each element.
2. Minidom -- Parse an XML document with `minidom`, then walk the DOM tree and show some information (tag, attributes, character data) for each element.

Here is a sample XML document that you can use for input:

```
<?xml version="1.0"?>
<people>
    <person id="1" value="abcd" ratio="3.2">
        <name>Alberta</name>
        <interest>gardening</interest>
        <interest>reading</interest>
        <category>5</category>
    </person>
    <person id="2">
        <name>Bernardo</name>
        <interest>programming</interest>
        <category></category>
        <agent>
            <firstname>Darren</firstname>
            <lastname>Diddly</lastname>
        </agent>
    </person>
    <person id="3" value="efgh">
        <name>Charlie</name>
        <interest>people</interest>
        <interest>cats</interest>
        <interest>dogs</interest>
        <category>8</category>
        <promoter>
            <firstname>David</firstname>
            <lastname>Donaldson</lastname>
            <client>
                <fullname>Arnold Applebee</fullname>
                <refid>10001</refid>
            </client>
        </promoter>
        <promoter>
            <firstname>Edward</firstname>
            <lastname>Eddleberry</lastname>
            <client>
                <fullname>Arnold Applebee</fullname>
                <refid>10001</refid>
            </client>
        </promoter>
    </person>
</people>
```

3. ElementTree -- Parse an XML document with ElementTree, then walk the DOM tree and show some information (tag, attributes, character data) for each element.
4. lxml -- Parse an XML document with lxml, then walk the DOM tree and show some information (tag, attributes, character data) for each element.
5. Modify document with ElementTree -- Use ElementTree to read a document, then modify the tree. Show the contents of the tree, and then write out the modified document.
6. XPath -- lxml supports XPath. Use the XPath support in lxml to address each of

the following in the above XML instance document:

- The text in all the `name` elements
- The values of all the `id` attributes

Solutions:

1. We can use the SAX support in the Python standard library:

```
#!/usr/bin/env python

"""
Parse and XML with SAX.  Display info about each
element.

Usage:
    python test_sax.py infilename
Examples:
    python test_sax.py people.xml
"""

import sys
from xml.sax import make_parser, handler

class TestHandler(handler.ContentHandler):
    def __init__(self):
        self.level = 0

    def show_with_level(self, value):
        print '%s%s' % ('    ' * self.level, value, )

    def startDocument(self):
        self.show_with_level('Document start')
        self.level += 1

    def endDocument(self):
        self.level -= 1
        self.show_with_level('Document end')

    def startElement(self, name, attrs):
        self.show_with_level('start element -- name:
"%s"' % (name, ))
        self.level += 1

    def endElement(self, name):
        self.level -= 1
        self.show_with_level('end element -- name:
"%s"' % (name, ))

    def characters(self, content):
        content = content.strip()
        if content:
            self.show_with_level('characters: "%s"' %
(content, ))
```

```
def test(infilename):
    parser = make_parser()
    handler = TestHandler()
    parser.setContentHandler(handler)
    parser.parse(infilename)

def usage():
    print __doc__
    sys.exit(1)

def main():
    args = sys.argv[1:]
    if len(args) != 1:
        usage()
    infilename = args[0]
    test(infilename)

if __name__ == '__main__':
    main()
```

2. The minidom module contains a `parse()` function that enables us to read an XML document and create a DOM tree:

```
#!/usr/bin/env python

"""Process an XML document with minidom.

Show the document tree.

Usage:
    python minidom_walk.py [options] infilename
"""

import sys
from xml.dom import minidom

def show_tree(doc):
    root = doc.documentElement
    show_node(root, 0)

def show_node(node, level):
    count = 0
    if node.nodeType == minidom.Node.ELEMENT_NODE:
        show_level(level)
        print 'tag: %s' % (node.nodeName, )
        for key in node.attributes.keys():
            attr = node.attributes.get(key)
            show_level(level + 1)
            print '- attribute name: %s  value: "%s"' %
(attr.name,
                attr.value, )
        if (len(node.childNodes) == 1 and
            node.childNodes[0].nodeType ==
minidom.Node.TEXT_NODE):
            show_level(level + 1)
            print '- data: "%s"' %
(node.childNodes[0].data, )
        for child in node.childNodes:
            count += 1
            show_node(child, level + 1)
    return count

def show_level(level):
    for x in range(level):
        print '    ',

def test():
    args = sys.argv[1:]
    if len(args) != 1:
        print __doc__
        sys.exit(1)
    docname = args[0]
    doc = minidom.parse(docname)
    show_tree(doc)

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    test()
```

3. ElementTree enables us to parse an XML document and create a DOM tree:

```
#!/usr/bin/env python

"""Process an XML document with elementtree.

Show the document tree.

Usage:
    python elementtree_walk.py [options] infilename
"""

import sys
from xml.etree import ElementTree as etree

def show_tree(doc):
    root = doc.getroot()
    show_node(root, 0)

def show_node(node, level):
    show_level(level)
    print 'tag: %s' % (node.tag, )
    for key, value in node.attrib.iteritems():
        show_level(level + 1)
        print '- attribute -- name: %s  value: "%s"' %
(key, value, )
    if node.text:
        text = node.text.strip()
        show_level(level + 1)
        print '- text: "%s"' % (node.text, )
    if node.tail:
        tail = node.tail.strip()
        show_level(level + 1)
        print '- tail: "%s"' % (tail, )
    for child in node.getchildren():
        show_node(child, level + 1)

def show_level(level):
    for x in range(level):
        print '   ',

def test():
    args = sys.argv[1:]
    if len(args) != 1:
        print __doc__
        sys.exit(1)
    docname = args[0]
    doc = etree.parse(docname)
    show_tree(doc)

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    test()
```

4. lxml enables us to parse an XML document and create a DOM tree. In fact, since lxml attempts to mimic the ElementTree API, our code is very similar to that in the solution to the ElementTree exercise:

```
#!/usr/bin/env python

"""Process an XML document with elementtree.

Show the document tree.

Usage:
    python lxml_walk.py [options] infilename
"""

#
# Imports:
import sys
from lxml import etree

def show_tree(doc):
    root = doc.getroot()
    show_node(root, 0)

def show_node(node, level):
    show_level(level)
    print 'tag: %s' % (node.tag, )
    for key, value in node.attrib.iteritems():
        show_level(level + 1)
        print '- attribute -- name: %s  value: "%s"' %
(key, value, )
    if node.text:
        text = node.text.strip()
        show_level(level + 1)
        print '- text: "%s"' % (node.text, )
    if node.tail:
        tail = node.tail.strip()
        show_level(level + 1)
        print '- tail: "%s"' % (tail, )
    for child in node.getchildren():
        show_node(child, level + 1)

def show_level(level):
    for x in range(level):
        print '   ',

def test():
    args = sys.argv[1:]
    if len(args) != 1:
        print __doc__
        sys.exit(1)
    docname = args[0]
    doc = etree.parse(docname)
    show_tree(doc)

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    test()
```

5. We can modify the DOM tree and write it out to a new file:

```
#!/usr/bin/env python

"""Process an XML document with elementtree.

Show the document tree.
Modify the document tree and then show it again.
Write the modified XML tree to a new file.

Usage:
    python elementtree_walk.py [options] infilename
outfilename
Options:
    -h, --help      Display this help message.
Example:
    python elementtree_walk.py myxmldoc.xml
myotherxmldoc.xml
"""

import sys
import os
import getopt
import time

# Use ElementTree.
from xml.etree import ElementTree as etree
# Or uncomment to use Lxml.
#from lxml import etree

def show_tree(doc):
    root = doc.getroot()
    show_node(root, 0)

def show_node(node, level):
    show_level(level)
    print 'tag: %s' % (node.tag, )
    for key, value in node.attrib.iteritems():
        show_level(level + 1)
        print '- attribute -- name: %s  value: "%s"' %
(key, value, )
    if node.text:
        text = node.text.strip()
        show_level(level + 1)
        print '- text: "%s"' % (node.text, )
    if node.tail:
        tail = node.tail.strip()
        show_level(level + 1)
        print '- tail: "%s"' % (tail, )
    for child in node.getchildren():
        show_node(child, level + 1)

def show_level(level):
    for x in range(level):
        print '   ',

def modify_tree(doc, tag, attrname, attrvalue):
    root = doc.getroot()
    modify_node(root, tag, attrname, attrvalue)

def modify_node(node, tag, attrname, attrvalue):
    if node.tag == tag:
        node.attrib[attrname] = attrvalue
    for child in node.getchildren():
        modify_node(child, tag, attrname, attrvalue)

def test(indocname, outdocname):
    doc = etree.parse(indocname)
    show_tree(doc)
    print '-' * 50
    date = time.ctime()
    modify_tree(doc, 'person', 'date', date)
    show_tree(doc)
    write_output = False
    if os.path.exists(outdocname):
        response = raw_input('Output file (%s) exists.
Over-write? (y/n): ' %
            outdocname)
        if response == 'y':
            write_output = True
    else:
        write_output = True
    if write_output:
        doc.write(outdocname)
        print 'Wrote modified XML tree to %s' %
outdocname
    else:
        print 'Did not write output file.'

def usage():
    print __doc__
    sys.exit(1)

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help',
            ])
    except:
        usage()
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 2:
        usage()
    indocname = args[0]
    outdocname = args[1]
    test(indocname, outdocname)

if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    main()
```

Notes:

- The above solution contains an `import` statement for ElementTree and another for lxml. The one for lxml is commented out, but you could change that if you wish to use lxml instead of ElementTree. This solution will work the same way with either ElementTree or lxml.

6. When we parse and XML document with lxml, each element (node) has an `xpath()` method.

```
# test_xpath.py

from lxml import etree

def test():
    doc = etree.parse('people.xml')
    root = doc.getroot()
    print root.xpath("//name/text()")
    print root.xpath("//@id")

test()
```

And, when we run the above code, here is what we see:

```
$ python test_xpath.py
['Alberta', 'Bernardo', 'Charlie']
['1', '2', '3']
```

For more on XPath see: XML Path Language (XPath) -- http://www.w3.org/TR/xpath

3.9.2 Relational database access

You can find information about database programming in Python here: Database Programming -- http://wiki.python.org/moin/DatabaseProgramming/.

For database access we use the Python Database API. You can find information about it here: Python Database API Specification v2.0 -- http://www.python.org/dev/peps/pep-0249/.

To use the database API we do the following:

1. Use the database interface module to create a connection object.
2. Use the connection object to create a cursor object.
3. Use the cursor object to execute an SQL query.
4. Retrieve rows from the cursor object, if needed.
5. Optionally, commit results to the database.
6. Close the connection object.

Our examples use the `gadfly` database, which is written in Python. If you want to use `gadfly`, you can find it here: http://gadfly.sourceforge.net/. `gadfly` is a reasonable choice if you want an easy to use database on your local machine.

Another reasonable choice for a local database is `sqlite3`, which is in the Python standard library. Here is a descriptive quote from the `SQLite` Web site:

> "SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain."

You can learn about it here:

- sqlite3 - DB-API 2.0 interface for SQLite databases -- http://docs.python.org/library/sqlite3.html
- SQLite home page -- http://www.sqlite.org/

- The pysqlite web page -- http://oss.itsystementwicklung.de/trac/pysqlite/

If you want or need to use another, enterprise class database, for example PostgreSQL, MySQL, Oracle, etc., you will need an interface module for your specific database. You can find information about database interface modules here: Database interfaces -- http://wiki.python.org/moin/DatabaseInterfaces

Excercises:

1. Write a script that retrieves all the rows in a table and prints each row.
2. Write a script that retrieves all the rows in a table, then uses the cursor as an iterator to print each row.
3. Write a script that uses the cursor's `description` attribute to print out the name and value of each field in each row.
4. Write a script that performs several of the above tasks, but uses `sqlite3` instead of `gadfly`.

Solutions:

1. We can `execute` a SQL query and then retrieve all the rows with `fetchall()`:

```
import gadfly

def test():
    connection = gadfly.connect("dbtest1",
"plantsdbdir")
    cur = connection.cursor()
    cur.execute('select * from plantsdb order by
p_name')
    rows = cur.fetchall()
    for row in rows:
        print '2. row:', row
    connection.close()

test()
```

2. The cursor itself is an iterator. It iterates over the rows returned by a query. So, we execute a SQL query and then we use the cursor in a `for:` statement:

```
import gadfly

def test():
    connection = gadfly.connect("dbtest1",
"plantsdbdir")
    cur = connection.cursor()
    cur.execute('select * from plantsdb order by
p_name')
    for row in cur:
        print row
    connection.close()
```

```
test()
```

3. The description attribute in the cursor is a container that has an item describing each field:

```
import gadfly

def test():
    cur.execute('select * from plantsdb order by
p_name')
    for field in cur.description:
        print 'field:', field
    rows = cur.fetchall()
    for row in rows:
        for idx, field in enumerate(row):
            content = '%s: "%s"' %
(cur.description[idx][0], field, )
            print content,
        print
    connection.close()

test()
```

Notes:

- The comma at the end of the `print` statement tells Python not to print a new-line.
- The `cur.description` is a sequence containing an item for each field. After the query, we can extract a description of each field.

4. The solutions using `sqlite3` are very similar to those using `gadfly`. For information on `sqlite3`, see: sqlite3 — DB-API 2.0 interface for SQLite databases http://docs.python.org/library/sqlite3.html#module-sqlite3.

```
#!/usr/bin/env python

"""
Perform operations on sqlite3 (plants) database.

Usage:
    python py_db_api.py command [arg1, ... ]
Commands:
    create -- create new database.
    show -- show contents of database.
    add -- add row to database.  Requires 3 args (name,
descrip, rating).
    delete - remove row from database.  Requires 1 arg
(name).
Examples:
    python test1.py create
    python test1.py show
    python test1.py add crenshaw "The most succulent
melon" 10
    python test1.py delete lemon
```

```
"""


import sys
import sqlite3

Values = [
    ('lemon', 'bright and yellow', '7'),
    ('peach', 'succulent', '9'),
    ('banana', 'smooth and creamy', '8'),
    ('nectarine', 'tangy and tasty', '9'),
    ('orange', 'sweet and tangy', '8'),
    ]

Field_defs = [
    'p_name varchar',
    'p_descrip varchar',
    #'p_rating integer',
    'p_rating varchar',
    ]


def createdb():
    connection = sqlite3.connect('sqlite3plantsdb')
    cursor = connection.cursor()
    q1 = "create table plantsdb (%s)" % (',
'.join(Field_defs))
    print 'create q1: %s' % q1
    cursor.execute(q1)
    q1 = "create index index1 on plantsdb(p_name)"
    cursor.execute(q1)
    q1 = "insert into plantsdb (p_name, p_descrip,
p_rating) values ('%s', '%s', %s)"
    for spec in Values:
        q2 = q1 % spec
        print 'q2: "%s"' % q2
        cursor.execute(q2)
    connection.commit()
    showdb1(cursor)
    connection.close()


def showdb():
    connection, cursor = opendb()
    showdb1(cursor)
    connection.close()


def showdb1(cursor):
    cursor.execute("select * from plantsdb order by
p_name")
    hr()
    description = cursor.description
    print description
    print 'description:'
    for rowdescription in description:
        print '    %s' % (rowdescription, )
    hr()
    rows = cursor.fetchall()
    print rows
    print 'rows:'
    for row in rows:
        print '    %s' % (row, )
    hr()
    print 'content:'
    for row in rows:
        descrip = row[1]
        name = row[0]
        rating = '%s' % row[2]
        print '    %s%s%s' % (
            name.ljust(12), descrip.ljust(30),
rating.rjust(4), )


def addtodb(name, descrip, rating):
    try:
        rating = int(rating)
    except ValueError, exp:
        print 'Error: rating must be integer.'
        return
    connection, cursor = opendb()
    cursor.execute("select * from plantsdb where p_name
= '%s'" % name)
    rows = cursor.fetchall()
    if len(rows) > 0:
        ql = "update plantsdb set p_descrip='%s',
p_rating='%s' where p_name='%s'" % (
            descrip, rating, name, )
        print 'ql:', ql
        cursor.execute(ql)
        connection.commit()
        print 'Updated'
    else:
        cursor.execute("insert into plantsdb values
('%s', '%s', '%s')" % (
            name, descrip, rating))
        connection.commit()
        print 'Added'
    showdb1(cursor)
    connection.close()


def deletefromdb(name):
    connection, cursor = opendb()
    cursor.execute("select * from plantsdb where p_name
= '%s'" % name)
```

```
    rows = cursor.fetchall()
    if len(rows) > 0:
        cursor.execute("delete from plantsdb where 
p_name='%s'" % name)
        connection.commit()
        print 'Plant (%s) deleted.' % name
    else:
        print 'Plant (%s) does not exist.' % name
    showdb1(cursor)
    connection.close()


def opendb():
    connection = sqlite3.connect("sqlite3plantsdb")
    cursor = connection.cursor()
    return connection, cursor


def hr():
    print '-' * 60


def usage():
    print __doc__
    sys.exit(1)


def main():
    args = sys.argv[1:]
    if len(args) < 1:
        usage()
    cmd = args[0]
    if cmd == 'create':
        if len(args) != 1:
            usage()
        createdb()
    elif cmd == 'show':
        if len(args) != 1:
            usage()
        showdb()
    elif cmd == 'add':
        if len(args) < 4:
            usage()
        name = args[1]
        descrip = args[2]
        rating = args[3]
        addtodb(name, descrip, rating)
    elif cmd == 'delete':
        if len(args) < 2:
            usage()
        name = args[1]
        deletefromdb(name)
    else:
        usage()

if __name__ == '__main__':
    main()
```

3.9.3 CSV -- comma separated value files

There is support for parsing and generating CSV files in the Python standard library. See: csv — CSV File Reading and Writing http://docs.python.org/library/csv.html#module-csv.

Exercises:

1. Read a CSV file and print the fields in columns. Here is a sample file to use as input:

```
# name  description  rating
Lemon,Bright yellow and tart,5
Eggplant,Purple and shiny,6
Tangerine,Succulent,8
```

Solutions:

1. Use the CSV module in the Python standard library to read a CSV file:

```
"""
Read a CSV file and print the contents in columns.
"""

import csv

def test(infilename):
    infile = open(infilename)
    reader = csv.reader(infile)
    print '====                 ===========
======'
    print 'Name                 Description
Rating'
    print '====                 ===========
======'
    for fields in reader:
        if len(fields) == 3:
            line = '%s %s %s' % (fields[0].ljust(20),
                fields[1].ljust(40),
fields[2].ljust(4))
            print line
    infile.close()

def main():
    infilename = 'csv_report.csv'
    test(infilename)
```

```
if __name__ == '__main__':
    main()
```

And, when run, here is what it displays:

```
====                 ===========
======
Name                 Description
Rating
====                 ===========
======
Lemon                Bright yellow and tart
5
Eggplant             Purple and shiny
6
Tangerine            Succulent
8
```

3.9.4 YAML and PyYAML

YAML is a structured text data representation format. It uses indentation to indicate nesting. Here is a description from the YAML Web site:

> "YAML: YAML Ain't Markup Language
>
> "What It Is: YAML is a human friendly data serialization standard for all programming languages."

You can learn more about YAML and PyYAML here:

- The Official YAML Web Site -- http://yaml.org/
- PyYAML.org - the home of various YAML implementations for Python -- http://pyyaml.org/
- The YAML 1.2 specification -- http://yaml.org/spec/1.2/

Exercises:

1. Read the following sample YAML document. Print out the information in it:

```
american:
  - Boston Red Sox
  - Detroit Tigers
  - New York Yankees
national:
  - New York Mets
  - Chicago Cubs
  - Atlanta Braves
```

2. Load the YAML data used in the previous exercise, then make a modification (for example, add "San Francisco Giants" to the National League), then dump the modified data to a new file.

Solutions:

1. Printing out information from YAML is as "simple" as printing out a Python data structure. In this solution, we use the pretty printer from the Python standard library:

   ```
   import yaml
   import pprint

   def test():
       infile = open('test1.yaml')
       data = yaml.load(infile)
       infile.close()
       pprint.pprint(data)

   test()
   ```

   We could, alternatively, read in and then "load" from a string:

   ```
   import yaml
   import pprint

   def test():
       infile = open('test1.yaml')
       data_str = infile.read()
       infile.close()
       data = yaml.load(data_str)
       pprint.pprint(data)

   test()
   ```

2. The YAML `dump()` function enables us to dump data to a file:

   ```
   import yaml
   import pprint

   def test():
       infile = open('test1.yaml', 'r')
       data = yaml.load(infile)
       infile.close()
       data['national'].append('San Francisco Giants')
       outfile = open('test1_new.yaml', 'w')
       yaml.dump(data, outfile)
       outfile.close()

   test()
   ```

   Notes:

   - If we want to produce the standard YAML "block" style rather than the "flow" format, then we could use:

     ```
     yaml.dump(data, outfile, default_flow_style=False)
     ```

3.9.5 Json

Here is a quote from Wikipedia entry for Json:

> "JSON (pronounced 'Jason'), short for JavaScript Object Notation, is a lightweight computer data interchange format. It is a text-based, human-readable format for representing simple data structures and associative arrays (called objects)."

The Json text representation looks very similar to Python literal representation of Python builtin data types (for example, lists, dictionaries, numbers, and strings).

Learn more about Json and Python support for Json here:

- Introducing JSON -- http://json.org/
- Json at Wikipedia -- http://en.wikipedia.org/wiki/Json
- python-json -- http://pypi.python.org/pypi/python-json
- simplejson -- http://pypi.python.org/pypi/simplejson

Excercises:

1. Write a Python script, using your favorite Python Json implementation (for example `python-json` or `simplejson`), that dumps the following data structure to a file:

   ```
   Data = {
       'rock and roll':
           ['Elis', 'The Beatles', 'The Rolling Stones',],
       'country':
           ['Willie Nelson', 'Hank Williams', ]
       }
   ```

2. Write a Python script that reads Json data from a file and loads it into Python data structures.

Solutions:

1. This solution uses `simplejson` to store a Python data structure encoded as Json in a file:

   ```
   import simplejson as json

   Data = {
       'rock and roll':
           ['Elis', 'The Beatles', 'The Rolling Stones',],
       'country':
           ['Willie Nelson', 'Hank Williams', ]
       }

   def test():
       fout = open('tmpdata.json', 'w')
       content = json.dumps(Data)
       fout.write(content)
   ```

```
    fout.write('\n')
    fout.close()

test()
```

2. We can read the file into a string, then decode it from Json:

```
import simplejson as json

def test():
    fin = open('tmpdata.json', 'r')
    content = fin.read()
    fin.close()
    data = json.loads(content)
    print data

test()
```

Note that you may want some control over indentation, character encoding, etc. For `simplejson`, you can learn about that here: simplejson - JSON encoder and decoder -- http://simplejson.googlecode.com/svn/tags/simplejson-2.0.1/docs/index.html.

4 Part 4 -- Generating Python Bindings for XML

This section discusses a specific Python tool, specifically a Python code generator that generates Python bindings for XML files.

Thus, this section will help you in the following ways:

1. It will help you learn to use a specific tool, namely `generateDS.py`, that generates Python code to be used to process XML instance documents of a particular document type.
2. It will help you gain more experience with reading, modifying and using Python code.

4.1 Introduction

Additional information:

- If you plan to work through this tutorial, you may find it helpful to look at the sample code that accompanies this tutorial. You can find it in the distribution under:

  ```
  tutorial/
  tutorial/Code/
  ```

- You can find additional information about `generateDS.py` here:
  http://http://www.davekuhlman.org/#generateds-py

  That documentation is also included in the distribution.

`generateDS.py` generates Python data structures (for example, class definitions) from an XML schema document. These data structures represent the elements in an XML document described by the XML schema. `generateDS.py` also generates parsers that load an XML document into those data structures. In addition, a separate file containing subclasses (stubs) is optionally generated. The user can add methods to the subclasses in order to process the contents of an XML document.

The generated Python code contains:

- A class definition for each element defined in the XML schema document.
- A main and driver function that can be used to test the generated code.
- A parser that will read an XML document which satisfies the XML schema from which the parser was generated. The parser creates and populates a tree structure of instances of the generated Python classes.
- Methods in each class to export the instance back out to XML (method `export`) and to export the instance to a literal representing the Python data structure

(method `exportLiteral`).

Each generated class contains the following:

- A constructor method (__init__), with member variable initializers.
- Methods with names `get_xyz` and `set_xyz` for each member variable "xyz" or, if the member variable is defined with `maxOccurs="unbounded"`, methods with names `get_xyz`, `set_xyz`, `add_xyz`, and `insert_xyz`. (Note: If you use the `--use-old-getter-setter`, then you will get methods with names like `getXyz` and `setXyz`.)
- A `build` method that can be used to populate an instance of the class from a node in an ElementTree or Lxml tree.
- An `export` method that will write the instance (and any nested sub-instances) to a file object as XML text.
- An `exportLiteral` method that will write the instance (and any nested sub-instances) to a file object as Python literals (text).

The generated subclass file contains one (sub-)class definition for each data representation class. If the subclass file is used, then the parser creates instances of the subclasses (instead of creating instances of the superclasses). This enables the user to extend the subclasses with "tree walk" methods, for example, that process the contents of the XML file. The user can also generate and extend multiple subclass files which use a single, common superclass file, thus implementing a number of different processes on the same XML document type.

This document introduces the user to `generateDS.py` and walks the user through several examples that show how to generate Python code and how to use that generated code.

4.2 Generating the code

Note: The sample files used below are under the `tutorial/Code/` directory.

Use the following to get help:

```
$ generateDS.py --help
```

I'll assume that `generateDS.py` is in a directory on your path. If not, you should do whatever is necessary to make it accessible and executable.

Here is a simple XML schema document:

And, here is how you might generate classes and subclasses that provide data bindings (a Python API) for the definitions in that schema:

```
$ generateDS.py -o people_api.py -s people_sub.py people.xsd
```

And, if you want to automatically over-write the generated Python files, use the `-f` command line flag to force over-write without asking:

```
$ generateDS.py -f -o people_api.py -s people_sub.py people.xsd
```

And, to hard-wire the subclass file so that it imports the API module, use the `--super` command line file. Example:

```
$ generateDS.py -o people_api.py people.xsd
$ generateDS.py -s people_appl1.py --super=people_api people.xsd
```

Or, do both at the same time with the following:

```
$ generateDS.py -o people_api.py -s people_appl1.py
--super=people_api people.xsd
```

And, for your second application:

```
$ generateDS.py -s people_appl2.py --super=people_api people.xsd
```

If you take a look inside these two "application" files, you will see and import statement like the following:

```
import ??? as supermod
```

If you had not used the `--super` command line option when generating the "application" files, then you could modify that statement yourself. The `--super` command line option does this for you.

You can also use the The graphical front-end to configure options and save them in a session file, then use that session file with `generateDS.py` to specify your command line options. For example:

```
$ generateDS.py --session=test01.session
```

You can test the generated code by running it. Try something like the following:

```
$ python people_api.py people.xml
```

or:

```
$ python people_appl1.py people.xml
```

Why does this work? Why can we run the generated code as a Python script? -- If you look at the generated code, down near the end of the file you'll find a `main()` function that calls a function named `parse()`. The `parse` function does the following:

1. Parses your XML instance document.
2. Uses your generated API to build a tree of instances of the generated classes.
3. Uses the `export()` methods in that tree of instances to print out (export) XML

that represents your generated tree of instances.

Except for some indentation (ignorable whitespace), this exported XML should be the same as the original XML document. So, that gives you a reasonably thorough test of your generated code.

And, the code in that `parse()` function gives you a hint of how you might build your own application-specific code that uses the generated API (those generated Python classes).

4.3 Using the generated code to parse and export an XML document

Now that you have generated code for your data model, you can test it by running it as an application. Suppose that you have an XML instance document `people1.xml` that satisfies your schema. Then you can parse that instance document and export it (print it out) with something like the following:

```
$ python people_api.py people1.xml
```

And, if you have used the `--super` command line option, as I have above, to connect your subclass file with the superclass (API) file, then you could use the following to do the same thing:

```
$ python people_appl1.py people1.xml
```

4.4 Some command line options you might want to know

You may want to merely skim this section for now, then later refer back to it when some of these options are are used later in this tutorial. Also, remember that you can get information about more command line options used by `generateDS.py` by typing:

```
$ python generateDS.py --help
```

and by reading the document at http://www.davekuhlman.org/#generateds-py

o

Generate the superclass module. This is the module that contains the implementation of each class for each element type. So, you can think of this as the implementation of the "data bindings" or the API for XML documents of the type defined by your XML schema.

s

Generate the subclass module. You might or might not need these. If you intend to write some application-specific code, you might want to consider starting with these skeleton classes and add your application code there.

super

This option inserts the name of the superclass module into an `import` statement in the subclass file (generated with "-s"). If you know the name of the superclass file in advance, you can use this option to enable the subclass file to import the superclass module automatically. If you do not use this option, you will need to edit the subclass module with your text editor and modify the import statement near the top.

root-element="element-name"

Use this option to tell generateDS.py which of the elements defined in your XM schema is the "root" element. The root element is the outer-most (top-level) element in XML instance documents defined by this schema. In effect, this tells your generated modules which element to use as the root element when parsing and exporting documents.

`generateDS.py` attempts to guess the root element, usually the first element defined in your XML schema. Use this option when that default is not what you want.

member-specs=list|dict

Suppose you want to write some code that can be generically applied to elements of different kinds (element types implemented by several *different* generated classes. If so, it might be helpful to have a list or dictionary specifying information about each member data item in each class. This option does that by generating a list or a dictionary (with the member data item name as key) in each generated class. Take a look at the generated code to learn about it. In particular, look at the generated list or dictionary in a class for any element type and also at the definition of the class `_MemberSpec` generated near the top of the API module.

version

Ask `generateDS.py` to tell you what version it is. This is helpful when you want to ask about a problem, for example at the generateds-users email list (https://lists.sourceforge.net/lists/listinfo/generateds-users), and want to specify which version you are using.

4.5 The graphical front-end

There is also a point-and-click way to run `generateDS`. It enables you to specify the options needed by `generateDS.py` through a graphical interface, then to run `generateDS.py` with those options. It also

You can run it, if you have installed `generateDS`, by typing the following at a command line:

```
$ generateds_gui.py
```

After configuring options, you can save those options in a "session" file, which can be loaded later. Look under the `File` menu for save and load commands and also consider using the "--session" command line option.

Also note that `generateDS.py` itself supports a "--session" command line option that enables you to run `generateDS.py` with the options that you specified and saved with the graphical front-end.

4.6 Adding application-specific behavior

`generateDS.py` generates Python code which, with no modification, will parse and then export an XML document defined by your schema. However, you are likely to want to go beyond that. In many situations you will want to construct a custom application that processes your XML documents using the generated code.

4.6.1 Implementing custom subclasses

One strategy is to generate a subclass file and to add your application-specific code to that. Generate the subclass file with the "-s" command line flag:

```
$ generateDS.py -s myapp.py people.xsd
```

Now add some application-specific code to `myapp.py`, for example, if you are using the included "people" sample files:

```
class peopleTypeSub(supermod.people):
    def __init__(self, comments=None, person=None, programmer=None,
        python_programmer=None, java_programmer=None):
        supermod.people.__init__(self, comments, person, programmer,
python_programmer,
            java_programmer)
    def fancyexport(self, outfile):
        outfile.write('Starting fancy export')
        for person in self.get_person():
            person.fancyexport(outfile)
supermod.people.subclass = peopleTypeSub
# end class peopleTypeSub

class personTypeSub(supermod.person):
    def __init__(self, vegetable=None, fruit=None, ratio=None,
id=None, value=None,
        name=None, interest=None, category=None, agent=None,
promoter=None,
        description=None):
        supermod.person.__init__(self, vegetable, fruit, ratio, id,
value,
            name, interest, category, agent, promoter, description)
    def fancyexport(self, outfile):
        outfile.write('Fancy person export -- name: %s' %
            self.get_name(), )
supermod.person.subclass = personTypeSub
# end class personTypeSub
```

4.6.2 Using the generated "API" from your application

In this approach you might do things like the following:

- `import` your generated classes.
- Create instances of those classes.
- Link those instances, for example put "children" inside of a parent, or add one or more instances to a parent that can contain a list of objects (think "maxOccurs" greater than 1 in your schema)

Get to know the generated export API by inspecting the generated code in the superclass file. That's the file generated with the "-o" command line flag.

What to look for:

- Look at the arguments to the constructor (`__init__`) to learn how to initialize an instance.
- Look at the "getters" and "setters" (methods name `getxxx` and `setxxx`, to learn how to modify member variables.
- Look for a method named `addxxx` for members that are lists. These correspond to members defined with `maxOccurs="n"`, where n > 1.
- Look at the build methods: `build`, `buildChildren`, and `buildAttributes`. These will give you information about how to construct each of the members of a given element/class.

Now, you can import your generated API module, and use it to construct and manipulate objects. Here is an example using code generated with the "people" schema:

```
import sys
import people_api as api

def test(names):
    people = api.peopleType()
    for count, name in enumerate(names):
        id = '%d' % (count + 1, )
        person = api.personType(name=name, id=id)
        people.add_person(person)
    people.export(sys.stdout, 0)

test(['albert', 'betsy', 'charlie'])
```

Run this and you might see something like the following:

```
$ python tmp.py
<people >
    <person  id="1">
        <name>albert</name>
    </person>
    <person  id="2">
        <name>betsy</name>
    </person>
    <person  id="3">
        <name>charlie</name>
    </person>
</people>
```

4.6.3   A combined approach

Note: You can find examples of the code in this section in these files:

```
tutorial/Code/upcase_names.py
tutorial/Code/upcase_names_appl.py
```

Here are the relevant, modified subclasses (`upcase_names_appl.py`):

```
import people_api as supermod

class peopleTypeSub(supermod.peopleType):
    def __init__(self, comments=None, person=None,
specialperson=None, programmer=None, python_programmer=None,
java_programmer=None):
        super(peopleTypeSub, self).__init__(comments, person,
specialperson, programmer, python_programmer, java_programmer, )
    def upcase_names(self):
        for person in self.get_person():
            person.upcase_names()
supermod.peopleType.subclass = peopleTypeSub
# end class peopleTypeSub

class personTypeSub(supermod.personType):
    def __init__(self, vegetable=None, fruit=None, ratio=None,
id=None, value=None, name=None, interest=None, category=None,
agent=None, promoter=None, description=None, range_=None,
extensiontype_=None):
        super(personTypeSub, self).__init__(vegetable, fruit, ratio,
id, value, name, interest, category, agent, promoter, description,
range_, extensiontype_, )
    def upcase_names(self):
        self.set_name(self.get_name().upper())
supermod.personType.subclass = personTypeSub
# end class personTypeSub
```

Notes:

- These classes were generated with the "-s" command line option. They are

subclasses of classes in the module people_api, which was generated with the "-o" command line option.
- The only modification to the skeleton subclasses is the addition of the two methods named upcase_names().
- In the subclass peopleTypeSub, the method upcase_names() merely walk over its immediate children.
- In the subclass personTypeSub, the method upcase_names() just converts the value of its "name" member to upper case.

Here is the application itself (upcase_names.py):

```
import sys
import upcase_names_appl as appl

def create_people(names):
    people = appl.peopleTypeSub()
    for count, name in enumerate(names):
        id = '%d' % (count + 1, )
        person = appl.personTypeSub(name=name, id=id)
        people.add_person(person)
    return people

def main():
    names = ['albert', 'betsy', 'charlie']
    people = create_people(names)
    print 'Before:'
    people.export(sys.stdout, 1)
    people.upcase_names()
    print '-' * 50
    print 'After:'
    people.export(sys.stdout, 1)

main()
```

Notes:

- The create_people() function creates a peopleTypeSub instance with several personTypeSub instances inside it.

And, when you run this mini-application, here is what you might see:

```
$ python upcase_names.py
Before:
    <people >
        <person  id="1">
            <name>albert</name>
        </person>
        <person  id="2">
            <name>betsy</name>
        </person>
        <person  id="3">
            <name>charlie</name>
        </person>
    </people>
--------------------------------------------------
After:
    <people >
        <person  id="1">
            <name>ALBERT</name>
        </person>
        <person  id="2">
            <name>BETSY</name>
        </person>
        <person  id="3">
            <name>CHARLIE</name>
        </person>
    </people>
```

4.7 Special situations and uses

4.7.1 Generic, type-independent processing

There are times when you would like to implement a function or method that can perform operations on a variety of members *and* that needs type information about each member.

You can get help with this by generating your code with the "--member-specs" command line option. When you use this option, `generateDS.py` add a list or a dictionary containing an item for each member. If you want a list, then use "--member-specs=list", and if you want a dictionary, with member names as keys, then use "--member-specs=dict".

Here is an example -- In this example, we walk the document/instance tree and convert all string simple types to upper case.

Here is a schema (`Code/member_specs.xsd`):

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="contact-list" type="contactlistType" />

  <xs:complexType name="contactlistType">
    <xs:sequence>
      <xs:element name="description" type="xs:string" />
        <xs:element name="contact" type="contactType"
maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="locator" type="xs:string" />
  </xs:complexType>

  <xs:complexType name="contactType">
```

```
    <xs:sequence>
      <xs:element name="first-name" type="xs:string"/>
      <xs:element name="last-name" type="xs:string"/>
      <xs:element name="interest" type="xs:string"
maxOccurs="unbounded" />
      <xs:element name="category" type="xs:integer"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:integer" />
    <xs:attribute name="priority" type="xs:float" />
    <xs:attribute name="color-code" type="xs:string" />
  </xs:complexType>

</xs:schema>
```

4.7.1.1 Step 1 -- generate the bindings

We generate code with the following command line:

```
$ generateDS.py -f \
  -o member_specs_api.py \
  -s member_specs_upper.py \
  --super=member_specs_api \
  --member-specs=list \
  member_specs.xsd
```

Notes:

- We generate the member specifications as a list with the command line option `--member-specs=list`.
- We generate an "application" module with the `-s` command line option. We'll put our application specific code in `member_specs_upper.py`.

4.7.1.2 Step 2 -- add application-specific code

And, here is the subclass file (`member_specs_upper.py`, generated with the "-s" command line option), to which we have added a bit of code that converts any string-type members to upper case. You can think of this module as a special "application" of the generated classes.

```
#!/usr/bin/env python

#
# member_specs_upper.py
#

#
# Generated Tue Nov  9 15:54:47 2010 by generateDS.py version 2.2a.
#

import sys
import member_specs_api as supermod

etree_ = None
Verbose_import_ = False
(   XMLParser_import_none, XMLParser_import_lxml,
    XMLParser_import_elementtree
    ) = range(3)
XMLParser_import_library = None
try:
    # lxml
    from lxml import etree as etree_
    XMLParser_import_library = XMLParser_import_lxml
    if Verbose_import_:
        print("running with lxml.etree")
except ImportError:
    try:
        # cElementTree from Python 2.5+
        import xml.etree.cElementTree as etree_
        XMLParser_import_library = XMLParser_import_elementtree
        if Verbose_import_:
            print("running with cElementTree on Python 2.5+")
    except ImportError:
        try:
            # ElementTree from Python 2.5+
            import xml.etree.ElementTree as etree_
            XMLParser_import_library = XMLParser_import_elementtree
            if Verbose_import_:
                print("running with ElementTree on Python 2.5+")
        except ImportError:
            try:
                # normal cElementTree install
                import cElementTree as etree_
                XMLParser_import_library = 
XMLParser_import_elementtree
                if Verbose_import_:
                    print("running with cElementTree")
            except ImportError:
                try:
                    # normal ElementTree install
                    import elementtree.ElementTree as etree_
                    XMLParser_import_library = 
XMLParser_import_elementtree
                    if Verbose_import_:
                        print("running with ElementTree")
                except ImportError:
                    raise ImportError("Failed to import ElementTree 
from any known place")

def parsexml_(*args, **kwargs):
    if (XMLParser_import_library == XMLParser_import_lxml and
        'parser' not in kwargs):
        # Use the lxml ElementTree compatible parser so that, e.g.,
        #   we ignore comments.
        kwargs['parser'] = etree_.ETCompatXMLParser()
    doc = etree_.parse(*args, **kwargs)
    return doc

#
# Globals
#

ExternalEncoding = 'ascii'

#
# Utility funtions needed in each generated class.
#

def upper_elements(obj):
    for item in obj.member_data_items_:
        if item.get_data_type() == 'xs:string':
            name = remap(item.get_name())
            val1 = getattr(obj, name)
            if isinstance(val1, list):
                for idx, val2 in enumerate(val1):
                    val1[idx] = val2.upper()
            else:
                setattr(obj, name, val1.upper())

def remap(name):
    newname = name.replace('-', '_')
    return newname


#
# Data representation classes
#

class contactlistTypeSub(supermod.contactlistType):
    def __init__(self, locator=None, description=None, contact=None):
        super(contactlistTypeSub, self).__init__(locator, 
description, contact, )
    def upper(self):
        upper_elements(self)
        for child in self.get_contact():
            child.upper()
supermod.contactlistType.subclass = contactlistTypeSub
# end class contactlistTypeSub


class contactTypeSub(supermod.contactType):
    def __init__(self, priority=None, color_code=None, id=None, 
first_name=None, last_name=None, interest=None, category=None):
        super(contactTypeSub, self).__init__(priority, color_code, 
id, first_name, last_name, interest, category, )
    def upper(self):
        upper_elements(self)
supermod.contactType.subclass = contactTypeSub
# end class contactTypeSub


def get_root_tag(node):
    tag = supermod.Tag_pattern_.match(node.tag).groups()[-1]
    rootClass = None
    if hasattr(supermod, tag):
        rootClass = getattr(supermod, tag)
    return tag, rootClass


def parse(inFilename):
    doc = parsexml_(inFilename)
    rootNode = doc.getroot()
    rootTag, rootClass = get_root_tag(rootNode)
    if rootClass is None:
        rootTag = 'contact-list'
        rootClass = supermod.contactlistType
    rootObj = rootClass.factory()
    rootObj.build(rootNode)
    # Enable Python to collect the space used by the DOM.
    doc = None
    sys.stdout.write('<?xml version="1.0" ?>\n')
    rootObj.export(sys.stdout, 0, name_=rootTag,
        namespacedef_='')
    doc = None
    return rootObj


def parseString(inString):
    from StringIO import StringIO
    doc = parsexml_(StringIO(inString))
    rootNode = doc.getroot()
    rootTag, rootClass = get_root_tag(rootNode)
    if rootClass is None:
        rootTag = 'contact-list'
        rootClass = supermod.contactlistType
    rootObj = rootClass.factory()
    rootObj.build(rootNode)
    # Enable Python to collect the space used by the DOM.
    doc = None
    sys.stdout.write('<?xml version="1.0" ?>\n')
    rootObj.export(sys.stdout, 0, name_=rootTag,
        namespacedef_='')
    return rootObj


def parseLiteral(inFilename):
    doc = parsexml_(inFilename)
    rootNode = doc.getroot()
    rootTag, rootClass = get_root_tag(rootNode)
    if rootClass is None:
        rootTag = 'contact-list'
        rootClass = supermod.contactlistType
    rootObj = rootClass.factory()
    rootObj.build(rootNode)
    # Enable Python to collect the space used by the DOM.
    doc = None
    sys.stdout.write('#from member_specs_api import *\n\n')
    sys.stdout.write('import member_specs_api as model_\n\n')
    sys.stdout.write('rootObj = model_.contact_list(\n')
    rootObj.exportLiteral(sys.stdout, 0, name_="contact_list")
    sys.stdout.write(')\n')
    return rootObj


USAGE_TEXT = """
Usage: python ???.py <infilename>
"""

def usage():
    print USAGE_TEXT
    sys.exit(1)


def main():
    args = sys.argv[1:]
    if len(args) != 1:
        usage()
    infilename = args[0]
    root = parse(infilename)


if __name__ == '__main__':
    #import pdb; pdb.set_trace()
    main()
```

Notes:

- We add the functions `upper_elements` and `remap` that we use in each generated class.
- Notice how the function `upper_elements` calls the function `remap` *only* on those members whose type is `xs:string`.
- In each generated (sub-)class, we add the methods that walk the DOM tree and apply the method (`upper`) that transforms each `xs:string` value.

4.7.1.3 Step 3 -- write a test/driver harness

Here is a test driver (`member_specs_test.py`) for our (mini-) application:

```
#!/usr/bin/env python

```

```
#
# member_specs_test.py
#

import sys
import member_specs_api as supermod
import member_specs_upper


def process(inFilename):
    doc = supermod.parsexml_(inFilename)
    rootNode = doc.getroot()
    rootClass = member_specs_upper.contactlistTypeSub
    rootObj = rootClass.factory()
    rootObj.build(rootNode)
    # Enable Python to collect the space used by the DOM.
    doc = None
    sys.stdout.write('<?xml version="1.0" ?>\n')
    rootObj.export(sys.stdout, 0, name_="contact-list",
        namespacedef_='')
    rootObj.upper()
    sys.stdout.write('-' * 60)
    sys.stdout.write('\n')
    rootObj.export(sys.stdout, 0, name_="contact-list",
        namespacedef_='')
    return rootObj


USAGE_MSG = """\
Synopsis:
    Sample application using classes and subclasses generated by
generateDS.py
Usage:
    python member_specs_test.py infilename
"""

def usage():
    print USAGE_MSG
    sys.exit(1)

def main():
    args = sys.argv[1:]
    if len(args) != 1:
        usage()
    infilename = args[0]
    process(infilename)

if __name__ == '__main__':
    main()
```

Notes:

- We copy the function `parse()` from our generated code to serve as a model for our function `process()`.

- After parsing and displaying the XML instance document, we call method `upper()` in the generated class `contactlistTypeSub` in order to walk the DOM tree and transform each `xs:string` to uppercase.

4.7.1.4 Step 4 -- run the test application

We can use the following command line to run our application:

```
$ python member_specs_test.py member_specs_data.xml
```

When we run our application, here is the output:

```
$ python member_specs_test.py member_specs_data.xml
<?xml version="1.0" ?>
<contact-list locator="http://www.rexx.com/~dkuhlman">
    <description>My list of contacts</description>
    <contact priority="0.050000" color-code="red" id="1">
        <first-name>arlene</first-name>
        <last-name>Allen</last-name>
        <interest>traveling</interest>
        <category>2</category>
    </contact>
</contact-list>
------------------------------------------------------------
<contact-list locator="HTTP://WWW.REXX.COM/~DKUHLMAN">
    <description>MY LIST OF CONTACTS</description>
    <contact priority="0.050000" color-code="RED" id="1">
        <first-name>ARLENE</first-name>
        <last-name>ALLEN</last-name>
        <interest>TRAVELING</interest>
        <category>2</category>
    </contact>
</contact-list>
```

Notes:

- The output above shows both before- and after-version of exporting the parsed XML instance document.

4.8 Some hints

The following hints are offered for convenience. You can discover them for yourself rather easily by inspecting the generated code.

4.8.1 Children defined with maxOccurs greater than 1

If a child element is defined in the XML schema with `maxOccurs="unbounded"` or a value of `maxOccurs` greater than 1, then access to the child is through a list.

4.8.2 Children defined with simple numeric types

If a child element is defined as a numeric type such as `xs:integer`, `xs:float`, or `xs:double` or as a simple type that is (ultimately) based on a numeric type, then the value is stored (in the Python object) as a Python data type (`int`, `float`, etc).

4.8.3 The type of an element's character content

But, when the element itself is defined as `mixed="true"` or the element a restriction of and has a simple (numeric) as a base, then the `valueOf_` instance variable holds the character content and it is always a string, that is it is not converted.

4.8.4 Constructors and their default values

All parameters to the constructors of generated classes have default parameters. Therefore, you can create an "empty" instance of any element by calling the constructor with no parameters.

For example, suppose we have the following XML schema:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="plant-list" type="PlantList" />

  <xs:complexType name="PlantType">
    <xs:sequence>
      <xs:element name="description" type="xs:string" />
        <xs:element name="catagory" type="xs:integer" />
        <xs:element name="fertilizer" type="FertilizerType"
maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="identifier" type="xs:string" />
  </xs:complexType>

  <xs:complexType name="FertilizerType">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:integer" />
  </xs:complexType>

</xs:schema>
```

And, suppose we generate a module with the following command line:

```
$ ./generateDS.py -o garden_api.py garden.xsd
```

Then, for the element named `PlantType` in the generated module named `garden_api.py`, you can create an instance as follows:

```
>>> import garden_api
>>> plant = garden_api.PlantType()
>>> import sys
>>> plant.export(sys.stdout, 0)
<PlantType/>
```