

# Math0086 practical classes

## Week 1: Introduction to **MATLAB**

Contact: [sean.jamshidi.16@ucl.ac.uk](mailto:sean.jamshidi.16@ucl.ac.uk)

The best way to learn **MATLAB**, or any programming language, is to try things out for yourself. This sheet is designed with that in mind, and I hope that you will try the extra challenges and discover some useful features of **MATLAB** yourself. To aid you in your discovery, there are many resources available. Some of the most often-used are listed below:

- The `help` command in **MATLAB**. Type something like `help abs` at the command line for a quick description of how the `abs` function works.
- The `doc` command in **MATLAB**. Type something like `doc abs` to open a new window with a full description of how the `abs` function works. Can be intimidating at first.
- The website <https://uk.mathworks.com/matlabcentral/answers/index> is a forum where lots of people have asked questions about how **MATLAB** works. Often if you google a question about **MATLAB**, this will be one of the first results that comes up.

### **How to download MATLAB**

If you would like to run **MATLAB** on your own computer, then you will need to download it. This can be done at the UCL software database, which is here: <https://swdb.ucl.ac.uk/>. All of the departmental machines should have **MATLAB** installed already, but it is highly recommended that you use your own computer for coursework.

If you are using your own computer now, I suggest you set **MATLAB** to download and begin with the first exercise, which can all be done online.

# 1 Getting started

## Exercise 1.

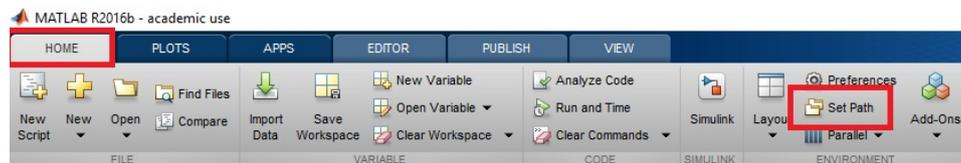
Complete the tutorial at <https://.learntocode.mathworks.com>.

Notice how every time you started a new section of the tutorial, all of the variables had to be re-defined. This was because the tutorial was teaching you how to program from the *command line*. Most programming in MATLAB is not done this way. Instead, it is more usual to create a file known as a *script* and then run it. The script can be saved, edited and shared easily, which is often much more convenient.

## Exercise 2.

On the moodle page you will find a file called 'learntocode\_script1.m', where the program that you have just designed has been turned into a file (called a 'dot m file') that can be read in MATLAB. Download the file, save it in a sensible place and open it. This should open MATLAB, with the script in the Editor window.

MATLAB will only run files that are stored in a certain location, called the *path*. In the 'Home' tab, click on 'Set Path', then 'Add Folder with Subfolders' and navigate to the folder in which you saved the download to allow MATLAB to run the script. Run the script by clicking on the 'Editor' tab and then pressing the 'Run' button<sup>1</sup>.



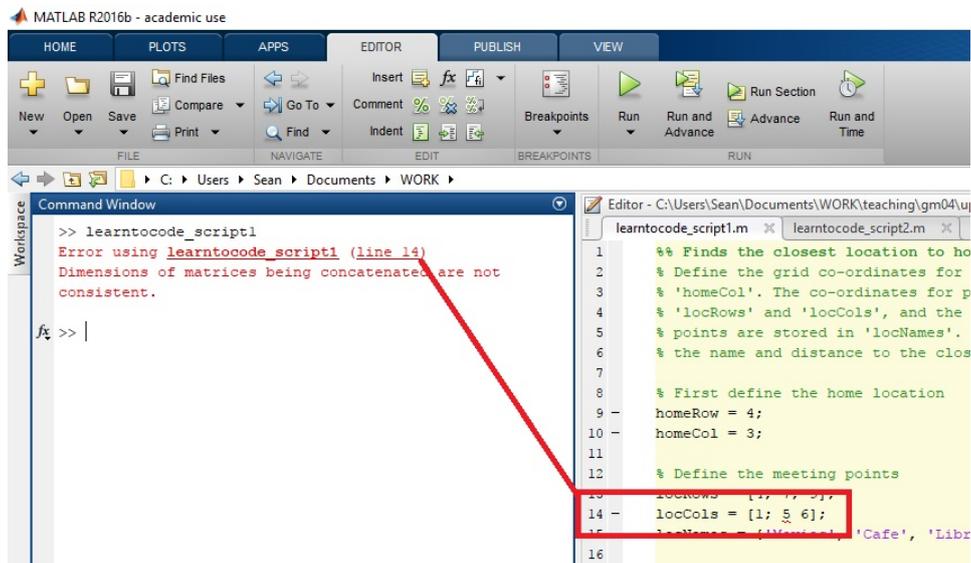
When you press run, notice that you get an error message on the command prompt. The script that you downloaded contains three errors that will stop it from working, and the red error message at the command prompt should identify the line number and type of error for the first problem that it encounters. If you look in the editor, the error is highlighted with a red underline.

## Exercise 3.

Remove the errors from the code and run the script (this process is known as *debugging*).

---

<sup>1</sup>You can also use the F5 key.



**Note:** Two of the mistakes are underlined in the editor in red, but MATLAB can't spot the third one. To work out what it is, you need to read the error message in the command prompt.

#### Exercise 4.

Change the code so that the home is located at row 5, column 2 and run the program again. This is a much faster way of editing than having to start again and input all the variables from scratch.

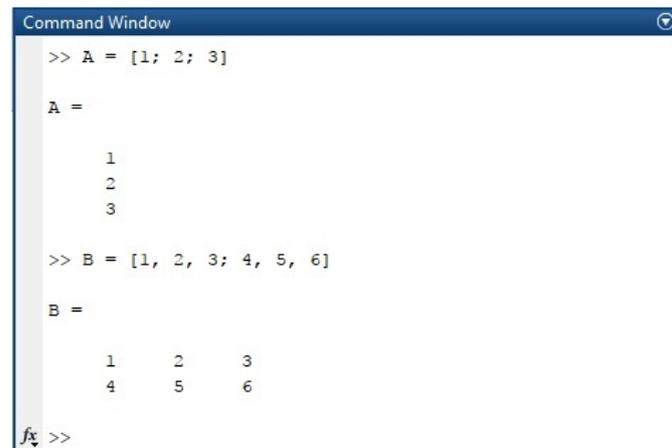
#### Using comments

Some lines in the code are coloured green, and begin with a %. These are *comment lines*, and will be ignored by MATLAB when it is running the file. You should use comments to annotate and organise your code and explain what different bits mean or do.

**Challenge:** At the end of the file, I have included code on line 29 and then turned it into a comment. This is another use for comments – to temporarily disable part of your program. Remove the % symbol from line 29 and run the file again. This time, you should see a message at the command prompt that tells you what the closest location is. To make MATLAB display this message, I used the function `sprintf`. Use the help files to explore how `sprintf` works and modify line 29 so that the displayed message reads: “The closest location is (*location name*) which is (*distance*) squares away”. You can also suppress line 27 with a semi-colon so that the closest location is only displayed once.

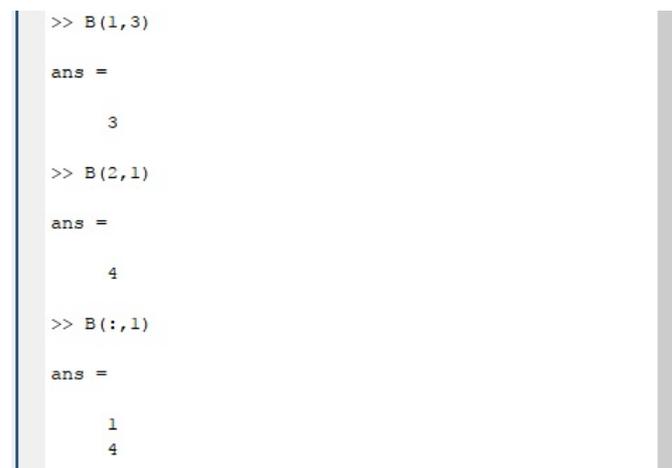
## 2 Using matrices

So far we have been storing all of our data in vectors. For example, we stored the list of location columns in a vector of length 3. Another important way of storing data is in matrices. In **MATLAB**, we separate the rows of a matrix using the semi-colon (;) symbol, and the columns using a comma (,) symbol. The whole matrix should be enclosed in square brackets, ].



```
Command Window
>> A = [1; 2; 3]
A =
     1
     2
     3
>> B = [1, 2, 3; 4, 5, 6]
B =
     1     2     3
     4     5     6
fx >>
```

To select entries in a matrix we need two indices to specify the row and column. The first index tells us what row we're in, and the second index gives the column. You can select a whole row or column by using a colon (:). So typing `B(:,1)` means 'everything in the first column'.



```
>> B(1,3)
ans =
     3
>> B(2,1)
ans =
     4
>> B(:,1)
ans =
     1
     4
```

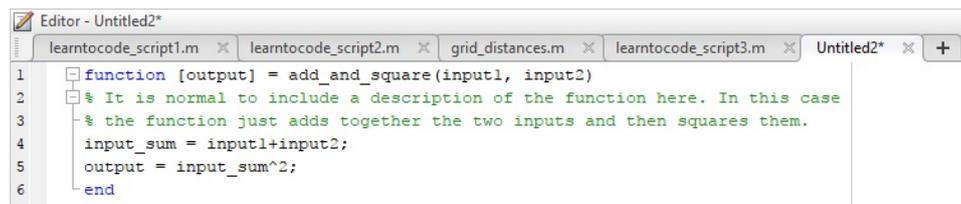
### Exercise 5.

Modify the code so that, instead of having separate vectors for columns and rows, we use a matrix for the locations. The first column of the matrix should give the information from `locRows`, and the second column should

give information from `locCols`. Your code should also store the home location as a single vector rather than two scalars. Save your modified code under a new name.

### 3 Writing functions

Sometimes we want to write a separate piece of code, called a *function*, that performs a self-contained and simple task. We can then use this function in lots of different projects without writing it out again every time. We will now create a function that calculates the distance from a home location to a list of meeting points, so that you can just ‘call’ the function every time you write a piece of code that requires this task. Just like a mathematical function, a function in **MATLAB** takes some inputs and gives some outputs. In our case, the function should take two inputs (the home co-ordinates and the list of meeting point co-ordinates) and return one output (a list of distances).



```
Editor - Untitled2*
learntocode_script1.m x learntocode_script2.m x grid_distances.m x learntocode_script3.m x Untitled2* x +
1 function [output] = add_and_square(input1, input2)
2 % It is normal to include a description of the function here. In this case
3 % the function just adds together the two inputs and then squares them.
4 input_sum = input1+input2;
5 output = input_sum^2;
6 end
```

The format for writing a function is shown in the screenshot above. The function must be on its script, and the name of the file must be the same as the name of the function. The first line of the script **must** start with the word **function**. Next, you need to specify which variable you will output, the name of the function, and the name of the variables that you input. In the screenshot, the function is called `add_and_square` and takes two inputs, `input1` and `input2`. You should be able to see that the code adds these two variables together and squares them, then saves the result as a new variable called `output`. The last line of the script **must** be the word `end`. The script must be saved with the same name as the function, so this script would have to be saved as `add_and_square`. You can then call it from the command line by typing, for example, `add_and_square(1,3)` which will return the value 16.

**Note:** When you are call a function, the only variables that it knows about are those listed as inputs. So, if we had some other variable called `input3`, then the function would not know about it and we couldn’t use it in the script. Similarly, the only things that exist after the function has run are the variables that are listed as outputs. So once we leave the function, the

intermediate variable `input_sum` will disappear.

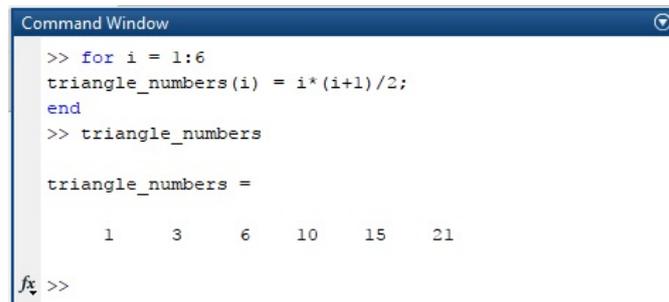
### Exercise 6.

Write a function, called `grid_distances`, that takes the home co-ordinates and location co-ordinates as inputs, and then outputs a vector that contains the distance from home to all the different locations. You have already written the code that does this, so just transfer it over into a new script. Modify your previous code to use `grid_distances`.

**Challenge:** Can you write `grid_distances` so that it computes the distances in just one line, without creating the intermediate variables `distRows` and `distCols`?

## 4 Using loops

We will now introduce `for` loops. The idea of a `for` loop is that you repeat the same set of lines a pre-determined number of times. Each time you go round the loop, the code executes in a slightly different way depending on the value of something called the *counting* variable. The counting variable normally increases by one every time the loop goes round.



```
Command Window
>> for i = 1:6
    triangle_numbers(i) = i*(i+1)/2;
end
>> triangle_numbers

triangle_numbers =

     1     3     6    10    15    21

fx >>
```

In example here, the first line tells `MATLAB` to loop over the counting variable `i`, which should start at 1 and then go through 2, 3, 4, 5, 6. The syntax `i = 1:6` means go through all the integers between 1 and 6, one by one.<sup>2</sup> The next line says that the `i`-th entry of the vector `triangle_numbers` should be given by the formula `i*(i+1)/2`, which is the formula for triangle numbers. The loop is closed using the `end` command. After the loop has finished executing, you can see that the vector `triangle_numbers` is as expected.

**Note:** If we wanted to carry on adding entries to `triangle_numbers` we could do this by writing

```
triangle_numbers = [triangle_numbers, 28]
```

---

<sup>2</sup>In general, this syntax would be `for i = start:gap:stop`, which starts at `start`, and then on each loop increments by `gap` until we get to a number bigger than `stop`.

which tells MATLAB to replace the vector `triangle_numbers` with a new vector which is made up of the old one, plus a new entry 28.

#### 4.1 Designing a fairer algorithm

So far we have chosen the meeting point that is closest to your house. Now we will design a (fairer?) algorithm that minimises the total distance that you and your friend would have to travel.

##### Exercise 7.

Let's say that your friend lives at a house with row co-ordinate 7 and column co-ordinate 4. Modify your script to include their home, along with yours, in a matrix called `homeLocs`. Then, use your `grid_distances` function in a loop to work out the distance from each location to each home. Your code should work something like this:

- For each home in `homeLocs`
  - Compute the distance to each meeting point using `grid_distances`
  - Add this to the total distance
- Then use `min` to work out which meeting point has the shortest possible distance

Now add in a third and fourth friend, who live at row 3, column 1 and row 4, column 4. Also add in a new meeting place, the park, at row 1, column 4. Run the script and see what the best location is.

##### Challenges:

- One way of measuring the efficiency of code is to count how many lines it needs. How short can you make your code?
- (harder) Let's consider a slightly different problem. You want to meet with lots of your friends (say 20 of them) and you want to choose which persons house you should meet at, again defined by the lowest total travelling distance. Can you write a script that does this?
- (open-ended) How long did the above script take to run, and how does that time scale as you increase the number of friends? Designing the most efficient algorithm for this problem is a deep question. Read this and see if you can design a faster code based on the ideas here.

## Practice questions

- Create a function that ...
  - takes a vector and returns the average of all the values in the vector
  - takes two points in the plane  $(x_1, y_1)$  and  $(x_2, y_2)$  and computes the Euclidean distance between them:  
 $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ .
  - computes the Euclidean distance between two vectors of length  $n$
  - takes the radius and length of a cylinder and returns its volume and surface area
  - takes a vector and sorts it into descending order
  - takes two inputs, a vector and a string, and sorts the vector into ascending order if the string is **ascend**, and descending order if the string is **descend**. If the string is not either of these things, return a custom error telling the user what they should have done.
  - takes an integer  $n$  as input, and returns the smallest integer  $m > 1$  such that  $n$  is divisible by  $m$ . If  $n$  is not an integer, the function should return an error.
- Write a program that works that computes the distance matrix between a series of locations, using some of the functions you created earlier. The program should then output the pair of locations that are closest to each other.
- Modify the above program so that the user also inputs an integer  $n$ , and the program outputs  $n$  pairs of locations starting with the two that are closest together.
- Modify all of the above programs to work in the taxi-cab norm (a different way of measuring distance)

# Math0086 practical classes

## Week 2: First-order Euler methods

Contact: sean.jamshidi.16@ucl.ac.uk

The forward Euler method is the simplest routine for solving first order differential equations. The idea is to approximate the derivative using what's known as a *finite difference method*. Specifically, using something called a *first order forward differencing scheme*. Recall the definition of the derivative:

$$\frac{dy}{dt} = \lim_{\Delta t \rightarrow 0} \frac{y(t + \Delta t) - y(t)}{\Delta t}.$$

Thus it seems reasonable that, for small  $\Delta t$ , one could write

$$\frac{dy}{dt} \approx \frac{y(t + \Delta t) - y(t)}{\Delta t}.$$

Consider the initial-value problem

$$\frac{dy}{dt} = f(t, y) \quad \text{with} \quad y(t_0) = y_0.$$

Using the approximate expression for the derivative, we can write

$$y(t + \Delta t) = y(t) + \Delta t f(t, y(t))$$

so the value of  $y$  at a time  $t$  is enough to (approximately) determine the value of  $y$  at a time  $t + \Delta t$ .

The forward Euler method iterates the above equation. Writing  $y_n$  for the approximate solution at time  $t_n = t_0 + n\Delta t$ , we get

$$\begin{aligned} y_0 &= y(t_0) && \text{(given),} \\ y_1 &= y(t_0 + \Delta t) = y_0 + \Delta t f(t_0, y_0), \\ y_2 &= y(t_0 + 2\Delta t) = y_1 + \Delta t f(t_0 + \Delta t, y_1). \end{aligned}$$

and so on. At each stage, the approximate solution  $y_n$  is computed based on data from *the previous time-step only*. In **MATLAB**, we will store the solution as a vector

$$\mathbf{y\_sol} = [y_0, y_1, y_2, y_3 \dots]$$

so that the  $j$ -th entry of  $\mathbf{y\_sol}$  contains the approximation to the solution at  $t_{j-1}$ .

**Exercise 1.**

Write a script that uses the forward Euler method to solve the ODE

$$\frac{dy}{dt} = 2y, \quad y(0) = 1 \quad (1)$$

over the range  $0 \leq t \leq 1$  using the time-step  $\Delta t = 0.05$ . Plot your solution.

**Exercise 2.**

For equation (1) it is possible to compute a solution by hand. This is known as an *analytical solution* or an *exact solution*, while the solution that we computed using Euler's method is known as a *numerical solution*. Compare the numerical and analytical solutions by plotting them on the same axes (you will need to use the command `hold on` to do this).

## Properties of the error

**Exercise 3.**

Compute numerical solutions using various step-sizes  $\Delta t$  and show graphically that the forward Euler method for this problem converges to the exact solution as  $\Delta t \rightarrow 0$ .

Notice that the solutions are not exactly the same however small you make  $\Delta t$  – solving a differential equation numerically will **always** introduce an error. A lot of the theory of numerical differential equations aims to reduce this error.

**Exercise 4.**

On a new set of axes, plot the difference between the analytical and numerical solutions as a function of  $t$ . This is known as the local error. How does the local error change as  $t$  increases?

**Exercise 5.**

Now use the forward Euler method to solve the equation

$$\frac{dy}{dt} = -50y \quad (2)$$

with the initial condition  $y(0) = 1$  over the interval  $0 < t < 1$ . Use the time-step  $\Delta t = 0.05$  and compare your numerical solution to the exact solution...Why does this go wrong? Use the 'stability' section of the lecture notes to find a more appropriate value of  $\Delta t$ .

**Exercise 6.**

Plot the local error in the numerical solution for this value of  $\Delta t$ . Also, compute the error at the final time-step for various values of  $\Delta t$ . How is this different to (1), and why?

## The global error

**Exercise 7.**

The forward Euler method is called a first order method because the difference between the exact and numerical solutions at a given time-step is proportional to  $\Delta t$  (as opposed to, say,  $\Delta t^2$ ). Verify the error at  $t = 1$  is  $O(\Delta t)$  by comparing the error for several values of  $\Delta t$ . What is the constant of proportionality for each equation, and how could a bound on this be derived analytically?

## The backward Euler method

The backward Euler method is similar to the forward one, but has different stability properties.

**Exercise 8.**

Use the backward Euler method to solve the two differential equations from the previous section. Be careful in how you compute  $y(t + \Delta t)$  – you need to do some re-arranging of the algorithm.

**Exercise 9.**

Investigate the stability of both equations under the backward Euler method analytically, then verify this numerically.

## Further use of the Euler method(s)

**Exercise 10.**

Adapt the forward Euler method to solve the differential equation

$$\frac{dy}{dt} = \sin(ty) \quad \text{with} \quad y(0) = \pi/2 \quad (3)$$

over the range  $0 \leq t \leq 6\pi$ . In this case, no analytical solution exists, so how can we trust our numerical solution?

**Challenge:** Solve equation (3) using the backward Euler method. For now, use `fsolve` (an automated routine for solving nonlinear equations using Newton's method) to get  $y(t + \Delta t)$  at each time-step.

**Challenge:** How can you verify that forward Euler is first-order accurate, given that no analytical solution exists for (3)?

Solving higher-order differential equations using finite difference methods often involves writing the equation as a system. For example, if our equation was

$$\frac{d^2y}{dt^2} + 3\frac{dy}{dt} - \cos(y) = 0 \quad (4)$$

we would write

$$\frac{d}{dt} \begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} y' \\ \cos(y) - 3y' \end{pmatrix}.$$

Equivalently, we introduce a function  $v = y'(t)$  and solve the system

$$\frac{d}{dt} \begin{pmatrix} y \\ v \end{pmatrix} = \begin{pmatrix} v \\ \cos(y) - 3v \end{pmatrix}.$$

The advantage of this is that we can apply forward Euler (or any method) to the derivatives on the left-hand side of the equation, and then solve for  $y(t + \Delta t)$  and  $v(t + \Delta t)$  at each time step.

**Exercise 11.**

Solve equation (4) using forward Euler over the range  $0 \leq t \leq \pi$  with the initial conditions  $y(0) = 0, y'(0) = \pi/2$ . Plot your solution  $y(t)$ .

**Exercise 12.**

Investigate the *critical points* of the system (4) and plot your numerical solution in the  $(y, y')$  phase plane. What initial conditions are needed to approach a different critical point?

# Math0086 practical classes

## Week 3: Other methods for ODEs

Contact: sean.jamshidi.16@ucl.ac.uk

There are many other finite difference methods, each with their own stability and convergence properties. All of the methods discussed below are listed in the lecture notes.

### 1 Second-order equations

Using exactly the same Taylor expansion as in forward-Euler, an approximation for the second derivative is

$$\frac{d^2y}{dt^2} = \frac{y(t_2) - 2y(t_1) + y(t_0)}{\Delta t^2}. \quad (1)$$

That is, to compute the solution at  $t$  we require data about the solution at the two previous time-steps. Thus, this is an appropriate method to solve a second-order *initial-value problem*, where we are given  $y$  and  $dy/dt$  at some  $t = t_0$ .

#### Exercise 1.

Use the forward-Euler method to solve the second-order equation

$$\frac{d^2y}{dt^2} = \frac{dy}{dt} - y \quad (2)$$

over the range  $0 < t < 2\pi$  along with the initial conditions  $y(0) = 0$ ,  $y'(0) = 1$ . Find the analytical solution and use it to investigate stability and the order of the method.

### 2 The leapfrog method

#### Exercise 2.

Solve the equation

$$\frac{dy}{dt} = \frac{\sin(2t) - 2ty}{t^2} \quad (3)$$

on the interval  $1 \leq t \leq 2$  with the initial condition  $y(1) = 2$ . Use the *leapfrog* method, thinking carefully about how you compute the solution at the first grid-point.

**Challenge:** Without using the analytical solution, can you show that the method is second-order?

### 3 Crank-Nicolson

Crank-Nicolson is another second-order accurate method, one which is unconditionally stable for a certain class of problems. The idea is to approximate the right-hand side by an average of two different time levels:

$$\frac{y(t + \Delta t) - y(t)}{\Delta t} = \frac{1}{2} [f(t, y(t)) + f(t + \Delta t, y(t + \Delta t))]. \quad (4)$$

But how do we compute  $f(t + \Delta t, y(t + \Delta t))$  when we don't know  $y(t + \Delta t)$ ? One way is to use forward Euler to approximate  $y(t + \Delta t)$  and then use that in the Crank-Nicolson formula as follows:

$$y^0(t + \Delta t) = y(t) + \Delta t f(t, y(t)) \quad (\text{Forward Euler})$$

$$y^1(t + \Delta t) = y(t) + \frac{\Delta t}{2} [f(t, y(t)) + f(t + \Delta t, y^0(t + \Delta t))] \quad (\text{Crank-Nicolson})$$

#### Exercise 3.

Implement this version of Crank-Nicolson for the equation

$$\frac{dy}{dt} = \frac{t}{y}, \quad y(t = 0) = 3. \quad (5)$$

#### Exercise 4.

Another option is to keep iterating equation (4) to get successive approximations for  $y(t + \Delta t)$ . That is, use forward Euler to compute a first approximation, use that in (4) to get a second approximation, then use *that* in (4) to get a third approximation and so on. Implement a method that repeats this until the difference between successive approximations is less than  $\epsilon = 0.01$ .

**Challenge:** You can also write an implicit version of Crank-Nicolson, where the right-hand side of (4) is solved numerically using a Newton solver. Compare the speed and accuracy of each of these three methods.

### 4 Higher-order methods

**Exercise 5.**

One of the most widely used schemes is the fourth-order Runge-Kutta method (commonly known as RK4). Use RK4 to solve (3) and show that the method is, indeed, 4th order.

**Exercise 6.**

There are many built-in ODE solvers in MATLAB, the most popular of which is called `ode45`. This is based on the RK4 method but uses an ‘adaptive time-step’ to vary  $\Delta t$  and aid stability. It can be used as a benchmark to test your own routines against. Use `ode45` to solve the *Lorenz equations*

$$\frac{dx}{dt} = 10(y - x) \tag{6}$$

$$\frac{dy}{dt} = -xz + 28x - y \tag{7}$$

$$\frac{dz}{dt} = xy - \frac{8}{3}z \tag{8}$$

Use the initial conditions  $\{x, y, z\} = \{1, 1, 1\}$  and integrate up to  $t = 100$ . Plot your solution in three dimensions using `plot3`.

Change the initial condition for  $x$  to 1.00001 and run your code again. What has happened to the solution at  $t = 100$ ? Why is this surprising?

# Math0086 practical classes

## Week 5: Explicit methods for PDEs

Contact: sean.jamshidi.16@ucl.ac.uk

Analytical solutions to partial-differential equations are far scarcer than they are for ODEs, so understanding the properties of numerical schemes becomes even more significant.

There are many ways to solve PDEs numerically. This worksheet explores explicit (also called time-marching) finite difference methods for PDEs in one spatial variable ( $x$ ) and one time variable ( $t$ ).

### 1 Forward-differencing scheme

We will consider a simple PDE: the one-dimensional wave equation. Its simplicity allows us to comprehensively analyse both the equation itself and any numerical method that we apply to it, and so it is a valuable tool for learning about the properties of PDEs. We can think of it as being a model for vibrations on a string. The wave equation is:

$$\frac{\partial}{\partial t}u(x, t) = \frac{\partial}{\partial x}u(x, t), \quad (1)$$

along with the initial data  $u(x, 0) = f(x)$ . The analytical solution is just  $u(x, t) = f(x + t)$ . That is, the initial data propagates through space, with unit speed, to the left.

The first approach is to approximate both derivatives in (1) using a first-order scheme, as in forward Euler. We write

$$\frac{\partial}{\partial t}u(x_0, t_0) \approx \frac{u(x_0, t_0 + \Delta t) - u(x_0, t_0)}{\Delta t} \quad (2)$$

$$\frac{\partial}{\partial x}u(x_0, t_0) \approx \frac{u(x_0 + \Delta x, t_0) - u(x_0, t_0)}{\Delta x}. \quad (3)$$

Substitution of these into (1) provides us with the algorithm

$$u(x_0, t_0 + \Delta t) = u(x_0, t_0) + \frac{\Delta t}{\Delta x} (u(x_0 + \Delta x, t_0) - u(x_0, t_0)). \quad (4)$$

Notice how, just like in forward Euler, knowledge of the solution at  $t = t_0$  allows us to compute the solution at the next time  $t_0 + \Delta t$ . Since we have the initial condition  $u(x, 0) = f(x)$  we can compute  $u(x, \Delta t)$ , and then repeatedly apply (4) up to some final time  $t = T$ .

**Important note:** A major issue with finite-difference methods for PDEs is how to impose boundary conditions. The PDE (1) is valid on the infinite domain  $-\infty < x < \infty$ , so for an analytic solution we might not worry about boundary conditions. However, for our computations we will of course restrict  $x$  to some finite range. The algorithm in (4) runs into trouble when we try and compute  $u(x_N + \Delta x, t)$  where  $x_N$  is the last grid-point. We need to make a decision about how the boundary condition will be approached. In our case, the solution is a wave propagating to the left and so we can presume that the value of  $u(x_N + \Delta x, t) = 0$  for all  $t$ .

### Exercise 1.

Write a code that uses (4) to solve (1) on the domain  $0 \leq t \leq 1$ ,  $-5 \leq x \leq 5$  subject to the initial conditions  $f(x) = \exp(-30x^4)$ . For the right-hand boundary condition, assume that  $u(5 + \Delta x, t) = 0$ . Use step sizes  $\Delta t = \Delta x = 0.01$ .

Your solution should be a matrix, with each row corresponding to the solution at a certain time. You can visualise it using the command `waterfall` or by writing an animation function that uses `pause`. Compare your results to the exact solution.

### Exercise 2.

Change the size of  $\Delta t$  and  $\Delta x$  to explore stability. What happens if  $\Delta t > \Delta x$ ? How could this be predicted in terms of characteristics?

## 2 Applying boundary conditions

Now suppose that the equation (1) was posed on a finite domain, with boundary conditions at either end. Suppose that there is a ‘wave-maker’ at  $x = 5$ . That is, we start from flat initial conditions  $f(x) = 0$  and instead wiggle the right-hand end at  $x = 5$  in a sinusoidal way. That is, we impose

$$u(5, t) = \sin(t). \tag{5}$$

### Exercise 3.

Apply the above boundary condition and run your code up to  $t = 4\pi$ . Since we know the value of  $u$  at the last grid-point  $x_N$ , we can use forward-differencing for the spatial derivative to compute  $\partial u / \partial x$  at every grid-point

where the solution is unknown (*ie* all grid-points apart from the last one). What happens if you apply both the initial condition  $f(x) = \exp(-30x^4)$  and the boundary condition?

Other problems involve ‘Neumann boundary conditions’, where  $\partial u/\partial x$  is specified along the boundary. These can be directly applied by changing an entry in the spatial discretisation of (4).

**Exercise 4.**

Solve the wave equation with initial conditions  $f(x) = \exp(-30x^4)$  and the Neumann boundary condition

$$\frac{\partial u}{\partial x} = 0 \tag{6}$$

at  $x = 5$ , which insists that the solution be flat at the right-hand side.

The method applied here seems fairly robust, but for more complicated PDEs it will quickly break down.

**Exercise 5.**

Apply the forward in time and space method to solve Burger’s equation

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0 \tag{7}$$

with the initial condition  $f(x) = \exp(-x^4)$ . You can set  $u = 0$  at the right-hand boundary condition, and run your code up to  $t = 1$ . Investigate whether you can make the code stable.

Burger’s equation is a difficult problem to solve numerically because it can develop a shock from smooth initial data. The forward-Euler method is particularly bad at handling shocks because it cannot represent the propagation of short waves—something that we will investigate next time.

# Math0086 practical classes

## Week 6: More explicit methods

Contact: sean.jamshidi.16@ucl.ac.uk

This worksheet explores several explicit methods for solving hyperbolic PDEs, and introduces some important ideas that can help in choosing the best method.

### 1 Von-Neumann stability

The Von Neumann method is one way of classifying the stability of an explicit method for a linear PDE. The basic idea is to decompose the solution into *Fourier modes*, and examine the growth/decay of a typical mode. It was shown in lectures that, for a first-order method, stability for the wave equation  $u_t + au_x = 0$  is equivalent to the CFL condition  $|a|\Delta t/\Delta x < 1$ .

#### Exercise 1.

Using the code from the previous sheet, show that the forward-Euler method is unstable when the CFL condition is not satisfied. Use the function `fft` to perform a Fourier decomposition and show that the growth is fastest for shorter wavelengths.

### 2 Up-wind methods

Recall Burger's equation:

$$\frac{\partial u}{\partial t} + \alpha u \frac{\partial u}{\partial x} = 0. \quad (1)$$

We saw in the previous exercises that we could not produce a stable solution to this equation when using the forward Euler method when  $\alpha = 1$ .

#### Exercise 2.

Solve Burger's equation with  $\alpha = 1$  using a first-order method where the discretisation is forward in time, and backward in space. Use the initial condition  $u(x, 0) = \exp(-x^4)$  and apply  $u = 0$  at the left-hand end of the domain. Your code should now be stable.

Which method (forward or backward in space) works when  $\alpha = -1$ ? Why?

### 3 Numerical dispersion and dissipation

For ODEs, it is generally true that higher-order truncation methods are ‘better’<sup>1</sup>. For PDEs, there is an extra layer of subtlety that is related to the *parity* of a method’s order. Odd-order methods introduce a phenomenon called numerical *dissipation*, and even-order methods introduce numerical *dispersion*. We will now investigate these.

Suppose that we approximate the  $x$ -derivative using a second-order discretisation.

$$\frac{\partial}{\partial x}u(x_0, t_0) = \frac{u(x_0 + \Delta x, t_0) - u(x_0 - \Delta x, t_0)}{2\Delta x}. \quad (2)$$

Note that now we need to apply boundary conditions at either end of the domain. For now, let us consider *periodic boundary conditions*, that is we impose  $u(x_0, t) = u(x_N, t)$  for all  $t$ , where the domain is  $x_0 < x < x_N$ .

#### Exercise 3.

Solve the wave equation  $u_t = u_x$  with initial data  $f(x) = \exp(-30x^4)$  using the second-order method for the  $x$ -derivative only, and periodic boundary conditions. Compare your solutions to those that were obtained using the first-order method.

What you are seeing here is *numerical dispersion*; the initial condition is breaking up into different Fourier modes, which are travelling at different speeds. The error in the first-order method is dominated by *numerical dissipation*, which is not so important for smooth initial conditions.

#### Exercise 4.

Solve Burger’s equation with  $\alpha = 1$  using backward space differencing and the initial condition  $u(x, 0) = H(x)$ , where  $H(x)$  is the Heaviside step function. The solution is stable, but becomes smoother over time. This is numerical dissipation in action.

#### Exercise 5.

Solve the wave equation using first-order differencing and the initial condition  $u(x, 0) = \tanh(10x)$ . Compare your results to the solution with the same initial condition and the central space-differencing method (impose  $u_x = 0$  at both ends for this). Do you think either one is the correct solution?

---

<sup>1</sup>Of course, higher order methods are more complicated to code and take longer to run, so there are diminishing returns from going to ever-higher orders. For most applications, a clever 4th-order method is sufficient

**Exercise 6.**

Design a method that is appropriate for solving the heat equation,  $u_t = u_{xx}$ . Use the initial condition  $u(x, 0) = \text{sech}^2(x^4)$  and apply no-flux conditions  $u_x = 0$  at the domain ends. Justify your choice of method, and how you have applied the boundary conditions.

Many higher-accuracy methods involve taking several smaller steps and combining them all into one larger step. For example, for ODEs the RK4 algorithm takes 4 partial steps and then combines them. We can apply the same idea to PDEs. One such method is called the Matsuno method. For an equation

$$\frac{du}{dt} = F(u)$$

the Matsuno method first computes an approximation  $\tilde{u}$ , and then uses that to calculate the value of  $u$  at the next time step:

$$\begin{aligned}\tilde{u} &= u^n + \Delta t F(u^n) \\ u^{n+1} &= u^n + \Delta t F(\tilde{u}).\end{aligned}$$

**Exercise 7.**

An advantage of the Matsuno method is that it damps short waves, which can remove many of the problems that we saw earlier. Use the Matsuno method and centred space differencing to solve the wave equation with initial conditions given by  $H(x)$ . Use a Fourier decomposition to show that short waves from the initial conditions have been damped. This is known as selective dissipation.

# Math0086 practical classes

## Week 7: Implicit methods

Contact: sean.jamshidi.16@ucl.ac.uk

The previous exercise sheet explored explicit (time-marching) methods for solving PDEs. In an explicit method, the solution at a grid-point  $(x_0, t_0)$  cannot depend on other values at the same time-step. That is, we use an algorithm where  $u(x_0, t_0)$  is expressed in terms of  $u(x_i, t_0 - \Delta t)$  for some  $x_i$ .

An implicit method uses an algorithm that cannot be written in this way. Instead, we solve a system of equations at time  $t_0$  in order to get the value of  $u$  at any point  $x_0$ . Implicit methods are thus more costly to program, but they have significant stability advantages.

We will first consider the heat equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

with the initial conditions  $u(x, 0) = u_0(x)$  and boundary conditions that  $u \rightarrow 0$  as  $|x| \rightarrow \infty$ . This is a model for any diffusive process such as diffusion of a chemical in the atmosphere, or the spreading of heat through an object.

### 1 Backward Euler

Consider a method based on using backward Euler in time and second-order central differencing in space.

$$\begin{aligned}\frac{\partial}{\partial t} u(x, t) &\approx \frac{u(x, t) - u(x, t - \Delta t)}{\Delta t} \\ \frac{\partial^2}{\partial x^2} u(x, t) &\approx \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{\Delta x^2}.\end{aligned}$$

Although this looks similar to forward Euler, the difference is that the spatial derivative is written in terms of unknown variables  $u(x, t)$ , as opposed to  $u(x, t - \Delta t)$ .

Re-arranging the heat equation to separate known and unknown variables, we have

$$\left( \frac{1}{\Delta t} + \frac{2}{\Delta x^2} \right) u(x, t) - \frac{1}{\Delta x^2} (u(x + \Delta x, t) + u(x - \Delta x, t)) = \frac{u(x, t - \Delta t)}{\Delta t}.$$

This makes clear the challenge of an implicit method. Given a solution  $u(x, t - \Delta t)$  we must solve a system of linear equations to find  $u(x, t)$ .

**Exercise 1.**

Write down the matrix associated with this method. That is, write the problem in the form

$$A\mathbf{u} = \mathbf{b}$$

where  $\mathbf{u}$  is the vector of unknowns, the values of  $u(x, t + \Delta t)$  at the grid-points  $x_1 < x_2 < x_3 \cdots < x_N$  for a given  $t$ . The right-hand side vector  $\mathbf{b}$  should be a vector consisting of contributions from  $u(x, t)$  and from the boundary conditions. For now, set the boundary conditions to be homogeneous ( $u = 0$ ).

We can now compute the problem by inverting the matrix  $A$  at each time-step. That is, given data  $\mathbf{u}^m$  at time  $t$ , the solution at time  $t + \Delta t$  is

$$\mathbf{u}^{m+1} = A^{-1}\mathbf{b}^m$$

where the vector  $\mathbf{b}^m$  is computed using  $\mathbf{u}^m$ .

**Exercise 2.**

Write a code that solves the heat equation using this method and initial conditions  $u(x, 0) = \text{sech}^2(10x)$ . You can invert the matrix using the backslash operator (`\`).

## 2 Methods of inverting large matrices

The main issue with implicit methods is the computation time needed to invert the matrix. When more spatial dimensions are added, this becomes even more costly.

**Exercise 3.**

For the backward Euler algorithm applied to the heat equation, the matrix  $A$  is tri-diagonal. Write a code that implements the Thomas algorithm to invert  $A$ . Check your results against the first exercise.

For a general (not necessarily tri-diagonal) matrix  $A$ , one approach is to write  $A = L + D + U$  where  $L$ ,  $D$  and  $U$  are the lower-triangular, diagonal and upper-triangular parts of  $A$ . Specialised inversion algorithms can then be used on each part separately. In the Jacobi method, we use the fact that a diagonal matrix is (very) cheap to invert, and write

$$\begin{aligned} D\mathbf{u} &= (D - A)\mathbf{u} + \mathbf{b} \\ \mathbf{u} &= (I - D^{-1}A)\mathbf{u} + D^{-1}\mathbf{b}. \end{aligned}$$

The latter equation is then solved iteratively, ie given some data  $\mathbf{u}^m \approx u(x, m\Delta t)$  we repeat

$$\mathbf{u}_n^{m+1} = (I - D^{-1}A)\mathbf{u}_{n-1}^{m+1} + D^{-1}\mathbf{b}^m,$$

until  $|\mathbf{u}_n^{m+1} - \mathbf{u}_{n-1}^{m+1}| < \epsilon$  for some pre-determined tolerance  $\epsilon$ . We then let  $u(x, (m+1)\Delta t) = \mathbf{u}_n^{m+1}$ , update  $A$  and  $b$  if necessary and begin iteration again for the next time-step.

**Exercise 4.**

Implement the Jacobi method for solving the backward Euler system at each time step. You should make use of the function `diag` to generate the matrix  $D$ . As a starting point for iteration, you could use the solution at the previous time-step. Use `while` to keep a loop running until the difference between iterations is less than  $\epsilon = 0.01$ . Record how long (on average) it takes to solve the system using `tic` and `toc`, and plot how this varies with the size of the matrix.

The next logical step is the Gauss-Seidel method. This additionally inverts the lower triangular part of the matrix;

$$\begin{aligned}(L + D)\mathbf{u} &= -U\mathbf{u} + \mathbf{b} \\ \mathbf{u}_n &= -(L + D)^{-1}U\mathbf{u}_{n-1} + (L + D)^{-1}\mathbf{b}.\end{aligned}$$

**Exercise 5.**

Compare performance of the Jacobi and Gauss-Seidel methods. You can use the functions `tril` and `triu` to generate the matrices  $L$  and  $U$ .

# Math0086 practical classes

## Week 8: Finite element for an ODE

Contact: sean.jamshidi.16@ucl.ac.uk

This worksheet applies the finite element method to ODEs. We begin with the second-order ODE

$$\frac{d^2T}{dx^2} + 2 = 0 \quad 0 < x < 6, \quad T(0) = 0, T'(6) = -3. \quad (1)$$

### Exercise 1.

Write down the exact solution to this equation and derive the system of equations, in terms of basis functions  $\phi_i$ , that needs to be solved for the finite element method.

### Exercise 2.

For piecewise linear basis functions on the grid  $[0, 2, 4, 6]$ , what is the mass matrix  $A$ ? What about for an arbitrary grid of evenly spaced points? Write a code that generates the matrix  $A$  for an evenly spaced grid.

### Exercise 3.

Use your code to solve the problem, and observe convergence to the exact solution as the grid-size  $\Delta x \rightarrow 0$ .

### Exercise 4.

Adapt your code to solve the equation  $T_{xx} + x = 0$ .

Often in the finite element method, the right-hand side and the boundary conditions are considered as “data”. That means it is assumed that they are supplied by the user of the code, so the programmer should write code that works with any sensible inputs. On the other hand the equation is not data, and the programmer writes code knowing what equation will be used.

### Exercise 5.

Can you adapt your code so that it works for an arbitrary right-hand side  $f(x)$ , if  $f(x)$  is supplied by the user as a function? What about for arbitrary boundary conditions? You will need some way of telling your algorithm whether to apply Neumann or Dirichlet conditions.

# Math0086 practical classes

## Week 9/10: 2D finite element method

Contact: sean.jamshidi.16@ucl.ac.uk

We will use the finite element method to solve the equation

$$\nabla^2 u = 1 \quad (1)$$

in the region  $x^2 + y^2 \leq 1$  along with the boundary conditions

$$\frac{\partial u}{\partial n} = \frac{1}{2}, \quad u = \frac{1}{4} \quad \text{on } x^2 + y^2 = 1. \quad (2)$$

The exact solution for this problem is  $u = r^2/4$ . For the finite element method, we consider the problem in the weak formulation

$$\iint_D \nabla u \cdot \nabla v \, dA = \int_{\partial D} v \frac{\partial u}{\partial n} \, ds - \iint_D v \, dA \quad (3)$$

and seek a numerical solution of the form

$$u = \sum_{i=1}^{i=N} \alpha_i \phi_i + \frac{1}{4} \quad (4)$$

where  $\phi_i$  are piecewise-linear basis functions on a triangular mesh. That is, we mesh the domain into a series of triangles (elements), and for each node  $i = 1 \dots N$  we define  $\phi_i = ax + by + c$  to be 1 at node  $i$  and 0 at all other nodes. Note that  $\phi_i$  is defined element-wise; the constants  $a, b, c$  take different values on each elements. The  $+1/4$  is needed in order to satisfy the Dirichlet boundary condition as (3) does not include any information about this. Instead, we use the finite element method to seek a solution  $u$  that is zero around the boundary and then just add  $1/4$ .

We take  $v = \phi_j$  in (3) and create the linear system

$$\sum_{i=1}^{i=N} \alpha_i \iint_D \nabla \phi_i \cdot \nabla \phi_j \, dA = \int_{\partial D} \frac{1}{2} \phi_j \, ds - \iint_D \phi_j \, dA, \quad j = 1 \dots N \quad (5)$$

which we write as

$$\mathbf{A}\mathbf{u} = \mathbf{b} \quad (6)$$

and then seek to implement in MATLAB for arbitrary grid spacing.

**Exercise 1.**

Download the package `distmesh` from here. Type `meshdemo2d` for a series of demos. Read through the `distmesh` documentation, and identify which of the demo examples produces a triangular mesh on the unit circle.

**Exercise 2.**

Write code to solve (5). It is helpful to break your code up into the following sections:

- Create a basis function  $\phi_i$  for each node that is not on the boundary. To do this, go through all the elements  $k$  that contain node  $i$  and then solve a 3x3 linear system to find the coefficients for the function  $\phi_i$  on the element  $k$ .
- Compute the entries of the matrix  $A$ . We can integrate over each individual element and then sum the total. An element only contributes to  $a_{ij}$  if it contains both nodes  $i$  and  $j$ . Use the geometry of the mesh, and the linearity of  $\phi$ , to help you.
- Compute entries of the right hand side vector. The area integral is just the volume of the tetrahedron made by  $\phi$ , and the line integral has a similar geometric interpretation.
- Solve for the coefficients, and visualise the solution  $u$  by plotting the coefficients in 3D over the mesh. Remember to add  $1/4$  to your solution!

**Exercise 3.**

Compute a numerical solution to the same problem using centred finite differences. Which code takes longer to run, for the same grid-size?