Practical 3: AMPA receptor kinetics

2006 version by Mark van Rossum
2019 version by Matthias Hennig and Theoklitos Amvrosiadis

October 16, 2019

1 Aims

This practical has three aims:

- To provide an introduction to the basics of programming with MATLAB
- To understand how to simulate differential equations (and by extension dynamical systems) in discrete time
- To simulate and understand the kinetics of the AMPA receptor in the presence of Glutamate

2 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.

2.1 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.

[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.

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:

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.

2.2 Basic functions used in this and the next couple of practicals

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).

[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.

[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).

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.

2.3 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.

2.4 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 the next 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).

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

3 Some background for AMPAR kinetics

AMPA-type glutamate (Glu) receptors mediate fast excitatory responses to Glu transmission in the brain. We can model the kinetics of the receptor as a Markov process, by including the different states for the receptor and the transition probabilities between them. Recall the state diagram from Lecture 4 (see Figure 1). The state diagram captures the three primary features of receptor gating, namely activation (binding of the neurotransmitter), deactivation (unbinding), and desensitization. More complicated models have been proposed since, but for our purposes this diagram will do just fine.

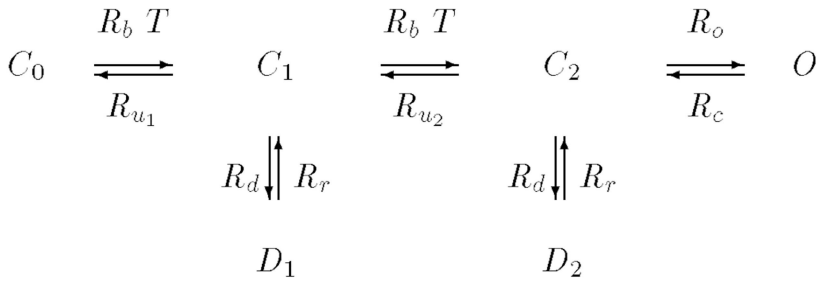

Figure 1: AMPAR state diagram (from "*Kinetic models of synaptic transmission*", Destexhe et al, in *Methods in Neuronal Modeling,* Koch and Segev, 1998). C_0, unbound receptor; C_1, singly-bound; C_2, doubly-bound; O, open; D_1, D_2, desensitised; R_b, transmitter binding rate; R_{u1}, R_{u2}, unbinding rates; R_d, R_r, desensitisation and resensitisation rates, respectively; R_o, R_c, opening and closing rates, respectively.

The parameters of this model have been fitted to experimental data and are:

- $R_b = 13 * 10^6 M^{-1} s^{-1}$
- $R_{u1} = 5.9 s^{-1}$
- $R_{u2} = 8.6 * 10^4 s^{-1}$
- $R_d = 900 s^{-1}$
- $R_r = 64 s^{-1}$
- $R_o = 2.7 * 10^3 s^{-1}$
- $R_c = 200 s^{-1}$

4 Running the simulation

Recall that we can keep track of the different states of the receptor with a **state vector**, $\boldsymbol{s}$, that contains all the possible states of the receptor. In this case, it would look like $\boldsymbol{s} = (C_0, C_1, D_1, C_2, D_2, O)$. We can then update the state vector by multiplying with the **transition matrix**, $\boldsymbol{W}$, which contains all the transition probabilities from one state to another, according to $\frac{d\boldsymbol{s}}{dt} = \boldsymbol{W.s}$ (refer to Lecture 4 slides and Lecture notes for more information). We will use the Euler method to solve this differential equation, meaning that we will calculate $\frac{d\boldsymbol{s}}{dt}$ in small discrete timesteps dt and update $\boldsymbol{s}$ iteratively.

Next, please go through the MATLAB script and try to understand what each part is doing. Make sure the transition matrix that is created has the appropriate values at the appropriate coordinates. After you have done that, run the script and look at the resulting plot. Do the trajectories for the different receptor states make sense?

5 Exploring the model

5.1 More inputs

Provide a second pulse of neurotransmitter shortly after the first one. What do you expect to see? Plot the results and check your predictions.

5.2 Input duration

Test what happens when you apply a longer pulse of Glu (e.g. double the duration of the pulse). Again, make your predictions beforehand and then check them by running the simulation.

5.3 Neurotransmitter concentration

Apply different concentrations of Glu and check how the open occupancy of the receptor depends on this parameter.

6 (Extra) Extending the model to account for stochastic behaviour

The model so far describes the collective behaviour of a large (infnite) number of channels. However, individual channels are not deterministic but act stochastically. Suppose you would like to create a simulation of a small population of channels which behave stochastically. How would you rewrite the simulation to describe stochastic simulations?