Introduction

This is a brief tutorial on the MATLAB toolbox described in the following article:

Lopes-dos-Santos V, Ribeiro S, Tort ABL (2013) *Detecting cell assemblies in large neuronal populations*, Journal of Neuroscience Methods, accepted for publication.

The toolbox is available upon request from vitor@neuro.ufrn.br (Vítor). Please send any suggestions, comments or criticisms regarding this tutorial or our paper. We really appreciate your feedback to help improving our work.

Below we briefly describe all inputs and outputs of the two main functions of the toolbox: `assembly_patterns` and `assembly_activity`. Next we provide some examples on how to use them.

`Patterns = assembly_patterns(Activitymatrix,opts)`: extracts assembly patterns from the spike matrix.

Description of inputs:

`Activitymatrix`: spike matrix. Rows represent neurons, columns represent time bins. Thus, each element of the matrix carries the spike count of a given neuron at a given bin.

`opts`: set parameters. All fields described below.

`opts.threshold.method`: defines the method to compute the threshold for assembly detection. Options are:

`'MarcenkoPastur'`: uses the analytical bound.

`'binshuffling'`: estimate eigenvalue distribution for independent activity from surrogate matrices generated by shuffling time bins.

`'circularshift'`: estimate eigenvalue distribution for independent activity from surrogate matrices generated by random circular shifts of original spike matrix.

`opts.threshold.permutations_percentile`: defines which percentile of the surrogate distribution of maximal eigenvalues is used as statistical threshold. It must be a number between 0 and 100 (95 or larger recommended). Not used when `'MarcenkoPastur'` is chosen.

`opts.threshold.number_of_permutations`: defines how many surrogate matrices are generated (100 or more recommended). Not used when `'MarcenkoPastur'` is chosen.

`opts.Patterns.method`: defines which method is used to extract assembly patterns. Options are: `'PCA'` or `'ICA'` (recommended).

`opts.Patterns.number_of_iterations`: number of iterations for fastICA algorithm (100 or more recommended). Not used when `'PCA'` is chosen.

Description of outputs:

`Patterns`: assembly patterns. Columns denote assembly # and rows neuron #.

By simply running `Patterns = assembly_patterns(Activitymatrix);` The following default options will be used:
`opts.Patterns.method`: `'ICA'`
`opts.threshold.method`: `'MarcenkoPastur'`
`opts.Patterns.number_of_iterations`: 500.

`Activities = assembly_activity(Patterns,Activitymatrix)`: computes the time course of the activity of assembly patterns defined in `Patterns` in the spike matrix `Activitymatrix` with single bin resolution.

Description of inputs:

`Activitymatrix`: spike matrix. Rows represent neurons, columns represent time bins.

`Patterns`: assembly patterns. Columns denote assembly # and rows neuron #.

Description of output:

`Activities`: Time course of the activity of assemblies. Rows represent assemblies and columns represent time bins.

This tutorial is divided in three parts:

1. **Simulation of spike matrices**, where we briefly describe how you can generate very simple simulated data for testing the codes;
2. **Extraction of assembly patterns**, where we present some examples of how to use our codes to extract assembly patterns from spike matrices; and
3. **Computing activity of detected assemblies**, where we show how to compute the time course of the activity of assemblies detected in the former step.

Before starting make sure you have the following files in your MATLAB path:

```
toy_simulation.m
assembly_patterns.m
assembly_activity.m
bin_shuffling.m
circular_shift.m
fast_ica.m
```

Of note, here we do not address any mathematical/statistical definition/derivation. For that, please read our paper and the references therein.

Simulating spike matrices

We will use simulated spike matrices in order to illustrate how the codes can be applied to real data.

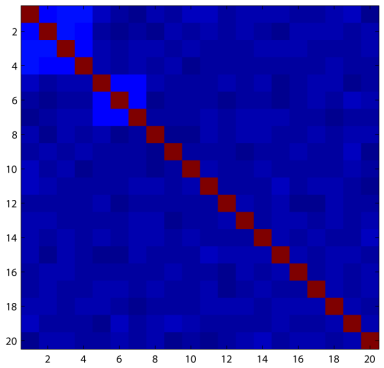
The function `toy_simulation.m` performs simulations by two steps: (1) each neuron is created as an independent Poisson processes (rows), and (2) a certain number of activation bins are randomly chosen for each neuron. In the activation bins the neurons also follow a Poisson distribution but with increased firing rate. In order to introduce correlations among assembly members, neurons in the same assembly have coincident activation bins. The function can be used as shown in the script below:

```
clear all % just clearing workspace

% define mean firing rate
Network_opts.meanspikebin = 1;

% define number of neurons
Network_opts.nneurons = 20 ;

% define number of bins
Network_opts.nbins = 10000;

% above define assembly membership

% line below sets neurons 1,2,3 and 4 to be in assembly 1
Assembly_opts.assembly_neurons{1} = [1 2 3 4];
% line below sets neurons 5,6 and 7 to be in assembly 2
Assembly_opts.assembly_neurons{2} = [5 6 7];

% defines number of activation bins
Assembly_opts.number_of_activations = 300;

% defines mean rate in activation bins
Assembly_opts.meanspikerate_activations = 3;

% running the function
Activitymatrix = toy_simulation(Network_opts,Assembly_opts);
```

The correlation matrix of the network can be visualized by running the code below:

```
correlationmat = corr(Activitymatrix');
figure(1),clf
imagesc(correlationmat)
```

A figure similar to the following one should pop up:



Finding assembly patterns

To extract assembly patterns from the spike matrix you will use the function `assembly_patterns.m` as shown below:

```
Patterns = assembly_patterns(Activitymatrix,opts);
```

The input variable `Activitymatrix` is a matrix in which rows are neurons and columns are time bins, and each element denotes the spike count. **Important**: the number of bins must be larger than the number of neurons.

The input variable `opts` is a structure (struct) that provides parameters to the function.

First, you will need to choose how the statistical threshold for assembly detection is calculated.

Numerical methods generate surrogate matrices and compute the maximum eigenvalue for each of them. The distribution of eigenvalues across the control data is used as null hypothesis distribution. Therefore you have to define (**1**) how to generate surrogate matrices, (**2**) how many surrogate matrices will be generated, and (**3**) which percentile of the null hypothesis distribution will be used as statistical threshold.
Two numerical methods are available in this toolbox:

(1) '**binshuffling**': shuffles the bins of each neuron independently. This method destroys correlation structure among neurons but maintains the distribution of spike counts.

(2) '**circularshift**': performs a circular shift in each neuron independently. This is a more conservative method. Maintains same distribution of spike counts and virtually same autocorrelations.

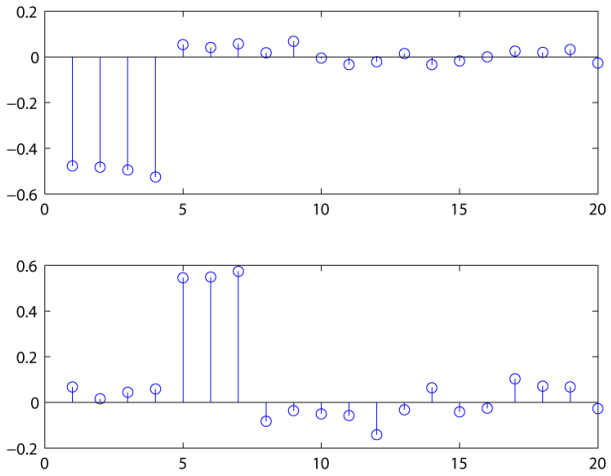
Second, you will need to choose if you want to run PCA (principal component analysis) or ICA (independent component analysis [1]). In our paper we show that ICA provides more robust results. However, we note that PCA can be useful for some applications.

See the code sample for running PCA with circular shift surrogates below:

```
opts.threshold.permutations_percentile = 95;
opts.threshold.number_of_permutations = 20;
opts.threshold.method = 'circularshift';
opts.Patterns.method = 'PCA';
Patterns = assembly_patterns(Activitymatrix,opts);

figure(2),clf
subplot(211)
stem(Patterns(:,1))
subplot(212)
stem(Patterns(:,2))
```

Substitute `'circularshift'` by `'binshuffling'` in order to run the bin shuffling procedure.

If you run the code above for the spike matrix generated in the example in the previous section, figure 2 will show something similar to the following plot:



Note that each assembly pattern represents one of the assemblies programmed in the last section.

In the next example, we use the Marčenko–Pastur distribution as null distribution of eigenvalues [2,3]. The use of the upper bound provided by this analytical distribution as statistical threshold avoids computational costs of surrogate methods and it is virtually equivalent to thresholds calculated numerically [4]. The assembly patterns are extracted by PCA, as the example above. This time, however, we simulate a network with overlapping assemblies (i.e., assemblies share neurons). As reviewed in our work [4], this is one of the cases where PCA cannot isolate assembly patterns.

```
clear all % just clearing workspace

Network_opts.meanspikebin = 1;
Network_opts.nneurons = 20;
Network_opts.nbins = 10000;
```

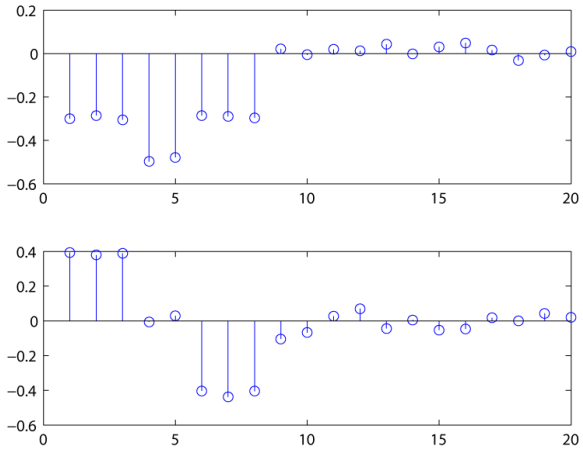
```
Assembly_opts.assembly_neurons{1} = [1 2 3 4 5];
Assembly_opts.assembly_neurons{2} = [4 5 6 7 8];
Assembly_opts.number_of_activations = 300;
Assembly_opts.meanspikerate_activations = 3;
Activitymatrix = toy_simulation(Network_opts,Assembly_opts);

% line below defines analytical threshold for assembly detection.
opts.threshold.method = 'MarcenkoPastur';
opts.Patterns.method = 'PCA';
Patterns = assembly_patterns(Activitymatrix,opts);
```

Note in the example above that you do not need to define the number of permutations and the statistical percentile when the analytical distribution is used.

Run the following code to plot results:

```
figure(2),clf
subplot(211)
stem(Patterns(:,1))
subplot(212)
stem(Patterns(:,2))
```

A figure similar to the one below will pop up:

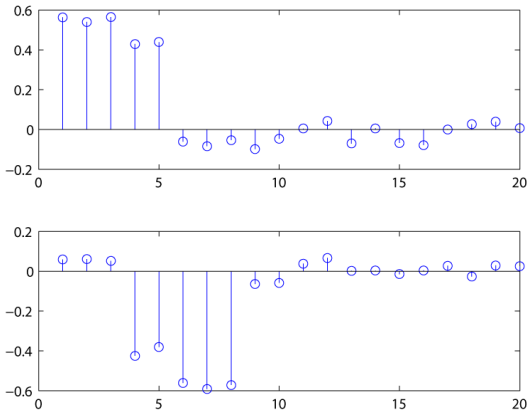


Note that PCA cannot isolate assembly patterns in this example. Fortunately, this limitation can be overcome by ICA, as we show in the next example.

Instead of using PCA for assembly pattern extraction, next we try running the ICA-based method. For that, we use the fastICA [5], which is an iterative algorithm; therefore you will also need to define the number of iterations to be performed, as in the example below.

```
opts.threshold.method = 'MarcenkoPastur';
opts.Patterns.method = 'ICA';
opts.Patterns.number_of_iterations = 200;
Patterns = assembly_patterns(Activitymatrix,opts);
```

Figure shows how the variable `Patterns` should look like when computed as above. Note that ICA-based extraction is able to isolate assembly patterns.



Computing assembly activity

Once you have the assembly patterns you can apply them to the spike matrix for computing the activity of each assembly with single-bin resolution. For that you will use the function `assembly_activity`, as shown below:

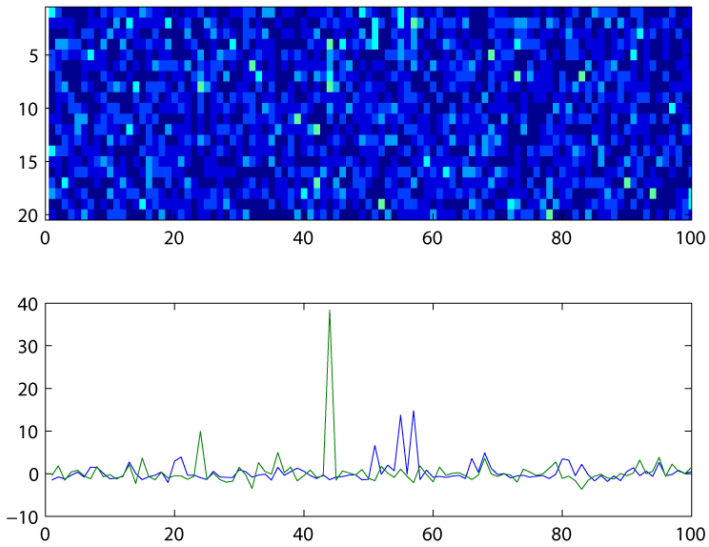
```
Activities = assembly_activity(Patterns,Activitymatrix);
```

The input `Patterns` is the assembly patterns calculated by the function `assembly_patterns`, as shown in the last section. The input `Activitymatrix` is the spike matrix from which you want to calculate the activity of the assemblies.

The output `Activities` is the activity of the assemblies. The rows represent assemblies (in the same order as in the input 'Patterns'), and columns represent time bins (same as in the input `Activitymatrix`).

The code below computes and plots the activity of the cell assemblies defined in the last example of the previous section:

```
Activities = assembly_activity(Patterns,Activitymatrix);

figure(4),clf
subplot(211)
imagesc(Activitymatrix)
xlim([0 100])
subplot(212)
plot(Activities')
xlim([0 100])
```



The upper panel shows an interval of the spike matrix and bottom panel shows the activity of assemblies.

Note in the figure that the green trace at the bottom panel peaks at co-activation of members of assembly 2: neurons 4, 5, 6, 7 and 8; while the blue trace peaks for activations of assembly #1: neurons 1, 2, 3, 4 and 5.

Note that you will not get exactly the same plot as above due to the randomness involved in the simulations.

Note also that you can extract assembly patterns from a spike matrix and compute their activity in other matrices (template-match paradigm). For instance, you may want to extract assembly patterns from a spike matrix recorded during a learning epoch (template) and calculate their activity during pre and post-sleep episodes (matches) [2].

Final comments

We suggest that you play with the codes by changing parameters of the simulations in order to develop a good intuition of the methods before applying them to your data.
This is the first version of the tutorial on detection and extraction of assemblies patterns from large neuronal populations. We will be further developing the method to improve its performance and we will be happy to hear any feedback from you. We hope you have found this tutorial useful.

Vítor Lopes dos Santos
vitor@neuro.ufrn.br
Universidade Federal do Rio Grande do Norte, Brazil

References

1. Hyvarinen A, Oja E (2000) Independent component analysis: algorithms and applications. Neural Networks 13: 411-430.
2. Peyrache A, Khamassi M, Benchenane K, Wiener SI, Battaglia FP (2009) Replay of rule-learning related neural patterns in the prefrontal cortex during sleep. Nature Neuroscience 12: 919-U143.

3. Peyrache A, Benchenane K, Khamassi M, Wiener SI, Battaglia FP (2010) Principal component analysis of ensemble recordings reveals cell assemblies at high temporal resolution. Journal of Computational Neuroscience 29: 309-325.
4. Lopes-dos-Santos V, Conde-Ocazionez S, Nicolelis M, Ribeiro S, Tort A (2011) Neuronal assembly detection and cell membership specification by Principal Component Analysis. Plos One 6: e20996.
5. Hyvarinen A, Oja E (1997) A fast fixed-point algorithm for independent component analysis. Neural Computation 9: 1483-1492.