
pyGCluster Documentation

Release 0.18.4

**Daniel Jaeger
Johannes Barth
Anna Niehues
Christian Fufezan**

November 04, 2013

CONTENTS

1	Introduction	1
1.1	Algorithm Workflow	1
1.2	General information	4
1.3	Implementation	4
1.4	Download	4
1.5	Citation	5
1.6	Installation	5
1.7	References	6
2	Module pyGCluster	7
3	Usage	23
3.1	Clustering	23
3.2	Clustered data visualization	25
4	Example scripts	31
4.1	Functionality check	31
4.2	Testscripts to demonstrate pyGCluster	32
4.3	Plot communities from a pyGCluster pkl	32
4.4	Retrieve pyGCluster pkl info	33
4.5	Plot simple expression map	33
4.6	Nodemap and expression map example	33
5	Indices and tables	37
	Python Module Index	39
	Index	41

INTRODUCTION

‘Omics’ technologies yield large datasets, which are commonly subjected to cluster analysis in order to group them into comprehensible communities, i.e. co-regulated groups, which might be functionally related (Si et al., 2011). A critical step in cluster analysis is cluster validation (Handl et al., 2005), the most stringent form of validation being the assessment of exact reproducibility of a cluster in the light of the uncertainty of the data. This issue is addressed by pyGCluster, an algorithm working in two steps. Firstly it creates many agglomerative hierarchical clusterings (AHCs) of the input data by injecting noise based on the uncertainty of the data and clusters them using different distance linkage combinations (DLCs). Secondly, pyGCluster creates a meta-clustering, i.e. clustering of the resulting, highly reproducible clusters into communities to gain a most complete representation of common patterns in the data. Communities are defined as sets of clusters with a specific pairwise overlap.

1.1 Algorithm Workflow

The workflow of pyGCluster can be divided in:

- **iterative steps**
 - re-sample the data based on mean and standard deviation
 - clustering of data using different distance linkage combinations (DLCs)
- meta-clustering of highly reproducible clusters into communities, i.e. sets of clusters with a specific overlap
- visualize results via node maps, expression maps and expression profiles

1.1.1 Re-sampling & clustering

For each iteration, a new dataset is generated evoking the re-sampling routine. pyGCluster uses by default a noise injection function that generates a new data set by drawing from normal distributions defined by each data point, i.e. object o in condition l is defined by $\mu_{ol} \pm \sigma_{ol}$. Clustering is then performed using SciPy or fastcluster routines.

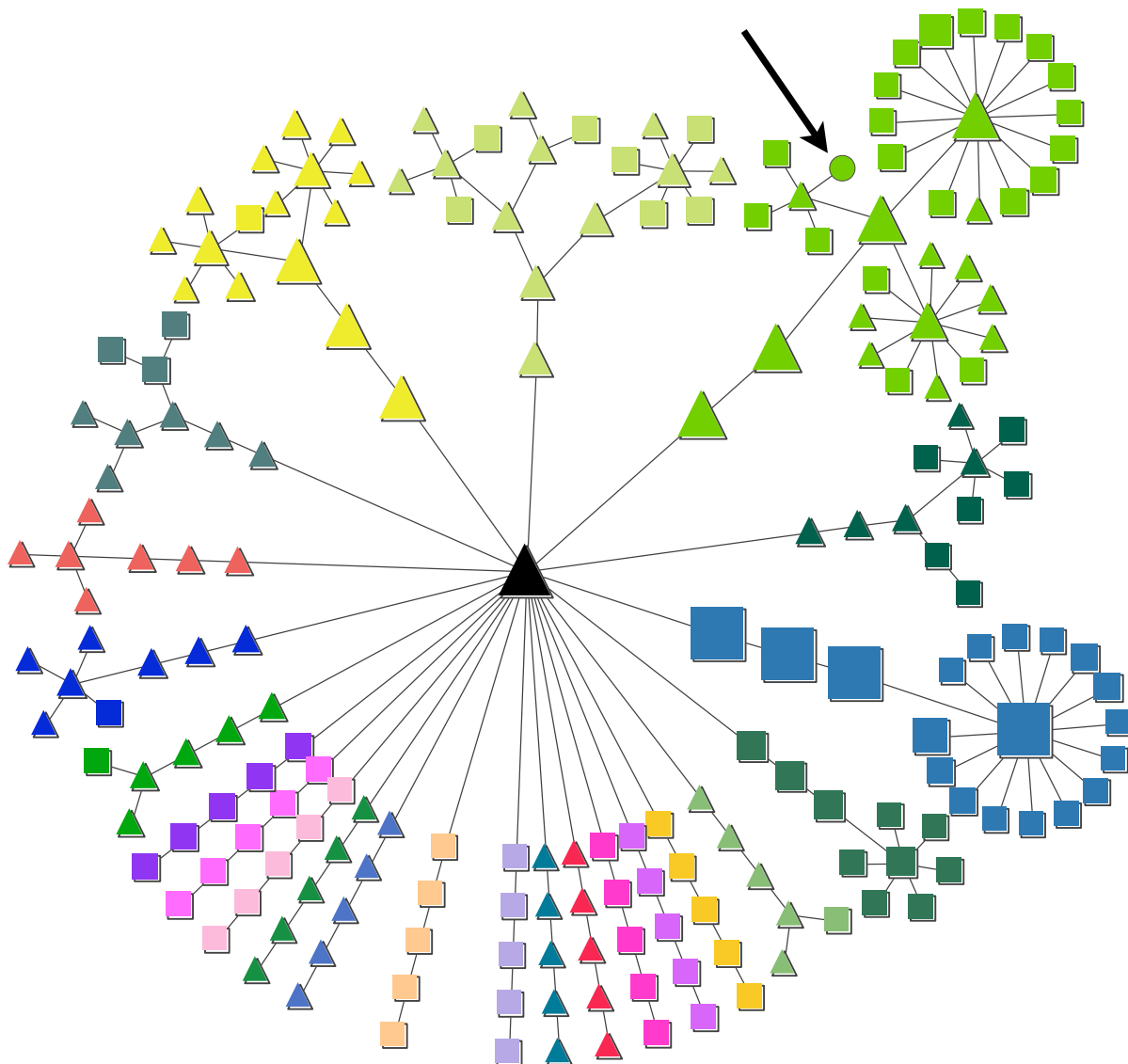
1.1.2 Community construction

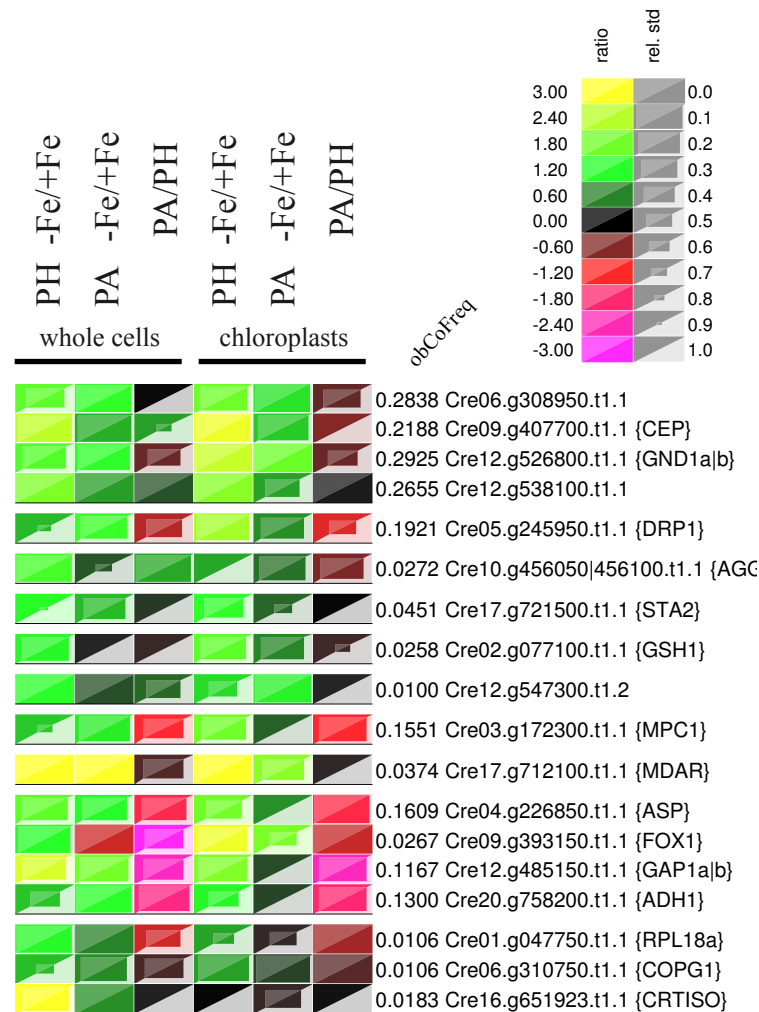
Communities are created after the iterations by a meta-clustering of the most frequent clusters, i.e. top X% or top Y number of clusters. Community construction is performed iteratively through an AHC approach with a specifically developed distance metric (see publication) and complete linkage. Complete linkage was chosen because it insures that all clusters or meta-clusters have overlapping objects. The customized distance metric ensures that a) smaller clusters are merged earlier in the hierarchy (closer to the bottom) and b) clusters that have a smaller overlap to each other than the threshold will merge after the root, i.e. never into the same branch. After each iteration, very closely related clusters (in terms of their object content) are merged in the hierarchy forming one branch or community starting

from the root. The final node map shows these iterations and where meta clusters are merged into the community. The closest node to the root in the final node map is the last iteration, in which no change to the community composition was detected. Using this approach the number of final clusters or communities to consider and analyze is reduced.

1.1.3 Node map and expression map example

The figures show an example of a node map and expression map generated by pyGCluster. The node map illustrates the data set of Höhner et al. (2013). The node shapes indicate whether a cluster was found using Euclidean distance (squares), correlation distance (circles) or both (triangles). The node color indicates the community membership. The strength of pyGCluster is shown in the green community in which Euclidean and correlation distance identified high frequency clusters (see arrow). Both distance metrics were required to identify all clusters. The black triangle in the middle represents the root node. Since the community construction is performed iteratively, the different iter steps are visible in the node map. For each community, the node closest to the root is the last iteration in which no change in the communities with respect to their composition was detected.





The example expression map is taken from Höhner et al. (2013)

Note: Some texts were copied from the original publication and are thus hereby marked as citation

1.2 General information

Copyright 2011-2013 by:

D. Jaeger,
J. Barth,
A. Niehues,
C. Fufezan

The latest Documentation was generated on: November 04, 2013

1.2.1 Contact information

Please refer to:

Dr. Christian Fufezan
Institute of Plant Biology and Biotechnology
Schlossplatz 8 , R 110.105
University of Muenster
Germany
eMail: christian@fufezan.net
Tel: +049 251 83 24861

<http://www.uni-muenster.de/Biologie.IBBP.AGFufezan>

1.3 Implementation

pyGCluster requires Python2.7 or higher, is freely available at <http://pyGCluster.github.io> and published under MIT license.

pyGCluster dependencies are:

numpy
scipy
fastcluster (optionally)
rpy2 (optionally)
graphviz (optionally)

Fastcluster (Müllner,D. (2013)) offers significant speed increase compared to the same SciPy routines.

1.4 Download

Get the latest version via github

<https://github.com/pygcluster/pyGCluster>

or the latest package at

<http://pyGCluster.github.com/dist/pyGCluster.tar.bz2>

<http://pyGCluster.github.com/dist/pyGCluster.zip>

The complete Documentation can be found as pdf

<http://pyGCluster.github.com/dist/pyGCluster.pdf>

1.5 Citation

Please cite us when using pyGcluster in your work.

Jaeger, D., Barth, B., Niehues, A. and Fufezan, C. (2013) pyGCluster, a novel hierarchical clustering approach

The original publication can be found here:

<http://bioinformatics.oxfordjournals.org...>

<http://bioinformatics.oxfordjournals.org...>

1.6 Installation

Please execute the following command in the pyGCluster folder:

```
sudo python setup.py install
```

1.6.1 Installation notes

If Windows XP (SP3) is used please make sure to install SciPy version 0.10.0

1.6.2 Functionality check

After installation, please run the script `test_pyGCluster.py` from the `exampleScripts` folder to check if pyGCluster was installed properly. Testscript to demonstrate functionality of pyGCluster

A synthetic dataset is used to check the correct installation of pyGCluster. This dataset contains 10 ratios (Gene 0-9) which were randomly sampled between 39.5 and 40.5 in 0.1 steps with a low standard deviation (randomly sampled between 0.1 and 1) and 90 ratios (Gene 10-99) which were randomly sampled between 3 and 7 in 0.1 steps with a high standard deviation (randomly sampled between 0.1 and 5)

5000 iterations are performed and the presence of the most frequent cluster is checked.

This cluster should contain the Genes 0 to 9.

Usage:

```
./test_pyGCluster.py
```

When the iteration has finished (this should normally take not longer than 20 seconds), the script asks if you want to stop the iteration process or continue:

```
iter_max reached. See convergence plot. Stopping re-sampling if not defined
otherwise ...
... plot of convergence finished.
See plot in "../exampleFiles/functionalityCheck/convergence_plot.pdf".
```

```
Enter how many iterations you would like to continue.
(Has to be a multiple of iterstep = 5000)
(enter "0" to stop resampling.)
(enter "-1" to resample until iter_max (= 5000) is reached.)
Enter a number ...
```

Please enter 0 and hit enter (The script will stop and the test will finish).

The results are saved into the folder functionalityCheck.

Additionally expression maps and expression profiles are plotted.

1.7 References

- Bréhélin,L. et al. (2008) Using repeated measurements to validate hierarchical gene clusters. *Bioinformatics*, 24, 682-628.
- Gansner,E.R. and North,S.C. (2000) An open graph visualization system and its applications to software engineering. *Software Pract. Exper.*, 30, 1203-1233.
- Handl,J. et al. (2005) Computational cluster validation in post-genomic data analysis. *Bioinformatics*, 21, 3201-3212.
- Höhner,R. et al. (2013) The metabolic status drives acclimation of iron deficiency responses in *Chlamydomonas reinhardtii* as revealed by proteomics based hierarchical clustering and reverse genetics. *Mol. Cell. Proteomics*, in press.
- Müllner,D. (2013) fastcluster: fast hierarchical agglomerative clustering routines for R and Python. *J. Stat. Softw.*, 53, 1-18.
- Saeed,A.I. et al. (2003) TM4: A free, open-source system for microarray data management and analysis. *Biotechniques*, 34, 374-378.
- Shannon,P. et al. (2003) Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Res.*, 13, 2498-2504.
- Si,J. et al. (2011) Model-based clustering for rna-seq data. Joint statistical meeting, Juli 30 - August 4, Florida.

MODULE PYGCLUSTER

pyGCluster is a clustering algorithm focusing on noise injection for subsequent cluster validation. By requesting identical cluster identity, the reproducibility of a large amount of clusters obtained with agglomerative hierarchical clustering (AHC) is assessed. Furthermore, a multitude of different distance-linkage combinations (DLCs) are evaluated. Finally, associations of highly reproducible clusters, called communities, are created. Graphical representation of the results as node maps and expression maps is implemented.

The pyGCluster module contains the main class `pyGCluster.Cluster` and some functions

```
pyGCluster.create_default_alphabet()
pyGCluster.resampling_multiprocess()
pyGCluster.seekAndDestry()
pyGCluster.yield_noisejected_dataset()
```

class `pyGCluster.Cluster` (*data=None, working_directory=None, verbosity_level=1*)

The pyGCluster class

Parameters

- **working_directory** (*string*) – directory in which all results are written (requires write-permission!).
- **verbosity_level** (*int*) – either 0, 1 or 2.
- **data** (*dict*) – Dictionary containing the data which is to be clustered.

In order to work with the default noise-injection function as well as plot expression maps correctly, the data-dict **has** to have the following structure.

Example:

```
>>> data = {
...     Identifier1 : {
...         condition1 : ( mean11, sd11 ),
...         condition2 : ( mean12, sd12 ),
...         condition3 : ( mean13, sd13 ),
...     },
...     Identifier2 : {
...         condition2 : ( mean22, sd22 ),
...         condition3 : ( mean23, sd23 ),
...         condition3 : ( mean13, sd13 ),
...     },
... }
>>> import pyGCluster
>>> ClusterClass = pyGCluster.Cluster(data=data, verbosity_level=1, working_directory=...)
```

Note: If any condition for an identifier in the “nested_data_dict”-dict is missing, this entry is discarded, i.e. not imported into the Cluster Class. This is because pyGCluster does not implement any missing value estimation. One possible solution is to replace missing values by a mean value and a standard deviation that is representative for the complete data range in the given condition.

pyGCluster inherits from the regular Python Dictionary object. Hence, the attributes of pyGCluster can be accessed as Python Dictionary keys.

A selection of the most important attributes / keys are:

```
>>> # general
>>> ClusterClass[ 'Working directory' ]
...     # this is the directory where all pyGCluster results
...     # (pickle objects, expression maps, node map, ...) are saved into.
/Users/Shared/moClusterDirectory
>>> # original data ca be accessed via
>>> ClusterClass[ 'Data' ]
...     # this collections.OrderedDict contains the data that has been
...     # or will be clustered (see also below).
... plenty of data ;)
>>> ClusterClass[ 'Conditions' ]
...     # sorted list of all conditions that are defined in the "Data"-dictionary
[ 'condition1', 'condition2', 'condition3' ]
>>> ClusterClass[ 'Identifiers' ]
...     # sorted tuple of all identifiers, i.e. ClusterClass[ 'Data' ].keys()
( 'Identifier1', 'Identifier2' , ... 'IdentifierN' )
>>> # re-sampling paramerters
>>> ClusterClass[ 'Iterations' ]
...     # the number of datasets that were clustered.
1000000
>>> ClusterClass[ 'Cluster 2 clusterID' ]
...     # dictionary with clusters as keys, and their respective row index
...     # in the "Cluster count"-matrix (= clusterID) as values.
{ ... }
>>> ClusterClass[ 'Cluster counts' ]
...     # numpy.uint32 matrix holding the counts for each
...     # distance-linkage combination of the clusters.
>>> ClusterClass[ 'Distance-linkage combinations' ]
...     # sorted list containing the distance-linkage combinations
...     # that were evaluted in the re-sampling routine.
>>> # Communities
>>> ClusterClass[ 'Communities' ]
...     # see function pyGCluster.Cluster.build_nodemap for further information.
>>> # Visualization
>>> ClusterClass[ 'Additional labels' ]
...     # dictionary with an identifier of the "Data"-dict as key,
...     # and a list of additional information (e.g. annotation, GO terms) as value.
{
    'Identifier1' :
        ['Photosynthesis related' , 'zeroFactor: 12.31' ],
    'Identifier2' : [ ... ] ,
    ...
}
>>> ClusterClass[ 'for IO skip clusters bigger than' ]
...     # Default = 100. Since some clusters are really large
...     # (with sizes close to the root (the cluster holding all objects)),
...     # clusters with more objects than this value
```

```
...      # are not plotted as expression maps or expression profile plots.
```

pyGCluster offers the possibility to save the analysis (e.g. after re-sampling) via `pyGCluster.Cluster.save()`, and continue via `pyGCluster.Cluster.load()`. Initializes `pyGCluster.Cluster` class

Classically, users start the multiprocessing clustering routine with multiple distance linkage combinations via the `pyGCluster.Cluster.do_it_all()` function. This function allows to update the `pyGCluster` class with all user parameters before it calls `pyGCluster.Cluster.resample()`. The main advantage in calling `pyGCluster.Cluster.do_it_all()` is that all general plotting functions are called afterwards as well, these are:

```
pyGCluster.Cluster.plot_clusterfreqs()
pyGCluster.Cluster.build_nodemap()
pyGCluster.Cluster.write_dot()
pyGCluster.Cluster.draw_community_expression_maps()
```

If one choses, one can manually update the parameters (setting the key, value pairs in `pyGCluster`) and then evoke `pyGCluster.Cluster.resample()` with the appropriate parameters. This useful if certain memory intensive distance-linkage combinations are to be clustered on a specific computer.

Note: Cluster Class can be initilized empty and filled using `pyGCluster.Cluster.load()`

build_nodemap(*min_cluster_size=4, top_X_clusters=0, threshold_4_the_lowest_max_freq=0.01, starting_min_overlap=0.1, increasing_min_overlap=0.05*)

Construction of communities from a set of most_frequent_cluster. This set is obtained via `pyGCluster.Cluster._get_most_frequent_clusters()`, to which the first three parameters are passed. These clusters are then subjected to AHC with complete linkage. The distance matrix is calculated via `pyGCluster.Cluster.calculate_distance_matrix()`. The combination of complete linkage and the distance matrix assures that all clusters in a community exhibit at least the “starting_min_overlap” to each other. From the resulting cluster tree, a “first draft” of communities is obtained. These “first” communities are then themselves considered as clusters, and subjected to AHC again, until the community assignment of clusters remains constant. By this, clusters are inserted into a target community, which initially did not overlap with each cluster inside the target community, but do overlap if the clusters in the target community are combined into a single cluster. By this, the degree of stringency is reduced; the clusters fit into a community in a broader sense. For further information on the community construction, see the publication of `pyGCluster`.

Internal structure of communities:

```
>>> name = ( cluster, level )
...      # internal name of the community.
...      # The first element in the tuple ("cluster") contains the indices
...      # of the objects that comprise a community.
...      # The second element gives the level,
...      # or iteration when the community was formed.
>>> self[ 'Communities' ][ name ][ 'children' ]
...      # list containing the clusters that build the community.
>>> self[ 'Communities' ][ name ][ '# of nodes merged into community' ]
...      # the number of clusters that build the community.
>>> self[ 'Communities' ][ name ][ 'index 2 obCoFreq dict' ]
...      # an OrderedDict in which each index is assigned its obCoFreq.
...      # Negative indices correspond to "placeholders",
...      # which are required for the insertion of black lines into expression maps.
...      # Black lines in expression maps separete the individual clusters
...      # that form a community, sorted by when
```

```
...         # they were inserted into the community.
>>> self[ 'Communities' ][ name ][ 'highest obCoFreq' ]
...         # the highest obCoFreq encountered in a community.
>>> self[ 'Communities' ][ name ][ 'cluster ID' ]
...         # the ID of the cluster containing the object with the highest obCoFreq.
```

Of the following parameters, the first three are passed to `pyGCluster.Cluster._get_most_frequent_clusters`

Parameters

- **min_cluster_size** (*int*) – clusters smaller than this threshold are not considered for the community construction.
- **top_X_clusters** (*int*) – form communities from the top X clusters sorted by their maximum frequency.
- **threshold_4_the_lowest_max_freq** (*float*) – [0, 1[form communities from clusters whose maximum frequency is at least this value.
- **starting_min_overlap** (*float*) –]0, 1[minimum required relative overlap between clusters so that they are assigned the same community. The relative overlap is defined as the size of the overlap between two clusters, divided by the size of the larger cluster.
- **increasing_min_overlap** (*float*) – defines the increase of the required overlap between communities

Return type none

calculate_distance_matrix (*clusters, min_overlap=0.25*)

Calculates the specifically developed distance matrix for the AHC of clusters:

1. Clusters sharing *not* the minimum overlap are attributed a distance of “self[‘Root size’]” (i.e. `len(self[‘Data’])`).
2. Clusters are attributed a distance of “self[‘Root size’] - 1” to the root cluster.
3. Clusters sharing the minimum overlap are attributed a distance of “size of the larger of the two clusters minus size of the overlap”.

The overlap between a pair of clusters is relative, i.e. defined as the size of the overlap divided by the size of the larger of the two clusters.

The resulting condensed distance matrix is not returned, but rather stored in `self[‘Nodemap - condensed distance matrix’]`.

Parameters

- **clusters** (*list of clusters. Clusters are represented as tuples consisting of their object’s indices.*) – the most frequent clusters whose “distance” is to be determined.
- **min_overlap** (*float*) –]0, 1[threshold value to determine if the distance between two clusters is calculated according to (1) or (3).

Return type none

check4convergence ()

Checks if the re-sampling routine may be terminated, because the number of most frequent clusters remains almost constant. This is done by examining a plot of the amount of most frequent clusters vs. the number of iterations. Convergence is declared once the median normalized slope in a given window of iterations is equal or below “`iter_tol`”. For further information see Supplementary Material of the corresponding publication.

Return type boolean

check_if_data_is_log2_transformed()

Simple check if any value of the data_tuples (i.e. any mean) is below zero. Below zero indicates that the input data was log2 transformed.

Return type boolean

convergence_plot (*filename='convergence_plot.pdf'*)

Creates a two-sided PDF file containing the full picture of the convergence plot, as well as a zoom of it. The convergence plot illustrates the development of the amount of most frequent clusters vs. the number of iterations. The dotted line in this plots represents the normalized slope, which is used for internal convergence determination.

If rpy2 cannot be imported, a CSV file is created instead.

Parameters *filename* (*string*) – the filename of the PDF (or CSV) file.

Return type none

create_rainbow_colors (*n_colors=10*)

Returns a list of rainbow colors. Colors are expressed as hexcodes of RGB values.

Parameters *n_colors* (*int*) – number of rainbow colors.

Return type list

delete_resampling_results()

Resets all variables holding any result of the re-sampling process. This includes the convergence determination as well as the community structure. Does not delete the data that is intended to be clustered.

Return type None

do_it_all (*working_directory=None, distances=None, linkages=None, function_2_generate_noise_injected_datasets=None, min_cluster_size=4, alphabet=None, force_plotting=False, min_cluster_freq_2_retain=0.001, pickle_filename='pyGCluster_resampled.pkl', cpus_2_use=None, iter_max=250000, iter_tol=1e-07, iter_step=5000, iter_top_P=0.001, iter_window=50000, iter_till_the_end=False, top_X_clusters=0, threshold_4_the_lowest_max_freq=0.01, starting_min_overlap=0.1, increasing_min_overlap=0.05, color_gradient='1337', box_style='classic', min_value_4_expression_map=None, max_value_4_expression_map=None, additional_labels=None*)

Evokes all necessary functions which constitute the main functionality of pyGCluster. This is AHC clustering with noise injection and a variety of DLCs, in order to identify highly reproducible clusters, followed by a meta-clustering of highly reproducible clusters into so-called ‘communities’.

The functions that are called are:

- `pyGCluster.Cluster.resample()`
- `pyGCluster.Cluster.build_nodemap()`
- `pyGCluster.Cluster.write_dot()`
- `pyGCluster.Cluster.draw_community_expression_maps()`
- `pyGCluster.Cluster.draw_expression_profiles()`

For a complete list of possible Distance matrix calculations see: <http://docs.scipy.org/doc/scipy/reference/spatial.distance.html> or Linkage methods see: <http://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html>

Note: If memory is of concern (e.g. for a large dataset, > 5000 objects), `cpus_2_use` should be kept low.

Parameters

- **distances** (*list*) – list of distance metrics, given as strings, e.g. [‘correlation’, ‘euclidean’]
- **linkages** (*list*) – list of distance metrics, given as strings, e.g. [‘average’, ‘complete’, ‘ward’]
- **function_2_generate_noise_injected_datasets** (*function*) – function to generate noise-injected datasets. If None (default), Gaussian distributions are used.
- **min_cluster_size** (*int*) – minimum size of a cluster, so that it is included in the assessment of cluster reproducibilities.
- **alphabet** (*string*) – alphabet used to convert decimal indices to characters to save memory. Defaults to string.printable, without ‘,’.

Note: If alphabet contains ‘,’, this character is removed from alphabet, because the indices comprising a cluster are saved comma-separated.

Parameters

- **force_plotting** (*boolean*) – the convergence plot is created after each iter_step iteration (otherwise only when convergence is detected).
- **min_cluster_freq_2_retain** (*float*) –]0, 1[minimum frequency of a cluster (only the maximum of the dlc-frequencies matters here) it has to exhibit to be stored in pyGCluster once all iterations are finished.
- **cpus_2_use** (*int*) – number of threads that are evoked in the re-sampling routine.
- **iter_max** (*int*) – maximum number of re-sampling iterations.

Convergence determination:

Parameters

- **iter_tol** (*float*) –]0, 1e-3[value for the threshold of the median of normalized slopes, in order to declare convergence.
- **iter_step** (*int*) – number of iterations each multiprocess performs and simultaneously the interval in which to check for convergence.
- **iter_top_P** (*float*) –]0, 1[for the convergence estimation, the amount of most frequent clusters is examined. This is the threshold for the minimum frequency of a cluster to be included.
- **iter_window** (*int*) – size of the sliding window in iterations. The median is obtained from normalized slopes inside this window - *should be a multiple of iter_step*
- **iter_till_the_end** (*boolean*) – if set to True, the convergence determination is switched off; hence, re-sampling is performed until iter_max is reached.

Output/Plotting:

Parameters

- **pickle_filename** (*string*) – Filename of the output pickle object
- **top_X_clusters** (*int*) – Plot of the top X clusters in the sorted list (by freq) of clusters having a maximum cluster frequency of at least threshold_4_the_lowest_max_freq (clusterfreq-plot is still sorted by size).

- **threshold_4_the_lowest_max_freq** (*float*) –]0, 1[Clusters must have a maximum frequency of at least threshold_4_the_lowest_max_freq to appear in the plot.
- **min_value_4_expression_map** (*float*) – lower bound for color coding of values in the expression map. Remember that log2-values are expected, i.e. this value should be < 0!
- **max_value_4_expression_map** (*float*) – upper bound for color coding of values in the expression map.
- **color_gradient** (*string*) – name of the color gradient used for plotting the expression map. Currently supported are default, Daniel, barplot, 1337, BrBG, PiYG, PRGn, PuOr, RdBu, RdGy, RdYlBu, RdYlGn and Spectral
- **expression_map_filename** (*string*) – file name for expression map. .svg will be added if required.
- **legend_filename** (*string*) – file name for legend .svg will be added if required.
- **box_style** (*string*) – the way the relative standard deviation is visualized in the expression map. Currently supported are 'modern', 'fusion' or 'classic'.
- **starting_min_overlap** (*float*) –]0, 1[minimum required relative overlap between clusters so that they are assigned the same community. The relative overlap is defined as the size of the overlap between two clusters, divided by the size of the larger cluster.
- **increasing_min_overlap** (*float*) – defines the increase of the required overlap between communities
- **additional_labels** (*dict*) – dictionary, where additional labels can be defined which will be added in the expression map plots to the gene/protein names

Return type None

For more information to each parameter, please refer to `pyGCluster.Cluster.resample()`, and the subsequent functions: `pyGCluster.Cluster.build_nodemap()`, `pyGCluster.Cluster.write_dot()`, `pyGCluster.Cluster.draw_community_expression_maps()`, `pyGCluster.Cluster.draw_expression_profiles()`.

draw_community_expression_maps (*min_value_4_expression_map=None*,
max_value_4_expression_map=None,
color_gradient='1337', *box_style='classic'*)

Plots the expression map for each community showing its object composition.

The following parameters are passed to `pyGCluster.Cluster.draw_expression_map()`:

Parameters

- **min_value_4_expression_map** (*float*) – lower bound for color coding of values in the expression map. Remember that log2-values are expected, i.e. this value should be < 0!
- **max_value_4_expression_map** (*float*) – upper bound for color coding of values in the expression map.
- **color_gradient** (*string*) – name of the color gradient used for plotting the expression map. Currently supported are default, Daniel, barplot, 1337, BrBG, PiYG, PRGn, PuOr, RdBu, RdGy, RdYlBu, RdYlGn and Spectral
- **box_style** (*string*) – name of box style used in SVG. Currently supported are classic, modern, fusion.

Return type none

```
draw_expression_map (identifiers=None, data=None, conditions=None, additional_labels=None, min_value_4_expression_map=None, max_value_4_expression_map=None, expression_map_filename=None, legend_filename=None, color_gradient=None, box_style='classic')
```

Draws expression map as SVG

Parameters

- **min_value_4_expression_map** (*float*) – lower bound for color coding of values in the expression map. Remember that log2-values are expected, i.e. this value should be < 0 !
- **max_value_4_expression_map** (*float*) – upper bound for color coding of values in the expression map.
- **color_gradient** (*string*) – name of the color gradient used for plotting the expression map. Currently supported are default, Daniel, barplot, 1337, BrBG, PiYG, PRGn, PuOr, RdBu, RdGy, RdYlBu, RdYlGn and Spectral
- **expression_map_filename** (*string*) – file name for expression map. .svg will be added if required.
- **legend_filename** (*string*) – file name for legend .svg will be added if required.
- **box_style** (*string*) – the way the relative standard deviation is visualized in the expression map. Currently supported are ‘modern’, ‘fusion’ or ‘classic’.
- **additional_labels** (*dict*) – dictionary, where additional labels can be defined which will be added in the expression map plots to the gene/protein names

Return type none

Data has to be a nested dict in the following format:

```
>>> data = {
...     fastaID1 : {
...         cond1 : ( mean, sd ) , cond2 : ( mean, sd ) , ...
...     }
...     fastaID2 : {
...         cond1 : ( mean, sd ) , cond2 : ( mean, sd ) , ...
...     }
... }
```

optional and, if needed, data will be extracted from

```
self[ 'Data' ]
self[ 'Identifiers' ]
self[ 'Conditions' ]
```

```
draw_expression_map_for_cluster (clusterID=None, cluster=None, file-name=None, min_value_4_expression_map=None, max_value_4_expression_map=None, color_gradient='default', box_style='classic')
```

Plots an expression map for a given cluster. Either the parameter “clusterID” or “cluster” can be defined. This function is useful to plot a user-defined cluster, e.g. knowledge-based cluster (TCA-cluster, Glycolysis-cluster ...). In this case, the parameter “cluster” should be defined.

Parameters

- **clusterID** (*int*) – ID of a cluster (those are obtained e.g. from the plot of cluster frequencies or the node map)

- **cluster** (*tuple*) – tuple containing the indices of the objects describing a cluster.
- **filename** (*string*) – name of the SVG file for the expression map.

The following parameters are passed to `pyGCluster.Cluster.draw_expression_map()`:

Parameters

- **min_value_4_expression_map** (*float*) – lower bound for color coding of values in the expression map. Remember that log2-values are expected, i.e. this value should be < 0 !
- **max_value_4_expression_map** (*float*) – upper bound for color coding of values in the expression map.
- **color_gradient** (*string*) – name of the color gradient used for plotting the expression map. Currently supported are default, Daniel, barplot, 1337, BrBG, PiYG, PRGn, PuOr, RdBu, RdGy, RdYlBu, RdYlGn and Spectral
- **box_style** (*string*) – name of box style used in SVG. Currently supported are classic, modern, fusion.

Return type none

draw_expression_map_for_community_cluster (*name*, *min_value_4_expression_map*=None, *max_value_4_expression_map*=None, *color_gradient*='1337', *sub_folder*=None, *min_obcofreq_2_plot*=None, *box_style*='classic')

Plots the expression map for a given “community cluster”: Any cluster in the community node map is internally represented as a tuple with two elements: “cluster” and “level”. Those objects are stored as keys in self[‘Communities’], from where they may be extracted and fed into this function.

Parameters

- **name** (*tuple*) – “community cluster” -> best obtain from self[‘Communities’].keys()
- **min_obcofreq_2_plot** (*float*) – minimum obCoFreq of an cluster’s object to be shown in the expression map.

The following parameters are passed to `pyGCluster.Cluster.draw_expression_map()`:

Parameters

- **min_value_4_expression_map** (*float*) – lower bound for color coding of values in the expression map. Remember that log2-values are expected, i.e. this value should be < 0 !
- **max_value_4_expression_map** (*float*) – upper bound for color coding of values in the expression map.
- **color_gradient** (*string*) – name of the color gradient used for plotting the expression map. Currently supported are default, Daniel, barplot, 1337, BrBG, PiYG, PRGn, PuOr, RdBu, RdGy, RdYlBu, RdYlGn and Spectral
- **box_style** (*string*) – name of box style used in SVG. Currently supported are classic, modern, fusion.
- **sub_folder** (*string*) – if specified, the expression map is saved in this folder, rather than in pyGCluster’s working directory.

Return type none

draw_expression_profiles (*min_value_4_expression_map*=None, *max_value_4_expression_map*=None)

Draws an expression profile plot (SVG) for each community, illustrating the main “expression pattern”

of a community. Each line in this plot represents an object. The “grey cloud” illustrates the range of the standard deviation of the mean values. The plots are named prefixed by “exProf”, followed by the community name as it is shown in the node map.

Parameters

- **min_value_4_expression_map** (*int*) – minimum of the y-axis (since data should be log2 values, this value should typically be < 0).
- **max_value_4_expression_map** (*int*) – maximum for the y-axis.

Return type none

frequencies (*identifier=None, clusterID=None, cluster=None*)

Returns a tuple with (i) the cFreq and (ii) a Collections.DefaultDict containing the DLC:frequency pairs for either an identifier, e.g. “JGI4|Chlre4|123456” or clusterID or cluster. Returns ‘None’ if the identifier is not part of the data set, or clusterID or cluster was not found during iterations.

Example:

```
>>> cFreq, dlc_freq_dict = cluster.frequencies( identifier = 'JGI4|Chlre4|123456' )
>>> dlc_freq_dict
... defaultdict(<type 'float'>,
... {'average-correlation': 0.0, 'complete-correlation': 0.0,
... 'centroid-euclidean': 0.0015, 'median-euclidean': 0.0064666666666666666,
... 'ward-euclidean': 0.0041333333333333335, 'weighted-correlation': 0.0,
... 'complete-euclidean': 0.0014, 'weighted-euclidean': 0.0066333333333333331,
... 'average-euclidean': 0.0020333333333333332})
```

Parameters

- **identifier** (*string*) – search frequencies by identifier input
- **clusterID** (*int*) – search frequencies by cluster ID input
- **cluster** (*tuple*) – search frequencies by cluster (tuple of ints) input

Return type tuple

info()

Prints some information about the clustering via pyGCluster:

- number of genes/proteins clustered
- number of conditions defined
- number of distance-linkage combinations
- number of iterations performed

as well as some information about the communities, the legend for the shapes of nodes in the node map and the way the functions were called.

Return type none

load (*filename*)

Fills a pyGCluster.Cluster object with the session saved as “filename”. If “filename” is not a complete path, e.g. “example.pkl” (instead of “/home/user/Desktop/example.pkl”), the directory given by self[‘Working directory’] is used.

Note:

Loading of pyGCluster has to be performed as a 2-step-procedure:

```
>>> LoadedClustering = pyGCluster.Cluster()  
>>> LoadedClustering.load( "/home/user/Desktop/example.pkl" )
```

Parameters **filename** (*string*) – may be either a simple file name (“example.pkl”) or a complete path (e.g. “/home/user/Desktop/example.pkl”).

Return type none

median (*_list*)

Returns the median from a list of numeric values.

Parameters **_list** (*list*) –

Return type int / float

plot_clusterfreqs (*min_cluster_size=4, top_X_clusters=0, thresh-
old_4_the_lowest_max_freq=0.01*)

Plot the frequencies of each cluster as a expression map: which cluster was found by which distance-linkage combination, and with what frequency? The plot’s filename is prefixed by ‘clusterFreqsMap’, followed by the values of the parameters. E.g. ‘clusterFreqsMap_minSize4_top0clusters_top10promille.svg’. Clusters are sorted by size.

Parameters

- **min_cluster_size** (*int*) – only clusters with a size equal or greater than min_cluster_size appear in the plot of the cluster freqs.
- **threshold_4_the_lowest_max_freq** (*float*) –]0, 1[Clusters must have a maximum frequency of at least threshold_4_the_lowest_max_freq to appear in the plot.
- **top_X_clusters** (*int*) – Plot of the top X clusters in the sorted list (by freq) of clusters having a maximum cluster frequency of at least threshold_4_the_lowest_max_freq (clusterfreq-plot is still sorted by size).

Note: if top_X_clusters is set to zero (0), this filter is switched off (switched off by default).

Return type None

plot_mean_distributions ()

Creates a density plot of mean values for each condition via rpy2.

Return type none

plot_nodetree (*tree_filename='tree.dot'*)

plot the dendrogram for the clustering of the most_frequent_clusters.

- node label = nodeID internally used for self[‘Nodemap’] (not the same as clusterID!).
- node border color is white if the node is a close2root-cluster (i.e. larger than self[‘for IO skip clusters bigger than’]).
- edge label = distance between parent and children.
- **edge - color codes:**
 - black = default; highlights child which is not a most_frequent_cluster but was created during formation of the dendrogram.
 - green = children are connected with the root.

- red = highlights child which is a most_frequent_cluster.
- yellow = most_frequent_cluster is directly connected with the root.

Parameters `tree_filename` (*string*) – name of the Graphviz DOT file containing the dendrogram of the AHC of most frequent clusters. Best given with “.dot”-extension!

Return type none

resample (*distances*, *linkages*, *function_2_generate_noise_injected_datasets*=None, *min_cluster_size*=4, *alphabet*=None, *force_plotting*=False, *min_cluster_freq_2_retain*=0.001, *pickle_filename*='pyGCluster_resampled.pkl', *cpus_2_use*=None, *iter_tol*=1e-07, *iter_step*=5000, *iter_max*=250000, *iter_top_P*=0.001, *iter_window*=50000, *iter_till_the_end*=False)

Routine for the assessment of cluster reproducibility (re-sampling routine). To this, a high number of noise-injected datasets are created, which are subsequently clustered by AHC. Those are created via `pyGCluster.function_2_generate_noise_injected_datasets()` (default = usage of Gaussian distributions). Each ‘simulated’ dataset is then subjected to AHC *x* times, where *x* equals the number of distance-linkage combinations that come from all possible combinations of “distances” and “linkages”. In order to speed up the re-sampling routine, it is distributed to multiple threads, if *cpus_2_use* > 1.

The re-sampling routine stops once either convergence (see below) is detected or *iter_max* iterations have been performed. Eventually, only clusters with a maximum frequency of at least *min_cluster_freq_2_retain* are stored; all others are discarded.

In order to visually inspect convergence, a convergence plot is created. For more information about the convergence estimation, see Supplementary Material of pyGCluster’s publication.

For a complete list of possible Distance matrix calculations see: <http://docs.scipy.org/doc/scipy/reference/spatial.distance.html> or Linkage methods see: <http://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html>

Note: If memory is of concern (e.g. for a large dataset, > 5000 objects), *cpus_2_use* should be kept low.

Parameters

- **distances** (*list*) – list of distance metrics, given as strings, e.g. [‘correlation’, ‘euclidean’]
- **linkages** (*list*) – list of distance metrics, given as strings, e.g. [‘average’, ‘complete’, ‘ward’]
- **function_2_generate_noise_injected_datasets** (*function*) – function to generate noise-injected datasets. If None (default), Gaussian distributions are used.
- **min_cluster_size** (*int*) – minimum size of a cluster, so that it is included in the assessment of cluster reproducibilities.
- **alphabet** (*string*) – alphabet used to convert decimal indices to characters to save memory. Defaults to `string.printable`, without ‘,’.

Note: If *alphabet* contains ‘,’ this character is removed from *alphabet*, because the indices comprising a cluster are saved comma-separated.

Parameters

- **force_plotting** (*boolean*) – the convergence plot is created after each `iter_step` iteration (otherwise only when convergence is detected).
- **min_cluster_freq_2_retain** (*float*) –]0, 1[minimum frequency of a cluster (only the maximum of the dlc-frequencies matters here) it has to exhibit to be stored in pyGCluster once all iterations are finished.
- **cpus_2_use** (*int*) – number of threads that are evoked in the re-sampling routine.
- **iter_max** (*int*) – maximum number of re-sampling iterations.

Convergence determination:

Parameters

- **iter_tol** (*float*) –]0, 1e-3[value for the threshold of the median of normalized slopes, in order to declare convergence.
- **iter_step** (*int*) – number of iterations each multiprocessing performs and simultaneously the interval in which to check for convergence.
- **iter_top_P** (*float*) –]0, 1[for the convergence estimation, the amount of most frequent clusters is examined. This is the threshold for the minimum frequency of a cluster to be included.
- **iter_window** (*int*) – size of the sliding window in iterations. The median is obtained from normalized slopes inside this window - *should be a multiple of iter_step*
- **iter_till_the_end** (*boolean*) – if set to True, the convergence determination is switched off; hence, re-sampling is performed until `iter_max` is reached.

Return type None

save (*filename='pyGCluster.pkl'*)

Saves the current pyGCluster.Cluster object in a Pickle object.

Parameters filename (*string*) – may be either a simple file name (“example.pkl”) or a complete path (e.g. “/home/user/Desktop/example.pkl”). In the former case, the pickle is stored in pyGCluster’s working directory.

Return type none

write_dot (*filename, scaleByFreq=True, min_obcofreq_2_plot=None, n_legend_nodes=5, min_value_4_expression_map=None, max_value_4_expression_map=None, color_gradient='1337', box_style='classic'*)

Writes a Graphviz DOT file representing the cluster composition of communities. Herein, each node represents a cluster. Its name is a combination of the cluster’s ID, followed by the level / iteration it was inserted into the community:

- The node’s size reflects the cluster’s cFreq.
- The node’s shape illustrates by which distance metric the cluster was found (if the shape is a point, this illustrates that this cluster was not among the most_frequent_clusters, but only formed during AHC of clusters).
- The node’s color shows the community membership; except for clusters which are larger than self[‘for IO skip clusters bigger than’], those are highlighted in grey.
- The node connecting all clusters is the root (the cluster holding all objects), which is highlighted in white.

The DOT file may be rendered with “Graphviz” or further processed with other appropriate programs such as e.g. “Gephi”. If “Graphviz” is available, the DOT file is eventually rendered with “Graphviz”’s dot-algorithm.

In addition, a expression map for each cluster of the node map is created (via `pyGCluster.Cluster.draw_expression_map_for_community_cluster()`).

Those are saved in the sub-folder “communityClusters”.

This function also calls `pyGCluster.Cluster.write_legend()`, which creates a TXT file containing the object composition of all clusters, as well as their frequencies.

Parameters

- **filename** (*string*) – file name of the Graphviz DOT file representing the node map, best given with extension “.dot”.
- **scaleByFreq** (*boolean*) – switch to either scale nodes (= clusters) by cFreq or apply a constant size to each node (the latter may be useful to put emphasis on the nodes’ shapes).
- **min_obcofreq_2_plot** (*float*) – if defined, clusters with lower cFreq than this value are skipped, i.e. not plotted.
- **n_legend_nodes** (*int*) – number of nodes representing the legend for the node sizes. The node sizes themselves encode for the cFreq. “Legend nodes” are drawn as grey boxes.
- **min_value_4_expression_map** (*float*) – lower bound for color coding of values in the expression map. Remember that log2-values are expected, i.e. this value should be < 0.
- **max_value_4_expression_map** (*float*) – upper bound for color coding of values in the expression map.
- **color_gradient** (*string*) – name of the color gradient used for plotting the expression map.
- **box_style** (*string*) – the way the relative standard deviation is visualized in the expression map. Currently supported are ‘modern’, ‘fusion’ or ‘classic’.

Return type none

write_legend (*filename='legend.txt'*)

Creates a legend for the community node map as a TXT file. Herein, the object composition of each cluster of the node map as well as its frequencies are recorded. Since this function is internally called by `pyGCluster.Cluster.write_dot()`, it is typically not necessary to call this function.

Parameters **filename** (*string*) – name of the legend TXT file, best given with extension “.txt”.

Return type none

`pyGCluster.create_default_alphabet()`

Returns the default alphabet which is used to save clusters in a lesser memory-intensive form: instead of saving e.g. a cluster containing identifiers with indices of 1,20,30 as “1,20,30”, the indices are converted to a baseX system -> “1,k,u”.

The default alphabet that is returned is:

```
>>> string.printable.replace( ',', '' )
```

Return type string

`pyGCluster.resampling_multiprocess` (*DataQ=None, data=None, iterations=5000, alphabet=None, dlc=None, min_cluster_size=4, min_cluster_freq_2_retain=0.001, func-*
tion_2_generate_noise_injected_datasets=None)

This is the function that is called for each multiprocesses that is evoked internally in pyGCluster during the re-sampling routine. Agglomerative hierarchical clustering is performed for each distance-linkage combination (DLC) on each of iteration datasets. Clusters from each hierarchical tree are extracted, and their counts are saved in a temporary cluster-count matrix. After *iterations* iterations, clusters are filtered according to

`min_cluster_freq_2_retain`. These clusters, together with their respective counts among all DLCs, are returned. The return value is a list containing tuples with two elements: cluster (string) and counts (one dimensional np.array)

Parameters

- **DataQ** (*multiprocessing.Queue()*) – data queue which is used to pipe the re-sampling results back to pyGCluster.
- **data** (*collections.OrderedDict()*) – dictionary (OrderedDict!) holding the data to be clustered -> passed through to the noise-function.
- **iterations** (*int*) – the number of iterations this multiprocessing is going to perform.
- **alphabet** (*string*) – in order to save memory, the indices describing a cluster are converted to a specific alphabet (rather than decimal system).
- **dlc** (*list*) – list of the distance-linkage combinations that are going to be evaluated.
- **min_cluster_size** (*int*) – minimum size of a cluster to be considered in the re-sampling routine (smaller clusters are discarded)
- **min_cluster_freq_2_retain** (*float*) – once all iterations are performed, clusters are filtered according to 50% (because typically forwarded from pyGCluster) of this threshold.
- **function_2_generate_noise_injected_datasets** (*function*) – function to generate re-sampled datasets.

Return type list

`pyGCluster.seekAndDestroy` (*processes*)

Any multiprocesses given by processes are terminated.

Parameters *processes* (*list*) – list containing multiprocessing.Process()

Return type none

`pyGCluster.yield_noisejected_dataset` (*data, iterations*)

Generator yielding a re-sampled dataset with each iteration. A re-sampled dataset is created by re-sampling each data point from the normal distribution given by its associated mean and standard deviation value. See the example in Supplementary Material in pyGCluster's publication for how to define an own noise-function (e.g. uniform noise).

Parameters

- **data** (*collections.OrderedDict()*) – dictionary (OrderedDict!) holding the data to be re-sampled.
- **iterations** (*int*) – the number of re-sampled datasets this generator will yield.

Return type none

USAGE

This Chapter deals with some features of pyGCluster and explains the basic usage.

The following examples are executed within the Python console (indicated by “>>>”) but can equally be incorporated in standalone scripts.

pyGCluster is imported and initialized like this:

```
>>> import pyGCluster
>>> cluster = pyGCluster.Cluster( )
```

3.1 Clustering

3.1.1 preparing input data

The pyGCuster input has to be nested python dictionary with the following structure

```
>>> data = {
...     Identifier1 : {
...         condition1 : ( mean11, sd11 ),
...         condition2 : ( mean12, sd12 ),
...         condition3 : ( mean13, sd13 ),
...     },
...     Identifier2 : {
...         condition1 : ( mean21, sd21 ),
...         condition2 : ( mean22, sd22 ),
...         condition3 : ( mean23, sd23 ),
...     },
... }
>>> import pyGCluster
>>> cluster = pyGCluster.Cluster( data = data )
```

Note: If any condition for an identifier in the “nested_data_dict”-dict is missing, this entry is discarded, i.e. not imported into the Cluster Class. This is because pyGCluster does not implement any missing value estimation. One possible solution is to replace missing values by a mean value and a standard deviation that is representative for the complete data range in the given condition.

3.1.2 clustering using do_it_all

A simple way to cluster away and to print the basic plots is to evoke `pyGCluster.Cluster.do_it_all()`. This function sets all important parameters for clustering and plotting. E.g.

```
>>> cluster.do_it_all(
...     distances = [ 'euclidean', 'correlation', 'minkowski' ],
...     linkages = [ 'complete', 'average', 'ward' ],
...     cpus_2_use = 4,
...     iter_max = 250000,
...     top_X_clusters = 0,
...     threshold_4_the_lowest_max_freq = 0.01,
...     min_value_4_expression_map = None,
...     max_value_4_expression_map = None,
...     color_gradient = '1337_2',
...     box_style = 'classic'
... )
```

For all available distance metrics and linkage methods, see the documentation of SciPy, sections `scipy.spatial.distance` and `scipy.cluster.hierarchy.linkage`.

3.1.3 clustering using resample

Alternatively, one can cluster only by using the `pyGCluster.Cluster.resample()` function. As for `pyGCluster.Cluster.do_it_all()`, one can specify a series of parameters. The minimal set is:

```
>>> distances = [ 'correlation', 'euclidean', 'minkowski' ]
>>> linkages = [ 'complete', 'average', 'ward' ]
>>> cluster.resample( distances = distances,
...                   linkages = linkages,
...                   pickle_filename = 'pyGCluster_resampled.pkl')
```

Warning: If no `pickle_filename` is specified, no pickle will be written!

3.1.4 saving the clustered data

Generally, the results are pickled into the working directory by calling `pyGCluster.Cluster.save()`

3.1.5 loading the clustered data

Clustering requires some time and is executed on servers or workstations. After successful clustering a pickle object is stored that can be analyzed on a regular Desktop machine. The `pyGCluster` Python pickle object, can be loaded and processed as follows:

```
>>> cluster.load( 'path_to_pyGCluster_pkl_file.pkl' )
```

Before starting, it may be necessary to define an output directory or 'Working directory' in the `pyGCluster.Cluster` object, where all the figures will be plotted into.

This can be done by simply editing the keys in the `pyGCluster.Cluster` object:

```
>>> cluster['Working directory'] = '/myPath/'
```

3.2 Clustered data visualization

Prior clustering several parameters have to be defined in order to control, e.g. memory usage. This is done by the kwargs 'minSize_resampling' and 'minFrequ_resampling'. Normally, it is not obvious how many clusters and communities are finally obtained. Therefore the user can specify how many of the top X clusters should be taken into consideration for plotting. Obviously, one can not specify a minimum cluster size smaller than the original cluster input parameter.

3.2.1 Communities assignment

The communities can be plotted using different options. Prior any visualization scripts the communities have to be specified.

This can be done by the `pyGCluster.Cluster.build_nodemap()` function.

This command create communities from a set of the top 0.5% most frequent clusters with a minimum cluster size of 4:

```
>>> cluster.build_nodemap( min_cluster_size = 4, threshold_4_the_lowest_max_freq = 0.005 )
```

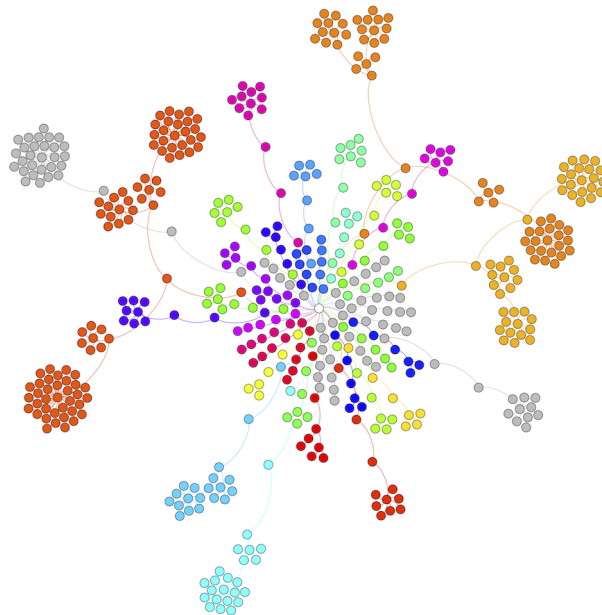
Note: These values can be modified in order to vary the community number and composition, the higher the threshold, the less clusters are included, i.e. the quality/stringency is increased.

3.2.2 Write DOT file

Create the DOT file of the node map showing the cluster composition of the communities. A filename can be defined and the number of nodes for the legend. The function `pyGCluster.Cluster.write_dot()` can be used:







```
>>> cluster.write_dot( filename = 'example_1promilleclusters_minsize4.dot', \
... n_legend_nodes = 5 )
```

The DOT file can be used as input for e.g. gephi and a nodemap can be build.



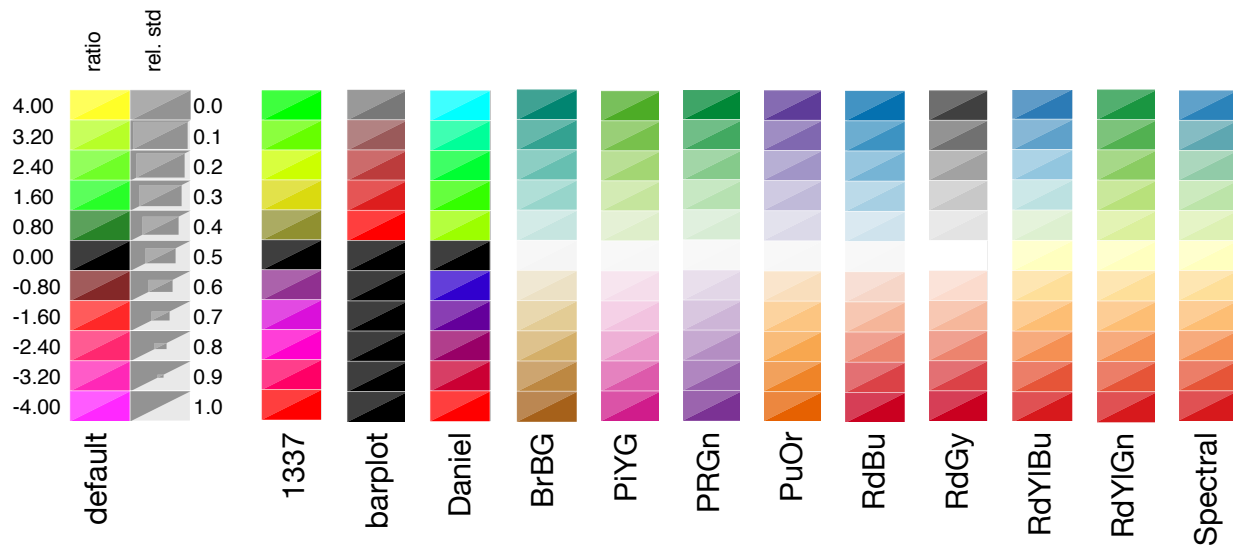
```
>>> cluster.draw_community_expression_maps(
...     min_value_4_expression_map = -3,
...     max_value_4_expression_map = 3,
...     color_gradient = 'default'
... )
```

Note: One may want to adjust the color range depending on the range of the values which should be visualized. The log2 range of 3 fits in the example case for proteomics data. When visualizing transcriptomics data, a broader range is required.

01_WC_PA_Fu4a	02_WC_PA_Fu4a	03_WC_PA4H	04_CP_PA_Fu4a	05_CP_PA_Fu4a	06_PA4H
					
<p>Cre07.2302050.11.1PAC16:1965670.0.0096 Cre12.6488300.11.1PAC16:19674719.0.0185 Cre06.2791650.11.1PAC16:1966673.0.0127 NCBI010000554.2c2AA00949.2c389083747TAA, exp: photosystem I P700 chlorophyll A apoprotein A2 [<i>Chlamydomonas reinhardtii</i>]. 0.0212 Cre05.2644500.11.1PAC16:1968777.0.0093 Cre09.421200.11.1PAC16:19662813.0.0109 NCBI010000554.2c2AA01471.132808376photosystem I P700 chlorophyll A apoprotein A1 [<i>Chlamydomonas reinhardtii</i>]. 0.0026</p>					

	ratio	rel. std.
3.00		0.0
2.40		0.1
1.80		0.2
1.20		0.3
0.60		0.4
0.00		0.5
-0.60		0.6
-1.20		0.7
-1.80		0.8
-2.40		0.9
-3.00		1.0

which would create a new profile with the name `myProfile`. The list contains tuples and within each tuple, the first number represents the relative induction to the plotted data set or relative to the defined min and max values and the second tuple represent the color in r,g,b format (0, .. , 255).



Box styles

The box styles that are currently part of pyGCluster are

The box styles are stored in `cluster['expression map']['SVG box styles']`. The general format is a string that contains SVG code using Python string format placeholders. E.g.

```
<g id="rowPos{0}_conPos{1}">
  <title>{ratio}&#177;{std} - [{x0}.{y0} w:{width} h:{height}</title>
  <rect x="{x0}" y="{y0}" width="{width}" height="{height}"
    style="fill:rgb({r},{g},{b});stroke:white;stroke-width:1;" title="{ratio}&#177;{std}" />
  <rect x="{x1}" y="{y1}" width="{widthNew}" height="{heightNew}"
    style="fill:None;stroke:black;stroke-width:1;" title="{ratio}&#177;{std}" />
</g>
```

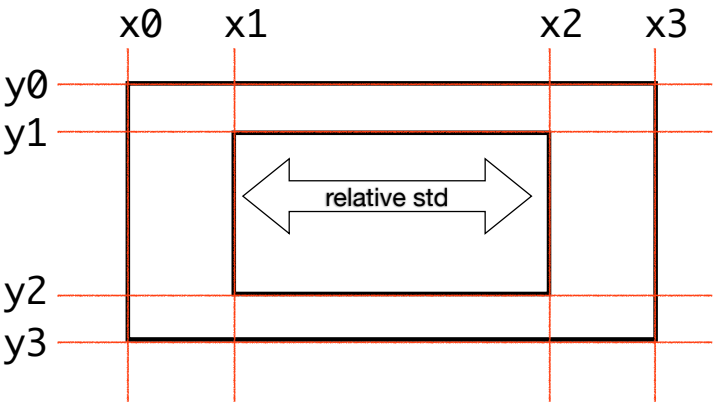
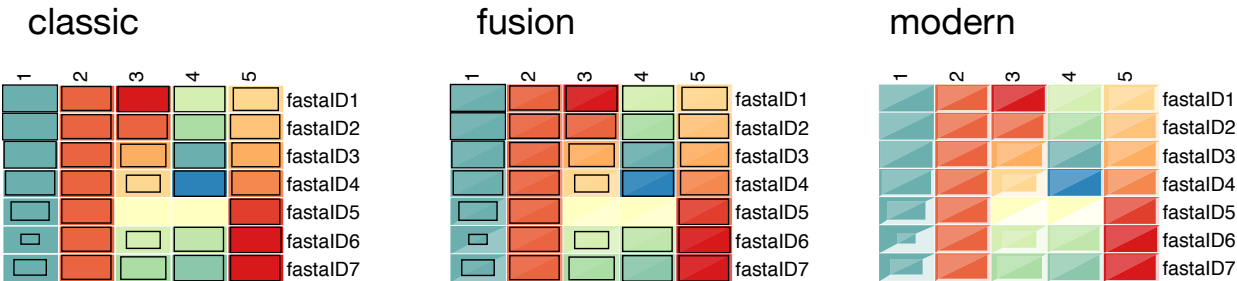
Coordinate definition is:

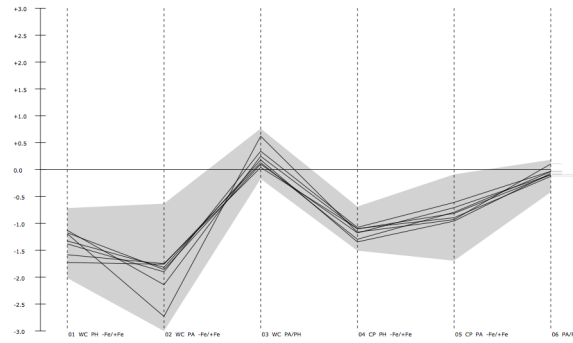
3.2.4 Plot expression profile

Expression profile can be plotted for every community, to further visualize the overall regulation of all objects in this community, by using the function `pyGCluster.Cluster.draw_expression_profiles()`:

```
>>> cluster.draw_expression_profiles(
...     min_value_4_expression_map = -3,
...     max_value_4_expression_map = 3
... )
```

An example of an expression profile is given below.



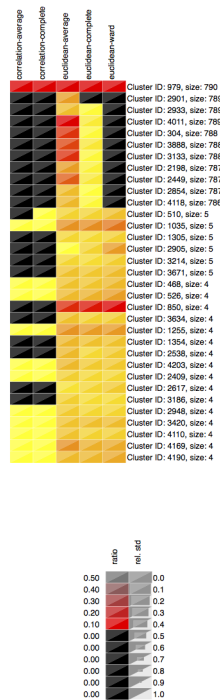


3.2.5 Plot cluster frequencies

Frequencies of clusters can be plotted using the `pyGCluster.Cluster.plot_clusterfreqs()` function, which also draws a own legend:

```
>>> cluster.plot_clusterfreqs( min_cluster_size = 4, top_X_clusters = 33 )
```

The cluster frequency plot is given below.



3.2.6 Save results

It is possible to store the modified pkl file again and save the results for further processing steps:

```
>>> cluster.save( filename = 'example_1promille_communities.pkl' )
```

Warning: One should be careful about editing essential stuff in the object like the 'Data' entry, because it will be overwritten, once saved.

EXAMPLE SCRIPTS

This section describes various example scripts to demonstrate pyGCluster

4.1 Functionality check

Note: After installation, please run the script `test_pyGCluster.py` from the `exampleScripts` folder to check if pyGCluster was installed properly.

Testscript to demonstrate functionality of pyGCluster

A synthetic dataset is used to check the correct installation of pyGCluster. This dataset contains 10 ratios (Gene 0-9) which were randomly sampled between 39.5 and 40.5 in 0.1 steps with a low standard deviation (randomly sampled between 0.1 and 1) and 90 ratios (Gene 10-99) which were randomly sampled between 3 and 7 in 0.1 steps with a high standard deviation (randomly sampled between 0.1 and 5)

5000 iterations are performed and the presence of the most frequent cluster is checked.

This cluster should contain the Genes 0 to 9.

Usage:

```
./test_pyGCluster.py
```

When the iteration has finished (this should normally take not longer than 20 seconds), the script asks if you want to stop the iteration process or continue:

```
iter_max reached. See convergence plot. Stopping re-sampling if not defined
otherwise ...
... plot of convergence finished.
See plot in "../exampleFiles/functionalityCheck/convergence_plot.pdf".
```

```
Enter how many iterations you would like to continue.
(Has to be a multiple of iterstep = 5000)
(enter "0" to stop resampling.)
(enter "-1" to resample until iter_max (= 5000) is reached.)
Enter a number ...
```

Please enter 0 and hit enter (The script will stop and the test will finish).

The results are saved into the folder `functionalityCheck`.

Additionally expression maps and expression profiles are plotted.

4.2 Testscripts to demonstrate pyGCluster

The basic script utilizes the function `pyGCluster.Cluster.do_it_all()` whereas the advanced script utilizes single steps to perform the clustering

4.2.1 Basic script

Testscript to demonstrate functionality of pyGCluster

This script imports the data of Hoehner et al. (2013) and executes pyGCluster with 250,000 iterations of resampling. pyGCluster will evoke 4 threads (if possible), which each require approx. 1.5GB RAM. Please make sure you have enough RAM available (4 threads in all require approx. 6GB RAM). Duration will be approx. 2 hours to complete 250,000 iterations on 4 threads.

Usage:

```
./basicClusterHoehnerExampleData.py <pathToExampleFile>
```

If this script is executed in folder `pyGCluster/exampleScripts`, the command would be:

```
./basicClusterHoehnerExampleData.py ../exampleFiles/hoehner_dataset.csv
```

The results are saved in `"../pyGCluster/exampleScripts/hoehner_example_run/"`.

4.2.2 Advanced script

Testscript to demonstrate functionality of pyGCluster

This script imports the data of Hoehner et al. (2013) and executes pyGCluster with 250,000 iterations of resampling. pyGCluster will evoke 4 threads (if possible), which each require approx. 1.5GB RAM. Please make sure you have enough RAM available (4 threads in all require approx. 6GB RAM). Duration will be approx. 2 hours to complete 250,000 iterations on 4 threads.

Usage:

```
./advancedClusterHoehnerExampleData.py <pathToExampleFile>
```

If this script is executed in folder `pyGCluster/exampleScripts`, the command would be:

```
./advancedClusterHoehnerExampleData.py ../exampleFiles/hoehner_dataset.csv
```

The results are saved in `"../pyGCluster/exampleScripts/hoehner_example_run/"`.

4.3 Plot communities from a pyGCluster pkl

Plot communities of a given pyGCluster pkl file. All output will be directed into the folder, where the input data is located

The `top_X_clusters` option can be used to use the top X clusters for community determination. The `threshold_4_the_lowest_max_freq` option define the threshold for the maximum frequency of the clusters which should be incorporated into the community determination.

Default values are:

```
-threshold_4_the_lowest_max_freq=0.005
```

Usage:

```
./plotCommunities.py <pathTopyGClusterPickle>
```

optional:

```
./plotCommunities.py <pathTopyGClusterPickle> <threshold_4_the_lowest_max_freq=0.005>  
OR <top_X_clusters=100>
```

4.4 Retrieve pyGCluster pkl info

Get some information from a pyGCluster pkl object

Usage:

```
./getpyGPickleInfo.py <MERGED_pyGCluster_pkl_object>
```

4.5 Plot simple expression map

Testscript to plot a simple heatmap.

This can be used to visualize the different box styles for the expression maps and to test the plotting function.

This can be also used as a basis to visualize own datasets by simply defining the ‘data’ dictionary in this script.

Usage:

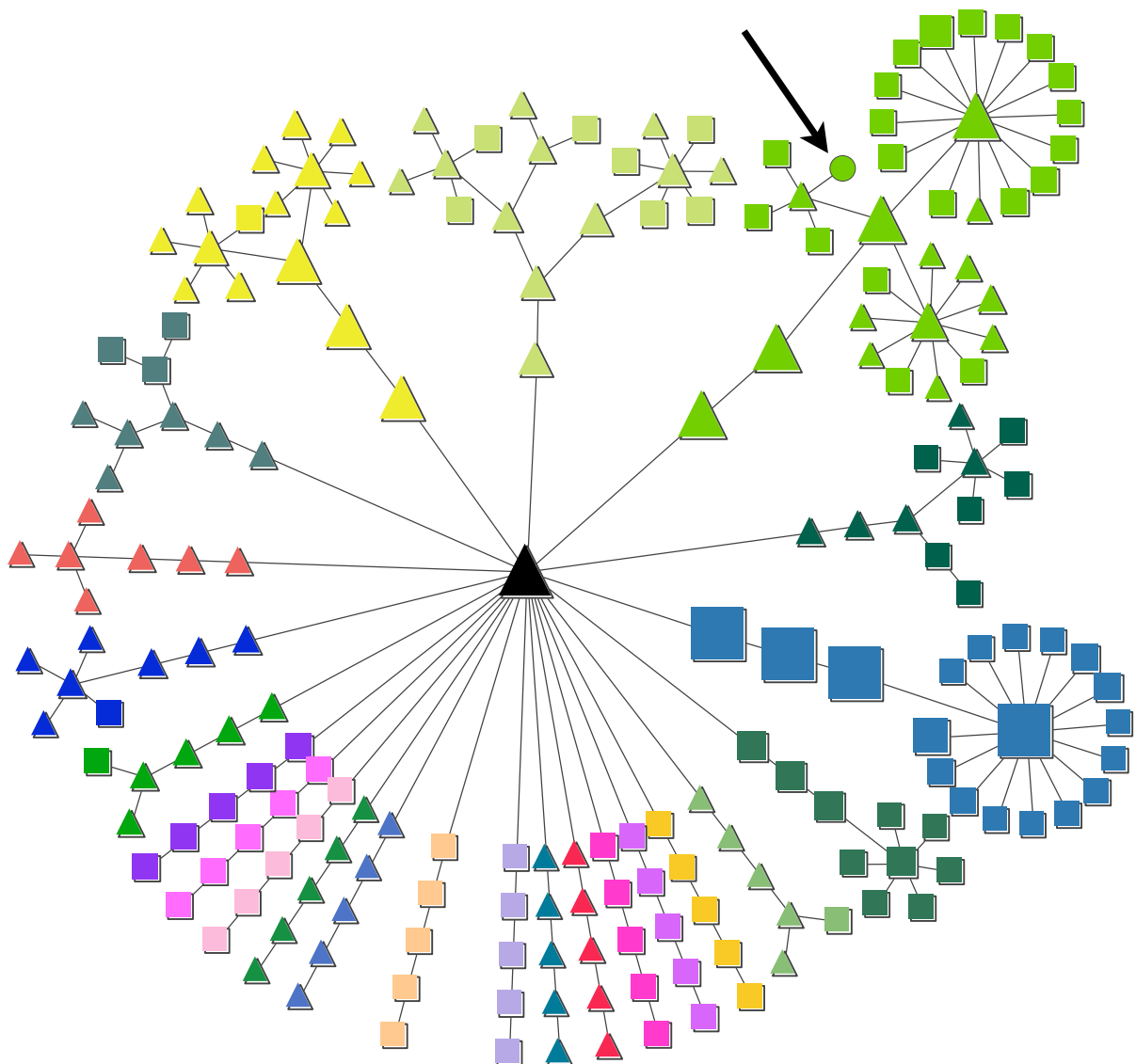
```
./simpleExpressionMap.py
```

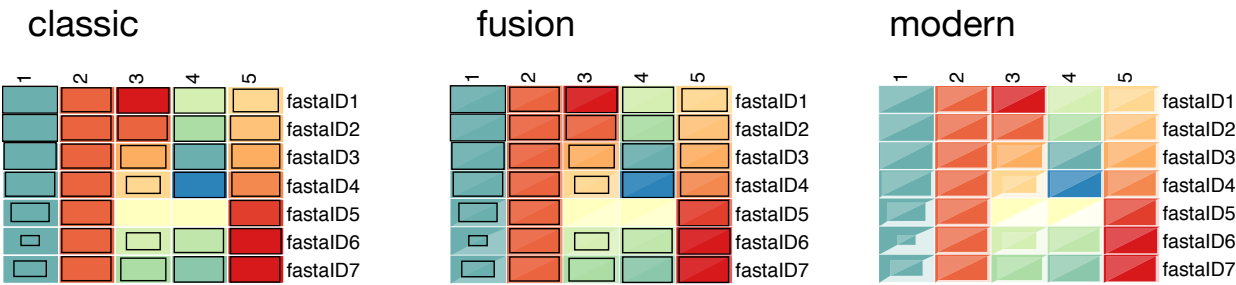
4.6 Nodemap and expression map example

The figures show examples of a nodemap and expression map generated by pyGCluster. The nodemap was taken from the manuscript.

4.6.1 Nodemap

4.6.2 Expression map





INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

a

`advancedClusterHoehnerExampleData`, [32](#)

b

`basicClusterHoehnerExampleData`, [32](#)

g

`getpyGPickleInfo`, [33](#)

p

`plotCommunities`, [32](#)

`pyGCluster`, [7](#)

s

`simpleExpressionMap`, [33](#)

t

`test_pyGCluster`, [5](#)

INDEX

A

advancedClusterHoehnerExampleData (module), 32

B

basicClusterHoehnerExampleData (module), 32

build_nodemap() (pyGCluster.Cluster method), 9

C

calculate_distance_matrix() (pyGCluster.Cluster method), 10

check4convergence() (pyGCluster.Cluster method), 10

check_if_data_is_log2_transformed() (pyGCluster.Cluster method), 10

Cluster (class in pyGCluster), 7

convergence_plot() (pyGCluster.Cluster method), 11

create_default_alphabet() (in module pyGCluster), 20

create_rainbow_colors() (pyGCluster.Cluster method), 11

D

delete_resampling_results() (pyGCluster.Cluster method), 11

do_it_all() (pyGCluster.Cluster method), 11

draw_community_expression_maps() (pyGCluster.Cluster method), 13

draw_expression_map() (pyGCluster.Cluster method), 13

draw_expression_map_for_cluster() (pyGCluster.Cluster method), 14

draw_expression_map_for_community_cluster() (pyGCluster.Cluster method), 15

draw_expression_profiles() (pyGCluster.Cluster method), 15

F

frequencies() (pyGCluster.Cluster method), 16

G

getpyGPickleInfo (module), 33

I

info() (pyGCluster.Cluster method), 16

L

load() (pyGCluster.Cluster method), 16

M

median() (pyGCluster.Cluster method), 17

P

plot_clusterfreqs() (pyGCluster.Cluster method), 17

plot_mean_distributions() (pyGCluster.Cluster method), 17

plot_nodetree() (pyGCluster.Cluster method), 17

plotCommunities (module), 32

pyGCluster (module), 7

R

resample() (pyGCluster.Cluster method), 18

resampling_multiprocess() (in module pyGCluster), 20

S

save() (pyGCluster.Cluster method), 19

seekAndDestry() (in module pyGCluster), 21

simpleExpressionMap (module), 33

T

test_pyGCluster (module), 5, 31

W

write_dot() (pyGCluster.Cluster method), 19

write_legend() (pyGCluster.Cluster method), 20

Y

yield_noisejected_dataset() (in module pyGCluster), 21