**Guide to using AdjacencyMatrix in MATLAB**

**Adam Bloniarz September, 2010**

AdjacencyMatrix is a MATLAB class which encapsulates several functions related to analyzing adjacency matrices of neural networks. These functions include:

* Loading and manipulation of adjacency matrix
  + Load matrix from .csv (comma-separated-values) file
  + View individual synaptic weights
  + Add together adjacency matrices (ex. elec + chem), and multiply adjacency matrices by scalar values
  + View interactions among a subset of neurons
  + Combine neurons into groups
  + Exporting adjacency matrices to .csv file
* Computation of graph theoretical properties
  + Diameter
  + Clustering coefficient
  + Characteristic path length
  + Betweenness centrality of nodes
* Analysis of community structure
  + Modularity Maximization algorithms – Newman and Leicht spectral method, and simulated annealing method
  + Exporting matrix to .csv file with cells ordered by module membership
  + Analysis of link communities
  + Analysis of ‘functional cartography’ – within module connectivity vs. between module connectivity of a neuron
* Analysis of motifs
  + Count doublets and triplets and test for statistical significance
* Analysis of degree distribution
  + Compute neighbor and weighted in-, out-, and total degree distributions
  + Fit distributions to exponential decay curve and power-law decay curve
  + Test goodness of fit and statistical significance of fit
* Access to lists of neuron types through static variables
  + Lists of male sensory, inter, motor neurons and muscles
  + Lists of hermaphrodite sensory, inter, and motor neurons
  + Lists of male neuron classes

In addition, there are two companion classes to AdjacencyMatrix, called SimilarityMatrix and MatrixComparison, which are used to compare nodes *within* an AdjacencyMatrix, and nodes *between* two AdjacencyMatrix objects, respectively. These classes are built to handle a variety of similarity algorithms, which can be set with parameters. These classes are described in a separate document.

NOTE: in order to use the AdjacencyMatrix class, you should have a familiarity with how cell arrays work in MATLAB.

1. **Loading and manipulation of an adjacency matrix**
   1. **Loading an adjacency matrix**

All the functions of AdjacencyMatrix are performed on an AdjacencyMatrix object. Before attempting to create an AdjacencyMatrix object, make sure to add the AdjacencyMatrix class folder to the MATLAB classpath. This can be done by right-clicking the ‘adjacencymatrix’ folder in the MATLAB file browser, and selecting ‘Add to path’ … ‘Selected folders and subfolders’.

* + 1. **Loading from a file**

An AdjacencyMatrix object can be created from a .csv (comma-separated-values) file with the following MATLAB command:

> adjMat = AdjacencyMatrix(‘file’, ‘N2Y\_chem\_release2.csv’)

* + 1. **Loading from a MATLAB array**

An AdjacencyMatrix object can be created directly from a MATLAB array using the following command:

> adjMat = AdjacencyMatrix(matrix,rowLabels,columnLabels)

‘matrix’ should be a MATLAB array

‘columnLabels’ should be a cell array of strings giving the column labels for ‘matrix’

‘rowLabels’ should be a cell array of strings giving the row labels for ‘matrix’

NOTE: When creating an AdjacencyMatrix object, all empty rows and columns are automatically removed.

* 1. **Viewing an adjacency matrix**
     1. **Viewing individual entries**

An AdjacencyMatrix object can be subsetted like a normal MATLAB array. To view a specific entry in the AdjacencyMatrix object, enter the following command, for example:

> adjMat('LUAL','LUAR')

* + 1. **Viewing the total input or output of a neuron**

To view a list of postsynaptic partners and weights, enter the following command:

> adjMat('LUAL',:)

To view a list of presynaptic partners and weights, enter the following command:

> adjMat(:,'LUAL')

Note: these commands will also return a struct array consisting of the fields ‘target’ and ‘weight’, containing the information displayed.

* + 1. **Viewing all connections in an AdjacencyMatrix object**

To view all connections in the adjacency matrix object, enter the following command:

> adjMat(:,:)

* + 1. **Retreiving a MATLAB matrix from an AdjacencyMatrix**

There are three ways to get a MATLAB matrix from an AdjacencyMatrix object. You can either retrieve a square matrix with rows and columns for all cells in the AdjacencyMatrix, a rectangular matrix with empty rows and columns removed, or a matrix with rows and columns for only specified neurons.

* + - 1. **Retrieving a square matrix, with row/column labels**

To get a square matrix and labels from an AdjacencyMatrix object, enter the following command:

>[A labels] = getSquareMatrix(adjMat);

A will be an NxN matrix, where N is the number of cells (nodes) in the AdjacencyMatrix, and labels will be a cell array of strings giving the column/row labels.

* + - 1. **Retrieving a rectangular matrix, with empty row/columns erased**

To get a rectangular matrix where all rows and columns are guaranteed to have at least one non-zero entry, enter the following command:

>[A rowLabels columnLabels] = getMatrix(adjMat);

‘rowLabels’ and ‘columnLabels’ will be cell arrays of strings giving the row labels and column labels for matrix A. Warning: ‘rowLabels’ and ‘columnLabels’ may not be in the same order, so diagonal elements of A are not necessarily the self-interactions.

* + - 1. **Retrieving a matrix representing only selected interactions**

If you are interested in getting a portion of the AdjacencyMatrix object representing only a subset of cells, use the ‘getSubMatrix’ command. The general syntax of ‘getSubMatrix’ is:

> [adjMat2 A]= getSubMatrix(chem, *rowSet*, *columnSet*)

where ‘rowSet’ and ‘columnSet’ are cell array of strings giving the row and column labels for the cells you are interested in. ‘adjMat2’ will be an AdjacencyMatrix object. ‘A’ will be a MATLAB array corresponding to adjMat2, with rows and columns in the same order as that given in rowSet and columnSet.

Thus for example to see the output of ray neurons onto interneurons, enter the following command:

> [raysWithInter A]= getSubMatrix(chem, rays, interneurons);

Where ‘rays’ is a cell array of strings with the names of the ray neurons, and interneurons is a cell array of strings with the names of the interneurons (this can be formed with the static variable ‘AdjacencyMatrix.inter\_male’).

NOTE: This is a handy method if you would like to retrieve the matrix with the rows and columns in a specific order.

You can also remove a set of neurons, using the parameter ‘remove’:

>[adjMat2 A] = getSubMatrix(chem, *set*, ‘remove’);

for example, replacing *set* with {‘R3BR’} removes R3BR from both axes.

* 1. **Manipulating an adjacency matrix**

While you cannot alter individual entries in an AdjacencyMatrix object, you can add AdjacencyMatrix objects together, multiply them by scalar values, threshold them (erasing all entries below or above a certain value), and group together cells (nodes).

* + 1. **Adding and subtracting adjacency matrices**

The AdjacencyMatrix class overloads the ‘+’ and ‘-‘ operators, so to add or subtract AdjacencyMatrix objects, simply enter the command:

> combined = elec + chem

Where ‘elec’ and ‘chem’ are AdjacencyMatrix objects. The resulting matrix will add entries for corresponding cells in ‘elec’ and ‘chem’. The cells in ‘comb’ will be the union of the set of cells in ‘elec’ and ‘chem’.

Entering the following command will result in an error:

> chem2 = chem – chem;

This is because an AdjacencyMatrix object cannot be created from an empty matrix.

* + 1. **Multiplying adjacency matrices by a scalar**

The AdjacencyMatrix class overloads the ‘.\*’ operator, which is the MATLAB scalar multiplication operator. So to multiply every entry in an AdjacencyMatrix by a scalar, enter the command:

> adjMat2 = .5 .\* adjMat

Thus, for example, to form a matrix which combines chemical and electrical matrices, weighted by ½, enter the following command:

> combined = chem +.5 .\* elec

* + 1. **Thresholding adjacency matrices**

Thresholding an AdjacencyMatrix object removes all entries in the matrix weaker than a certain level. To form a thresholded matrix, enter the following command:

> chem2 = threshold(chem,5)

‘chem2’ will contain no entries < 5.

You can also threshold all entries above a certain value, with the optional parameter ‘ceiling’:

> chem2 = threshold(chem,5,'ceiling')

* + 1. **Grouping together nodes in an AdjacencyMatrix**

Use the command ‘combineCells’ to group together cells (nodes) in the AdjacencyMatrix. You can group things separately in the rows and columns. The general syntax of ‘combineCells’ is:

> groupedAdjMat = combineCells(adjMat, *rowGroups*, *columnGroups*)

rowGroups should be a cell array of strings, alternating group names with a cell array of that group’s contents. For example:

>groupLUA = {‘LUA’,{‘LUAL’,’LUAR’}};

would form a grouping which puts together the L/R homologs for LUA. To get a group with multiple different cells grouped, use, e.g.:

groupLR={‘AWA’,{‘AWAL’,’AWAR’},’AWB’,{‘AWBL’,’AWBR’},’AWC’,{‘AWCL’,’AWCR’}, and so forth…}

If either ‘rowGroups’ or ‘columnGroups’ is an empty cell array {}, then there will be no groups formed in the rows or columns respectively. Thus, for example, to see the output of all neurons onto sensory,inter,motor, and muscles, enter the following command:

>groupedByClass = combineCells(adjMat,{},{‘sensory’, ...

sensoryGroup,’inter’,interGroup,’motor’,motorGroup, ...

muscle,’muscleGroup’);

Where ‘sensoryGroup’ is a cell array of strings which give the labels for sensory neurons, ‘interGroup’ is the same for interneurons, etc.

* 1. **Outputting an adjacency matrix to a .csv file**

While the AdjacencyMatrix class does not have a method to directly output the object to a file, it does have a static method which takes a MATLAB array, row labels, and column labels as arguments, and then makes a .csv file. Given a MATLAB array A, with cell arrays rowLabels and columnLabels, you can output a file with the following command:

> AdjacencyMatrix.writeToFile(‘*filename*.csv',A,rowLabels,columnLabels);

You can use the AdjacencyMatrix methods ‘getMatrix’ or ‘getSquareMatrix’ to get the matrix A, the row labels, and column labels.

1. **Computing graph-theoretic properties**

The AdjacencyMatrix class has a few methods to compute basic graph-theoretic properties of a network. These are outlined below.

* 1. **Clustering Coefficient**

The clustering coefficient of an unweighted, undirected graph gives the probability that two nodes that share a common neighbor will themselves be connected by an edge. Random graphs have a low clustering coefficient. To extend the definition of clustering coefficient to weighted networks, I use the definition found in ‘Clustering in complex directed networks’, Fagiolo, Physical Review E, 2007. To calculate the clustering coefficient of a graph described by and AdjacencyMatrix object, enter the command:

> CC = clusteringCoefficient(adjMat)

This will, by default, form an unweighted network from adjMat by counting all edges with >5 weight. You can manually set the threshold for creating an edge by entering the parameter ‘threshold’:

> CC = clusteringCoefficient(adjMat,’threshold’,5)

To perform the *weighted* calculation outlined in Fagiolo, enter the parameter ‘weighted’

> CC = clusteringCoefficient(adjMat,’weighted’)

To perform a test of statistical significance, enter the parameter ‘pTest’. This will generate 200 random graphs (you can change N by changing the variable ‘nIters’ in the code for function) with the same degree distribution of the given graph, compute the clustering coefficient for each of those graphs, and then determine the p value for the given graph (probability that a random graph will have a clustering coefficient greater than that of the given graph). WARNING: this takes a long time. To perform this p-test, enter the following command:

> CC = clusteringCoefficient(adjMat,’pTest’)

This will print out results of the p-test.

* 1. **Characteristic path length**

The characteristic path length of a network gives the mean shortest path between two nodes of a network. To calculate the characteristic path length of a graph described by an AdjacencyMatrix object, enter the following command:

> C = charPathLength(adjMat)

In calculating the path length, edge weights are ignored. To filter out weak edges, set the optional parameter ‘threshold’:

> C = charPathLength(adjMat,’threshold’,5);

The default setting is threshold = 5.

To perform a test of statistical significance, enter the parameter ‘pTest’:

> C = charPathLength(adjMat,’pTest’);

* 1. **Betweenness Centrality**

The betweenness centrality of a node is the number of shortest paths in the network that pass through that node. To calculate the betweenness centrality for every node in the network, enter the following command:

> [BCvals labels] = betweennessCentrality(adjMat);

‘BCvals’ will be a vector of betweenness-centrality values corresponding to the nodes in ‘labels’.

In calculating the shortest paths, the distance between nodes is set to be the inverse of the weight between them (if weight = 2, then distance = .5) .

To get the betweenness centrality for an individual neuron (LUAL for example), enter the command:

> BCval = betweennessCentrality(adjMat,’LUAL’)

This betweenness centrality routine does not allow for thresholding.

1. **Analysis of community structure**

The AdjacencyMatrix class implements a few algorithms to analyze the community structure of the network.

* 1. **Modularity maximization algorithms**

AdjacencyMatrix implements two modularity maximization algorithms: the Newman and Leicht spectral method, and the simulated annealing algorithm of Guimera and Amaral.

* + 1. **Modularity Maximization by Newman and Leicht algorithm**

The AdjacencyMatrix class implements the modularity maximization algorithm of Leicht and Newman, as outlined in ‘Community structure in directed networks’, Physical Review Letters, 2008. This algorithm computes the leading eigenvector of the modularity matrix. To find the communities that maximize modularity, enter the following command:

> [commLabels Q reducedMat] = getCommunitiesMM(adjMat);

Output:

‘commLabels’ – This will be a cell array of cell arrays of strings giving the members in each community.

‘Q’ – This is the modularity for this community division

‘reducedMat’ – This will be an AdjacencyMatrix object, formed by putting together all community members into meta-nodes, so you can see the interactions between communities as a whole.

NOTE: This method will also produce a dendrogram showing the hierarchy of divisions that occur while running the Newman Leicht algorithm. The leaves in the dendrogram represent the final communities. To prevent this, and to suppress the text output of this method, enter the parameter ‘quiet’:

> [commLabels Q reducedMat] = getCommunitiesMM(adjMat,’quiet’)

To perform a statistical test for significance of this modularity value, enter the parameter ‘pTest’:

> [commLabels Q reducedMat] = getCommunitiesMM(adjMat,’pTest’)

This will create 1000 randomized matrices, compute the maximum modularity for these matrices, and show the p-value for the given matrix ‘adjMat’

* + 1. **Modularity maximization by Simulated Annealing**

The AdjacencyMatrix class implements a simulated annealing algorithm for modularity maximization, as described in ‘Functional cartography of complex metabolic networks’, Guimera and Nunes Amaral, Nature, Feb 2005. To find the communities that maximize modularity by simulated annealing, enter the command:

> [commLabels Q reducedMat] = getCommunitiesMMSA(adjMat);

The output will be the same as that outlined above for getCommunitiesMM.

To change the initial temperature and cooling schedule in the simulated annealing, edit the variables ‘T’,’f’,’relax’, and ‘nSteps’ in the file ‘modularity\_dir\_SA.m’

WARNING: This function takes several hours for large networks. You should set this up to run overnight.

* 1. **Iterative community detection**

Because modularity maximization has a resolution limit, submodules can be found using iterative methods. AdjacencyMatrix implements an iterative community detection algorithm similar to that outlined in J Ruan, W Zhang ‘Identifying network communities with a high resolution’, Physical Review E 2008. This method first computes the modules using Newman and Leicht, and then finds submodules of these modules, testing for statistical significance. To run this algorithm, enter the following command:

> matrices = getCommunitiesHMM(adjMat)

Output:

‘matrices’ will be a cell array whose structure represents the community structure of the network. The hierarchy of labels represents the hierarchy of the community structure.

* 1. **Exporting .csv file ordered by modules**

AdjacencyMatrix allows you to export the matrix to a .csv file, with rows and columns ordered by community membership. This file does not show what the actual communities are (you will have to do this manually in Excel by adding lines to separate communities). This spreadsheet will reflect the community structure – ie. most nonzero entries will be clustered around the diagonal. To export an AdjacencyMatrix object in this way, enter the command:

> makeCommunitiesCSV(adjMat,‘*filename*.csv’,commLabels);

The argument ‘commLabels’ should be a cell array of cell arrays of strings giving the community membership. This can be the output of the functions getCommunitiesMM, or getCommunitiesHMM or getCommunitiesMMSA.

* 1. **Link communities**

Communities in networks can be determined by looking at edges rather than nodes; this way, nodes can belong to multiple communities. To find link communities, AdjacencyMatrix implements the algorithm outlined in Ahn, Bagrow, Lehmann, ‘Link communities reveal multiscale complexity in networks’, Nature Letters, 2010. To find the link communities of an AdjacencyMatrix object, enter the command:

> [commLabelsLinks commLabelsNodes] = getLinkCommunities(adjMat);

Output:

‘commLabelsLinks’ will be a cell array of cell arrays of strings. Each element of the first dimension of the cell array gives the links that are a member of that community.

‘commLabelsNodes’ will be a cell array of cell arrays of strings. This shows the nodes that are members of each community.

To view this output in a more readable form, use the function ‘dispLinkCommunities’:

>dispLinkCommunities(commLabelsLinks)

Or

>dispLinkCommunities(commLabelsNodes)

This will display a list of the non-trivial (having more than 2 members) communities

* 1. **Functional cartography**

Determining the modular structure of a network allows you to analyze the so-called ‘functional cartography’ of the network. This describes the degree to which nodes are connected within their module, versus their connectivity between modules. AdjacencyMatrix implements the algorithm described in Guimera and Amaral, ‘Functional cartography of complex networks’, Nature, Feb. 2005. To find the functional cartography of a network, enter the command:

> [M labels] = functionalCartography(chem);

Output;  
‘M’ – this will be an N x 2 matrix, where N is the number of nodes in the network. The first column of M will be the within-module Z-score, and the second column will be the participation coefficient.

‘labels’ – this gives the list of neurons that ‘M’ refers to.

This method will also produce a plot of the functional cartography, and will show the various categories of nodes as outlined in the paper.

1. **Analysis of motifs**

AdjacencyMatrix has methods to count doublets and triplets in the network, and test for statistical significance. This procedure is outlined in Reigl, Alon, and Chklovskii, ‘Search for computational modules in the C. elegans brain’, BMC biology 2004.

* 1. **Doublet Motifs**

To count doublet motifs, enter the command:

> [doubletCounts doubletCountsRand] = doubletFrequency(adjMat);

Output:

‘doubletCounts’ will be a 1 x 3 vector of counts for the individual doublet motifs. The motifs are as follows:



1 2 3

‘doubletCountsRand’ will be a 1000 x 3 vector of counts for the randomized matrix.

This will also create a plot of the results. This plot gives the ratio of a motif count to the expected value (the average counts of the random matrices). This plots the points for the randomized matrices as red crosses, and for the given matrix as a blue square.

Optional parameters:

‘threshold’ – in calculating the motifs, the algorithm must convert a weighted matrix into a binary matrix. By default, this method only counts edges with weight >= 5. Use the threshold parameter to set the level at which an edge is counted:

> [doubletCounts doubletCountsRand] = ...

doubletFrequency(adjMat,’threshold’,10);

‘nIters’ – This parameter manually allows you to set the number of random matrices to create when performing the statistical test. The default value is 1000. For example to change nIters to 200, enter the command:

> [doubletCounts doubletCountsRand] = ...

doubletFrequency(adjMat,’nIters’,200);

* 1. **Triplet motifs**

Triplet motifs are calculated in the same way as doublet motifs, except that in creating the randomized matrices, doublet counts must be preserved.

To count triplet motifs, enter the command:

[tripletCounts tripletCountsRand] = tripletFrequency(adjMat);

Output:

Similar to doubletFrequency, this method will produce vectors giving the counts for the given matrix and for the random matrices. There are sixteen triplets, so tripletCounts will be a 1x16 vector. The triplets are:

Similar to doubletFrequency, this function will create a plot of the results (read above in section 4.1 about how to interpret this plot).

This function also takes the same parameters as doubletFrequency (‘nIters’ and ‘threshold’).

* 1. **Single step min P procedure**

To perform the single step min-P significance test as outlined in the Alon, Chklovskii, and Reigl paper, there is a companion function ‘singleStepMinP’. The output data from ‘tripletFrequency’ or ‘doubletFrequency’ can be directly used in this function. Thus, after running tripletFrequency, you could simply enter the command:

> Ps = singleStepMinP(tripletCounts,tripletCountsRand);

Output:

‘Ps’ will be a 1 x 16 vector (or 1 x 3 in the case of doublets) giving the P value for each type of motif. Low p-values indicate that the motif is overrepresented.

1. **Analysis of degree distributions**

AdjacencyMatrix implements several methods to analyze the degree distribution of a network. These functions allow you to plot the distributions, and to fit an exponential or power-law curve to these distributions. These methods also allow you to distinguish between in and out degrees, as well as weighted and unweighted (neighbor) degrees.

NOTE: To use some of the curve fitting functions, you must add the folder ‘Clauset,Newman,Shalizi - Power law fit’ to the MATLAB classpath. Right click on the folder, and then select ‘Add to path’ ... ‘Selected folders and subfolders’.

* 1. Plotting the degree distribution

To plot the degree distribution of an AdjacencyMatrix object as a bar graph, simply enter the command:

> plotDegree(adjMat)

There are a number of optional parameters:

* ‘degreeType’ – should be ‘in’,’out’, or ‘total’ (default). This will determine whether the in-degree distribution, out-degree distribution, or total degree (in + out) distribution (default) will be plotted. Ex:

> plotDegree(adjMat,’degreeType’,’in’)

* ‘weighted’ – should be true (default) or false. This will determine whether to plot the weighted or unweighted degree distribution. For example, to calculate the neighbor distribution, enter:

> plotDegree(adjMat,’weighted’,false);

* ‘threshold’ – if plotting a neighbor distribution, you can change the threshold to determine which edges are counted. Thus, if you only want to count edges with weight >= 5, for example, enter the command:

> plotDegree(adjMat,’weighted’,false,’threshold’,5)

* ‘nBins’ – when plotting the distribution, you can alter the number of bins to use in the histogram. The default is to simply have a bar for every possible value. To use only 10 bins, for example, enter the command:

> plotDegree(adjMat,’nBins’,10);

* 1. **Fitting a curve to the degree distribution**

AdjacencyMatrix implements methods to fit an exponential curve or a power-law curve to the degree distributions. These curve fitting functions follow the method described in the paper of Clauset, Shalizi, Newman, ‘Power-law distributions in empirical data’, SIAM Review 2009. This involves using a most-likelihood-estimator to determine the coefficient of fitting (alpha or lambda), and using the Kolmogorov-Smirnov goodness of fit statistic to determine the appropriate lower bound for the fit (xmin).

* + 1. **Fit exponential curve:**

To fit an exponential curve to the degree distribution and plot the results, enter the following command:

> [lambda p gof] = expFitDeg(adjMat);

Output:

‘lambda’ - the lambda value for the exponential fit

‘p’ - the p-value for this fit (probability that a randomly generated data set using this value of lambda will yield a goodness of fit statistic worse than that calculated from the given network)

‘gof’ – the Kolmogorov statistic for this fit

This function will also display the statistics related to this fit. The value of ‘xmin’ displayed gives the lower cut-off point for the fit.

This function takes the same parameters as ‘plotDegree’, described above. There is one additional parameter, ‘pTest’, which you can set to false to prevent the test for the p-value (this test is very computationally intensive). Thus, to avoid the p-test, enter the command:

> lambda = expFitDeg(adjMat,’pTest’,false);

* + 1. **Fit power-law curve:**

The function to fit the power-law curve, plFitDeg, is essentially identical to expFitDeg:

> [alpha p gof] = plFitDeg(adjMat);

It takes the same parameters, and will similarly produce a plot of the results.

1. **Access to lists of neuron types**

The AdjacencyMatrix class stores lists of neuron types in static variables, defined in the class definition file (AdjacencyMatrix.m). These lists allow you to quickly access groups of neurons. They are stored as cell arrays of strings. Access the lists with the following commands:

>AdjacencyMatrix.sensory\_male

>AdjacencyMatrix.inter\_male

>AdjacencyMatrix.motor\_male

>AdjacencyMatrix.muscles\_male

>AdjacencyMatrix.sensory\_herm

>AdjacencyMatrix.inter\_herm

>AdjacencyMatrix.motor\_herm

>AdjacencyMatrix.class5

>AdjacencyMatrix.class4

>AdjacencyMatrix.class2

>AdjacencyMatrix.class1