COpy Number and EXpression In Cancer (CONEXIC) is an algorithm that integrates matched copy number (amplifications and deletions) and gene expression data from tumor samples to identify driving mutations and the processes they influence. CONEXIC is inspired by Module Networks (Segal et al, 2003), but has been augmented by a number of critical modifications that make it suitable for identifying drivers. CONEXIC uses a score-guided search to identify the combination of modulators that best explains the behavior of a gene expression module across tumor samples and searches for those with the highest score within the amplified or deleted region.

When using CONEXIC, please cite the following article:

Akavia, U.D.*, Litvin O.*, Kim J., Sanchez-Garcia F., Kotliar D., Causton H.C., Pochanard P., Mozes E, Garraway L.A., Pe'er D. An Integrated Approach to Uncover Drivers of Cancer. Cell 2010; 143:1005-1017
*Equal Contribution

This document briefly describes the algorithm, and specifies all parameters available when running it. Despite the fact the algorithm is robust, the exact choice of optimal parameters is complex and requires human interaction. Currently, parameters to CONEXIC are given in a text file (format given below). This program is not plug-and-play, since it requires editing of text files and batch submitting. Please do not use this unless you can perform either task.
Please check www.c2b2.columbia.edu/danapeerlab/html/software.html for the most updated version of CONEXIC. A tutorial is forthcoming.

SINGLE MODULATOR

Single Modulator clustering is a method for forming initial disjoint clusters of genes, associated 1-1 with a list of regulators, which can be used as the starting point for Module Network Learning. It can be called from within a run of module-network learning, or it can run standalone, creating an output file which can later be used as input to module-network learning.

Overview of the Single Modulator clustering algorithm

The algorithm works by combining the expression data and copy-number data of the potential regulators. The input copy-number specifies, for each sample, whether the gene is Normal, Amplified or Deleted in this sample. The algorithm accepts a list of regulators for which amplification data should be used, and a list of regulators for which deletion data should be used. (The two lists may overlap if we want to use both amplification and deletion data for some regulators.)

The steps of the algorithm are as follows:

A. For each potential regulator on the amplification list, if both expression data and copy-number data are available for it, verify that there are at least two normal samples and at least two amplified samples. Also optionally do a Welch t-test to verify that expression level in the amplified samples is significantly different (either greater or smaller) from expression level in the normal samples.

B. For those regulators that have passed the test in A, do k-means clustering on their expression values, using k=2 and using the normal and amplified samples as the two initial clusters. Create a one-level regression tree with this regulator, and with the boundary between the two resulting clusters as the split point. At stages A and B, deleted samples (if present) are ignored.

B1. Optional step: for each regulator for which we're creating a regression tree, also create another regression tree that directly has this regulator's copy-number status as regulator, with deleted and normal data points on one side and amplified ones on the other. This step is not considered useful for the clustering, and will rarely be used.

C. Do the equivalent of steps A and B for the deletion list. At this stage, amplified samples (if present) are ignored. If step B1 is performed, the split is between deleted samples on one side and normal and amplified samples on the other.

D. Create the set of clusters among which the genes are to be assigned. Create either one or two clusters associated with each of the regression trees created in B or C (either one cluster, or one "up" cluster and one "down" cluster, depending on the noUpDown parameter described below); also create one additional cluster for genes for which no regulator can be found. The "up" cluster contains genes positively

correalted with the expression of the regulator, while the "down" cluster contains genes negatively correlated with the expression of the regulator. The default is to create both types.

E. For each gene, go over all regression trees created in steps B or C and calculate its score (i.e. the score for a singleton module containing this gene) with each of them. For the regression tree that gives the best score, verify that it gives a better score than the empty regulatory program. Since the program compares to the empty regulatory program, you need to remove the penalty lines. Comparing each single gene to the empty regulatory program with penalties will result in 90% (or more) of the genes failing this test. If a p-value threshold is specified, check the score's p-value by creating a number of random permutations of the regulator's values, calculating the score for each of them, and calculating in how many of them the score is better than in the actual order. If the p-value is below the required threshold, assign the gene to the cluster associated with this regression tree. (If there are "up" and "down" clusters, assign the gene to the "up" cluster if its expression level is positively correlated with that of the regulator, to the "down" cluster if it is negatively correlated.) If the p-value is above the required threshold, try the permutation test for the tree providing the second-best score, and so on until either you find one that passes the p-value test or you've tried all those that are better than the empty regulatory program. If it fails the p-value test for all of them, assign the gene to the no-regulators-found cluster.

F. At this point every gene has been assigned to some cluster. If the number of members in some of the clusters is less than the MinClusterSize threshold, break up the smallest cluster associated with a regression tree (do not break up the no-regulators-found cluster, even if it is the smallest one) and repeat E for its members to reassign them to other clusters. Repeat until all clusters have at least the required number of members.

Running Single Modulator

Single Modulator clustering is run by executing class `conexic.LearnModules` (the same class used for module-network learning). A spec file for a standalone run of Single Modulator clustering needs to contain the following:

- A "score" line specifying the scoring function. The scoring function must be NormalGamma (parameters are described on below).

- A "data" line specifying the input expression data (parameters are described below).

- The line `moduleInitiation RegCopyNumberClustering`
followed by specification of parameters for the algorithm, described below.

- The line
`ClusterOnly`
to specify that this is a stand-alone run.

- If this run is part of a bootstrapping group of runs, the line
`Sample <number>`
specifying the number of data points to be randomly chosen out of the data (with replacement) and used in this run.

Note that in a standalone run of Single Modulator clustering, **prior penalty lines** (Prior LeafPenalty and Prior RegulatorPenalty, described below) **should not be included** in the spec file. These priors would cause the empty regulatory program to score higher than any other regulator for most genes, and therefore cause most genes to be assigned to the no_regulators_found cluster, usually making the results of Single Modulator clustering useless.

A standalone run of Single Modulator clustering will result in three output files: a step1 output file, listing the clusters and their associated regression trees at the end of step E below; a step2 output file, listing the clusters and their associated regression trees at the end of the entire algorithm (i.e. the end of step F below); and a modules1 output, listing only the clusters without the associated regression trees.

If you run Single Modulator clustering in the same run as module-network learning, the spec file should be as described below. The step1 and step2 outputs will still be created; the clusters will then be used by the module-network learning algorithm, and the modules1 output will contain the final result of module-network learning. In such a run, the "score=NormalGamma" parameter, described below, should be used, in order to prevent the prior penalties specified for module-network learning from being used during Single Modulator clustering.

Parameters for the algorithm

The "moduleInitiation RegCopyNumberClustering" line in the spec file specifies the following parameters. All parameters are case-insensitive.

Required parameters:

amplified_list=<file name>
deleted_list=<file name>

The algorithm can be run with either one or both of the above parameters. The parameter points to a file containing a list of the potential regulators for which amplification data or deletion data should be considered.
If a regulator appears in both lists, the algorithm will consider both possible expression options (Deleted vs. Normal and Amplified vs. Normal), and two expression clusters using these regulators may be created.
The algorithm only looks at the first column of the file, so acceptable files are outputted by JISTIC and are called AMP.genes.All.matrix and DEL.genes.All.matrix.

AllowSelfRegulation=<true|false>

This parameter is **required** unless the ExcludeRegulators parameter is used (see below). The AllowSelfRegulation parameter specifies whether a regulator may be placed in the cluster associated with its own regression tree. If "false" is specified, such self-regulation is prevented. If "true" is specified, self-regulation is allowed and will likely occur for most or even all regulators in the final result of the algorithm.

Optional parameters:
input=<list of file names>
A list of one or more file names, separated by commas, providing copy-numbers for genes whose expression is specified in the input expression data. The relevant copy-number data is for genes that are included in the expression data's list of regulators; copy-number data for other genes is ignored. Columns in the copy-number data files must refer to the same data samples as in the expression data, and in the same order; each line should contain in each column the letter N for normal, A for amplified, or D for deleted. If copy-number data for a regulator appears in more than one of the listed files, the data from the file listed first is used. An acceptable input file for this parameter is outputted by JISTIC, if the tumors parameter was used in the JISTIC run, and this file is called biolearn.gene.matrix.
The program takes a gene copy number from the first file it appears. If it does not appear in the first file, it looks at the second file and so on. Since default JISTIC is to print out all genes, even if they are normal, you should just use one file, and the suggested file is the output of limitedPeelOff.
If you've set JISTIC to skip normals, there is value in using multiple files.

This argument is now optional. As an alternative to using this argument, the CNV input may be specified in the "**CNVData**" argument of the data line in the spec file, as described in the module network section (below); in that case, the data line must also contain a **CNVRegulators** argument pointing to the amplified-list and deleted-list lines. If step B1 in the algorithm is performed, CNV data must be specified in the data line; otherwise it may be specified in either of these two ways

CNVRegulators
This argument specifies that step B1 in the algorithm should be performed; in addition to regression trees based on regulator expression values, regression trees are also created based directly on regulator copy-number status. If the CNVRegulators option is used, CNV data must be specified in the data line in the spec file, using the CNVData argument (as described below), rather than using the input argument in the single-reg specification line.

first_output=<file name>
second_output=<file name>
Files to which to write the step1 and step2 output. If one or both of these parameters are omitted, the corresponding output is not created.

rejected_list=<file name>
Optional output file listing all potential regulators rejected in steps A or C of the algorithm, and for which therefore no regression tree and no clusters were created.

split_point_list=<file name>
Optional output file listing all regulators for which a regression tree was created in step B or C, and for each one the split point chosen for it. Most of the information in this list is redundant, since it also appears in the step1 output; but there may be some regulators that have no gene placed in their clusters, and so the only output in which to find their split point will be in this list.

MinClusterSize=<number>
Specifies the minimum number of genes that each cluster (with the possible exception of the no-regulators-found cluster) must have, as a stopping condition for step F. Default is 20.

WelchTTestthreshold=<pvalue>
Specifies the required p-value for passing the test in steps A and C, verifying that expression levels in the amplified or deleted samples is significantly different than in the normal samples. If not specified, the Welch t-test is not performed; all regulators listed in the amplified_list and deleted_list files are used to create regression trees, as long as they have at least two amplified or deleted samples and at least two normal samples.

pvalueThreshold=<pvalue>
Minimum pvalue that a member's score needs to have in order to pass the permutations test and be entered in a cluster. If not specified, the permutations test is not performed; in steps E and F each gene is always assigned to the cluster whose regression tree gives the best score, and there is no no-regulators-found cluster.

permutations=<number>
If pvalueThreshold is specified, the number of permutations to use in the permutations test in steps E and F. Default is 1000.

regulatorsTested=<number>
If pvalueThreshold is specified, the number of regression trees to test for each gene in steps E and F. If the specified number of top-scoring regression trees has been tested, and all have failed the permutations test, the gene is placed in the no-regulators-found cluster. Default is 3.

noUpDown
If this parameter is specified, step D of the algorithm creates one cluster for each of the regression trees created in steps B and C, which contains genes positively and negatively regulated with the expression of the regulator. By default, step D creates two clusters, a "up" and "down" cluster, for each regression tree.

excludeRegulators
If this parameter is specified, potential regulators are not included as members of the resulting clusters. In that case the AllowSelfRegulation parameter is meaningless and is not required.

alpha=<number>
lambda=<number>
These parameters provide alpha and lambda values for the NormalGamma score used during the algorithm. By default, the same alpha and lambda values are used as in the "score" line of the spec file. If you wish to use different values than those specified in the "score" line, you can specify them here. This makes sense only if you run Single Modulator clustering in the same run as module-network learning.

score=NormalGamma
If you are running Single Modulator clustering in the same run as Module Network learning, you usually want the scoring function to be different between the two steps. The scoring function for Module Network learning usually has prior penalties which should not be used during Single Modulator clustering (such penalties would make the empty regulatory program higher-scoring for most genes than any of the regulators, which will cause most genes to be placed in the no_regulators_found cluster). To specify this, the parameter "score=NormalGamma" must be specified; this creates a separate NormalGamma scoring function to be created for the Single Modulator run, without any of the prior penalties. When this parameter is specified, the alpha and lambda parameters must also be specified.

The UpDownSplit application

If you run Single Modulator clustering with the `noUpDown` parameter, only one cluster is created for each regression tree, containing both members whose expression is positively correlated with that of the regulator and those whose expression is negatively correlated. You can then run a separate application, by using class `conexic.UpDownSplit`, to create separate up and down clusters for each regression tree.

UpDownSplit takes either two or three command-line arguments: `the spec file`, which should be the same spec file used in the Single Modulator clustering run; `the step2 output` of the Single Modulator clustering run (the output file containing the final result of the algorithm including regression trees); and optionally the keyword "`ClusterOnly`". Its output (written to the standard output) is a new set of clusters in which each cluster that was associated with a regression tree is replaced by two separate clusters; members of the original cluster are assigned to one of the two new clusters depending on whether they are positively or negatively correlated with that of the regulator. If the "ClusterOnly" argument was specified, only the clusters are written to the output, without the associated regression trees; otherwise the regression trees are included in the output as well. If the output is intended for use as initial clusters in a subsequent run of module network learning, the "ClusterOnly" argument should be used.

Note that running Single Modulator clustering with the noUpDown parameter, followed by running UpDownSplit, will often result in different clusters than you would get by running without the noUpDown parameter in the first place. If the number of members in a regulator's cluster is above the MinClusterSize threshold, but the number of each type (positively correlated and negatively correlated) is below the threshold, then when running without noUpDown the two clusters for this regulator will be broken up because they are too small, and so this regression tree will not appear in the final result; if you run with noUpDown and then run UpDownSplit, the result will contain this regression tree and the two clusters for it, with each of these clusters containing a number of members below the threshold. This may lead to very small modules (one gene), which are not necessarily unified and cleaned when running the complete module networks learning algorithm (it depends on the score parameters and penalties).

MODULE NETWORKS

Summary of the algorithm:

Module network learning is the 3rd step in a run of CONEXIC. Given an initial list of candidate regulators, and a set of initial gene modules, CONEXIC refines the list of regulators and modules using an iterative algorithm to learn the 'regulatory program', that specifies the behavior of genes in the module as a function of the expression of its regulators. The output is a module_network file, which characterizes a driver network.

The driver network divides the regulated genes into modules, and associates each module with a driver tree. Each node in the tree is associated with a driver gene and a threshold expression level, and divides the expression values of the module's members into samples in which the driver gene's expression is below the threshold and those in which the driver gene's expression is above the threshold. Each side of the split can contain further splits using other driver-gene/expression-threshold pairs; we allow a total of up to five splits in each driver tree.

Starting from the initial clustering provided, we alternately go through iterations of learning a driver tree for each module, and then re-assigning the regulated genes among the modules. Each iteration improves the network's score; the iterations are ended when, during the gene re-assignment step, fewer than 10% of regulated genes have been re-assigned to a different module from their previous one.

The driver network is scored by using Segal et. al.'s Normal Gamma score, with the prior distribution added in the Geronemo system by Lee et. al. (described by Lee et. al, supporting materials, p. 5); the parameters we used for the Normal Gamma score and the priors are alpha=2, lambda=1, beta=20, x=15, y=0.

When learning the driver tree for a module, at each new split the driver gene that provides the best score is chosen as long as it is verified to be statistically significant. Significance is tested using Lee et. al.'s permutations test (described above); up to the three top-scoring driver genes are tried, and if none of them pass the permutations test no more splits are added in this driver tree.

It filters the set of driver genes by non-parametric bootstrapping. The iterative learning algorithm is run N times, using the initial set of potential driver genes and using a random sample of 62 tumor samples chosen with replacement. We then filter the set of potential driver genes, leaving only genes that appeared in at least one driver tree in the results for at least 40% of the runs (the filtering in this step is less restrictive than in step B, because the driver trees in the final result are important, and so we need to make sure we don't filter out too many driver genes that are actually significant). We then make one final run of the driver-network learning algorithm, using this filtered set of potential driver genes and the original data set.

The Module Network learning algorithm is modeled after the algorithm used in the Geronemo program, with several extensions.

The stages in the module-network learning algorithm are as follows:
A. Learn regulatory programs for each module in the initial clustering. Initial clusters can be given or generated by using Single Modulator or KMeans.
B. Reassign each gene to the module that gives it the best score.
C. In the case of using regression trees, if the number of genes reassigned in step B is below a given threshold, try to create singleton modules for each gene.
D. Repeat steps B and C a constant number of times.
E. Repeat A thru D until either a maximum number of iterations has been performed, or until the number of genes moved in step D is below a given threshold.
F. Learn regulatory programs for the resulting modules.
G. When using regression trees, if garbage-module parameters have been specified:
G1. Remove the genes that have the worst fit to the model to a garbage module.
G2. Repeat steps A thru D a specified number of times.
G3. Learn regulatory programs for the resulting modules.
H. When using regression trees, if final pruning has been specified, prune the regression trees in the final regulatory programs.

Running the algorithm:
Module network learning is run by executing the java class conexic.LearnModules. The class takes either one or two command-line arguments. The first, required, argument is the spec file, describing the various parameters of the run. The command for running module network learning is:

```
java [-Xmx1500] -classpath
$path2program$/Conexic.jar:$path2program$/commons-math-
1.2.jar conexic.LearnModules <spec-file-name> [<starting-
point-file-name>]
```

The optional second argument provides a file specifying a module network that provides a starting-point for the search. If you don't have an initial cluster to use as the second argument see the below section "Initial Clustering of Genes".

Specification of input data

Parameters to this program must be given in a spec file. Arguments constituting the beginning of a new line are specified in bold below which follows a list of the relevant optional parameters. The spec file must contain a "data" line specifying the input expression data. The line is of the following form where only the first parameter is required, and the remaining parameters are optional:

data GeneExpressionFile <expression-file-name> [optional parameters]
or
data VariablePerLine <expression-file-name> [optional parameters]

(VariablePerLine and GeneExpressionFile are synonyms and mean exactly the same). Note that in most applications the data represents gene expression levels, and the discussion here assumes that; however, module-network learning can in principle be applied to other types of data, which should be presented in the same format as described here.

The expression file should be tab-separated, with each line representing a gene and each column representing an experiment. In each line after the header line, either the first column represents the name of the gene, or the first column represents the gene's ORF name and the second column represents the gene's name; all subsequent columns represent expression values in the corresponding experiment. The first line is a header line containing the names of the experiments. If the lines contain both the gene's ORF and name, then the first two columns in the header line must correspondingly be "ORF" and "Name".

The optional parameters in the data line are as follows:

regulators=<regulators-list-file>
A file containing a list of potential regulators. The regulators listed in the file must be a subset of the genes in the expression file. By default, all genes in the expression file are treated as potential regulators. If you do not want any gene expression regulators (because you specify genotype markers or other types of regulators in separate data files) the regulators file should be empty.

markers=<genotype-markers-file>
A file containing genotype markers that can be used as regulators (in addition to the gene expression regulators). This is a tab-separated file in the same format as the expression file, with the columns representing the experiments in the same order. The name of each genotype marker must be of the form <letter><chromosome-number>_<start-position>_<end-position> (the initial letter in our current marker files is always 'M', but in principle it can be any other letter). The genotype value in each column must be 0, 1 or 2. If genotype markers are used as regulators, CONEXIC needs to associate each gene with its closest marker. The user has two options for how to provide that information:

a. by using the "locations" optional parameter in the same line
locations=<gene-locations-file>
A file containing the locations of genes. Currently CONEXIC can accept two possible formats for the location file. For examples of the two possible formats, see:
- file Human_All_coding_Genes_Info.txt, in directory /ifs/home/c2b2/dp_lab/uda2001/TCGA.Glioblastomas/Gene_Info
- file SGD_features.tab, in directory /ifs/data/c2b2/dp_lab/mozes
CONEXIC will use the location information in this file to associate each gene with its closest genotype marker.

b. OR by specifying in a separate line in the spec file
ClosestMarkers <closest-markers-file>
The closest markers file explicitly lists the markers associated with each gene. Each line in the file starts with the name of a gene, followed by one or more marker names; the first of these marker names that appears in the markers file is associated with this gene.

otherRegulators=<other-regulators-file>
A file containing other continuous-valued regulators, other than gene expression values. This can be used for any regulators with continuous values. The file format is the same as the expression file, with the experiments in the same order.

discreteRegulators=<discrete-regulators-file>
A file containing other discrete-valued regulators, other than genotype markers. The file format is the same as the expression file, with the experiments in the same order, but all data values must be discrete.

multiSplitRegulators=<multi-split-regulators-file>
A file containing discrete-valued regulators that must be split at each value. The file format is the same as the discrete regulators file. When learning a regression-tree regulatory program, when a split is considered on this regulator, the algorithm does not consider a single split at one split point, but instead considers multiple splits at all possible split points, subject to the MinSplit limit.

CNVData=<list-of-gene-cnv-files>
A comma-separated list of files containing gene copy-number variation values in discrete form. The format is similar to that of the expression data, and the gene names should be the same as names in the expression data. However, each value is represented not by a number but by a letter; 'N' for normal, 'A' for amplified or 'D' for deleted. Usually these files would be the biolearn.gene.matrix files outputted by JISTIC.

Unlike for other data files, the parameter need not be a single file name, but can be several file names separated by commas. If the same gene name is used in several of the listed files, data from the earlier-listed file is used.

Discrete gene copy-number values can be used in the module-network learning process in two ways. Gene cnvs can be directly used as potential regulators, by specifying the "CNVRegulators" argument described below. Also, when using regression trees, cnvs can be used in deciding which gene expression regulators will be allowed to regulate the module in which they are members; this is described below in the section on self-regulation.

CNVRegulators=<list_of-cnv-regulators-list-file>
A comma-separated list of files, with each containing a list of names of genes whose CNV values are to be used as potential regulators. This is a valid argument only if the CNVData argument is also specified.

Specification of Bootstrapping
By default, module-network learning is done using all the data. There is also the option, however, of running using a random sample of the experiments. This is done by specifying the line
Sample <number-of-experiments>
CONEXIC then takes a random sample, with replacement, of the specified number of experiments out of the data, and runs module-network learning using this data.

Usually such random sampling is used as part of bootstrapping; i.e. running a large number of runs of module-network learning, each with a different random sample of the data, in order to find out features of the resulting network that are consistently found in the run results. There are two ways to run such bootstrapping:

a. Run all bootstrapping runs in parallel in separate processes. This is the preferred method when running on a cluster. To do this, create an identical copy of the spec file for each run.

b. Run the bootstrapping runs sequentially. This takes much more time, but may be necessary if a cluster is not available. To do this, include in the spec file the line
NumRuns <number-of-runs>

The program will then run module-network learning the required number of times, creating a separate output for each run. If a "Sample" line was specified, each run uses a different random sample of the data. The samples used are printed in the file ending with **spec.btsamples**.

Specification of Parameters to the learning algorithm:

assignmentAlgorithm GeronemoIteration [parameters]

All parameters have default values and are optional. There are four groups of parameters:

1) parameters controlling number of iterations

MaxIterations=<number>
Specifies the maximum number of iterations in step E. Default is 10. Note that specifying "MaxIterations=0" will cause steps A thru E to be skipped, and the algorithm will start with step F; if no garbage-module parameters are specified, this means that the modules are fixed and the run will consist only of learning their regulatory programs.

StopThreshold=<fractional-number>
Specifies a fraction of the genes that serves as a stopping criterion in step E. The algorithm stops in step E either if MaxIterations has been reached, or if the total number of genes moved in step D is less than the specified fraction of the total genes. Default is .1, i.e. the algorithm stops if the total number of genes moved in step D is less than 10% of the total genes.

SingletonStepThreshold=<fractional-number>
Specifies a fraction of the genes that provides the criterion for step C. If the number of genes moved in step B is less than the specified fraction of the total genes, the algorithm tries to create singleton modules in step C. Default is .05. This parameter is meaningful only when running with regression trees.

ReassignmentSteps=<number>
The number of iterations in step D (i.e. the number of times B and C are repeated after each run of A). Default is 3.

2) parameters controlling creation of the garbage module (relevant only to regression trees)

The garbage module is created in step G of the algorithm, by finding and removing the genes with the worst fit to the model. By default, this step is not performed; it is only performed if the criteria for choosing the genes with the worst fit have been explicitly specified.

To create a garbage module, one of the following two parameters needs to be specified:

LikelihoodCutoff=<number>
When the LikelihoodCutoff parameter is specified, the criterion for the worst fit is the difference between the likelihood of a gene's expression values under the regulatory program of its module and the likelihood of the data with a plain gaussian distribution. The difference each calculated for each gene, and a distribution is created of the differences; genes for which the differences are more than the specified number of standard deviations below the mean are removed to the garbage module.

ScoreCutoff=<number>
When the ScoreCutoff parameter is specified, the criterion for the worst fit is the improvement to the score of the gene's module from removing this gene. For each module, the program calculates the score improvement from removing each gene, creates a distribution of these improvements (unlike with the likelihood cutoff, the program prepares a separate distribution for each module), and then removes to the garbage module genes for which the score improvement is mroe than the specified number of standard deviations above the mean.
If both LikelihoodCutoff and ScoreCutoff are specified, genes are removed to the garbage module only if they fulfill both conditions.

IterationsAfterFiltering=<number>
Specifies the number of iterations to perform in step G2; i.e. the number of module-network learning iterations to perform after removing genes to the garbage module. Default is 1. This parameter is relevant only if either the ScoreCutoff or the LikelihoodCutoff parameter has been specified.

3) parameters controlling regression-tree pruning (relevant only to regression trees)

CONEXIC may use various heuristics to prune the regression trees, removing splits that are expected not to be good. Such pruning can either be performed either as part of step A, while learning the regulatory programs; or only once at the end, in step H. By default, tree pruning is not performed; it is only performed if the relevant parameters are specified.

pruningLeaveoutFraction=<fractional-number>
improvementDropThreshold=<fractional-number>
immediateOutliersTest
When the two parameters pruningLeaveoutFraction and improvementDropThreshold are specified, CONEXIC prunes any splits in which the score improvement caused by the split depends on a small fraction of the data.
If immediateOutliersTest is specified, this test is performed during step A, each time a split in the tree is considered. For each such split, CONEXIC calculates the score improvement of the split if the extreme pruningLeaveoutFraction of the values on either side of the split are left out. If leaving out the data on either side causes a drop of more than improvementDropThreshold in the score improvement, the split is rejected.
If immediateOutliersTest is not specified, the test is performed at the end during step H. It is performed for each lowest-level split. Any lowest-level split that fails the test is removed; if removing this split causes the split above it to become the lowest-level split, the same check is recursively applied to it.

slopeRatioThreshold=<number>
correlationThreshold=<number>
immediateLinearityTest
When the two parameters slopeRatioThreshold and correlationThreshold are specified, CONEXIC does not allow subtrees under a split with a regulator with a strong linear correlation with the data.
If immediateLinearityTest is specified, this test is performed during step A, each time a new split is added in the tree. CONEXIC does linear regression between the regulator values and module member values on each side of the split, and also for the entire data. If the ratio of the slope of the linear regression on one side to the combined regression is greater than slopeRatioThreshold or less than 1/slopeRatioThreshold, and Pearson correlation between the regulator and module member values on the same side is more than correlationThreshold, then further subsplits on this side of the split are disallowed.
If immediateLinearityTest is not specified, the test is performed at the end during step H. It is performed for each split that is not lowest-level. If the data on any side of a split passes the test, the subtree under this side of the split is removed.

Scoring function (relevant only to regression trees)

The spec file needs to contain a line specifying the scoring function to be used. When using regression trees, this line is of the form:

score NormalGamma alpha=<number> lambda=<number> [optional parameters]
Alpha and lambda are required parameters to the NormalGamma score. In addition, the following optional parameters can be specified:

MinSplit=<number>
Minimum number of samples in each leaf of the regression tree. Default is 5.

MinSplittable=<number>
Minimum number of samples in a leaf of the regression treeto be considered for splitting. If a leaf has fewer than this number of samples, no further splits will be considered. Default is 12.

MinSplitValue=<number>
Minimum absolute value for all split values. This is useful, assuming the data is normalized, to make sure that split points are not too close to the center of the range of possible values.

NoMeanPenalty
This parameter changes the normal-gamma score so as not to penalize a mean of the values that is very far from 0.

The user can also specify two priors that provide score penalties on the number of leafs in each regression tree, and on the number of different regulators used. The penalties are specified using the lines:

Prior LeafPenalty <penalty-value>
Prior RegulatorPenalty <penalty-value>

The user can also specify a maximum number of leaves that any regressino tree can have, by specifying the line:

Constraint LeafMaximum <number>

Regulatory-program learning algorithm (relevant only to regression trees)

When using regression trees, the regulatory program is learned using greedy hill-climbing. At each point in the hill-climbing search, the next split is chosen using a test similar to that used in Geronemo; a split can only be performed if a permutations test on the new regulator demonstrates a sufficiently low pvalue.

The algorithm for learning regulatory programs is specified using the following two lines:

algorithm GreedyHillClimbing [local]
choiceTest GeronemoTest [optional parameters]
The optional "local" parameter in the "algorithm" line specifies that the regulatory programs for each module will be learned one at a time. This greatly speeds up the learning process, and should always be used unless the Acyclic constraint is used (see below in the section on self-regulation).

Optional parameters in the "choiceTest" line are as follows:

permutations=<number>
This specifies the number of permutations to be used in the permutations test. Default is 1000.

threshold=<pvalue>
This specifies the threshold pvalue needed for a split to be performed. Default is .05.

regulatorsTested=<number>
Maximum number of regulators to test from each class of regulators for each potential split. If the program has test this number of regulators in each class and all failed the permutations test, no split is done at this point in the regression tree. Default is 3.

CNVPermutationsPvalue=<pvalue>
CNVWelchPvalue=<pvalue>
These two parameters specify how CNV data is to be used to allow some expression regulators to regulate the module that contains them. Their meaning is described below in the section on self-regulation.

Controlling self-regulation (relevant only to regression trees)

The module network learning algorithm provides four possible levels of control of self-regulation, i.e. of cases in which an expression regulator regulates the module that contains it. The various levels are as follows:

A. Preventing cycles. One possible level is to completely prevent cycles; i.e. if regulator R1 is a member of module M1, then it not only is not allowed to be a regulator of M1, but also cannot be a regulator of module M2 if M2 contains regulator R2 which is a regulator of M1 (or a regulator of module M3 which contains R3 which is a regulator of M1, etc. preventing cycles of any length). To specify prevention of cycles, the spec file should contain the line:
Constraint Acyclic

When preventing cycles, it is not possible to learn the regulatory program of each module separately, as the results will heavily depend on the order of learning the modules. All regulatory programs must be learned together, with all splits to all regression trees considered at each step. **Therefore when preventing cycles (and only then) the argument "local" should be omitted from the "algorithm" line**.

B. Preventing self-regulation. This is the **default** level. If regulator R1 is a member of module M1, then it cannot be a regulator of M1; but there is no checking on cycles of more than one module.

C. Allowing self-regulation in delimited cases using copy-number variation data. To use this level, CNVData must be specified in the data line, and either the CNVPermutationsPvalue or CNVWelchPvalue argument (or both) must be specified in the choiceTest line. Decisions on whether to allow self-regulation are made as follows:

If CNVWelchPvalue is specified, then for each gene for which cnv data is available, if it has deleted samples a Welch t-test is performed between its expression values in normal and deleted samples; if it has amplified samples a Welch t-test is performed between its expression values in normal and amplified samples. The gene expression is allowed to regulate the module that contains it if it has passed both t-tests (or if it only has one type of aberration, the one t-test) with a p-value no greater than specified in the parameter; if it has both types of aberration, it is also required that its expression level be a monotonic function of its cnv, i.e. mean expression in normal samples should be in between the mean expressions in deleted and amplified samples.

If CNVPermutationsPvalue is specified, then for each split that is considered, if the best-scoring regulator is a member of the module and cnv data is available for it, the gene's cnv is tried as a regulator at this split point, and a permutations test on its cnv values is performed. If the pvalue is no greater than the specified parameter, the gene expression is allowed as a regulator in this split point.

If both CNVPermutationsPvalue and CNVWelchPvalue are specified, a gene is allowed to regulate the module that contains it only if both tests pass.

D. Drop all restrictions on self-regulation. This level is specified by including the line
AllowSelfRegulation
in the spec file.

Initial clustering of genes

The module network learning algorithm needs to start with some initial clustering of the genes. There are two ways to provide initial clustering: providing the initial clusters, or specifying an algorithm for generating them.

The user can provide a fixed initial clustering of the genes into modules. The file containing the initial clustering is provided in the second command-line argument. Each line in the file optionally starts with "<module-name>:", and then contains a tab-separated list of the genes in one module. Any genes that are not included in the initial clustering file are excluded from the modules (i.e. considered to be in the garbage module).

The initial clustering file may optionally provide an initial network, containing regulatory programs as well as module assignments (this can be done if the output of a previous run of module-network learning is used). In that case step A at the start of the algorithm is skipped. If the initial network file contains regulatory programs and argument MaxIterations=0 is specified, steps A thru F are skipped completely (this is useful if the user wishes to perform only steps G and H, i.e. creating the garbage module and final tree pruning, on the results of a previous run).

If an initial clustering file is not provided, CONEXIC needs to generate the initial clusters as the first step in the run, before starting the module network algorithm. There are two algorithms that can be used for generating the initial clusters:

a. Single-regulator clustering (see above).
b. KMeans clustering. To use KMeans clustering, the spec file should contains the line:

moduleInitiation RandomKMeansAssignment K=<number-of-clusters> [optional arguments]

The optional arguments are as follows:

MaxIterations=<number>
The maximum number of KMeans iterations. Default is 10.

restarts=<number>
The number of restarts of KMeans algorithm. The KMeans algorithm runs the specified number of times plus 1, and the best clustering result from all the runs - i.e. the run that gives the highest ration of average distance among cluster means to average distance among pairs within each cluster - is used. Default restarts number is 0, i.e. the KMeans algorithm will be run once.

Module Expansion

After getting the final regulatory program from Module Networks, we can expand the modules. This is based on our belief that genes are not regulated by one regulatory program, but possibly more than one. In a way, it can be thought as the complement of the garbage module: the garbage module removes genes which are not good, the expansion adds genes which are good (into multiple modules).
As a result a gene may get assigned to more than one module.

Expansion rules are pretty simple
Using up to **N** levels, where N is a parameter, calculate the normalGamma difference for each gene (in the module, and out of the module).
We want to limit the number of levels, to avoid a situation where no gene is a good fit, because of the low-level (low number of samples) splits.

Use the distribution of normalGamma values for the genes inside the module to decide which genes should be added. Add all the genes that are above percentile **P** in the distribution of genes in the module.

For each module *m*
  For each genes *g*
    calculate *delta score* = score of gene with regulatory program of module m - score of gene with no regulatory program
end
  Calculate *D*, the distribution of delta scores for all genes in module m
  Add all genes that are not in the module, but are better than the Pth percentile of D
end

Regulators are not added to their module at all, even if they are at a split level lower than N.

```
java -classpath
$path2program$/Conexic.jar:$path2program$/commons-math-
1.2.jar conexic.ExpandModules spec-file modules1-file
threshold=P [MaxModulesPerGene=N] [depth=N] [output=file-
name]
```

spec-file, modules1-file
modules1 file to expand, spec file that created it.
Required.

threshold
minimal percentile for expansion. **Required**. Is given in fraction form - if P is 0.8, a gene will be added into a module if its delta score is better than at least 80% of the module's members.

MaxModulesPerGene
Maximal number of modules that the same gene can be added to. If one gene is found to be above the threshold for more than the specified number of modules, it is not added to any modules. Optional; the default is 10% of the number of modules.

depth
how many levels of the regulatory tree to use - the expansion can be done on the higher levels (say 2). Optional, but recommended. If it is omitted, the entire regulatory tree is used.

output
file to write expanded module to. Optional parameter, when it is not given, default is modules1.expanded. Useful when you want to expand one program more than one time.

Start from Single Module

As an alternative to learning the module network from scratch, users can work from an existing cancer signature, using algorithms from module network learning and module expansion to refine it. Single-module iterative refinement works as follows:
A. Learn regulatory program for initial module, using step A of module network learning.
B. Use the garbage-module algorithm to remove the worst members from the module, using step G1 from module network learning.
C. Add good new members to module, using the module expansion algorithm.
D. Learn regulatory program for current module, as in step A.
E. Compare regulatory program to the program from previous iteration. If the set of regulators in the top levels of the tree is identical, stop. Also stop if we have made the maximum number of iterations. Otherwise, go back to step B.

Invoking single-module iterative refinement

The application is invoked similarly to module network learning, by executing the java class conexic.LearnModules. There are two command-line arguments, both **required**. The first argument is the spec file, describing the various parameters of the run. The second argument provides the starting module to be refined; the file must contain exactly one line, listing the members of the module.

The spec file needs to contain specifications of the input data, the scoring function, and the regulatory-program learning algorithm, specified in the same way as for module network learning. Note that single-module refinement uses regression trees, so the sections above marked as "relevant only to regression trees" apply here. The "assignmentAlgorithm GeronemoIteration" line, used for module network learning, is not used in this application. Instead, the spec file needs to contain the line:

assignmentAlgorithm SingleModuleRefinement LikelihoodCutoff=[number] ScoreCutoff=[number] ExpansionThreshold=[fractional-number] MaxIterations=[number] RelevantTreeLevels=[number]

LikelihoodCutoff
ScoreCutoff

Parameters to the garbage-module algorithm in step B; same meaning as in the "AssignmentAlgorithm GeronemoIteration" line in module network learning.

ExpansionThreshold

Threshold to use in module expansion in step C; same meaning as in the module expansion application.

MaxIterations
RelevantTreeLevels

Stopping criteria for step E. The iteration stops either when we have performed MaxIterations iterations, or when the set of regulators in the RelevantTreeLevels top levels of the tree is identical to the previous step.