This guide has been written for Maybrain version 2, released in June 2014. For further information, or to contribute to the project, please contact the authors or see the Maybrain website at XXX.

Chapter 1

Getting Started

1.1 Installation

Maybrain is written in the Python programming language. To run Maybrain you will need a Python2.7 installtion and several other packages on which parts of the code depend. The following are required for analysis:

- Numpy 1.8.1 `http://www.numpy.org/`
- NetworkX 1.8.1, `http://networkx.github.io/`

The following is required for plotting functions:

- Mayavi2, as part of Enthought Tool Suite 4.4.2 `http://code.enthought.com/projects/mayavi/`

The following provides some extra functionality for input of certain data types:

- NiBabel 1.3.0, `http://nipy.org/nibabel/`

Software versions higher than those given may also work, updates will be given on the project website. If you are not familiar with Python it is recommended that you install a pre-packaged version that will include most of the above, for example pythonxy (`http://code.google.com/p/pythonxy/`) or Enthought Canopy (`https://www.enthought.com/downloads/`). Installation for each package can be found on the individual websites.

1.1.1 Installation of Maybrain

Source code can be downloaded from the project website, www.code.google.XXX. Place all the files in one subfolder either in the working folder of the script that calls maybrain or in the site-packages folder of your Python insatllation. The latter allows any script to call the package. If you only want to use the GUI, you can install in any folder and double-click the maybrainGUI.py file to execute.

1.1.2 Development

The authors are happy for developers to extend, customize, improve and debug our code. You can create a clone from the online project in the normal way using git. If you would like your changes to be incorporated into the master branch, please get in touch. We will create an innovative and meaningful hall of fame for anyone contributing a bug-fix and promise to buy you a beer (or acceptable non-alcoholic alternative) when we meet.

1.2 Basic principles

The purpose of Maybrain is to allow easy visualisation of brain connectome and related data and perform various analyses. It is coded in Python, an open source and free programming language. We have attempted to create a pythonic, object oriented, user-friendly program. The code is built around two principal classes, brainObj and plotObj. The first contains all the information about the brain and numerous functions to change, measure and highlight those. At its heart is a networkx object, brainObj.G, via which all networkx functions are available. Some have also been incorporated directly into our code for convenience.

brainObj contains all the information about the brain and numerous functions to change, measure and highlight those. At its heart is a networkx object, brainObj.G, via which all networkx functions are available. Some have also been incorporated directly into our code for convenience.

The plotObj class is a wrapper to enable easy plotting using Mayavi2. It can plot several brain objects and subsets of them (highlights) simultaneously and change basic plot properties. Other Mayavi commands can be acessed via maybrain.XXX. Plots are shown in the mayavi GUI for easy access to a whole range of other mayavi functions.

1.3 Data Input

Several types of data can be input. The basic connectome is made up of two files: a coordinate file and an adjacency matrix. In fact only the second of these is strictly required.

The coordinate file defines the position of each node. It is a text file where each line has four entries: the node index, x, y and z coordinates. e.g.

```
1   0.0   3.1   4.4
2   5.3   7.6   8.4
3   3.2   4.4   3.1
```

The adjacency matrix defines the strength of connection between each pair of nodes. For n nodes it is an $n \times n$ text matrix. Nodes in maybrain are labelled 1,2,... and the order of the rows and columns in the adjacency

matrix is assumed to correspond to this. Entering an adjacency matrix with the wrong dimensions will lead to certain doom.

To load these basic files a few shortcut recipes are available. They will load the files and possibly perform a couple of other functions:

`loadFiles(adjname, coordname)` Loads the adjacency and coordinate files given in the arguments.

`loadAndThreshold(adjname, coordname, threshold)` Loads the given adjacency and coordinate files and applies the (absolute) threshold given.

`loadAndPlot(adjname, coordname, threshold, opacity = 1.0)` Loads the files, applies the given (absolute) threshold and plot the connectome with the option to add an opacity value.

Other data that can be input include the following:
backgrounds
isosurface

1.3.1 Properties and highlights

Frequently you may wish to add additional data to a brain network or highlight a sub-network. Maybrain provides a facility for these. You can add properties to nodes or edges from a text file. The format should be as follows:

```
colour
1 red
3 blue
5 red
6 blue
1 5 red
2 4 green
```

Note that the first line contains the property name. Subsequent lines refer to edges if they contain 3 terms and nodes if they contain 2. The above will give node 1 the property 'colour' with value 'red' and the edge connecting nodes 1 and 5 (only if it exists CHECK) the same property and value. The properties are stored in the networkx objects. An example of the code needed to apply the properties is:

```
brain.importProperties(propertyFilename)
```

Properties can be filtered using highlights. A highlight is simply a list of nodes or edges. They can be generated, for example, as follows:

```
highlightFromConds('colour', 'eq', 'red', mode = 'node', colour = (1.,0.,0.))
```

Here, the term is property name, the second is the relation between the property and value and the third is the value (or range of values) the highlight will cover. The relations can be one of the following, the first few of which are based on the equivalent LaTeXmarkup:

'eq' equal to, value is a single number/string.

'gt' greater than, value is a single number

'lt' less than, value is a single number

'leq' less than or equal to, value is a single number

'geq' greater than or equal to, value is a single number

'in()' in the exclusive interval, value is two numbers

'in[]' in the inclusive interval, value is two numbers. Note that in this case the end-points are considered part of the interval, unlike the previous case. You can also mix brackets, so '(]' '[)' are also acceptable.

'contains' the property of the node contains the given string, applies to strings only.

1.4 Changing the Network

Once the data have been input, you can modify things in a number of ways, outlined in this section.

1.4.1 Rethresholding the adjacency matrix

```
applyThreshold(edgePC = None, totalEdges = None, tVal = -1.1)
```

Use this to apply a new threshold to the edge weights that are defined as linked. The options are: only take weights above a certain value (define `tVal`), have a percentage of connectivity (define `edgePC`) or give a number of edges that should be connected (define `totalEdges`).

1.4.2 Modifying a single data point

Do this by accessing the node properties via the NetworkX graph (`brainObj.G`) and modifying the desired property. See the NetworkX documentation for how to do this.

1.4.3 Degenerate

```
degenerate(weightloss=0.1, edgesRemovedLimit=1, threshLimit=None, pcLimit=None,
weightLossLimit=None, toxicNodes=None, riskEdges=None, spread=False,
updateAdjmat=True, distances=False, spatialSearch=False)
```

Simulation of a degenerating brain. Edges are randomly removed from connections of a set of nodes denoted `toxicNodes`, or from a set of edges denoted `riskEdges`. This occurs either until a set number of edges (`edgesRemovedLimit`) is reached or until the weight loss limit has been reached (for a weighted graph). The limit can also be determined by a given weight threshold (`threshLimit`) or percentage connectivity (`pclimit`). For a binary graph, weight loss should left at the default value of 1.

- The spread option recruits connected nodes of degenerating edges to the toxic nodes list.
- By default this function will enact a random attack model, with a weight loss of 0.1 each iteration.
- Weights are taken as absolute values, so the weight in any affected edge is set to 0.
- Spread can either be False, or a number specifying the weight above which to add nodes to the list of at-risk nodes.
- The `distances` option records the edge length of edges lost.
- The adjacency matrix can optionally be updated to record lost edges by setting `updateAdjmat` to be True (the default option).

1.4.4 Contiguous spread degeneration

```
contiguousspread(edgeloss, startNodes = None)
```

Degenerate nodes in a continuous fashion. The starting node is chosen at random, or from a set `startnodes`. The maximum number of nodes lost is `edgeloss`. It returns a list of the nodes degenerated and a list containing the degenerated nodes at each step, e.g.

```
a, b = brain.contiguousspread(5)
print(a)
    [5,3,7,6,2]
print(b)
    [[5],[5,3],[5,3,7],[5,3,7,6],[5,3,7,6,2]]
```

1.4.5 NNG

`NNG` text ???

1.5 Metrics

Access is provided to a bewildering number of metrics via links to the brain connectivity toolbox (WEBLINK) and XXX. Some measures are also natively included.

1.5.1 The Brain Connectivity Toolbox

How?

```
makebctmat
assignbctmatResult
```

1.5.2 Minimum spanning edges/tree

```
minimum_spanning_edges(weight='weight')
minimum_spanning_tree(
```

This code (like the following description) is based on that written by David Eppstein 2006 (`http://www.ics.uci.edu/~eppstein/PADS/`). It generates edges in a minimum spanning forest of an undirected weighted graph. A minimum spanning tree is a subgraph of the graph (a tree) with the minimum sum of edge weights. A spanning forest is a union of the spanning trees for each connected component of the graph.

The input parameter is:

- weight : a string, the edge data key to use for weight.

The edges version returns

- edges : an iterator type object, a generator that produces edges in the minimum spanning tree. The edges are triples (u,v,w) where w is the weight.

The tree incarnation returns:

- G : A NetworkX Graph, a minimum spanning tree or forest.

The code uses Kruskal's algorithm.

If the graph edges do not have a weight attribute, a default weight of 1 will be used.

1.5.3 Other ways to modify (explanations coming soon...)

```
hubidentifier
pseudohubIdentifier
modularity
robustness
```

1.6 Graphical user interface

A limited funtionality is available via the GUI. This can be a good way to dive in and test out the program. The GUI can log commands that can be used in a Python script, see the logging tab once the GUI is running.

To run the GUI, click on the maybrainGUI.py file. The opening view is for inputting data.