GRAPH ANALYSIS LIBRARY (v0.0 beta)
(A preliminary manual)
by Dr. Gorka Zamora-López

1) INTRODUCTION

1.1) WHAT IS GAlib?
GAlib is a library for the analysis of graphs and complex networks in Python/NumPy. It is intended for researchers studying complex networks and those performing data analysis with a reasonable knowledge of graph theory and Python/NumPy programming. The library is very easy to modify and extend in order to adapt it for the users' personal needs. As a network scientist and developer of data analysis methods myself, flexibility was the original motivation to write GAlib. It does not include visualization functionalities but it can be easily integrated in larger projects.

GAlib does not define a graph object, it represents networks by their adjacency matrices as rank-2 ndarrays. This choice limits the size of the networks GAlib can handle but it allows to exploit NumPy to boost performance far beyond code written in pure Python.

The library contains 3 modules. See the Reference Guide or the interactive documentation for a detailed list of functions within each module.
- *gatools.py* : miscelaneous helper functions, e.g. data-type transformations.
- *galib.py* : basic graph descriptors, e.g. degrees and clustering.
- *gamodels.py* : Generation of synthetic networks and randomization.

In an interactive session type help(modulename) to see a list of functions included in each of the modules or help(functionname) to access the information of each function. In Ipython simply type `modulename?` or `functionname?`

In addition to the library files GAlib comes with several example scripts to illustrate its use (see .../GAlib/Examples/ folder). For network scientists who are not familiar with the Python / NumPy environment, GAlib includes also a few helper scripts to generate and save into files ensembles of random or scale-free networks, as-well-as ensembles of rewired networks conserving the degrees. Users only need to modify a few selected fields and let GAlib do the rest. See .../GAlib/HelperScripts/ folder.

1.2) INSTALLATION
As a library, GAlib does not require any installation but the modules need to be placed in the python path such that they can be imported. The only requirement is to have NumPy/SciPy installed together with Python. If NumPy/Scipy is not yet installed in the computer, visit the following webpage for further information (http://www.scipy.org).

For beginners to Python/Numpy/Scipy it is highly recommended to install a Python distribution that already includes NumPy, SciPy and other utilities for numerical and scientific tools. These are easy to install and stable distributions. For example Canopy (https://www.enthought.com/products/canopy/) or Anaconda (https://store.continuum.io/cshop/anaconda/)

If you don't aim to modify the code in GAlib the simplest way to install it is to copy the modules into the *site-packages* folder. This is a special folder that is installed with Python to place third party libraries and packages. However, its exact location depends both on the operating system and the Python version(s) or distribution(s) currently installed.

1) Find the current python version installed.
This might be more complicated and annoying than it sounds because there might be several versions of Python installed in the computer. For example, Mac OS X and Linux come with a pre-installed version of Python. When re-installing Python through a distribution the older version is not removed and each version has its own "site-packages" folder. To find the current Python version open a terminal and type "python". An interactive shell will open (type exit() to leave the interactive shell). The current Python version is displayed in the header. If a Python distribution has been installed that includes NumPy and SciPy, very likely iPython has been installed too. iPython is an advanced interactive shell for Python. If this is the case, open another terminal and type `ipython` to open another interactive shell. In the header the version of Python used by iPython is displayed. Make sure that the two version displayed in the headers are the same. If the two versions are different you will have to copy GAlib into the site-packages folder of both Python versions to warranty that it will always work independently of how to run the scripts.

2) Locate the "site-packages" folder.
Once the current version of installed Python installed is known, search the filesystem for folders named "site-packages". Usually the whole path looks like "…/lib/pythonX.Y/site-packages" where X.Y is the Python version number. If several Python versions are installed in the computer, locate the "site-packages" folders for those versions you will be using.

3) Copy the modules of GAlib into the "site-packages" folder.

4) Check GAlib is correctly installed.
Close any open interactive shells and open them again by typing `python` or `ipython` in the terminal. Import the modules by typing `import galib`, `import gatools` and `import gamodels`. If no error is returned, the library is correctly installed. Otherwise a warning will be raised that the modules are not found.

Now, if you intend to modify the library or to include your own functions to it, the modules in the "site-packages" are usually not possible to modify due to permission restrictions. Alternatively, you can place the modules of GAlib in any folder you wish, e.g. "/usr/myusername/GraphAnalysisLibrary". To allow Python to find GAlib a text file can be created in the site-packages folder that contains additional folders to be included in the path. Follow steps 1) and 2) previously and do the following:

3) Include a .pth file in the site-packages folder
Open a new text document in an editor and type in the files the path you want to include, e.g. the file will have a single line with the text */usr/myusername/GraphAnalisysLibrary*. Save the text document in the *site-packages* folder and give it the extension *.pth*, for example: *extrapythonpaths.pth*. You can include as many paths as you want in a .pth file, every path being in one independent line.

1.3) FUTURE PLANS
At this starting stage GAlib includes basic graph analysis tools. In future releases further functionalities, graph measures and network model generators will be included:
- measures for weighted and/or directed networks,
- classical graph models,
- functions to compute the roles of nodes in networks with modular organization,
- improved data conversions for graph formats used in other packages (Pajek, graph-tools, NetworkX, Gephi, etc.)
- support for sparse matrices to increase the size of networks handled.

Any third party collaboration to improve GAlib is highly welcome.

1.4) HOW TO FIND DOCUMENTATION
While working in an interactive session type:

```
>>> help(modulename)
```

where modulename can be *galib*, *gamodels* or *gatools*, to see a list of functions included in the module. To find further information of each individual function: aim, parameters and returned values, type:

```
>>> help(functionname)
```

In IPython the help command is replaced by a question mark after the module or function name:

```
>>> modulename?
>>> functionname?
```

1.5) GAlib and my own research
GAlib is an accidental by-product of my scientific research. In 2004 I started my Ph.D. in the group of Prof. Jürgen Kurths at the University of Potsdam (Germany) to make research in brain connectivity. My goal was to uncover the topological organization of long-range connections in the cat and the macaque brains. Back in those times complex networks was a exploding field and the computational tools were scarce. There were some very good programs, e.g. Pajek, but such pre-packaged programs were too rigid environments for a field constantly in change. I couldn't just wait until someone would include new graph measures and models to those programs and I just started coding myself every graph measure that I needed. Hence, the need for flexibility in a research environment was the main reason for GAlib to happen, and remains its main scope. GAlib is not intended for unexperienced users that need a "blackbox" to put their data in. I enjoyed that process very much and I ended making my own contributions to graph theory, particularly on the meaning and quantification of significance of graph measures.

Over the years I was often tempted to publicly release the library and so contribute to the open-source ecosystem of Python and NumPy. The evolution of Numeric and Numarray into NumPy delayed my intentions, and it was for good. The extra experience I have gained in the mean time allowed me to improve every single function from code that "just works" to code that I hope to be very much optimized. I discovered that optimizing Python code is very non-trivial. Far from the mandates given by the Zen of Python, the combination of Python and NumPy allows to very often do the same job in several different manners, each having its own pros and cons. in Appendix II I summarize some of those "tricks".

2) USER GUIDE

2.1) GETTING STARTED

In GAlib, graphs and networks are represented by their adjacency matrices, that is, 2-dimensional NumPy arrays, called `ndarrays`. The library makes extensive use of NumPy's array manipulation tools to compute the metrics of the networks. An empty network of N nodes is thus represented by a NxN numpy array full of zeros.

```
>>> from numpy import*
>>> N = 20
>>> emptynet = zeros((N,N), dtype=uint8)
```

The dtype is optional and depends on whether the network shall be binary or weighted. Usually, for binary networks the unsigned 8-bit type can be used (uint8) and regular floatting-point dtype (float) for real-valued weighted networks. See the numpy documentation for the available data types.

After importing GAlib its functions can be applied to existing networks, or synthetic networks can be generated for their analysis. In the following, a random graph of the Erdös-Renyi type is generated with N = 20 nodes and link probability 0.2 for its further analysis.

```
>>> from galib import *
>>> from gamodels import *
>>> N = 20
>>> p = 0.2
>>> net = ErdosRenyiGraph(N, p)
```

In order to know the dtype of the generated network type:

```
>>> net.dtype
dtype('uint8')
>>>
```

The number of links in the network can be recovered from counting the number of 1s in the adjacency matrix. As the network is binary, we only need to sum over its elements. The 0.5 factor is applied because in this case the network is undirected:

```
>>> L = 0.5 * net.sum()
>>> L
36.0
>>> density = (2*L)/(N*(N-1))
>>> density
0.18947368421052632
>>>
```

NOTE:
In the case of weighted networks, the array method net.sum() will return the sum of the weights of the links. In this case, one has to find the number of non-zero elements in the adjacency matrix which is done by the NumPy method a.nonzero(). So, to compute the number of links type L = 0.5 * len(net.nonzero()[0]).

We compute the degree of every node using the `Degree()` function and compute the mean degree <k> using the numpy method `array.mean()`.

```
>>> knodes = Degree(net)
>>> knodes
array([5, 3, 1, 3, 2, 6, 3, 4, 3, 3, 4, 5, 2, 3, 3, 6, 6, 3, 6, 1])
>>> meank = knodes.mean()
>>> meank
3.6000000000000001
>>>
```

GAlib works also with directed networks and, whenever applicable, functions include the optional boolean parameter `directed`. By default all GAlib functions and network generators that can be used with directed networks come specified for undirected networks, i.e. with parameter `directed=False`. In the following example we generate a random directed graph.

```
>>> dinet = ErdosRenyiGraph(N, p, directed=True)
>>> Ld = dinet.sum()
>>> Ld
68
>>> density = float(Ld) / (N*(N-1))
>>> density
0.17894736842105263
>>>
```

As a directed network, its nodes have different input and output degrees (in-k and out-k), therefore the `Degree()` function has to be called with the parameter `directed=True` and with two names for each of the arrays to be unpacked:

```
>>> inknodes, outknodes = Degree(dinet,True)
>>> inknodes
array([2, 4, 6, 1, 4, 2, 4, 2, 4, 6, 4, 2, 2, 2, 2, 5, 3, 5, 5, 3])
>>> outknodes
array([3, 3, 0, 3, 5, 8, 1, 2, 1, 4, 5, 4, 2, 3, 2, 7, 7, 1, 2, 5])
>>>
```

NOTE:
Notice that when functions are called for directed network measures, they often return 2-dimensional arrays that need to be adequately unpacked as in the example above.

Another popular graph measure is the clustering coefficient, that is, the fraction of triangles in a network out of all possible triangles the network could have. It is therefore a measure of transitivity or cohesiveness. A tree graph has no triangles and its clustering coefficient C = 0. On the contrary, a complete graph of size N >= 3 has C = 1 because all nodes form triangles with all their neighbours. Additionally one can define the local clustering of individual nodes. The function `Clustering()` computes both the clustering coefficient of the network and the local clustering of every node, returning a floating number between 0 and 1, and an array of length N with the local clustering of every node. Remember to adequately call the function to unpack both results. For the random graph generated above we have:

```
>>> clustcoef, clustnodes = Clustering(net)
>>> clustcoef
0.1271186440677966
>>> clustnodes
array([ 0.3       ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.13333333,  0.33333333,  0.        ,  0.        ,  0.        ,
        0.16666667,  0.        ,  0.        ,  0.33333333,  0.        ,
        0.13333333,  0.2       ,  0.33333333,  0.06666667,  0.        ])
>>>
```

WARNING:
Notice that the clustering coefficient C and the mean of the local clustering are not equal. In the previous case, the mean value of the local clustering is:

```
>>> clustnodes.mean()
0.099999999999999992
```

that is different from the actual value `clustcoef`. This is not an error, but a well-known property which arises from the manner the local clustering is defined. Indeed, in random graphs the local clustering is known to anti-correlate with the degree of the node.

NOTE:
The function `Clustering()` only accepts undirected networks. The reason is that there is no agreed manner on how to define what a triangle is in directed networks and several approaches might be taken. In future releases GAlib will have support for Clustering in directed networks.

2.2) DATA I/O
Reading networks from files and saving them into files is done using the standard data I/O tools of numpy: functions loadtxt() and savetxt() are available to work with text files. Funtions `load()` and `save()` work with the standard binary data format of numpy. For large datasets we recommend `load()` and `save()` because of their better performance. In the following example we save the random network created before into a file and read it again. Then we make sure that the network is the same using the `ArrayCompare()` function in the *gatools.py* module

```
>>> savetxt('filepath.txt', net, fmt='%d')
>>> newnet = loadtxt('filepath.txt', dtype=uint8)
>>> from gatools import*
>>> ArrayCompare(net, newnet)
True
>>>
```

to save and read the data as binary files, we could instead do the following:

```
>>> save('filepath.npy', net)
>>> newnet = load('filepath.npy')
>>> ArrayCompare(net, newnet)
True
>>>
```

Notice that `savetxt()` and `loadtxt()` require data type formatter options to be explicitely introduced, while `save()` and `read()` functions remember the dtype of the array. The function `ArrayCompare()` is a simple wrap around NumPy's array comparison utilities. When a1 and a2 are two NumPy arrays of the same shape, `a1 == a2` returns another array of the same shape with the element-wise result of the comparison. `ArrayCompare()` is True is all the elements are equal and False if there is at least one element in both arrays that is not equal.

GAlib also includes support to read and write network files in popular network formats used by other graph analysis software, e.g. Pajek. See the example script *DataConversions.py* in *GAtools/Examples*.

2.3) USING THE "UTILITY SCRIPTS"

GAlib includes several scripts useful for network scientists independently of their usual programming environment or experience. These scripts are intended to easily perform common tasks such as the generation of ensembles of random or rewired networks. In general there are two steps to follow to use the Utility Scripts:

1) *Specify the parameters and the options desired.*
Open the script in a text editor and manually modify the parameters desired for the generated networks. Other options are to be also manually introduced such as the target directory where the results will be saved and the format of the output data.

2) *Run the script.*
Open a terminal and move to the location of the script, e.g.,

```
$ cd /somepath/GAlib/UtilityScripts
```

Run the script by calling Python

```
$ python scriptname.py
```

Utility scripts save the results into files, which can later be further analysed with GAlib or any other preferred software.

2.3.1) Generate ensembles of random networks

The Utility Script *GenerateRandomNetwork.py* generates ensembles of random networks easily even for users with little or no programming experience. After setting of some parameters and options to personalize the results, the script generates the networks and saves them into files for later use. It can generate two types of random graphs: Erdös-Renyí graphs and random graphs with specified number of links. The only difference between the two methods is that the Erdös-Renyí model starts from an empty network and includes a link between any two nodes with probability p. This means that different realizations of the model return networks with slightly different number of link. On the ensemble average, networks have link density $\rho = p$, but realizations differ from each other. This is not desirable when we have a real network with N nodes and L links and we want to create random networks of the same characteristics for comparison. For those cases, the function `GenerateRandonGraph()` of the *gamodels.py* module creates random networks all having the same number of links. The option `model` in the script allows to select between Erdös-Renyí networks and random networks with specified number of links.

NOTE :
Before specifying the number of links for `GenerateRandomGraph()`, remind that the density of links in a directed graph is $\rho = L / (N*(N-1))$ and in undirected graphs is $\rho = 2L / (N*(N-1))$.

The main choices are:

NETWORK PARAMETERS
- `N` : Size of the network (number of nodes).

- `realiz` : Number of networks to be generated.
- `directed` : True to generate directed network, False for undirected.
- `model` : type of random networks desired. The Erdos-Renyi (`model='ER'`) model generates random networks with given link probability, p. The option `model='RG'` option generates random networks all with the same number of links.
- `p` : Link probability for the 'ER' case.
- `L` : Number of links in the 'RG' case.

OUTPUT OPTIONS
- `outpath` : Path where the resulting networks will be saved.
- `filename` : Basename for the output files. Final name will be *basename_reX.dat* where X is the realization number.
- `filetype` : Format of the output network file. Option `filetype='text'` saves the adjacency matrices as text files with a *.dat* extension. The option `filetype='binary'` saves the adjacency matrices in the numpy binary format with a *.npy* extension. The option `filetype='pajek'` saves the networks into text files with the Pajek format and extension *.net*.

RUNNING THE SCRIPT
After personalizing the network and output options in the script, open a
terminal window and move to the folder containing the script, e.g. by typing:

```
$ cd ~/GraphAnalysisLibrary/HelperScripts
```

then type the following command:

```
$ python GenerateRandomNetworks.py
```

The program will feedback the parameters and save the generated files in the specified path.

2.3.2) Generating ensembles of rewired networks that conserve the degree sequence.
Given a network, the script *GenerateRewiredNetworks.py* creates ensembles of randomized networks with the same degree sequence as the original network. The program automatically finds whether the input network is directed or undirected and generates the networks accordingly.

INPUT PARAMETERS
Input networks can be entered in three file formats: 1) a text file containing the adjacency matrix (typically *.txt* or *.dat*), 2) a numpy binary file containing the adjacency matrix (.npy extension) or 3) a text file with the network in Pajek format. The input parameters are:
- inputfile : The path to the file in which the network is found.

REWIRING PARAMETERS
- `realiz` : Number of rewired networks to be generated.
- `prew` : Rewiring 'probability'. Specifies the number of iterations of the Markov chain. Networks will be rewired prew*L times, where L is the number of links in the network.
- `checkresult` : if True, the programm will check for every rewired network whether the degree sequence(s) have been conserved, as-well-as for the lack of self-loops and multiple edges. If False, the programm will skip the check and run faster.

> --------------
> NOTE :
> Due to the nature of the Markov chain process, prew = 2 is the minimally acceptable value to achieve a network with all internal topology randomized, apart from the degrees of the nodes. If computational time is not a strong limitation, use prew = 10. Otherwise, prew = 5 is the smallest recommended value.
> --------------

OUTPUT PARAMETERS
As for the input files, the resulting network can also be saved into three types of files: text files, numpy binary or Pajek-readable files. The output parameters to specify are:

- `outpath` : Path where the resulting networks will be saved.
- `filename` : Basename for the output files. Final name will be *basename_reX.dat* where X is the realization number.
- `filetype` : Choice to save networks. `'text'` saves the adjacency matrices as text files with a *.dat* extension. `'binary'` saves the adjacency matrices in the numpy binary format with a *.npy* extension. `'pajek'` saves the networks into text files with the Pajek format and extension *.net*.

RUNNING THE SCRIPT
After personalizing the script and saving the changes, open a terminal window and move to the folder containing the script, e.g.:

```
$ cd ~/GraphAnalysisLibrary/HelperScripts
```

then type the following command:

```
$ python GenerateRewiredNetworks.py
```

The program will feedback the parameters and save the generated files in the specified path.

2.3.3) Generating ensembles of scale-free networks

The script *GenerateSFNetworks.py* is meant to easily generate ensembles of scale-free networks. After setting of some parameters to personalize the results, the script generates the networks and saves them into files for later use. The main choices are:

NETWORK PARAMETERS
- `N` : Size of the network (number of nodes).
- `dens` : The density of links of the network generated.
- `gamma` : The exponent of the tail, p(k) ~ k^(-gamma)
- `directed` : True to generate directed networks, False for undirected. In case of directed networks, the input and output degrees are correlated, i.e., the input hubs are also the output hubs.

OUTPUT PARAMETERS
- `outpath` : Path where the resulting networks will be saved.
- `filename` : Basename for the output files. Final name will be *basename_reX.dat* where X is the realization number.
- `filetype` : Choice to save networks. `'text'` saves the adjacency matrices as text files with a *.dat* extension. `'binary'` saves the adjacency matrices in the numpy binary format with a *.npy* extension. `'pajek'` saves the networks into text files with the Pajek format and extension *.net*.

RUNNING THE SCRIPT
After personalizing the network and output options in the script, open a terminal window and move to the folder containing the script, e.g. by typing:

```
$ cd somepath/GAlib/HelperScripts
```

where *somepath* is the path in which GAlib has been copied to. Then enter the following command:

```
$ python GenerateRandomNetworks.py
```

The program will feedback the parameters and save the generated files in the specified path.

3) LIBRARY REFERENCE

(Section to be improved…)

GAlib consists of three modules: *gatools.py*, *galib.py* and *gamodels.py*.
- *gatools.py* : miscelaneous helper functions, e.g. data-type transformations.
- *galib.py* : basic graph descriptors, e.g. degrees and clustering.
- *galib_models.py* : Generation of synthetic networks and randomization.

3.1) GRAPH DESCRIPTORS (galib.py)

The module *galib.py* contains the core functions to perform graph analysis.

3.1.1) Basic connectivity descriptors

Degree(net[, directed])	Computes the number of neighbours of every node.
Intensity(net[, directed])	Computes the total strength of a node in a weighted network.
Reciprocity(net[, weighted])	Computes the fraction of reciprocal links to total number of links.
ReciprocalDegree(net[, normed])	Returns the reciprocal degree and excess degrees of every nodes.
AvNeighboursDegree(net[, knntype, …])	Average neighbours' degree of nodes with given degree k, for all k.
Clustering(net)	Returns the clustering coefficient and the local clustering of every node.
RichClub(net[, weighted, rctype])	Computes the density of subnetworks with degree > k', for k' = 0 to kmax.
MatchingIndex(net[, normed])	Computes the number of common neighbours of every pair of nodes.

3.1.2) Paths and graph distance

FloydWarshall(net)	Returns the pathlength between all pairs of nodes in a NxN matrix.
PathsAllinOne(net)	Returns pathlength and betweenness. Finds all shortest paths and cycles.
AllShortestPaths(net, start, end, length)	Finds all the shortest paths between two nodes.

3.1.3) Components, modules, higher-order organization

AssortativityMatrix(net, partition[, norm, …])	Returns the assortativity matrix of network given a partition of nodes.
ConnectedComponents(distnet)	Finds all the connected components in a network out of a distance matrix.
Modularity(net, partition[, degree])	Computes the Newman modularity given a partition of nodes.
K_Core(net, kmin)	Finds the K-core of a network with degree k >= kmin.
K_Shells(net)	Returns the K-shells of a network for all k from kmin to kmax.

3.1.4) Roles of nodes in network with modular organization

ParticipationMatrix(net, partition)	Given a partition of the network, it returns the participation matrix.
ParticipationIndex(participmatrix, partition)	Returns the participation index of all nodes given a partition.
ParticipationIndex_GA(participmatrix)	Returns the participation index as defined by Guimera & Amaral.
LocalHubness_GA(participmatrix)	Returns the within-module degree defined by Guimera & Amaral.

3.2) NETWORK GENERATION MODELS AND NETWORK RANDOMIZATION

3.2.1) Random network generators

Lattice1D(N,z)	Returns a ring lattice where each node connects its 2z closest neighbours.
WattsStrogatzGraph(N, z, prew[, lattice])	Returns a small-world network as in the Watts & Strogatz model.
ErdosRenyiGraph(N, p[, directed])	Returns a random graphs following the Erdos & Renyi model.
RandomGraph(N, L[, directed])	Returns a random graph with N nodes and L links.
BarabasiAlbertGraph(N, m0, m[,outdtype])	Returns a scale-free network after the Barabasi & Albert model.
ScaleFreeGraph(N, density, exponent[, …])	Returns a scale-free graph of given size and exponent.

3.2.2) Network rewiring and randomization algorithms

RewiredNetwork(net, prewire[, directed, …])	Returns a network with links rewired with same degrees as net.

3.2.3) Hierarchical and modular (HM) network models

HMpartition(hmshape)	Returns a partition of nodes for a hierarchical/modular random network.
HMRandomNetwork(hmshape, avklist[,…])	Returns a random hierarchical/modular network of given shape.
RavaszBarabasiModel(xxxxxx)	Returns a hierarchical/modular network of the Ravasz & Barabási model.

3.3) EXTRA TOOLS FOR THE GRAPH ANALYSIS LIBRARY (gatools.py)

The module GAtools.py contains functions that are helpful in the analysis of graphs although they are not graph measures.

3.3.1) I/O and data conversions

LoadLabels(filepath)	Reads the labels of nodes from a text file.
SaveLabels(filepath, labels)	Saves the labels of nodes into a text file.
ReadPartition(filepath)	Reads a partition of nodes from a text file.
SavePartition(filepath, partition)	Saves a partition of nodes into a text file.
LoadFromPajek(filepath[, getlabels)	Reads a network from a text file with Pajek format.
Save2Pajek(filepath, net[, labels, directed])	Saves a network into a text file with Pajek format.
ExtractSubmatrix(net, nodelist1[, nodelist2])	Returns the sub-matrix composed by a set of nodes.
SymmetriseMatrix(net)	Converts a directed network into undirected by averaging the weights.
LaplacianMatrix(net)	Returns the Laplacian matrix of a given network.
CleanPaths(pathlist)	Finds and removes in-place repeated, opposite paths from a list of paths.

3.3.2) Array and matrix comparisons

ArrayCompare(array1, array2)	Compares whether two arrays are identical or not.
HammingDistance(array1, array2[, normed])	Computes the Hamming distance between two arrays of same shape.

3.3.3) Additional math / combinatorics functions

NonZeroMin(data)	Returns the smallest non-zero value in an array.
CumulativeDistribution(data, nbins[, …])	Computes the cumulative distribution of a dataset.
Factorial(x)	Computes the factorial of an integer number.
BinomialCoefficient(n,m)	Computes the binomial coefficient of n over m.
StdDeviation(data)	Returns the mean value and standard deviation of a dataset.
Quartiles(data)	Finds the 1st, 2nd and 3th quartiles of a dataset.
AllPermutations(data)	Given a set, it returns all possible permutations.
AllCominations(data)	Given a set, finds all combinations of given size.
AllBipartitions(data)	Given a set, finds all its possible bipartitions.

4) ACKNOWLEDGMENTS

This library has been developed along several years as a by-product of my scientific work. Therefore my main gratitude is for all the anonymous european and german tax-payers who have contributed to the financial support for my work. In return, GAlib is free for everyone to use, modify and extend after the terms of the included license. I am also grateful to those institutions and organizations that have distributed the financial support in form of research grants: The Helmholtz Institute for Supercomputational Physics, University of Potsdam, Bundesministerium für Bildung und Forschung (Germany), Deutsche Forschungemeinschaft, Bernstein Center for Computational Neuroscience – Berlin, Humboldt-Universität zu Berlin and the 7th Framework Programme for Research of the European Commission.

I also owe special gratitude to the developers of IPython and Enthought for simplifying my life so much and providing the tools I work with every single day. Without you doing research would have only been half the fun.

APPENDIX I: ON CODE STYLE AND PEP-8

I have tried to follow all recommendations in PEP-8 (Style Guide for Python Code) with the exception of the rules for naming functions. In GAlib I use *CamelCase* for the names of the functions for several reasons. Contrary to the main philosophy behind the style guides in PEP-8, that code is more often read than written, I spend 85% of my time writing code (or working interactively) and only 20% reading my code. I very much disagree with the two main recommendations in PEP-8 to name functions:

- names of functions should be lowercase.
- functions should have short names.

First, naming functions in lowercase is to give functions the same hierarchical importance of variables and, in my opinion, this is not acceptable. Second, when giving functions very short names there is a very strong risk to accidentally overwrite functions while writing new code or working interactively in the shell. This happens to me very often when working with NumPy, whose names for functions follow very close to Matlab's functions names. Short function names could be useful when working interactively but given that any proper interactive shell has tab-completion capacities, this is not so important. As an example, NumPy's functions to declare an array as a matrix are mat() and matrix(), two names one would always like to have free to name their variables. This forces you to declare variable in silly and complicated ways as for example:

```
somematrix = mat(somedata)
```
or
```
somematrix = matrix(somedata)
```

with the very important difference that the variable `somematrix` has been declared to be used later on and the function `mat()` will not be used again. Therefore I would prefer that those functions are named `toMatrix()` or `Matrix()`, then I could just declare:

```
mat = Matrix(somedata)
```

so that the variable `mat`, which will be used often in the script has a shorter name.

In summary, naming functions with short names in lowercase does not save typing because it forces you to give variables longer names to distinguish them from the functions, with the difference that variables are more often used in the code while functions are called just once or twice in a script.

APPENDIX II: TRICKS FOR PYTHON AND NUMPY TO HELP IMPROVE PERFORMANCE

Despite that one of the rules of the Zen of Python claims that "There should be one – and preferably only one – obvious way to do it", the ecosystem composed by the standard library of Python, NumPy and SciPy offer many different ways to solve one problem. This flexibility is positive because it enhances the chance to find an optimal solution to problems but the trouble comes with finding which of those options is the optimal. Hence, this Appendix aims to be a listing of tricks and of comparative examples testing the efficiency of the different solutions to a problem. These should also serve as guidelines for efficient coding during the extension of GAlib with new functions.

(To be continued …)

1) Python loops are too slow. For every loop in your code, look if there is any numpy function, method or array manipulation functionality with which the loop(s) can be replaced.

2) Referencing ndarrays as a[i,j] is much faster than array[i][j].