Software for BMI for Vato et.al (Plos Computational Biology, 2012)

SI-CODE: Towards new Brain-Machine Interfaces:
State-dependent information coding

FET Open – Grant Agreement Number 284553

Authors: Alessasndro Vato and Stefano Panzeri

uzh | eth | zürich

Table of Contents

In this report we describe the Matlab Functions that constitute the bidirectional brain-machine interface algorithms presented in the paper: "Shaping the dynamics of a Bidirectional Neural Interface", PLoS Comput Biol 8(7): e1002578. The following document is divided in three sections: the data structure used by the interface, the calibration procedure algorithms and the simulation procedure algorithms.

Shaping the Dynamics of a Bidirectional Neural Interface: a report on the calibration and simulation algorithms.

1. Data Structure

The multi-channel recorded signals evoked by electrical stimulation are organized into a set of recorded unit responses. Let the following variables be defined.

$N =$ number of recorded neurons
$\mathrm{T} =$ number of time intervals recorded on each neuron
$S =$ number of distinct stimulation patterns
$R =$ number of repetitions of each stimulation pattern

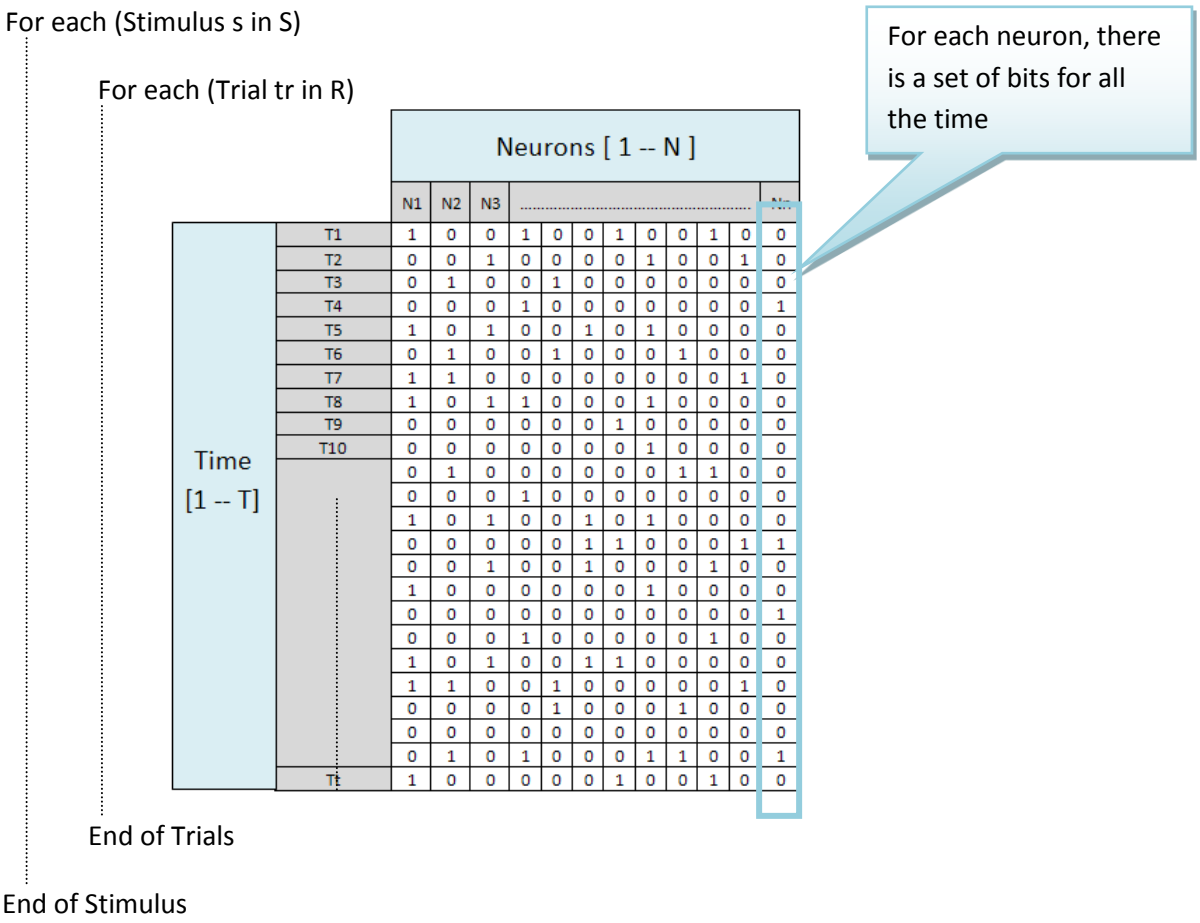
During the calibration each stimulation patterns is repeated R times and, accordingly, $R \times N$ neural responses are recorded. Each response is an array of T values, representing – for example – the number of spikes in each bin.

In the data files stored for the algorithms, there are two arrays:

- a 4-Dimensional array which contains bits (0 | 1) representing neuron signals, here is the structure of this 4-D array:

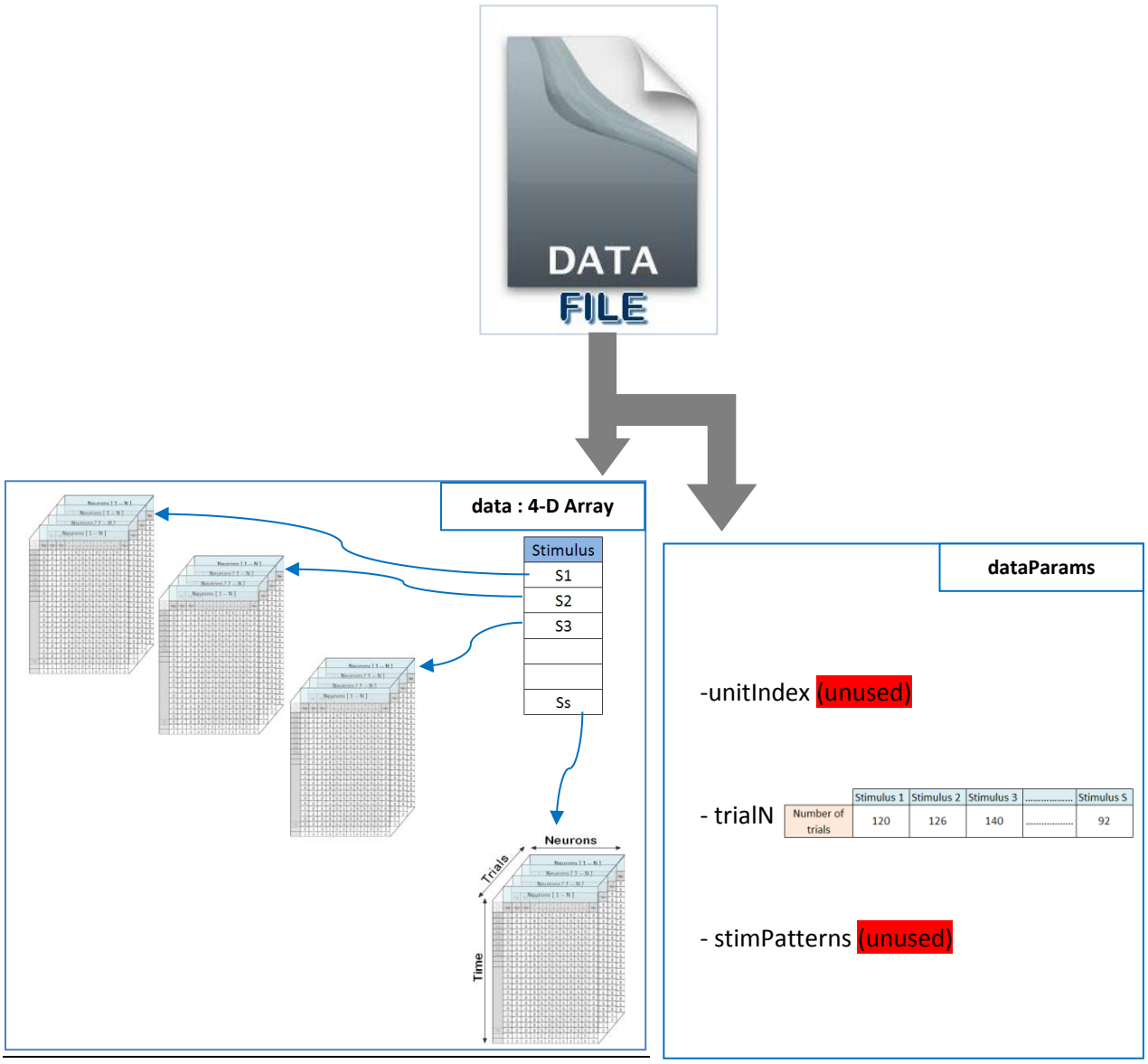
  [T(time) , R (Number of trial for the corresponding stimulus) , N (Neurons) , S (Stimulus)]

  For each stimulus, there are a number of trials which can be flexible according the simulation and for each trial, there is a dataset of bits defining the state (1 or 0) of the neuron signal at specific time t between $[0, T]$.



- a second field named *dataParams*, defined as a structure, containing two or three properties depending to the file:

- *trialN*: It's an single array (1 dimension), which contains the number of trials for each stimulus of the 1st 4-D array (data). The number of trials may be different according to the stimulus, so the size of this array is the number of stimulus. The importance of the *trialN* is to know for each stimulus, how many trials there are going to be.
- *unitIndex*: A 2-D array containing two lines : the 1st line refers to the identifying number of the electrode (*knowing that we obtained these data by first implanting an array of 16 micro-electrodes*), the 2nd line refers to the identifying number of the neuron read for the electrode. The size of the array is $[2 \times N]$ but it won't be used in the algorithms.
- *stimPatterns:* is a two dimensional array with the size of $[S \times S]$, representing a which electrodes are activated during each stimulation pattern. This array won't be used either in the algorithms.


DATA
FILE
data : 4-D Array
Stimulus
S1
S2
S3
Ss
Neurons
Trials
Time
dataParams
-unitIndex (unused)
- trialN
Number of trials
Stimulus 1
Stimulus 2
Stimulus 3
Stimulus S
120
126
140
92
- stimPatterns (unused)

2. Calibration Procedure

2.1 Introduction

During the calibration each stimulation patterns is repeated R times and, accordingly, $R \times N$ neural responses are recorded. Each response is an array of T values, representing – for example – the number of spikes in each bin.

The calibration responses are then represented as $S \times R$ N-dimensional vector functions:

$$\psi_s^r(t) = \begin{bmatrix} \psi_{s,1}^r(t) \\ \psi_{s,2}^r(t) \\ \cdots \\ \psi_{s,N}^r(t) \end{bmatrix} \qquad (s = 1,\ldots,S;\ r = 1,\ldots,R;\ t = 1,\ldots,T) \tag{1}$$

From these calibration responses, we extract S mean responses

$$\varphi_s(t) = \frac{1}{R}\sum_{r=1}^{R} \psi_s^r(t)\,. \tag{2}$$

Following the same convention, a neural response vector is an N-dimensional vector function

$$\mathrm{f}(t) = \begin{bmatrix} f_1(t) \\ f_2(t) \\ \cdots \\ f_N(t) \end{bmatrix} \qquad (t = 1,\ldots,T) \tag{3}$$

The inner product of two neural responses is defined as an extension of the standard Euclidean inner product:

$$\langle \mathrm{f} \,|\, \mathrm{g} \rangle = \sum_{n=1}^{N}\sum_{t=1}^{T} f_n(t)\cdot g_n(t) \tag{4}$$

The S mean calibration responses are considered to be n basis fields and are used to approximate all recorded neural responses. In particular, each calibration response is approximated as a sum of mean responses:

$$\psi_s^r(t) \approx \sum_{i=1}^{S} d_{s,i}^r \cdot \varphi_i(t) \tag{5}$$

We take the inner product of each side of Equation (5) with each basis function, to obtain S vector/matrix equations

$$\Psi_s^r = \Phi \cdot \mathrm{d}_s^r \tag{6}$$

where

$$\begin{aligned} \left[\Psi_s^r\right]_i &= \left\langle \varphi_i \middle| \psi_s^r \right\rangle \\ \left[\Phi\right]_{i,j} &= \left\langle \varphi_i \middle| \varphi_j \right\rangle \\ (i, j = 1, \ldots, S). & \end{aligned} \tag{7}$$

Equation (7) can be solved for d_d^r provided that $\det(\Phi) \neq 0$ (if the projection matrix is singular, one can still use a pseudo-inverse. But this does not seem to be a likely situation).

With this, each calibration response is mapped into a S-dimensional vector

$$\mathrm{d}_s^r = \begin{bmatrix} d_{s,1}^r \\ d_{s,2}^r \\ \cdots \\ d_{s,S}^r \end{bmatrix} \tag{8}$$

Each response corresponds to a d-vector and vice-versa, each d-vector corresponds to a unique approximation of the response (the likelihood that two distinct signals map onto the same d-vector is vanishingly small). Therefore, we take the S-dimensional D-vectors as representations of the neural responses.

PCA is applied to the $S \times R$ S-dimensional calibration vectors, collected in a single data matrix X with $S \times R$ rows and S columns. The two leading principal components are collected in a projection operator

$$\mathrm{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,S} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,S} \end{bmatrix} \tag{9}$$

This operator defines the 2D plane with maximum variance over the set of S stimuli. The calibration responses are then transformed into $S \times R$ points over this plane as:

$$\mathrm{c}_s^r = \mathrm{W} \cdot \mathrm{d}_s^r \tag{10}$$

We apply the two projection matrices, Φ and W, and the mean calibration responses, $\varphi_i(t)$, to the data collected during the experiment to map each response, $\mathrm{f}(t)$ into a corresponding 2-d vector

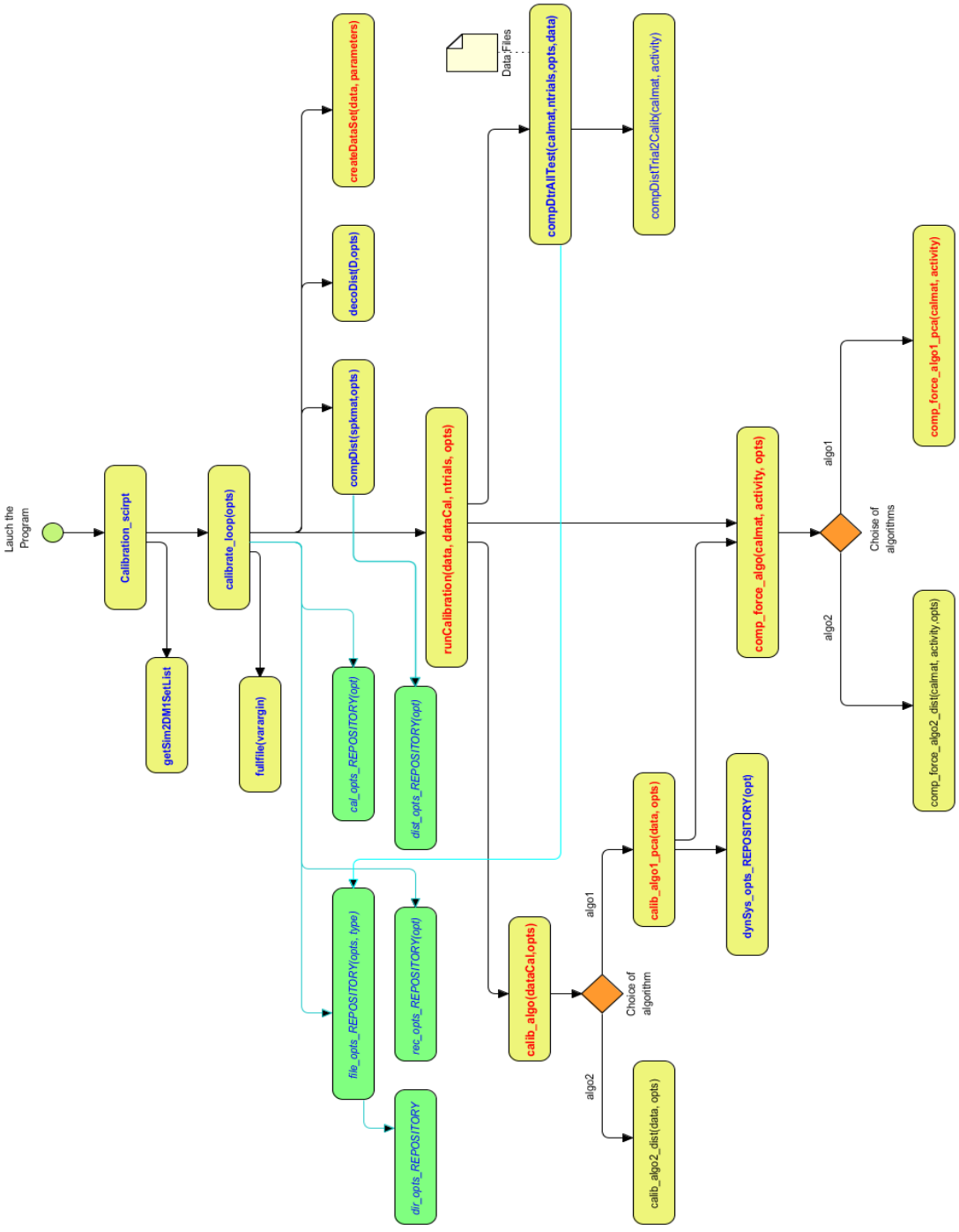
$$\mathrm{c} = \mathrm{W} \cdot \Phi \cdot \mathrm{F} \tag{11}$$

$$\mathrm{F} = \begin{bmatrix} \langle \varphi_1 | \mathrm{f} \rangle \\ \langle \varphi_2 | \mathrm{f} \rangle \\ \cdots \\ \langle \varphi_S | \mathrm{f} \rangle \end{bmatrix} \tag{12}$$

Equations (11) and (12) are calculated in real time.

Dynamic shaping involves mapping the 2D range of the calibration vectors c_s^r over the range of the desired forces and the corresponding states over the stimuli that generated the calibration vectors.

2.2 Scheme of the functions called by the calibration algorithm



2.3 Description of the functions called by the calibration algorithm

Script « calibrate_script »

This script was created in order to set parameters and to call the main functions realized for implementing the algorithm. At the very beginning the script calls the function named « getSim2DSetList » to set the different types of list to be used for the simulation part.

After that, it set several parameters to define the calibration option, recording option, the dynamic systems and the different algorithms to be used in the study. By looping for each setList (defined before) and each algorithm, the function « calibrate_loop(opts) » is called for proceeding to the calibration.

```
for aa=1:length(algos)
    opts.algo = algos{aa};
    for ff=1:length(setList)
        opts.set = setList{ff};
        disp(opts.set);

        for dd=1:length(dynSys),
            tic;
            opts.dynSys = dynSys{dd};

            calibrate_loop(opts);

            toc;
        end;
    end;
end;
```

Function «calibrate_loop(opts)»

This function receives as input the structure (*opts*) representing information about the sampling rate, the parameters of the dynamical system and others. The main task of this function is to set up the data, variables, to process and to reshape the N-Dimensional arrays before calling the real calibration function which is «run_Calibration(data, dataCal, ntrials, opts)»

This function first of all load the information about the data by using «file_opts_REPOSITORY(opts, type)» which creates and returns the path of the data file that is to be opened.

With that operation, two variables are created: *data* and *dataParams* (defined in the data files description)

```
%************************************************************%
[dirName, fileName] = file_opts_REPOSITORY(opts, 'raw mat');
%************************************************************%
load(fullfile(dirName, fileName));  % load data and dataParams;
%************************************************************%
```

The recording parameters are load from the called function «rec_opts_REPOSITORY(opts.rec)» and then the sampling parameters (*BIN, OFFSET, WINDOW*) are set.

```
% -------------------------
% load sampling parameters
% -------------------------
rec_opts = rec_opts_REPOSITORY(opts.rec);
WINDOW = rec_opts.window;
OFFSET = rec_opts.offset;
BIN = rec_opts.bin;
end;
```

After initializing all the parameters, the function define how using the data:

- *normal* : it means that calibration data and simulation are separated in to two 4-D array
- *shuffle* :it means that, other than separate the data , they will also be mixed in order to observe the chance that the calibration might work in that case.

These operation represent the pre-processing procedure to prepare the data for the calibration which is called by: « runCalibration(data, dataCal, ntrials, opts) »

```
  out = runCalibration(data, dataCal, ntrials, opts);
  % store results
  nStim=out.nStim;
  cal.RecPoint(uu,jj,kk,ii,1:nStim,:) = out.RecPoint';
  cal.RecForce(uu,jj,kk,ii,1:nStim,:) = out.RecForce';
  cal.ForceCal(uu,jj,kk,ii,:,1:nStim,:)=out.ForceCal;
  cal.ForceTest(uu,jj,kk,ii,:,1:nStim,:)=out.ForceTest;
```

Finally, the outputs data returned by «runCalibration» are saved on the drive.

```
% save results
[dirName, fileName] = file_opts_REPOSITORY(opts, 'cal');
save(fullfile(dirName, fileName), 'cal');
```

Function « runCalibration(data, dataCal, ntrials, opts)» :

Runs the calibration corresponding to the desired algorithm and derives the forces from the testing set.

Inputs:

- *data* : data loaded from data file
- *dataCal* : calibration data
- *nTrials* : number of trials
- *opts* : other options

Outputs:

- *cal* : this variable contains information about the forces, the number of stimuli, the recording points etc …

The calibration step is launched by calling the algorithm contained in the function named «calib_algo(dataCal, opts)», which returns the recording points useful for defining sensory regions.

After that, the *dataCal* (i.e. calibration data) is updated as well as the field of the variable *cal.*

```
% run calibration
calmat = calib_algo(dataCal,opts);
nStim=calmat.nStim; dataCal=dataCal(:,:,:,1:nStim);
cal.nStim=nStim;
cal.RecForce = calmat.RecForce;
cal.RecPoint = calmat.RecPoint;
```

At this point the forces are computed from the neural activity by initializing all the variables by lunching the function «comp_force_algo(calmat, activity, opts)» which actually does compute the forces for each neural response of each stimulus and returns results that are stored into the structure variable out. The calibration forces are stored into a 3-D array called *ForceCal*.

Finally, we initialize the *ForceTest* array before launching the computation of forces corresponding to data in simulation set by calling the same function called before for the computation of the calibration of forces «comp_force_algo(calmat, activity, opts)».

```
% compute Forces corresponding to data in test set

ForceTest=nan(max(ntrials),nStim,2);

for kk = 1:nStim
    for ii=1:ntrials(kk)
        activity = zeros(nT, nUnit);    % needed in case nT == 1
        activity(:,:) = squeeze(data(:,ii,:,kk));
        if strcmp(opts.algo(1:5),'algo2')
            calmat.Dtr=squeeze(DtrAllTest(kk,ii,:,:));
        end;
        out=comp_force_algo(calmat, activity,opts);
        ForceTest(ii,kk,:) = out.Force;
    end;
end;
```

The results are then stored in the *ForceTest* array.

Function « calib_algo(dataCal,opts) » :

This function uses as input the calibration data (*dataCal*) and the option (*opts*) which actually contains the name of the algorithm that to be used.

From a switch on the name of the algorithm, we call the function «calib_algo1_pca or calib_algo2_dist (dataCal, opts)» and return its result into the output variable *calmat*.

```
function calmat = calib_algo(dataCal,opts)

switch opts.algo(1:5)
    case 'algo1'
        calmat = calib_algo1_pca(dataCal, opts);
    case 'algo2'
        calmat = calib_algo2_dist(dataCal, opts);
    otherwise
        error('algo inexistent');
end;
```

Function « calib_algo1_pca (data, opts) »:

This function computes the matrix necessary to map the recorded activity into the 2-Dimensional force vector.

Inputs:

- *data* (time x trials x nUnits x nStimuli) a matrix containing spike count
- *opts* : contains dynamical system parameters

Outputs:

- *meanData* : mean of training set across trials (PHI)
- *invPHI* : inverse of PHI matrix
- *MeanD* : mean of d-vectors matrix
- *scaleMatrix* : transformation matrix

First of all, we load the dynamical system parameters by calling the function «dynSys_opts_REPOSITORY(opts.dynSys)» using the calibration matrix, the neuronal activity and the options. After that the force range is calculated, it computes PHI and the average response to each stimulation.

```
% calculate force range
[FxR, FyR]= rangeF(Stiffness, Xrange, Yrange);
% -------------------------
% get data dimensions
[nT, nTrial, nUnit, nStim] = size(data);

% compute phi, the average response to each stim
meanData = zeros(nT, nUnit,nStim);  % needed in case nUnit = 1
meanData(:,:,:) = squeeze(nanmean(data,2)); % phi

% Build the matrix PHI = <phi, phi>
PHI = zeros(nStim);
for ii = 1:nStim
    for jj = 1:nStim
         PHI(ii,jj) =
sum(sum(squeeze(meanData(:,:,ii)).*squeeze(meanData(:,:,jj))));
    end
end
invPHI = pinv(PHI);
```

Then we project all the calibration responses and collect all the S-Dimensional d-vectors in a data matrix X and calculate its mean.

Then the principal components analysis process over the matrix X using the function «pc= princomp(zscore(X))'». The two leading components are then collected in a projection operator W. Before mapping the matrix X to 2-D by using the expression: xydata = W*detrend(X,0)', the function detrend(X, 0) removes mean across column (*meanD*) and find the maximum variation along the two dimensions.

```
% Calculate the PCs over X

pc = princomp(zscore(X))';

% The 2 leading PCs are collected in a projection operator W
W = [pc(1,:);pc(2,:)];

% Map to 2D by PCA
xydata = W*detrend(X,0)'; % detrend(X,0) removes mean across column (meanD)
```

Then it comes the computing of the scaling matrix and we derive the net transformation matrix, we then save the results in the structure calmat and compute the force corresponding to each stimulus using the function «comp_force_algo(calmat, activity, opts)» in order to derive the corresponding recPoints obtained.

➢ ***Function « dynSys_opts_REPOSITORY(opt) » :***

Using the option parameter with a switch treatment, this function set the dynamic system parameters :

- mass
- stiffness
- damping
- X-range, Y-range
- Radius, etc …

➢ ***Function « comp_force_algo(calmat, activity, opts) » :***

According the option chosen (algo1 or algo2), launch the required algorithm :

- comp_force_algo1_pca(calmat, activity)
- comp_force_algo2_dist(calmat, activity, opts)
-

```
function out = comp_force_algo(calmat, activity, opts)

switch opts.algo(1:5)
    case 'algo1'
        out=comp_force_algo1_pca(calmat, activity);
    case 'algo2'
        out=comp_force_algo2_dist(calmat, activity, opts);
    otherwise
        error('algo inexistent');
end;
```

➢ ***Function « comp_force_algo1_pca(calmat, activity) » :***

This function computes the force corresponding to a given neural activity by loading transformation matrix out of the calibration procedure and deriving the force from the neural activity using the following formula:

$$F = \begin{bmatrix} \langle \varphi_1 | f \rangle \\ \langle \varphi_2 | f \rangle \\ \cdots \\ \langle \varphi_S | f \rangle \end{bmatrix}$$

Function « dir_opts_REPOSITORY » :

Creates directories for different kind of data, those different directories are about :

- Analysis figures (stationarity, psth, etc)
- calibration figures
- interpolated field figures
- convergence rate and MI figures
- figures for analysis with multi-neuron distances
- figures for analysis with different dynamical systems and simulated neurons
- figures for NIPS: comparison algorithms
- contains raw plx file
- contains raw mat file
- contains result of information analysis between spike trains and stimuli
- contains results of calibration
- contains results of simulation and generated trajectories
- contains result of information analysis between expected and generated force
- contains result of information analysis between expected and generated force
- contains distances between multi-neuron spike trains

Function « rec_opts_REPOSITORY(opt) » :

Sets recording parameters according to the experiment type specified

Input:

- *opt*: specifies experiment type

Output:

- *rec_opts*: recording parameter
    - BIN
    - WINDOW
    - OFFSET

Function « file_opts_REPOSITORY(opts, type) » :

Input parameters:

- *opts* : contains information of the type of data
- *type* : specifies the kind of data

This function is used to create and return from the type of data, the directory corresponding to that type as well as the name of the corresponding file

Function «createDataSet(data, parameters)» :

This function resamples data matrix according to the parameters.

Inputs:

- *data* (time X trials X nUnits X nStimuli) matrix containing spike count
- *parameters* : array of 3 elements
    - MIN : start point of data collection after simulation
    - MAX : end point of data collection after simulation
    - BIN : sampling interval

Outputs:

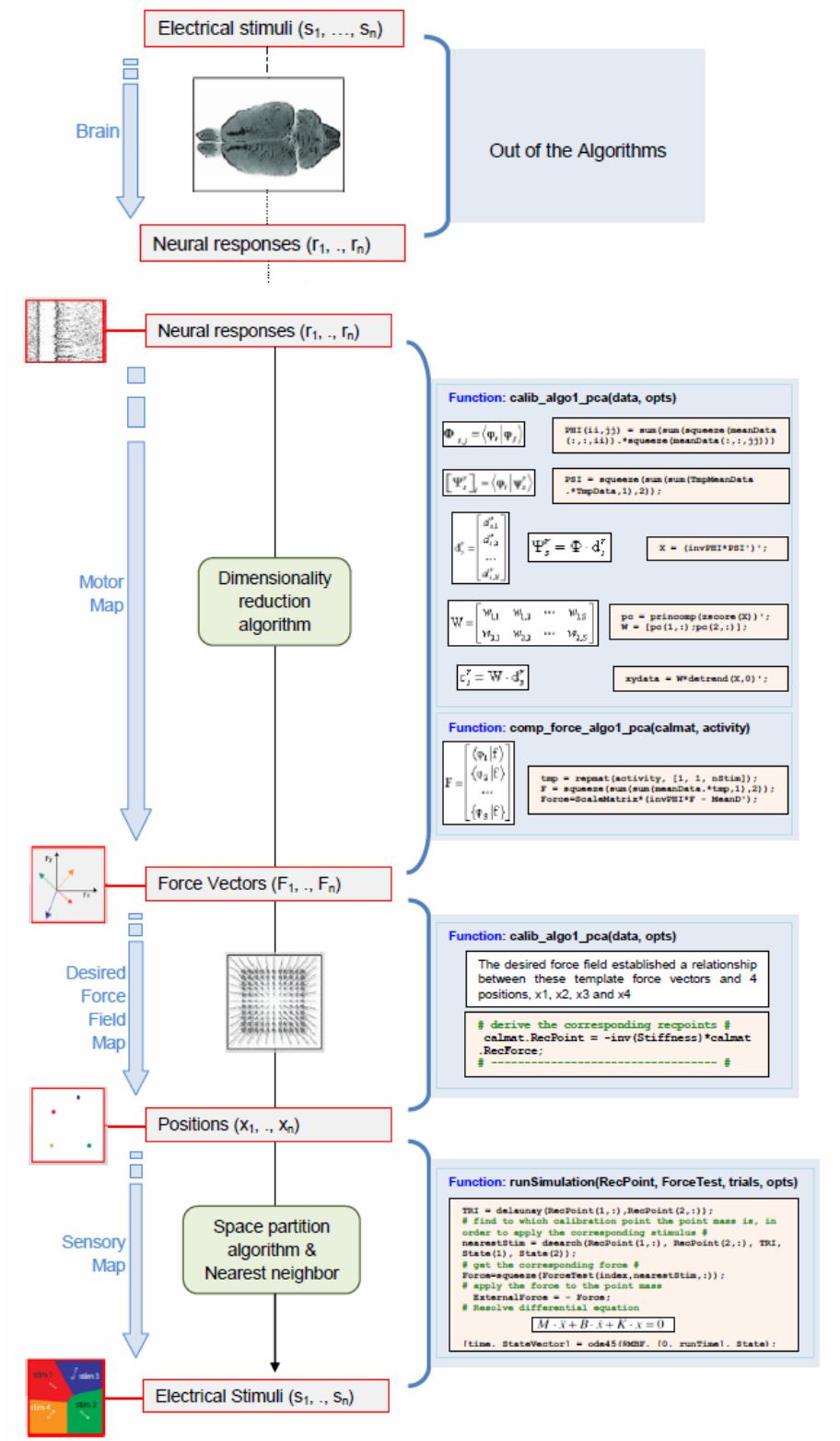
- *newData* (newTime X maxTrials X nUnits X nStimuli) matrix containing resampled spike count

Using the parameters (max, min, BIN, LAG), this function reshape the data loaded from data files and rebuilds it into a N-Dimensional array with different sampling.

```
MIN = parameters(1)+1;
MAX = parameters(2);
BIN = parameters(3);
LAG = [MIN:BIN:MAX]; % new time vector
[nT, maxTrial, nUnit, nStim] = size(data);

tmpData = reshape(data(MIN:MAX,:,:,:),MAX-MIN+1,maxTrial*nUnit*nStim)';
timeMat = repmat(MIN:MAX,maxTrial*nUnit*nStim,1);
tmpData = timeMat.*tmpData;
```

2.4 Overview of the Calibration Procedure


Electrical stimuli (s1, …, sn)
Brain
Out of the Algorithms
Neural responses (r1, ., rn)
Neural responses (r1, ., rn)
Motor Map
Dimensionality reduction algorithm
Function: calib_algo1_pca(data, opts)
Function: comp_force_algo1_pca(calmat, activity)
Force Vectors (F1, ., Fn)
Desired Force Field Map
Function: calib_algo1_pca(data, opts)
The desired force field established a relationship between these template force vectors and 4 positions, x1, x2, x3 and x4
Positions (x1, ., xn)
Sensory Map
Space partition algorithm & Nearest neighbor
Function: runSimulation(RecPoint, ForceTest, trials, opts)
Electrical Stimuli (s1, ., sn)

3. Simulation Procedure

3.1 Introduction

To test the above concepts, we made neural activity interact with a simulated point mass in a viscous medium which moves on a plane. Our desired FF was a linear FF defined by the equation $F = K \cdot x$, whose magnitude depends only on the radial distance x from the origin of the plane. This resulted in the following linear differential equation:

$$M \cdot \ddot{\rho} + B\dot{\rho} + K \cdot \rho = 0$$

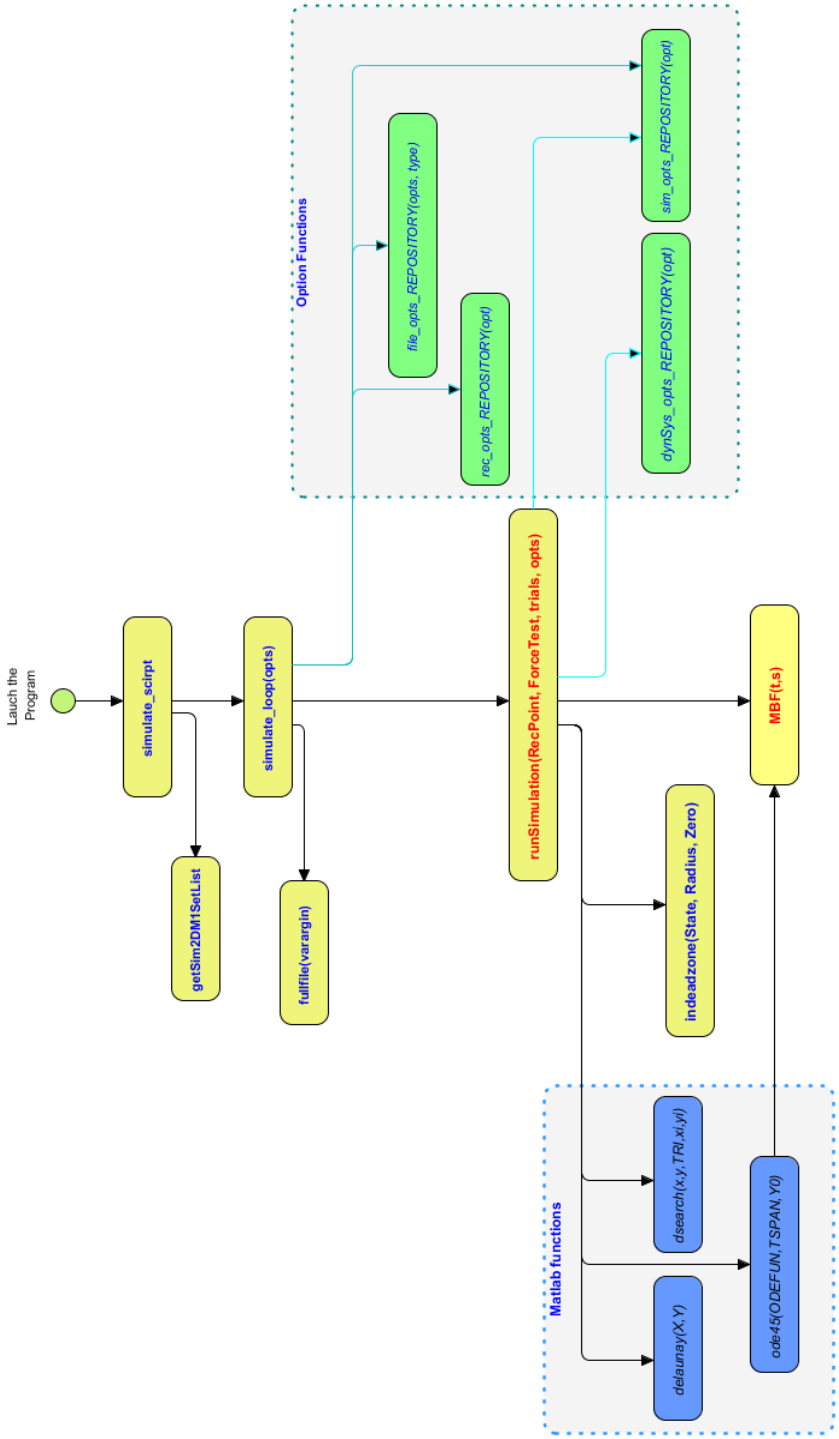
The critical parameter determining the dynamics of this system is the damping ratio, defined as follows:

$$\zeta = B / (2 \cdot \sqrt{K \cdot M})$$

We explored different behaviors: we varied the damping ratio from 1.03 (system almost critically damped) to 2.93 (system over damped). This was achieved by fixing the stiffness K and the mass M to 4 N/m and 10 kg, respectively while varying the viscosity B from 13 to 37 N·s·m-1.

The force decoded by the motor interface from the simulated neural activity was supplied as an input to the dynamical simulation which was integrated for 1 s. The position of the simulated point mass was then retrieved and fed to the sensory interface to determine the next stimulus to be applied.

3.2 Scheme of the functions called by the simulation algorithm



3.3 Description of the functions called by the simulation algorithm

Script «simulate_script»:

This script was created in order to set primary parameters and to call the different main functiosn which have been realized for implementing calibration algorithm.
First of all, in the script, we start by calling the function named « getSim2DM1SetList » which actually set the different types of list used for the simulation part.

After that, we define several parameters as:

- the calibration option
- recording option
- the dynamic systems which might be considered in the study (an array used as an enumeration)
- the different algorithms which might be used (an array used as an enumeration)

And then looping in the dynamic systems for each setList (defined before) and each algorithm, the function « simulate_loop(opts) » is called for proceeding to the calibration.

```
for aa=length(algos)
    opts.algo = algos{aa};
    for ff=[6 7 8 9 11 12]%1:length(setList)
        opts.set = setList{ff};
        disp(opts.set);
        for dd=1%:length(dynSys),
            tic;
            opts.dynSys = dynSys{dd},
            simulate_loop(opts);
            toc;
        end;
    end;
end;
```

Function « simulate_loop(opts) » :

This function receives as input a structure (*opts*) representing options containing information about sampling, dynamic system and others. Most of the treatments of this function is more about setting up data, variables, processing and reshaping N-Dimensional arrays before calling the real calibration function which is « runSimulation (RecPoint, ForceTest, trials, opts) ».

Inputs :
- opts : option needed for the simulation

Outputs :
- State
- Force
- Stimulation

- Converge
- TriFail : a boolean variable which tells whether the triangulation worked or not

First, we load needed data :
- the sampling parameters : using the function « rec_opts_REPOSITORY(opts.rec) »
- the results of the calibration : using the function « file_opts_REPOSITORY(opts, 'cal')»
- the simulation parameters : using the function « sim_opts_REPOSITORY(opts.sim) »

Then we initialize the simulation variables, to the corresponding results of the runSimulation function.

Finally, for each OFFSET of each WINDOW of each BIN of each UNITS, we set the recpoint array and the forces data before running the simulation function named « runSimulation(recpoint, forces, ntrials, opts) » and we save the results in a structure named sim containing the force, the state, the stimulation, the failed trials, etc …

```
for uu=1:length(units)
      for ii = 1:numel(BIN) % choose bin size
        for jj = 1:numel(WINDOW)    % choose window size
            for kk = 1:numel(OFFSET)   % choose offset
                recpoint=double(squeeze(RecPoint(uu,jj,kk,ii,:,:))');
                forces=squeeze(ForceTest(uu,jj,kk,ii,:,:,:));
                Sim = runSimulation(recpoint, forces,ntrials, opts);
                % store results
                sim.State(uu,jj,kkfor,ii,:,:,:) = Sim.State;
                sim.Force(uu,jj,kk,ii,:,:,:) = Sim.Force;
                sim.Stimulation(uu,jj,kk,ii,:,:) = Sim.Stimulation;
                % sim.Activity(uu,jj,kk,ii,:,:,:,:) = Sim.Activity;
                sim.Converge(uu,jj,kk,ii,:) = Sim.Converge;
                sim.TriFail(uu,jj,kk,ii) = Sim.TriFail;
            end
        end
```

Function « runSimulation(RecPoint, ForceTest, trials, opts) » :

This function is the main function of the simulation, it runs the simulation using of the recpoints, the force parameters, the numbers of trials and the options.

Inputs : Recpoint, ForceTest , Trials , Opts
Outputs : Sim

This function defines 3 global variables which will be used in another functions :
«global ExternalForce Mass Damping»

First of all, we load the dynamical system parameters from the function «dynSys_opts_REPOSITORY(opt)».

After that we load the simulation options as the running time, the total time and the number of runs using the called function « sim_opts_REPOSITORY(opts.sim)».

Then, we set the variables used to run the simulation before launching the triangulation.

```
% number of time steps per run
nTicks = round(totalTime/runTime);

% init matrices
dataState = zeros(nRun, nTicks, 4);
dataForce = zeros(nRun, nTicks, 2);
dataStimulation = zeros(nRun, nTicks);
% dataActivity = zeros(nRun, nTicks, nT, nUnit);
dataConverge = NaN(nRun,1);
TriFail = 0;
```

From a try – catch block, we call the triangulation function on the Recpoint array « Delaunay (RecPoint(1,:), RecPoint(2,:)) » knowing that this function creates a Delaunay triangulation of a set of points in 2-D or 3-D space, a 2-D Delaunay triangulation ensures that the circumcircle associated with each triangle contains no other point in its interior. This definition extends naturally to higher dimensions.

After that, the incoming treatment depend on the fact that the triangulation worked. If this is the case we loop with nRuns times, each time we start by the defining limit of the possible coordinates, before picking one point and checking if we are not too close to the target and if we are, in a loop we keep picking a point randomly until we get an acceptable distance. Then we set that point as the initial state and in another loop corresponding to each tick, we check if the state vector is not in a dead zone using the function « indeadzone(State, Radius, Zero) », if so the second loop stops there.

If the state vector is not in the dead zone, we find the closer calibration point to the point mass in order to apply the corresponding stimulus (this part corresponds to the nearest neighbor algorithm) using the called function « dsearch(RecPoint(1,:), RecPoint(2,:), TRI, State(1), State(2)) », which searches Delaunay triangulation for nearest point, returns the index (x , y) of the nearest point to the point (xi, yi). dsearch requires a triangulation TRI of the points x,y obtained using Delaunay. If xi and yi are vectors, K is a vector of the same size. Knowing that :

- x : RecPoint(1,:)
- y : RecPoint(2,:)
- xi : State(1)
- yi : State(2)

Taking that nearest stimulus, we stimulate and record the neural activity in order to get the corresponding force by squeezing the ForceTest Matrix.

```
% find to which calibration point the point mass is, in
% order to apply the corresponding stimulus
nearestStim = dsearch(RecPoint(1,:), RecPoint(2,:), TRI, State(1), State(2));
% stimulate and record
index = ceil(rand*trials(nearestStim));
% get the corresponding force
Force=squeeze(ForceTest(index, nearestStim,:));
% apply the force to the point mass
 ExternalForce = - Force;
```

Then using the force applied to the point mass, we calculate the new value of the state vector by resolving the non-stiff differential equation, medium order method :

$$M \cdot \ddot{x} + B \cdot \ddot{x} + K \cdot x = 0$$

The function ODE45 called by the expression [time, StateVector] = ode45(@MBF, [0, runTime], State) returns us the state vector containing all the positions of the mass with it speed at each time instant between [0 – runtime], then we take out the position at the last instant we do this for each runTime instant in the variable state, before saving the state, the force and the nearest stimulus in N-D arrays. At the end of the second loop, we check if we have reached the target (state in the dead zone), if so we update the dataConverge matrix.

```
                % WARNING: the time argument isn't used when solving the
                  % differential equation
                [time, StateVector] = ode45(@MBF, [0, runTime], State);
                  State = StateVector(end,:)';

                % record data
                  dataState(ii, jj, :) = State;
                  dataForce(ii, jj, :) = Force;
                  dataStimulation(ii, jj) = nearestStim;
%
            end
     end

    if indeadzone(State, Radius, Zero)
            dataConverge(ii) = jj;
    end
```

Finally, we save all the results in the simulation structure.

Function « indeadzone(State, Radius, Zero) » :

This function determines if the system is in the dead zone using as inputs:
- State: state vector containing X and Y position and velocity
- Radius: radius of dead zone
- Zero: coordinates of dead zone center

Function « MBF(t, s)» :

The function MBF describes the behavior of a point mass under the effect of a force in a viscous medium.

Inputs :

- t: time vector
- s: state vector

Output :
- sp: new state vector

We compute the acceleration using this formula: a = -IM*(Damping*[s(3) s(4)]'+ ExternalForce)
And we store it in the same vector that contains the speed, which actually represents the new state to be returned as result.

Function « rec_opts_REPOSITORY(opt) » :

Sets recording parameters according to the specified experiment type .

Input:
- opt: specifies experiment type

Output:
- rec_opts: recording parameters

These recording parameters are :
- BIN
- WINDOW
- OFFSET

Function « file_opts_REPOSITORY(opts, type) » :

Using as input parameters :

- opts : contains information of the type of data
- type : specifies the kind of data

This function is used to create and to return from the type of data, the directory corresponding to that type as well as the name of the corresponding file

Function « dynSys_opts_REPOSITORY(opt) » :

Using the option parameter with a switch treatment, this function set the dynamic system parameters : mass, stiffness , damping , X-range, Y-range, radius, etc …

Function « sim_opts_REPOSITORY(opt) » :

The function sets the simulation temporal parameters according to the defined option in the variable <<opt>> using a switch block between two possible values : sim1, sim2.

Input :
- opt: specifies experiment type

Output :
- sim_opts: parameters for simulation