Practical 4: The Integrate & Fire neuron

2014 version by Mark van Rossum
2018 version by Matthias Hennig and Theoklitos Amvrosiadis

16th October 2018

1 Introduction to MATLAB basics

You can start MATLAB by typing *matlab* on the command line or clicking on its shortcut. Commands to MATLAB can be entered directly on its command line, or can be written as part of a **function** or a **script**, which save the commands and execute them in the written order when called by their name in the command line. The main difference between **functions** and **scripts** is that the first take **inputs** and return **outputs** (at least usually), whereas the latter work with the variables specified in the body of the program. Note that these two forms of .m files can be combined with each other, i.e. a **script** can contain multiple **functions** and vice versa. In this practical we will make use of a **script** file.

The code provided includes a lot of comments to explain the aim of each command, and a general introduction to basic functions will be given here, but for everything else you can use MATLAB's **help** search (top-right corner of the window usually), or search online. MATLAB is documented very well generally, and you will find that most times the Mathworks forum already contains the answers to your questions. You are encouraged to look up how functions work to better understand why, when, and how they are used. There are also custom-written functions by the online community that do extra things, not included in the default MATLAB functions (for example, tools for plotting etc). If you are already experienced with MATLAB you can skip the rest of section 1 and start with the practical itself.

1.1 Fundamentals

As you probably know by now[1], in MATLAB we usually work with **vectors** (1D numerical arrays) and **matrices** (>1D numerical arrays).[2] These arrays store numerical values[3] and allow us to perform calculations in an efficient way, without need for writing lengthy iterative loops (more about those later). There are also different data types in MATLAB, like **strings, cells, structures** and so on, each with its own specific properties and uses, but we will not deal with them too much in this practical.

To store any data in MATLAB, we have to assign it to a **variable name**, by which it is stored in memory and retrieved again when needed. For example, to store the vector (15, 8), we can assign it to the name *a*, with the command *a = [15, 8]; .*[4] This creates a 1-by-2 **row** vector (1 row, 2 columns). For a **column** vector we would have to separate the two values with a semicolon. To create a 3-by-3 matrix we could type in *B = [1 2 3; 4 5 6; 7 8 9]; .* To see the value of any of these variables, just type its name in the command line.

We can retrieve a specific element (or a number of elements) of an array by providing the *indices* of the element like so:

[1] Short introduction in Practical 2.

[2] There's also a 0D (1-by-1) array, which is called a **scalar** and is just a number. Finally, there is also the **empty** (0-by-0) array, which does not contain any elements. This might seem pointless, but it is actually used quite commonly for various purposes.

[3] There are different numerical types, the default one being **double**. You can look up the different types and what distinguishes them. In this practical we will be working with **double-**typed variables.

[4] Note, that we use *square* brackets to create a vector or a matrix. We could have omitted the comma between the two values and just separated them with a space.The semicolon, *;*, following our command makes sure that the value of the variable isn't printed out right after the assignment, as that can be a bit annoying and take *a lot* of space.

b = B(2, 3);

Note the use of *parentheses* for indexing into arrays. This would then assign the value of the element in the second row and third column of the array B (which in this case is 6) to a new variable b. In a very similar way, we can also *change* specific elements of an array. With the command

B(3,1) = 13;

we would end up with the array B = *[1 2 3; 4 5 6; 13 8 9]*.

We could also take *multiple* elements of the array without having to write the indices of each one separately. Look through the following examples:

B(:, 1) → *[1; 4; 13]* (took *all* the elements of the first column)
B(2, 2:3) → *[5 6]* (second row, columns 2 **to** 3)
B(1:2, 2:3) → *[2 3;5 6]* (first **to** second row, columns 2 **to** 3)

We can also quickly create arrays using the *colon* operator as above, to specify starting and ending values. For example, the command

c = [3:15]

will create a row vector containing the numbers from 3 to 15, incrementing by 1.[5]

All the known matrix operations apply and it would be best if you refresh these if you are not too familiar with them. *Transposition* of arrays is accomplished by typing an *apostrophe* (') after the variable name or the array itself.

Errors are sure to result at some point or another. Although they are annoying, they can also be quite informative as to how various functions don't work. Don't be afraid of them, the most dangerous bugs are the ones that do not produce obvious errors but alter the program's function in insidious ways.

1.2 Basic functions used in this practical

We can quickly create large arrays of specified size[6], with the functions ***zeros*** and ***ones***. These functions take as input the desired size of the array (number of rows as first argument and number of columns as second; if only one input, x, is provided, then they create an x-by-x array) and output an array of the specified size full of *zeros* and *ones,* respectively.

To calculate the sum and average of the elements of an array, we can use the built-in functions *sum* and *mean,* respectively. The functions *min* and *max* can be used to find the elements with the minimum and maximum values in an array, respectively.

The function *randn* creates an array of specified size with elements of random values. There are different random functions in MATLAB and they have interesting and useful properties.

The function *floor* rounds a number towards the nearest smaller integer (contrast with the function *ceil,* which does the opposite).

The function *find* is used to find the indices of all the *non-zero* elements in array. It can also be used to find elements that fulfill a specified condition.

The function *diff* takes an array and calculates the differences between the neighbouring elements and returns an array whose length is smaller by 1 than the original.

The above functions are simple, but there are also subtleties for their use, so look them up in MATLAB's documentation as you encounter them in the code.

[5] This is the default, but we can also specify the step size, by typing *c = [3:2:15]* we would get *[3, 5, 7, 9, 11, 13, 15]*.

[6] This is useful to do from the start of the program, because it is computationally less expensive to replace the elements of an existing array than to change the size of the array (by adding or removing elements, say).

1.3 If statements and for loops

If statements are powerful tools that allow us to check whether a condition is true and execute a block of code in that case. They can be combined with *elseif* and *else* statements, which check alternative conditions and specify alternative blocks of code to be executed. They always have to be terminated with an *end* statement. The basic syntax goes something like this:

If *condition-to-be-checked-is-true*
execute commands
(**elseif/else**
alternative commands)
end

For loops allow us to sequentially iterate some commands a specified number of times.[7] The basic syntax looks like this:

for *iIndex = start:nIterations*
execute commands (depending on iIndex)
end

This loop will go through every index from the specified *start* value to the *nIterations* (incrementing by 1 every time, by default) and execute the commands written in the body of the loop on each iteration.

If statements and *for* loops can be combined with each other, and also with themselves, in which case they are called *nested.*

There's much more to these kinds of statements than can be covered here, which you can only gain by using them in different applications. But for now, the above will do.

1.4 Plotting

Visualizing your results properly is crucial, because it is the only way to simply and clearly communicate your program's output. No one likes going through paragraphs of numbers and trying to imagine what they mean. The most basic plotting function in MATLAB is, well, *plot(x, y).* This plots a 2D line for the data specified in array x against the corresponding data in array y, and they will correspond to the *x-axis* and *y-axis* values, respectively.

With the same function we can also set the properties of the line, like **color**, **style**, **width**, **marker** type, etc. We can also change these properties using the *set* function. There are many options you can explore in the **help** section.

The functions *xlim, ylim,* allow us to change the limits of the x- and y- axis respectively. Otherwise they have default values, to include all of the data points plotted.

We can add a title to our figure and labels to the axes with the functions *title('*yourtitle*'), xlabel('*yourxlabel*'), ylabel('*yourylabel*'),* respectively.

Another thing we can do is plot more than one graph in a single figure with the function *subplot,* by specifying how many subplots we want and the desired coordinates for each plot in the resulting grid (similar to an array).

Finally, in this practical, we also make use of the function *histogram,* which bins our data automatically and plots the histogram of counts for every bin, with the option of normalization (e.g. to have values as a fraction of the total).

2 Simulating a noisy input to Integrate & Fire neurons

2.1 Model Setup

First, we will construct an array of I&F neurons. To do that, we have to give them some basic properties to make them functional (resting membrane potential, V_{rest}, reset voltage, V_{reset}, spike generation threshold, V_{thr}, membrane time constant, τ_m, input resistance. R_m is considered to be 1 for simplicity).

[7] Due to the nature of MATLAB, unnecessary use of *for* loops is discouraged, in favour of matrix utilization for iterative operations.

We also specify the time parameters of our simulation (duration of the simulation, timestep size, and number of timesteps).

Moreover, we will provide our neurons with an input current and two noise inputs, one that is common to all cells (can be seen as part of the stimulus, or as part of the global network activity[8]), and one that is independent for each neuron (can be seen as reflecting the unique set of connections that each neuron receives).

Finally, we create some arrays to hold our data for the duration of the simulation.

Please look through the first section of the code and make sure you understand what each variable stands for.

2.2 Running the simulation

Once we have all the relevant variables specified and our data structures ready to store data, we can run the simulation. This simply consists of going one timestep at a time and:

- providing each neuron with an input current
- adding the common and independent noise components
- calculating the new membrane potential for the current timestep
- determining whether any cell has reached the AP threshold
- resetting the membrane potential of all the cells that have fired to the reset voltage

The values of the common input current, the membrane voltage of each cell, and a binary value for the spikes are all stored in the corresponding arrays we created during the model setup.

Look carefully through this part of the code, identify each of the steps detailed above, and make sure you understand what is going on.

2.3 Plotting the results

What we want to do after the end of the simulation is to look at the spikes our cells fired in response to the input we provided. We can do this by making a *raster* plot, which you have encountered in the lectures. While here we are looking at the spikes for many different neurons at the same time, you can imagine that the different neurons are equivalent to recordings of the *same* neuron in different *trials,* as is usually the case with raster plots from experiments. What do you observe in the raster plot compared to the input amplitude? Discuss with your neighbour.

You should end up with something like Figure 1. We are using the data structure we used to hold our spiking data for the raster plot, and we are also visualizing the input current as a separate subplot. The two plots are aligned in terms of simulation time. Look through the code used to generate the plots and try to understand the commands. Most of the commands are used to make the figure more presentable.

[8] Can you guess as to what might influence a cortical network as a whole?

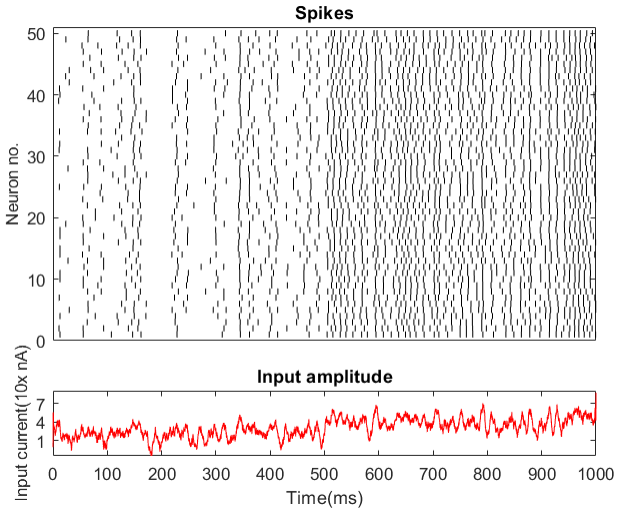

Figure 1: Raster plot and input current

Finally, we are computing the InterSpike Interval (ISI) distribution and visualizing it with the *histogram* function. It should look like Figure 2. Make sure you understand its generation. Discuss your thoughts with your neighbour. More below.

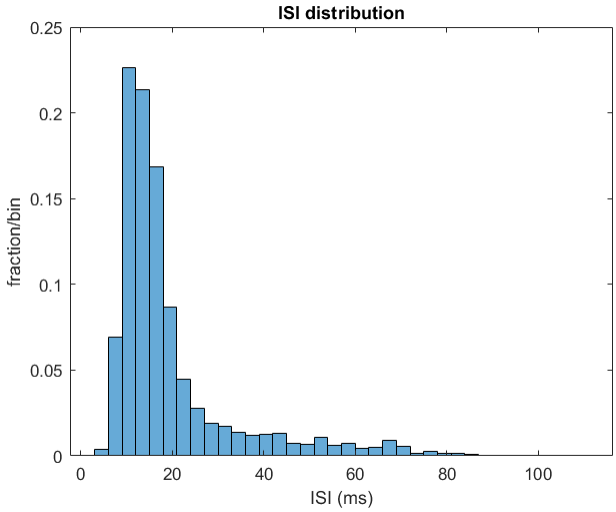

Figure 2: ISI distribution

3 Exploring the model's parameters

3.1 Neuron parameters

Try changing the parameters that characterize the neurons (voltages, time constant). Before each run of the simulation try to predict the outcome qualitatively. Check the correspondence of your intuitions with the results. For visualization purposes, also try varying the number of the neurons in the simulation (beware of increasing runtime!).

3.2 Input parameters

- What role does the input current play? How would you expect the spike trains to change if you vary its amplitude? Check your predictions in the raster plot but also the ISI distribution. Try incorporating different inputs at different time points.
- What role does each noise parameter play? What is the difference between them? What will happen if you set either one, or both, to zero? Make predictions before looking at the results! Discuss with your neighbours. What is stimulus locking? Recall the lectures and try to identify in the graphs.

4 ISI statistics

- What information does the ISI distribution provide? Plot the histogram for the different input conditions separately (i.e. first and second half of simulation, or as many different input conditions as you have included). Are they different? Why/why not?
- (Optional) You can look at the fit to the distribution with the function *histfit.* Look up how it works first, there are various options.
- Calculate the coefficient of variation (CoV) for the ISI distribution. What does CoV signify in general? What does it mean for a neuron? Try doing this for the different input conditions separately and try different window sizes as well. How do you expect its value to change?
- (Optional) Do the above for the Fano factor as well. Plot the resulting Fano factor with different window sizes.

5 (Extra) Generalizing

Suppose we would like to write a more general Integrate & Fire simulator in which the timestep is user-adjustable. How should one normalise the noise term, such that the effect of the noise does not depend on the timestep?

6 (Extra) Adding adaptation

We can add adaptation to the model as follows:

- If the neuron spikes, increase the Ca^{2+} concentration by 1.
- At every timestep, the Ca^{2+} concentration decays exponentially back to zero with a time-constant of $50ms$.
- The Ca^{2+} activates a K_{Ca} current as $I_{K_{Ca}} = [Ca]{*}g_{K_{Ca}}(V^{rev}_{K_{Ca}} - V)$. Take $V^{rev}_{K_{Ca}} = V_{rest}$ and $g_{K_{Ca}} = 2$.

Apply step currents of different strengths. How much and how quickly does the neuron adapt?