Coding Guide and Hints for Practical session 1: Solving differential equations on a computer

The aim of this guide is to give hints and tips for the coding required for Practical Session 1, particularly targeted towards people with limited experience with Matlab. Furthermore, hints are for some of the trickier tasks.

Part 1: The Euler Method

Task 1

Opening Matlab (CUBRIC users): I have checked all the code for this workshop using Matlab version R2017b, but I believe it should work on all other versions of Matlab. To access Matlab from a CUBRIC desktop or via NoMachine, you can either go to Applications -> Programming -> MATLAB 2017b or open the terminal and enter `matlab` (for R2015a) or `/opt/MALTAB/R2017b/bin/matlab` (for R2017b).

Changing directory: To change your directory, find the window in Matlab called "Command Window" (marked 1 in Fig 1) and use the command `cd DIR`, where DIR is the path to the directory. For CUBRIC users, if you extracted the github folder to your scratch folder, you would enter

```
cd /cubric/scratch/XYZ/intro_to_modelling/practical1
```

Here, XYZ is your Cardiff ID. You can validate you are in the correct directory by looking at the bar marked 2 in Fig 1 which shows the path of the current directory. Furthermore, the "Current Folder" window (marked 3 in Fig 1) should have the same contents as is shown in Fig 1.

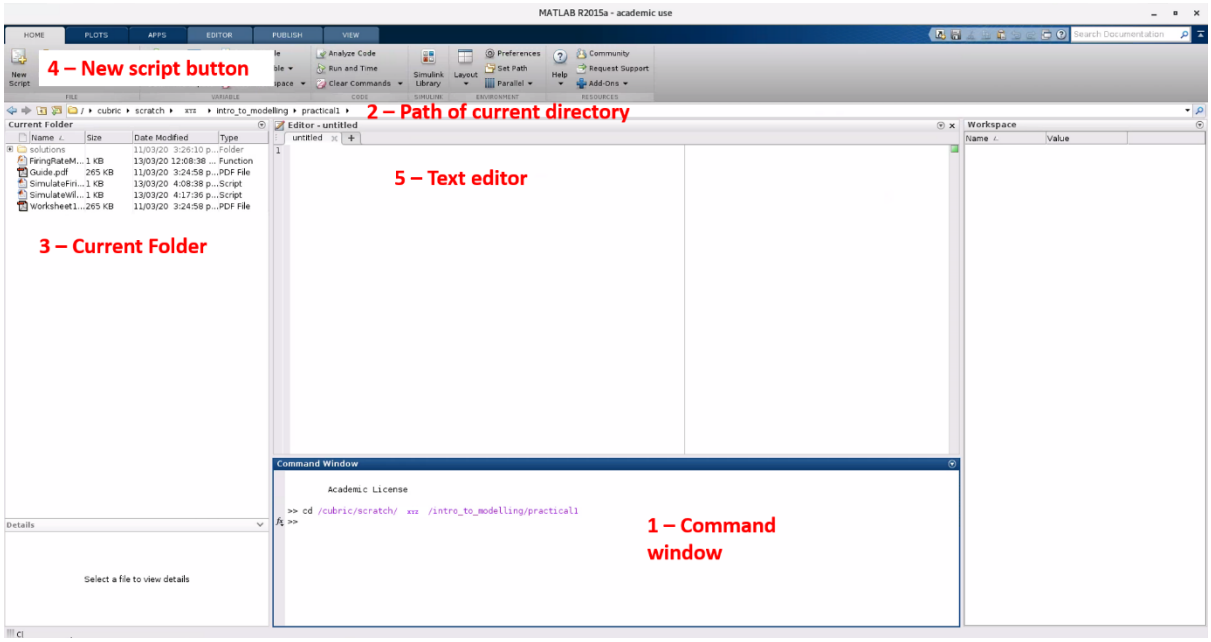

Figure 1: Layout of Matlab window. Note, when opening Matlab for the first time, the text editor (5) may not be open, and the command window (1) may fill its space. To open the text editor for writing scripts, you can click New Script (4).

Writing a function: To write a function, click the New Script button (marked 4 in Fig 1) to open a new instance of the text editor (marked 5 in Fig 1). In the text editor, on the first line, type

```
function x=FunctionName(input1,input2,...)
```

Here, the … just represents that you can have an arbitrary number of inputs, and is not syntax. So, to write a function called EulerODE with three inputs (t, x0, and f), you should type

```
function x=EulerODE(t,x0,f)
```

If you select the EDITOR tab in the panel above the New Script button, and click the save button (near to where the New Script button was in the HOME tab), a screen should pop up prompting you to give a file name and choose a directory. By default, the current directory is used and the function name is the file name. Click save. EulerODE.m should appear in the Current Folder window. Note: It is vital that the file name be the function name, i.e. the function EulerODE MUST be saved as EulerODE.m.

The contents of the function are entered below this line of text, and must define the variable `x` which is the output. The function is called using parentheses from the console. For an example function, see Fig 2. In task 1, you must write this function such that the output `x` is the solution to the ODE `f` at times `t` given initial conditions `x0`, using the Euler algorithm. This algorithm is described in the next section.

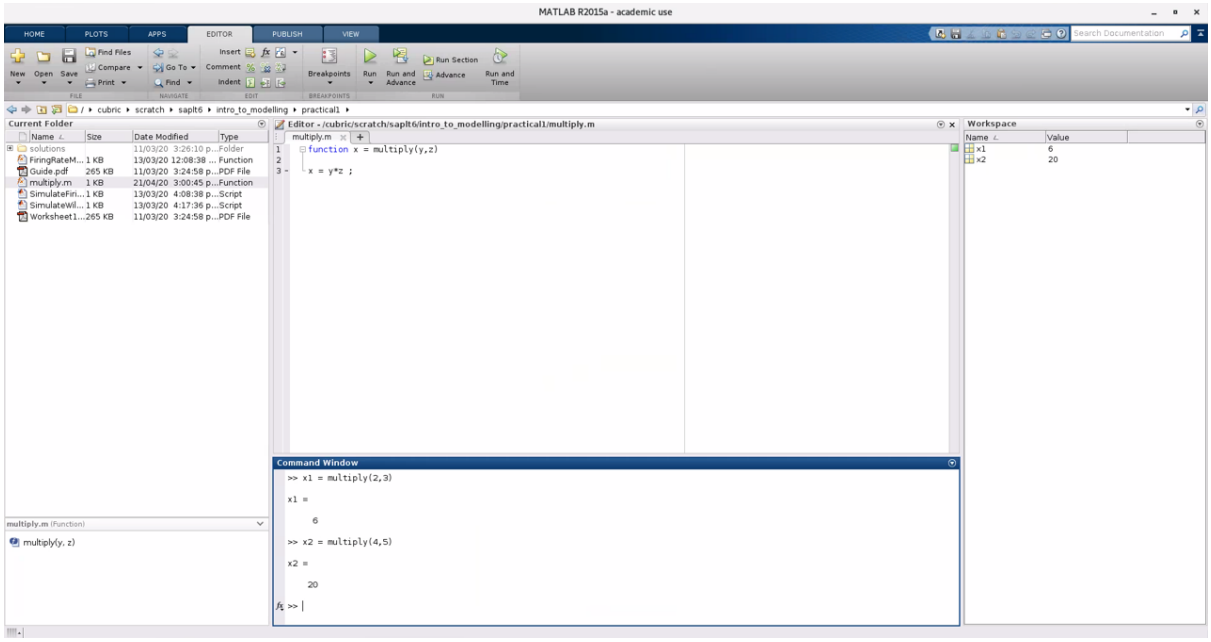

Figure 2: An example function. This function takes two numbers y and z as an input and multiplies them. The function is called multiply, as can be seen on line 1 (`function x=multiply(y,z)`), and in the file name multiply.m. In the command window, we call the function twice, first multiplying 2 and 3, then multiplying 4 and 5.

Function handles: The third input to EulerODE, `f`, is a *function handle*. This is essentially a variable which points to a function, and is declared using the `@` symbol. Consider the function `x=multiply(y,z)` in figure 2, which takes in two inputs y and z and multiplies them. If I define a function handle `f=@multiply`, then `f(3,10)` will output 30. In EulerODE, `f` is the ODE function. Whenever we type `f(x(:,i))`, we are evaluating the ODE for the values of `x` at time point `i`.

The Euler Algorithm: Task 1 of Part 1 asked you to write a function called EulerODE which "takes as input an array of time points (`t`: size 1xT), an array of initial conditions (`x0`: size Nx1) and an ODE function handle `f`, which outputs the solution to the ODE solved using the Euler method. It should perform steps 2-4 of the algorithm [Algorithm: Euler Method]". We will describe below how each step

can be implemented in Matlab. These steps should be entered in the file EulerODE.m you created in the previous step.

Step 2: Initialize an array `x` with size `[N,T]` where `N` is the dimensionality of the system and `T` is the number of time points.

To **initialize an array** of size [N,T], you can use the command `x=zeros(N,T)`. This defines `x` as an array of zeros. To define `N` and `T`, we can use the Matlab default `length` function on inputs `x0` and `t` respectively. This function calculates the number of elements in a vector array, i.e. if `m` is a 1xM or Mx1 array, then `M=length(m)`.

Step 3: Set `x(:,1)=x0`

Here, we are simply filling in the first column of `x`, which corresponds to the first time point, to be the initial conditions `x0`. You just need to copy the line `x(:,1)=x0`. The syntax `x(:,i)` means every element in the `i`'th column of `x`, while the syntax `x(i,:)` would mean every element in the `i`'th row of `x`.

Step 4: Loop over `i` in the range `1` to `T-1`, finding `h=t(i+1)-t(i)` and then solving `x(:,i+1)=x(:,i)+h*f(x(:,i))`

To **loop over a variable** in a range 1 to M, the syntax is

```
for i = 1:M
      % Contents of loop
end
```

You can use this syntax to loop over `i` from `1` to `T-1`. Inside the loop, you should first find `h` and then solve for `x` using the snippets of code described in the algorithm step.

Task 2

Running a script: In Matlab, scripts (like functions) are saved as FileName.m (where FileName is a relevant name for the script). First check this .m file is in the current directory by finding it in the Current Folder window (marked 3 in Fig 1). Type FileName (without the .m extension) into the Command Window. Alternatively, if you wish to also see the contents of the script, you can type open FileName.m in the Command Window, and click the Run button in the EDITOR tab.

Therefore to run `SimulateFiringRateModel.m`, you should type

```
SimulateFiringRateModel
```

into the Command Window OR type

```
open SimulateFiringRateModel.m
```

into the Command Window and click Run in the EDITOR tab.

Function handles with fixed inputs: This section explains line 13 of FiringRateModel.m – you don't have to implement anything, but you may be confused by the 3rd input given to EulerODE on this line, so here we clarify this. In EulerODE, we call `f(x(:,i))`. Yet our ODE function FiringRateModel has two inputs, `x` and `P`. How does EulerODE know which value of `P` to use? On line 13 of

SimulateFiringRateModel, we have set the third input to EulerODE, `f`, to be `@(x) FiringRateModel(x,P)`. Let us break this command down. Firstly, we have declared `f` with an `@` symbol, meaning it is a function handle. Immediately after the `@` symbol, we wrote `(x)`. This means that we are using only one input to the function handle named `x`, and all other inputs will be fixed. We then write `FiringRateModel(x,P)`. This says the function we are assigning is the FiringRateModel function. The first input is variable because we have called it `x`, and `x` was in brackets after the `@` symbol. The 2nd input is fixed because we have called it `P`, and `P` was not in brackets after the `@` symbol. The value used for the 2nd input is the value of `P` declared in the script on line 8. Therefore calling `f(x(:,i))` in EulerODE is the same as fixing `P` and then calling `FiringRateModel(x(:,i),P)`.

In short, `@(x) FiringRateModel(x,P)` creates a function handle to FiringRateModel which only takes one input `x`, and fixes `P` to the value already declared.

Task 3

Editing a script: Scripts can be opened in the Matlab editor by entering `open FileName.m` into the command window. You can then simply edit the script in the Matlab editor (marked 5 in Fig 1). If running the script from the command window, you MUST save any changes before running the script again, otherwise the changes will not have effect. If running the script from the Run button in the EDITOR tab, changes will automatically be saved.

Part 2: The Wilson-Cowan Equations

Task 4

Semicolon syntax: In the task description, we say that x=[E;I] and xdot=[Edot;Idot]. The syntax x=[A;B] simply means that we have a 2x1 array, where the first element is x(1)=A and the 2nd element is x(2)=B, i.e. semi-colons 'stack' variables in columns. If we instead write x=[A,B], this is very similar, but the array is 1x2, i.e. commas 'stack' variables in rows.

Adapting FiringRateModel.m into WilsonCowan.m: The function FiringRateModel.m is very similar to the function WilsonCowan.m, but has only three parameters (vs WilsonCowan's 10 parameters given the Table on page 2 of the worksheet), and one population meaning there is only one sigmoid function and one ODE (vs WilsonCowan's two populations). However, these are easily adaptable, and I will outline this below.

Parameters: The parameters of both models are variables which must be given a value in the code. For example, FiringRateModel.m defines a variable `tau` on line 4, which is the time constant of the population. The Wilson-Cowan model has time constants for both the E and I populations, so in WilsonCowan.m you should instead define two variables `tauE` and `tauI`. You can treat the remaining variables similarly. Don't forget to also add the parameters for the connectivity between populations.

Dealing with two populations: FiringRateModel.m has a single population `x`. The Wilson-Cowan model has two populations E and I, so the input `x` is instead a 2x1 array `x=[E;I]`. To deal with this, we first need to 'unpack' `x` writing `E=x(1)` and `I=x(2)` in WilsonCowan.m. Similarly, there will be two

equations – one equation for `Edot` and one for `Idot`, so at the end of WilsonCowan.m we must make sure to include a line which reads `xdot = [Edot;Idot]`.

Sigmoid functions: FiringRateModel.m has a single sigmoid which is calculated on line 9, given by the variable `S`. In the Wilson-Cowan model, there are two populations, each with their own sigmoid functions. We therefore require two equations here, one for `SE` (the excitatory sigmoid) and one for `SI`. You will need to think here about what the input to each population is. Remember, the sigmoids are given by $S = 1/(1 + \exp(-k(input - \theta))$. For the firing rate model, the only input is the external input P, hence the term `-k*(P-th)` on line 9. In the Wilson-Cowan model, the input to the E population is $P + c_{EE}E - c_{IE}I$, so the equation for `SE` must be adjusted accordingly. Similarly, the input to the I population is $c_{EI}E - c_{II}I$, so the equation for `SI` must be adjusted.

Differential equations: FiringRateModel.m has a single population `x` with derivative `xdot`, and this derivative is calculated on line 12. The Wilson-Cowan model has two populations E and I, so we must include two equations, one for `Edot` and one for `Idot`, which can be done by adapting line 12 of FiringRateModel.m once for each population. Remember to pack these up into a single variable `xdot = [Edot;Idot]` at the end!

Task 5

Running a script: See help for task 2.

Index 'end' syntax: The last element of an array `x` is given by `x(end)`. Similarly, the last column is `x(:,end)`, and the last row is `x(end,:)`. Therefore to print the final values of E and I, type `x(:,end)` into the command window. If the system has reached a steady state during the course of the simulation, this should give the value of the steady state, since once a system is at a steady state it does not move away from this point unless perturbed. You can therefore use this to check the steady state is at `x=[0.0181;0.0207]`.

Task 6

Editing a script: See help for task 3.

Task 7

Editing a function: Your parameters c_{EI} and c_{IE} should be defined in the function WilsonCown.m. Editing a function is the same as editing a script – see help for task 3. Make sure to save before running SimulateWilsonCowan.m to test the effects of these changes.

Part 3: Stochastic differential equations (SDEs)

Task 8

Drawing normally distributed random numbers: The Matlab function `randn(N)` draws an NxN array of random numbers from the normal distribution with standard deviation 1. To draw an NxM

array, use `randn(N,M)`. To draw numbers with a mean of `U` and standard deviation `A`, you can use the command `U-A*randn(N,M)`. In the EulerSDE solution, at each time step we want to draw N random numbers (where N is the dimensionality of the system) with mean 0 and standard deviation `stdnoise`, so we use `stdnoise*randn(N,1)`.

Task 9

Comments: Anything followed by a `%` character is ignored by Matlab, and should be displayed in green. Therefore this is useful for commenting your code. To comment whole sections of code, you can highlight and either press the Comment button (a green % sign) in the EDITOR tab or type Ctrl+/ on Unix or Ctrl+R on Windows (although sometimes it seems to be Ctrl+R on Unix!). To uncomment, highlight some commented code and press the Uncomment button (next to the Comment button) or Ctrl+T.