PROJECT 2

This project is required to pass this course. There must be a main.m file (can be a script or a function). The file must be so that when we press F5 to run it, it runs without any error, warning or undesired screen output (so remember to put those semi-colons at the end of every line). The only thing resulting from running main.m must be the plots shown below.

If you didn't turn in the first project, or if you didn't receive your corrected files back to your email (which means that your project was too late or too incomplete to count), you must do the Extra Exercise (at the end) to pass the course. If you did the first project, you don't have to do it.

The project is due by the end of 05.02.2015. We will pass the list of those who passed the seminar on the next morning, so an extension is not possible (we need time to check the files too!).

As before, all files should be sent (preferably in a zip file) to dario.cuevas_rivera@tu-dresden.de.

For any help with the project, you can contact us by email. If necessary, we can arrange a meeting.

SIMULATING A SPIKING NEURON

The goal of this project is to simulate a spiking neuron using the FitzHugh-Nagumo model. To do this, you will follow a number of steps to, generate some input to the neuron, simulate its activity (obtaining spikes) and plot these data.

The simulation will be ran from $t_{ini} = 0$ to $t_{end} = 1000$. The time interval is dt = 0.1. That means we have 10,000 time points.

GENERATING INPUT

We will use four types of inputs: constant input, pulse input and two randomized inputs. Create a main.m script that generates these inputs (as described below) and store them in variables input_constant, input_pulse, input_randn and input_randnC. Each one of these vectors should contain $(t_{end} - t_{ini})/dt$ elements, that is, 10,000 elements. You can find these vectors in the matlab files included with the project. These are for comparison purposes; you should generate your own in your code.

INPUT_CONSTANT

Generate input_constant with a constant value of 0.35. That means that every element of input_constant equals 0.35.

INPUT_PULSE

Generate a vector input_pulse such that it has a value of 0.35 from t = 200 to t = 300, a value of 0.31 from t = 700 to t = 900, and a value of zero for the rest.

INPUT_RANDN

Generate a vector input_randn that has a random value at every time point. Use the function randn for this.

INPUT_RANDNC

Create a vector input_randnC like in the previous point. Then, every value smaller than zero should be set to zero and every value bigger than 0.4 should be set to 0.7. Hint: use logical indexing.

SIMULATING A NEURON

We will use the FitzHugh-Nagumo equations to simulate the membrane potential of a neuron. You can find these equations in Wikipedia. We will use the following parameters for the equations:

$\tau = 12.5$, $a = 0.7$, $b = 0.8$

Initial conditions: $v(t=0) = -1.2$, $w(t=0) = -0.5$

Note that I_{ext} is input (i.e. input_constant, etc.).

Write a function called Integrate_FHN whose inputs are the initial conditions for *v* and *w* (call them v0 and w0) and the cell's input (for example input_randnC). The function should use the Euler forward to integrate the FitzHugh-Nagumo equations with the given parameters, for all time points. The output of the function is the resulting vectors of the simulation, *v* and *w*. Hint: use a for-loop inside the function that runs for all time points.

The function should look something like this:

function Integrate_FHN(v0,w0,input)

define the values $\tau = 12.5$, $a = 0.7$, $b = 0.8$

for ii from 1 to $(t_{end} - t_{ini})/dt$

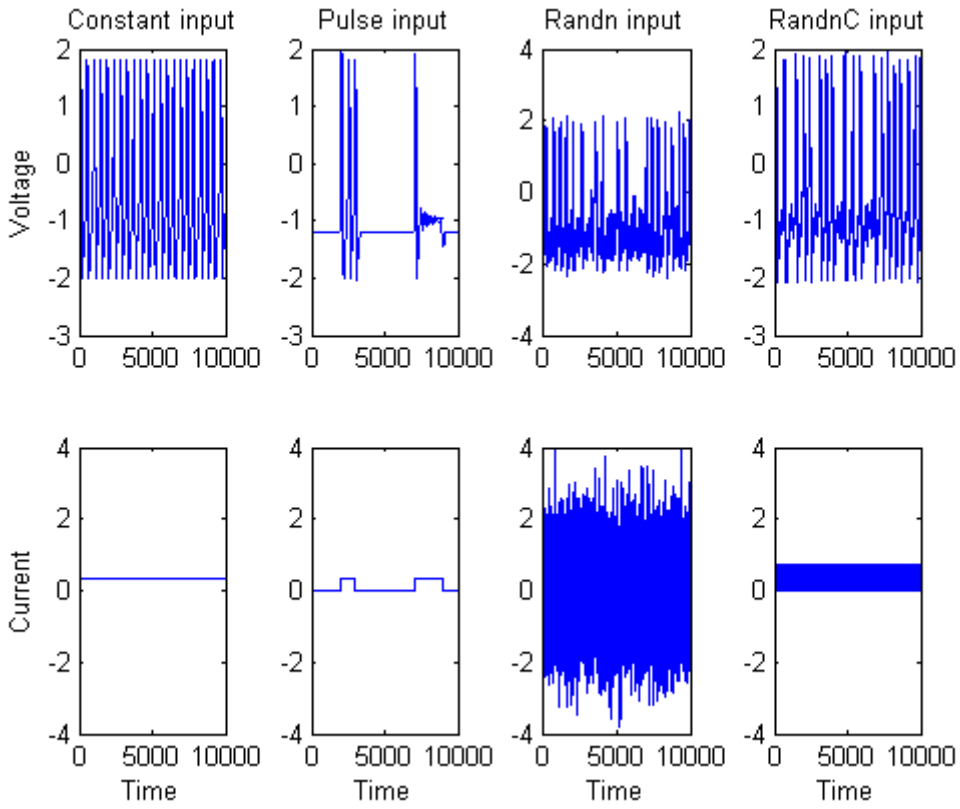
    integrate the FHN equations for each time point ii

end for

end function

Using this function Integrate_FHN, you should simulate a neuron once for each of the inputs above. This must be done from the main.m file (that is, you must do v_constant = Integrate_FHN(…), etc. from the main.m). For each of these simulations, you should store the resulting vector *v* in a different variable, for example, *v_constant, v_pulse, v_randn* and *v_randnC*.

Create a subplot with 2x4 plots. In the lower row of plots, each one of the inputs should be plotted. In the top row, the corresponding values of *v* should be plotted. The plot title, legends and ranges. should be exactly as shown in the included figure (except for the randomness of the inputs in columns 3 and 4).



COUNTING SPIKES

A simple way of determining whether a neuron has spiked is with a threshold. This means that if the membrane potential (*v* in our case) goes above a certain value, we say that it has spiked. We will now use the previously stored variables (*v_constant, v_pulse, v_randn* and *v_randnC)* to count the spikes generated by our neuron.

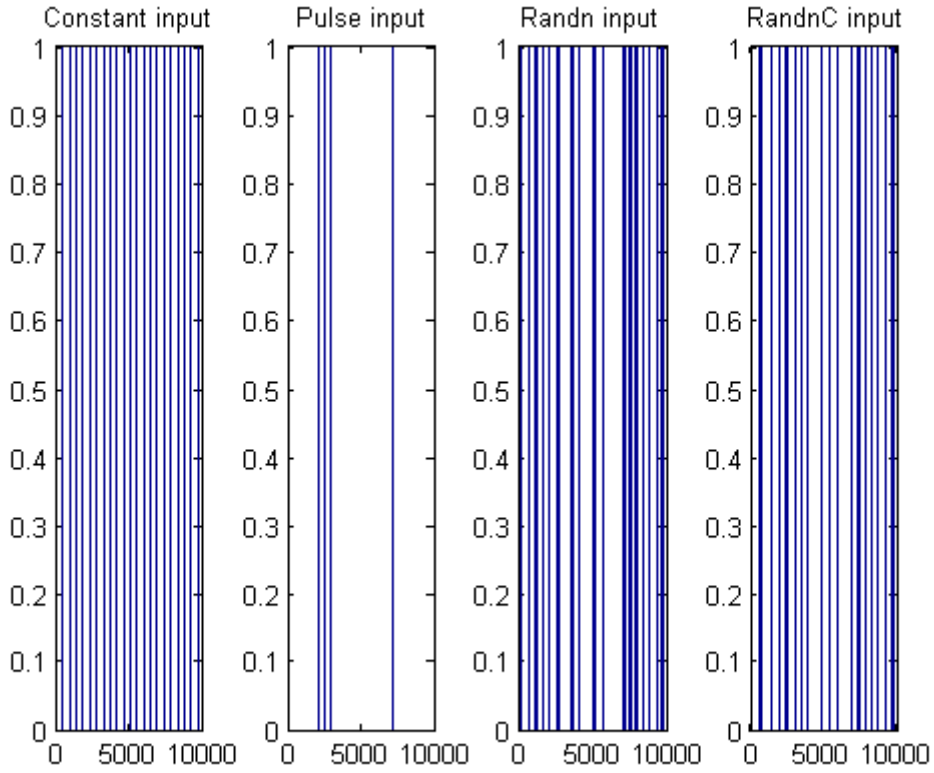
Create a function Counting_Spikes whose inputs are a vector *v* and a scalar T (the threshold). The function should count how many spikes there are in *v*. To do this, the function should go through all the time points in the vector *v*. Create a vector Spikes with as many points as *v*, whose values are all zeroes. For a time point N, the function should check if *v(N)* is smaller than the threshold and if *v(N+1)* is bigger than the threshold. If both conditions are true, then Spikes(N) should be set to 1.

The code should do something like this:

```
function [Spikes] = Counting_Spikes(v, T)

create vector Spikes with zeroes (as many as numel(v))

for N from 1 to numel(v)

        if v(N)<T AND v(N+1)>=T

                Spikes(N+1) = 1

        end if

end for
```

end function

Call (run) the function Counting_Spikes for each *v* (*v_constant, v_pulse, v_randn* and *v_randnC*) with a threshold T = 1. Use *bar* to plot the outputs obtained in a subplot of 4 horizontally arranged plots. The result should be like in the following figure.



EXTRA PROJECT

This is only for those who wish to pass this class but didn't do the first project. If you did the first project, you don't have to do this. As before, the function must run without any error or warning messages and without any output to the screen.

CALCULATING PI WITH RANDOM NUMBERS

The goal is to create a function that will calculate Pi (3.1415…) with any desired level of accuracy. To do this, we will use the random number generator in matlab called rand(a,b). With this function, we will generate a set of N two-dimensional points (that is, points with 2 coordinates) which lie in the square of side 1 (see the figure). We will then count how many of these points landed inside of the quarter-circle; call this number M. Then, Pi = 4*M/N (for mathematical details of this, look at the link below). Follow the steps below.

Create a function called CalPi, with one input (accuracy level, called epsilon) and two outputs: the estimated value of Pi (call it Pi2) and the number of steps required to obtaining such value (call it N; we will explain this below). So:

```
function [Pi2, N] = CalPi(epsilon)
```

This algorithm for calculating Pi gives more accurate results if you use more random points (call these points RS). Thus, the idea is that we start with a small number of points (N = 100, for example) and calculate Pi: we create a vector RS with random entries whose size is going to be Nx2. Each row will be one point with two coordinates (the first column represents X and the second Y, in a Cartesian plane). For example, for N = 1 we can have RS = rand(1,2), which will create one point with two coordinates.

Once we have generated RS for N = 100, we will count how many of these points lie inside the quarter-circle. Because the circle is of radius = 1, we need to check, for each point in RS, if its distance to the origin (to (0,0)) is smaller than 1. Using the Pythagorean theorem, we have that, for the first point in RS, we need to check if `sqrt(RS(1,1)^2 + RS(1,2)^2)<=1` . We count how many of these RS points satisfy this condition and call that number M. Then, we calculate our estimate of Pi, which we will call Pi2, with Pi2 = 4*M/N. You will need a for-loop to do this (although it can be done faster and in fewer lines with logical indexing).

Once we have this estimate of Pi (Pi2), we need to check how accurate it is. We will use the built-in value of Pi in matlab (called "pi") to check this. Say `error = abs(Pi2 - pi)` (check "help abs" to see what it does). If error>epsilon (where epsilon is the input to the function), we repeat the previous process for some bigger N (for example, N = N + 1000). Once the new Pi2 is calculated, we check again if error>epsilon and repeat the process until error<epsilon. To do this automatically, you will need a while loop:

```
While error>epsilon

        Calculate Pi2 with the for-loop

        Calculate error = abs(Pi2 - pi)

end while
```

This function should run for any value of epsilon. In particular, you can try running it for epsilon = 0.001 (to check whether your function works) and see what you obtain. The estimate of pi you obtain should be 3.14xx,

where xx are some random numbers. The value of N you obtain should be a few thousands (max. around 25,000). It will change from run to run because of the randomness of the process.