CEPAGE: Central Pattern Generator toolbox

Matteo Lodi

April 19, 2017

Contents

Chapter 1

Overview

The Central Pattern Generator toolbox (or CEPAGE for short) is an open source, Matlab-based toolbox for the simulation and analysis of neurons network, with emphasis on Central Pattern Generator.

1.1 Installation

1.1.1 Prerequisites

- MATLAB R2014a or later;
- A supported C and C++ compiler
- boost package (you can download it from `http://www.boost.org/`);

1.1.2 Installation procedure

First install boost c++ libraries.
On Unix-like system:

1. Download and unzip boost libraries
2. Create the folder where you want to install boost (for example */home/boostLibrary*)
3. open a terminal and move to boost root directory
4. run `./bootstrap.sh --prefix=path/to/installation/prefix`, for exmple */home/boostLibrary* if you use as installation directory run `./bootstrap.sh --prefix=/home/boostLibrary`
5. run `./b2 install`

On Windows system:

1. Download and unzip boost libraries
2. Create the folder where you want to install boost (for example */home/boostLibrary*)
3. open a terminal and move to boost root directory
4. run `./bootstrap.sh --prefix=path/to/installation/prefix`, for exmple */home/boostLibrary* if you use as installation directory run `./bootstrap.sh --prefix=/home/boostLibrary`
5. run `./b2 install`

In order to install CEPAGE you need to follow these steps:

1. Download CEPAGE Toolbox at `www.ncas.dibe.unige.it/software/..`;
2. Launch script `installCEPAGE.m` and follow the instructions.

Chapter 2

Classes

CEPAGE Toolbox relies on Object Oriented Programming (OOP) and therefor each "object" is represented through a MATLAB class. Each class contains several properties and methods. The properties are always private, therefore it is not possible to access them from outside the class (i.e., `age = person.age;` or `person.age = 23;`) but it is possible to access them through the public methods (i.e., `age = person.getAge();` or `person = person.setAge(23);`). In this sections all classes and their public methods are detailed. An overview of all CEPAGE classes and their hierarchical relationships is shown in Fig. 2.1.

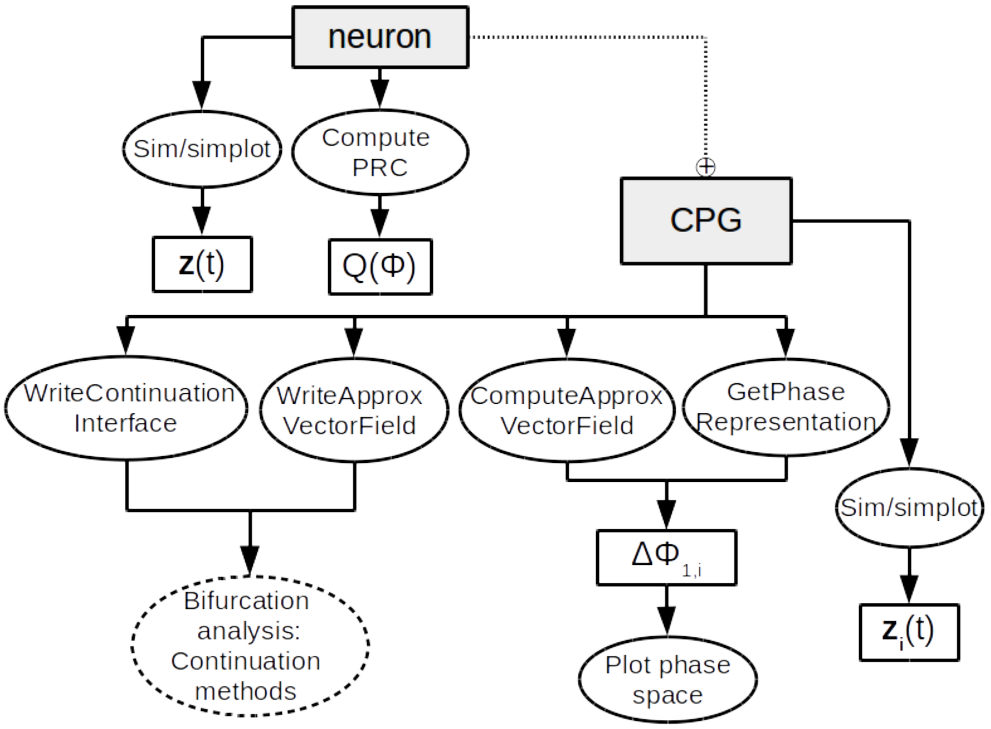

Figure 2.1: Organization of CEPAGE Toolbox classes. Grey blocks correspond to abstract classes, which cannot be istantiated.

NOTE: The methods of a generic class `OBJ` can be accessed equivalently either by passing the class `OBJ` as first argument (e.g. `Y = eval(OBJ,X)`) or with the "dot" operator (e.g. `Y = OBJ.eval(X)`). In this manual the first syntax is shown.

2.1 @neuron_model

This is an abstract class (it cannot be instantiated) which defines a generic neuron model in the form:

$$\dot{x} = f(x, I_{ext}) \tag{2.1}$$

where $x \in \mathbb{R}^{n_x}$ is the set of states that characterized the neuron and I_{ext} is an external current that flow into the neuron.

2.1.1 Methods

computePRC

```
[PRC,limitCycle] = computePRC(object,nPoints,Ttrans,opts);
```
Compute the limit cycle and the phase resetting curve of the neuron using the method described in [1].
`PRC` is a structure with fields phi and PRC representing the phase in which the PRC is computed and its value. `limitCycle` is a structure with fields T and X that represent the limit cycle over which the function compute PRC. `nPoints` is the number of points over which the function eval the PRC, `Ttrans` is the transitory time used before finding a limit cycle and `object` is the neuron_model object. `opts` is a structure with the following fields:

- `Vth`: Membrane potential threshold value used to find neuron limit cycle (Default 0).
- `x0`: initial condition of the neuron used to find limit cycle (Default 0).
- `Tlimit`: maximum time used for finding limit cycle during integration of neuron state equation (Default 10^7).

getnx

```
nx = getnx(OBJ);
```
Gets the number of state variable nx

getStateNames

```
names = getStateNames(OBJ);
```
Gets the mnemonic name of the neuron states

sim

```
[T,X] = sim(object,Tspan,x0,opts);
```
Simulates the neuron model OBJ, starting from initial condition `X0` in the time range `Tspan`. Output `T` is time and output `X` is the state evolution of the neuron. `opts` is a structure with the following fields:

- `integrator`: string indicating the solver used to integrate the differential equations. It can be either 'ode45', 'ode23', 'ode113', 'ode15s', 'ode23s', 'ode23t' , 'ode23tb', 'odeint' or 'eulero. If you choose eulero or odeint the simulation is made in C thorough mex file and a supported mex compiler is required (Default ode45).
- `integrProp`: options to provide to the ODE solver. Type doc odeset to get help. If you choose eulero the variable must have a field dt that describe the integration step size. (Default odeset).

simplot

```
[T,X] = sim(object,Tspan,x0,opts);
```
Performs a simulation by means of method neuron_model/sim and plots the results. The syntax is the same of method neuron_model/sim.

2.2 @HH_model

This class represent a neuron described by the Hodgkin-Huxley model. The class is derived from `neuron_model` and inherit all its function. The state equations are:

$$\begin{cases} C\dot{V} = -I_{Na} - I_{k2} - I_l - I_{app} - I_{ext}; \\ \tau_{Na}\dot{h} = h_\infty - h \\ \tau_{k2}\dot{m} = m_\infty - m \end{cases} \tag{2.2}$$

where

$$\begin{aligned} h_\infty &= \frac{1}{1+e^{500(X+0.333)}} \\ n_\infty &= \frac{1}{1+e^{-150(X+0.305)}} \\ m_\infty &= \frac{1}{1+e^{-83(X+(0.018+V_{shiftK2}))}} \\ n &= n_\infty \\ I_{Na} &= g_{na}n^3h(V-E_{Na}) \\ I_{k2} &= g_{k2}m^2(V-E_k) \\ I_l &= g_l(V-E_l) \end{aligned}$$

The model has three state variable; V represent the membrane potential. g_{na}, E_{Na}, g_{k2}, E_k, g_l, E_l, τ_{Na}, τ_{k2}, C,I_{app} and $V_{shiftK2}$ are parameters that allows to change neuron behaviors.

2.2.1 Methods

constructor

`OBJ = HH_model()`
Builds an HH_model object OBJ with all parameters equal to 0.

`OBJ = HH_model(gna,ENa,gk2,Ek,gl,El,tNa,tk2,C,Iapp,VshiftK2)`
Builds an HH_model object OBJ with user assigned parameters value

disp

`disp(object)`
Displays some information about the HH_model object

generateC

`generateC(OBJ,filename)`
Generates filename.c and filename.h files for the computation of model vector field

`generateC(OBJ,filename,folder)`
Generates filename.c and filename.h files for the computation of model vector field in the directory folder

getJacobian

`J = getJacobian(object,x)`
Compute the Jacobian J of the model in point x

getXdot

```
 xdot = getXdot(object,t,x,Iext)
```
compute the time derivative xdot of the model at time instant t, in state x and with an external current Iext; if Iext is not provided it is set equal to 0.

2.3 @HR_model

This class represent a neuron described by the Hindmarsh-Rose model. The class is derived from `neuron_model` and inherit all its function. The state equations are:

$$\begin{cases} \dot{x} = y - x^3 + bx^2 - z + I - I_{ext} \\ \dot{y} = 1 - 5x^2 y \\ \dot{z} = \mu(s(x - x_{rest}) - z) \end{cases} \tag{2.3}$$

The model has three state variable; the first represent the membrane potential. b,I,μ,x_{rest} and s are parameters that allows to change neuron behaviors.

2.3.1 Methods

constructor

```
OBJ = HR_model()
```
Builds an HR_model object OBJ with all parameters equal to 0.

```
OBJ = HR_model(b,mu,s,I,x_rest)
```
Builds an HR_model object OBJ with user assigned parameters value

disp

```
disp(object)
```
Displays some information about the HR_model object

generateC

```
generateC(OBJ,filename)
```
Generates filename.c and filename.h files for the computation of model vector field

```
generateC(OBJ,filename,folder)
```
Generates filename.c and filename.h files for the computation of model vector field in the directory folder

getJacobian

```
J = getJacobian(object,x)
```
Compute the Jacobian J of the model in point x

getXdot

```
 xdot = getXdot(object,t,x,Iext)
```
compute the time derivative xdot of the model at time instant t, in state x and with an external current Iext; if Iext is not provided it is set equal to 0.

2.4 @FN_relaxation_model

This class represent a neuron described by a variation of the FitzHughNagumo model. The class is derived from `neuron_model` and inherit all its function. The state equations are:

$$\begin{cases} \dot{V} = VV^3x + I - I_{ext} \\ \dot{x} = \varepsilon(X_\infty - X) \end{cases} \tag{2.4}$$

where

$$X_\infty = \frac{1}{1 + e^-10V}$$

The model has two state variable; V represent the membrane potential. ε and I are parameters that allows to change neuron behaviors.

2.4.1 Methods

constructor

`OBJ = FN_relaxation_model()`
Builds an FN_relaxation_model object OBJ with all parameters equal to 0.

`OBJ = FN_relaxation_model(I,eps)`
Builds an FN_relaxation_model object OBJ with user assigned parameters value

disp

`disp(object)`
Displays some information about the FN_relaxation_model object

generateC

`generateC(OBJ,filename)`
Generates filename.c and filename.h files for the computation of model vector field

`generateC(OBJ,filename,folder)`
Generates filename.c and filename.h files for the computation of model vector field in the directory folder

getJacobian

`J = getJacobian(object,x)`
Compute the Jacobian J of the model in point x

getXdot

`xdot = getXdot(object,t,x,Iext)`
compute the time derivative xdot of the model at time instant t, in state x and with an external current Iext; if Iext is not provided it is set equal to 0.

2.5 @ML_model

This class represent a neuron described by a variation of the Morris-Lecar model. The class is derived from `neuron_model` and inherit all its function. The state equations are:

$$\begin{cases} C_M\dot{V} = -g_l(V-Vl) - g_{Ca}M_\infty(V-V_{Ca}) - g_K N(V-Vk_)) + I - I_{ext} \\ \dot{N} = \lambda_N(N_\infty - N) \end{cases} \tag{2.5}$$

where

$$\begin{aligned}
&M_\infty = 0.5(1+tanh((V-V_1)/V_2))\\
&N_\infty = 0.5(1+tanh((V-V_3)/V_4))\\
&\lambda_N = \phi cosh((V-V_3)/(2V_4))\\
&C_M = 5\\
&g_K = 8\\
&g_l = 2\\
&V_{Ca} = 120\\
&V_k = -80\\
&V_l = -60\\
&V_1 = -1.2\\
&V_2 = 18
\end{aligned}$$

The model has two state variable; V represent the membrane potential. and I are parameters that allows to change neuron behaviors.

2.5.1 Methods

constructor

`OBJ = ML_model()`
Builds an ML_model object OBJ with all parameters equal to 0.

`OBJ = ML_model(gCa,V3,V4,phi,I)`
Builds an ML_model object OBJ with user assigned parameters value

disp

`disp(object)`
Displays some information about the ML_model object

generateC

`generateC(OBJ,filename)`
Generates filename.c and filename.h files for the computation of model vector field

`generateC(OBJ,filename,folder)`
Generates filename.c and filename.h files for the computation of model vector field in the directory folder

getJacobian

`J = getJacobian(object,x)`
Compute the Jacobian J of the model in point x

getXdot

```
 xdot = getXdot(object,t,x,Iext)
```
compute the time derivative xdot of the model at time instant t, in state x and with an external current Iext; if Iext is not provided it is set equal to 0.

2.6 @CPG

This class represents a generic CPG model. The network is characterized by a set of states (x). Each neuron i-th of the network is influenced by neuron j-th by mean the synapses current:

$$Isyn = g_{in}^{(i,j)} \frac{V_i - E_{In}}{1 + e^{\nu(V_j - \theta)}} + g_{ex}^{(i,j)} \frac{V_i - E_{ex}}{1 + e^{\nu(V_j - \theta)}} + g_{el}^{(i,j)}(V_i - V_j)$$

where V_i and V_j are the membrane potentials of neurons i and j, g_{in}, g_{ex} and g_{el} are matrices that describe the topology of the network, θ and ν modify synapses activation function and E_{in} and E_{eq} are the inversion potential of the synapsis.

2.6.1 Methods

constructor

```
OBJ = CPG()
```
Builds a CPG object OBJ with all parameters and matrix g_{in}, g_{ex} and g_{el} equal to 0.

```
OBJ = HR_model(g_in,g_ex,g_el,Esyn_In,Esyn_Ex,tetaSyn,uSyn)
```
Builds a CPG object OBJ with user assigned parameters value

disp

```
disp(object)
```
Displays some information about the CPG object

generateC

```
generateC(OBJ,filename)
```
Generates filename.c and filename.h files for the computation of neurons network vector field. The function also generate .c and .h file necessary to compute each neuron vector field.

```
generateC(OBJ,filename,folder)
```
Generates filename.c and filename.h files for the computation of neurons network vector field in the directory folder. The function also generate .c and .h file necessary to compute each neuron vector field.

getXdot

```
 xdot = getXdot(object,t,x)
```
compute the time derivative xdot of the model at time instant t, in state x.

sim

```
[T,X] = sim(object,Tspan,x0,opts)
```
Simulates the neuron model OBJ, starting from initial condition `X0` in the time range `Tspan`. If `X0` have M rows the function make M simulation of the network starting from all the M initial condition. Output `T` is time and output `X` is the state evolution of the neuron. `opts` is a structure with the following fields:

- `integrator`: string indicating the solver used to integrate the differential equations. It can be either 'ode45', 'ode23', 'ode113', 'ode15s', 'ode23s', 'ode23t' , 'ode23tb', 'odeint' or 'eulero. If you choose eulero or odeint the simulation is made in C through mex file and a supported mex compiler is required (Default ode45).
- `integrProp`: options to provide to the ODE solver. Type doc odeset to get help. If you choose eulero the variable must have a field dt that describe the integration step size (Default odeset).

simplot

`[T,X] = sim(object,Tspan,x0,opts)`
Performs a simulation by means of method neuron_model/sim and plots the results. The syntax is the same of method neuron_model/sim.

getCIfromPhi

`CI = getCIfromPhi(object,deltaPhi,opts)`
Get the initial condition of a CPG in which the phase difference between neuron 1 and neuron i is described by vector deltaPhi. The methods use numerical integration and events over membrane potential state variable. In a network of N neurons deltaPhi must be a $M \times (N-1)$ matrix; each row is a desired phase difference of the network. CI is a cell array with M elements; each element is the initial condition of the network that ensure the desired phase difference. A variable opts could be provided; it must be a structure with fields:

- Vth - threshold that define events, it must be between max and minimum value of the membrane potential (default 0)
- Ttrans - Transitory time of the numerical integration (default 100)
- x0 - Numerical integration initial condition of the neuron (default 0)

plot

`plot(object)`
Plot the neurons network topology

plotPhaseSpace

`h = plotPhaseSpace(object,phi)`
Plot the time evolution of the phase difference of the neurons in the network described by variable phi. If network is composed by 3 or 4 neurons the function plot also the phase space evolution of the network. h is the figure(s) handle. phi must be a cell array; each element of phi must be a matrix describing the phase different evolution of the network.

getPhaseRepresentationFromTrack

`phi = getPhaseRepresentationFromTrack(object,T,X,Vth )`
Gets the phase difference between neurons composing the network starting from neurons state evolution. phi is a matrix with $N-1$ columns describing the evolution of the phase difference between neuron 1 and neuron i+1. T and X are the time and state vector that describe the state evolution of the network, Vth is the value of the membrane potential that correspond to an event (it must be between the maximum and the minimum value of the membrane potential), object is the CPG object.

getPhaseRepresentation

`phi = getPhaseRepresentation(object,T,CI,opts)`
Gets the phase evolution of the network. phi is a cell array with N-1 elements describing the evolution of the phase difference between neuron 1 and neuron i+1. The network is simulated for T seconds. CI is the initial conditions of the networks. A structure opts could be provided with the following fields:

- integrator - string indicating the solver used to integrate the differential equations. It can be either 'ode45','ode23', 'ode113','ode15s','ode23s','ode23t' , 'ode23tb', 'odeint' or 'eulero. If you choose eulero or odeint the simulation is made in C through mex file and a supported mex compiler is required (Default ode45).
- integratorOptions - options to provide to the ODE solver. Type doc odeset to get help. If you choose eulero the variable must have a field dt that describe the integration step size (Default odeset).
- Vth - is the value of the membrane potential that correspond to an event (Default 0).

computeApproxVectorialField

`[deltaPhiMatrix, deltaPhiDot] = computeApproxVectorialField(object,PRC,orbit,opts)`
Get the phase differencevectorial field using standard phase reduction method (4.1). deltaPhiMatrix is a $nStepPhasexN-1$ matrix describe the phase different point in which the phase evolution are computed; deltaPhiDot is a $nStepPhasexN-1$ matrix that describe the evolution of the phase difference; PRC is a structure with fields phi and PRC that describe the phase resetting curve of the neuron model; orbit is a structure with fields T and X that describe the time evolution of the single orbit of the neuron over which the PRC is calculated. To compute PRC and orbit you can use method computePRC of neuron_model object. An additional variable opts could be provided; it is a structure with fields:

- nStepPhase - the number of subdivision over each of the N-1 dimension of the phase difference domain nStepIntegral: the step over which the integral is computed (Default 10).
- nStepIntegral - the step over which the integral is computed (Defult 20).

Chapter 3

Example

All the examples are stored in folder `example`.

3.1 Single neuron simulation

In this example a neuron described by Hindmarsh Rose model is simulated through the tool contained in CEPAGE toolbox. The example is described in file `ex_HR_model.m` and is divided in four parts:

- In the first section an object that represent a neuron described by Hindmarsh-Rose model is created and its is displayed through method `disp`
- In the second part the neuron is simulated through the method `sim` using ode45; the results show that with the parameters chosen in the first section the neuron behavior is *bursting*
- In the first part neuron parameters are changed to obtain *quiescence* behavior; the neuron time evolution is obtained calling `simplot` method; in this case we decided to use eulero integrator
- In the last section the parameters are changed again in order to obtain *spiking* behavior; the simulation of the neuron in this case used odeint integrator.

3.2 3 neurons CPG - exact solution

In this example a 3 cell Central pattern generator is simulated through the tool contained in CEPAGE toolbox; the neurons are modeled through the Hodgkin-Huxley model. The example is described in file `ex_HH_Network.m` and is divided in three parts:

- In the first section an object that represent a neuron described by Hodgkin-Huxley model and an object that described the CPG are created; the network structure is displayed through `plot` method.
- In the second section we analyze the network finding the evolution of the phase difference between neurons; we first find a set of initial condition that ensure different phase difference between neurons through method `getCIfromPhi` and then find the state evolution of the network through methods `sim` using odeint integrator; starting from state evolution we find phase difference evolution using `getPhaseRepresentationFromTrack` method and plot the results by mean of `plotPhaseSpace` method
- In the third part we change network structure and compute network phase difference through `getPhaseRepresentation` method; using this method is faster and required less RAM memory then using `sim` and `getPhaseRepresentationFromTra` but with this method the state evolution are not saved.

3.3 3 neurons CPG - approximate phase space representation

In this example a network composed of Fitzhugh-Nagumo neurons is analyzed through both exact and approximate phase representation. The example is described in file `ex_FN_PRC.m` and is divided in four parts:

- In the first section an object that represent a neuron described by Fitzhugh-Nagumo model and an object that described the CPG are created; the network structure is displayed through `plot` method.
- , In the second section we analyze the network finding the evolution of the phase difference between neurons; we first find a set of initial condition that ensure different phase difference between neurons through method `getCIfromPhi` and then find the phase difference evolution of the network through method `getPhaseRepresentation`
- In the third part we compute neuron model phase resetting curve and compute phase difference vector field through method `getApproxPhaseRepresentation`
- in the last section we compare the results obtained with the methods `getPhaseRepresentation` and `getApproxPhaseRepresentation`

Chapter 4

Appendix

4.1 Approximate phase space representation

Many neuron models can be recast as the nonlinear system (4.1), where V is the membrane potential (or a related variable) and vector **x** contains the other state variables.

$$\dot{\mathbf{z}} = \begin{bmatrix} \dot{V} \\ \dot{\mathbf{x}} \end{bmatrix} = \begin{bmatrix} f(V, \mathbf{x}) \\ \mathbf{p}(V, \mathbf{x}) \end{bmatrix} \tag{4.1}$$

We assume that the neuron state describes a periodic orbit $\hat{\mathbf{z}}(t) = [\hat{V}(t), \hat{\mathbf{x}}(t)]$ of period T. We map the periodic orbit to a phase variable $\varphi \in [0, 1)$: the phase increases constantly, thus it is described by the equation $\dot{\varphi}(\hat{\mathbf{z}}) = \omega = 1/T$. If an infinitesimal perturbation $\epsilon(t)$ acts on the neuron we can model it with a phase description using Phase Resetting Curve (PRC) $Q(\varphi)$, as described in Eq. (4.2).

$$\dot{\varphi}(t) = \omega + Q(\varphi)\epsilon(t) \tag{4.2}$$

PRC could be built from the periodic orbit according to different methods [1], [2]. Consider now a network of N neurons; the i-th neuron could be modeled starting from Eq. (4.1) as:

$$\dot{\mathbf{z}}_i = \begin{bmatrix} \dot{V}_i \\ \dot{\mathbf{x}}_i \end{bmatrix} = \begin{bmatrix} f(V_i, \mathbf{x}_i) + \sum_{j=0}^{N-1} g_{i,j}^{in} h_{in}(V_i, V_j) + \sum_{j=0}^{N-1} g_{i,j}^{ex} h_{ex}(V_i, V_j) + \sum_{j=0}^{N-1} g_{i,j}^{el} h_{el}(V_i, V_j) \\ \mathbf{p}(V_i, \mathbf{x}_i) \end{bmatrix} \tag{4.3}$$

where h_{in},h_{ex} and h_{el} describe inhibitory, excitatory and electrical synapses actions, respectively, and $g_{i,j}^{in}$,$g_{i,j}^{in}$ and $g_{i,j}^{el}$ describe inhibitory, excitatory and electrical synapses strength respectively. If the synaptic current that acts on the neuron is small enough (i.e. if the synaptic strengths small enough), we can consider that as a small perturbation that affects the model in [**?**]. By this assumption (By assuming that), we can model each neuron as follows, in terns of the "new" state variable $\varphi_i(t)$:

$$\begin{aligned} \dot{\varphi}_i(t) = \omega_i + Q(\varphi_i)(\sum_{j=0}^{N-1} g_{i,j}^{in} h_{in}(V_i(\varphi_i), V_j(\varphi_j)) + \sum_{j=0}^{N-1} g_{i,j}^{ex} h_{ex}(V_i(\varphi_i), V_j(\varphi_j)) + \\ + \sum_{j=0}^{N-1} g_{i,j}^{el} h_{el}(V_i(\varphi_i), V_j(\varphi_j))) \end{aligned} \tag{4.4}$$

Henceforth we will use the short notation $h_x(\varphi_i, \varphi_j)$ for $h_x(V_i(\varphi_i), V_j(\varphi_j))$.
We now describe the network considering the phase differences between neurons; we describe the i-th neuron through its state variable φ_i; we consider neuron 1 as reference and then describe the network with the new state variables

$\Delta_{1,i} = \varphi_1 - \varphi_i$. Using Eq. (4.4) and supposing all neurons identical ($\omega_i = \omega_j$), we can model the system dynamics as follows:

$$\begin{aligned}\dot{\Delta}_{1,i} = Q(\varphi_1)(\sum_{j=0}^{N-1} g_{1,j}^{in} h_{in}(\varphi_1,\varphi_j) + \sum_{j=0}^{N-1} g_{1,j}^{ex} h_{ex}(\varphi_1,\varphi_j) + \sum_{j=0}^{N-1} g_{1,j}^{el} h_{el}(\varphi_1,\varphi_j))+ \\ -Q(\varphi_i)(\sum_{j=0}^{N-1} g_{i,j}^{in} h_{in}(\varphi_i,\varphi_j) + \sum_{j=0}^{N-1} g_{i,j}^{ex} h_{ex}(\varphi_i,\varphi_j) + \sum_{j=0}^{N-1} g_{i,j}^{el} h_{el}(\varphi_i,\varphi_j))\end{aligned} \tag{4.5}$$

The system is now described by state variables $\Delta = [\Delta_{1,2}, \dots, \Delta_{1,N}]$ and φ_1. Since we assumed that the synaptic strengths are small, variable φ_1 has dynamics faster than Δ. Then we can approximate the vector field that describe the system trough its mean over φ_1. The new formulation is described in (4.6).

$$\begin{aligned}\dot{\Delta_{1,i}} = \sum_{j=0}^{N-1} g_{1,j}^{in} w_{in}(\varphi_j) + \sum_{j=0}^{N-1} g_{1,j}^{ex} w_{ex}(\varphi_j) + \sum_{j=0}^{N-1} g_{1,j}^{el} w_{el}(\varphi_j)+ \\ -\sum_{j=0}^{N-1} g_{i,j}^{in} M_{in}(\varphi_i,\varphi_j) - \sum_{j=0}^{N-1} g_{i,j}^{ex} M_{ex}(\varphi_i,\varphi_j) - \sum_{j=0}^{N-1} g_{i,j}^{el} M_{el}(\varphi_i,\varphi_j)\end{aligned} \tag{4.6}$$

where

$$\begin{aligned} w_k(\varphi_j) &= \int_0^1 Q(\varphi_1) h_k(\varphi_1,\varphi_j) d\varphi_1 \\ M_k(\varphi_i,\varphi_j) &= \int_0^1 Q(\varphi_j) h_k(\varphi_i,\varphi_j) d\varphi_1 \end{aligned}$$

If we group all synaptic strengths in a vector **g**, we can finally write the system as

$$\dot{\Delta} = \mathbf{F}(\Delta, \mathbf{g}) \tag{4.7}$$

Bibliography

[1] Viktor Novičenko and Kestutis Pyragas. Computation of phase response curves via a direct method adapted to infinitesimal perturbations. *Nonlinear Dynamics*, 67(1):517–526, 2012.

[2] Willy Govaerts and Bart Sautois. Computation of the phase response curve: a direct numerical approach. *Neural computation*, 18(4):817–847, 2006.