MATLAB - An Introduction for Mathematicians

Scott Morgan

October 12, 2016

Introduction - The Matlab Interface

- Matlab is a computational software package originally written by numerical analyst Clive Moler in the 1970s.
- Unlike other programming languages used in numerical computations like Fortran and C++, Matlab makes use of highly developed external libraries and functions which allows you to relieve yourself of many of the mundane tasks associated with programming. For example, Matlab has built in functions that can calculate matrix inverses, products and eigenvalues. All of these things would have to be hard-coded into a Fortran program, eating up time that could be spent on more important issues.
- With a very readable and easy to navigate interface, Matlab is a great starting tool for anyone interested in programming.
- Although it has many qualities, Matlab is not a perfect package for all computational tasks and what it gains in usability it loses on computational time. A program written in Fortran or C++ will run a lot faster in general than the same program written in Matlab, which may be of concern for very large programs. This, however, is most certainly *not* of concern here, and all of the programs written in this course should take only a fraction of a second to run on a workstation.
- It should be noted that these notes are designed only as a companion to the classes which they accompany - a student should not expect to gain any insight from these notes without attending class.
- E-mail corrections and/or comments are always welcome and can be directed to `MorganSN@cardiff.ac.uk`

Installing Matlab

- A student Matlab subscription is currently £29[1] and is highly worth the spend. However, if you are currently a student at Cardiff then you can get the academic license key for installation on your home computer for free. The download and install process is described in detail on the Intranet at
  `https://intranet.cardiff.ac.uk/students/support-and-services/technical-help-and-support/software-support/software-for-home-use/matlab`
  Note that you will need to be logged into the Intranet to access this page. There is also a thorough guide to installation from MathWorks, available at
  `http://uk.mathworks.com/help/install/ug/install-mathworks-software.html`

- Alternatively, you can use Matlab at any of the workstations around the university, available through *Cardiff Apps*, or you can use GNU Octave.

- GNU Octave is free, open-source software which can run many Matlab commands. However, as with everything in life, you get what you pay for and thus, Octave can be buggy, slow and is lacking certain features that may make your life a misery.

- Installing Octave on Windows can also be a pain, although it is described in detail on the web at `https://www.gnu.org/software/octave/download.html`.

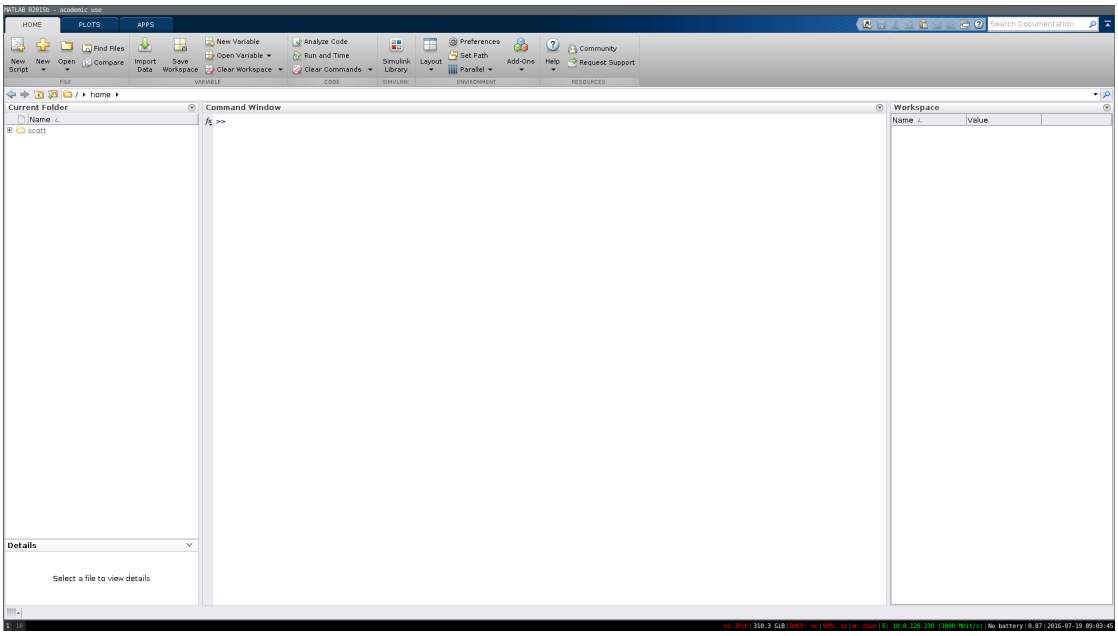
- Installing Octave on Ubuntu is easy and can be accomplished with

```
$ sudo add-apt-repository ppa:octave/stable
$ sudo apt-get update
$ sudo apt-get install octave
```

[1] Correct as of 19/07/2016

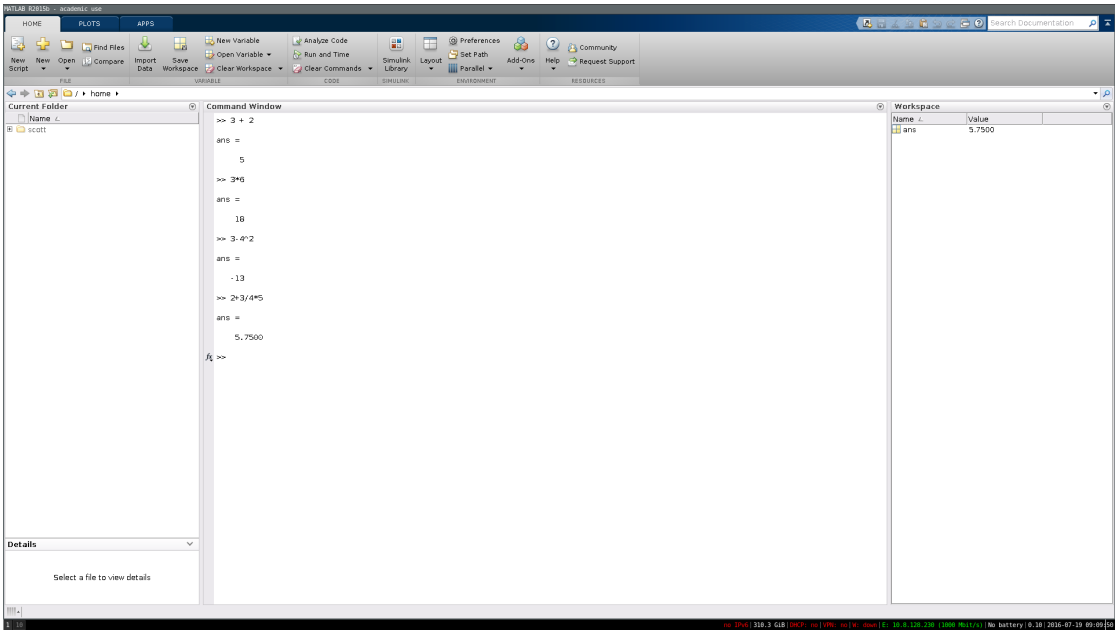
Using Matlab for the first time

- On opening Matlab for the first time you should be presented with a window similar to this:



- The main window in the centre is the *command window* and contains the *command prompt*. You can enter commands directly at the command prompt and Matlab will give some sort of output. The left hand side shows the current directory and the folder structure. This will be important later.

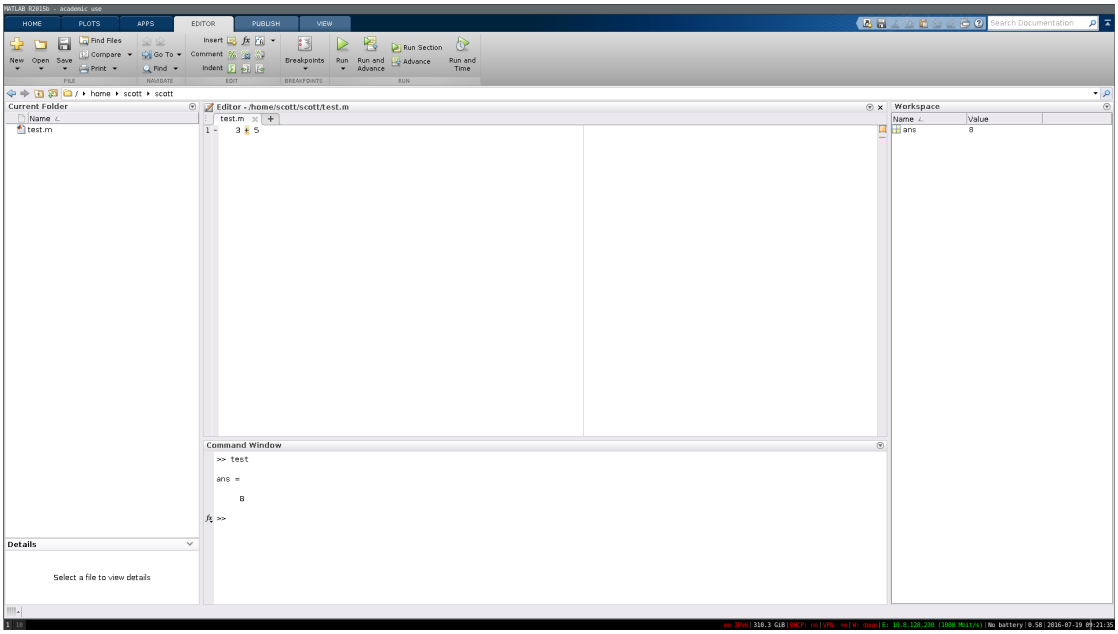
- Calculations can be carried out in the usual way and follow the usual rules for ordering of operations. The figure below shows some example commands. After running these commands, you should note now that the *workspace* on the right hand side contains a new value. This *ans* variable takes the value of the last entered command - in this case, 5.75.



- While in theory you can work entirely in the command window, in practise this will get very untidy very quickly and so most of the calculations we will do will be inside what are called *scripts*.
- Scripts are basically text files which can be *called* from inside the command window. Follow the procedure below to create a script.
  - In the top left corner of the screen, click the yellow plus sign above the word *new*.
  - Click script.
  - A window called *Editor* should now appear above the command window, with the script named as *untitled*.
  - To use the script, type a calculation into the Editor and save the file *in the current directory* as *test.m*
  - The script is then *called* in the command window by writing

```
>> test
```

at the prompt. The figure below shows the process.



- Now that you know the basic ideas behind creating a script and running commands in Matlab, you can start to have a play around with the interface. Matlab has lots of built in functions like *sin, cos, tan, exp, log, abs, pi, sqrt* and many more. The best resource by far is Google! If you think Matlab may have a built in function that you want to use then chances are it does.

- The command

```
>> demo
```

is a great place to start learning about what everything does.

Remarks

- Before we start getting into the basics of coding, there are a few remarks to make.
  - There is more than one way to write a program! People have *styles*, as you'll begin to see soon. If your program is different to mine then that's not a problem - you will develop your own way of doing things, your own way of naming things and as long as you're consistent with yourself then it won't matter. Someone once described it to me as learning a language, but with an accent - we can all understand each other but we say things in a slightly different way! It's like that with programming, don't be worried if your program looks different - if the results are the same it shouldn't matter too much.
  - *However*, that being said, you should always strive to write code that is *readable by another person*! You should always write comments explaining what complicated lines do and write your algorithms in a coherent way. This is not exclusive to your code being readable by someone else either - if you write a code that is not properly commented and don't look at it for a few months, you will not have any idea what it does the next time you look at it! This has happened to me so many times that I am determined to not let you make that mistake.
- A *comment* in Matlab is achieved with the % sign:

```
>> 3 + 4; %this is a comment - it won't alter the result
```

- Using a semi-colon will *suppress* the output of the command - and the answer will not be printed to the command prompt.

Basic Coding Ideas

Variables

- You can assign a value to a *variable* in the following way

```
>> x = 3; %assigns the value 3 to the variable x
```

This will give the variable x the value 3, which can then be used in future calculations such as

```
>> x = 3; %assigns the value 3 to the variable x
>> 3 + x %uses the assigned value in the calculation 3+x
ans =
    6
```

These values will remain until they are overwritten or *cleared*. You can clear specific variables using the *clear* command.

```
>> x = 3;
>> 3 + x
ans =
    6
>> clear x;
```

Trying to use the variable x again will result in an error.

- Unlike many other programming languages such as VB, Fortran or C++, there is no need to *declare* variables to have a certain *type* in Matlab. Matlab

will automatically figure out whether you are specifying an integer, a real number or a complex number.

- Variables can be named any combination of letters and numbers both lower case and upper case but must start with a letter.
- You can see the variables which are currently stored by Matlab in the *workspace*. Alternatively, you can see the same information by running the command

```
>> whos
```

Loops

If there is nothing else you take from this document, let the thing you learn be how to use *for* loops and *if* statements. These are central to most programs you will write during this course and usually play an important role in many other programs.

The very simple example below illustrates what these statements do.

```
for j = 1:10
    disp(j) %display the value assigned to j at each point in the loop
end

for j = 1:10
    if j == 5
        disp(j + 3) %if j = 5, display the value 8 — i.e. 5+3
    elseif j == 6
        disp(j + 1) %if j = 6, display 7 — i.e. 6+1
    else
```

```
        disp(j) %if j is not equal to 5 or 6, display j
    end
end
```

Remarks on Loops

- You can use any variable you like as a loop counter name, although most people use i or j. However, care must be taken when doing this. Matlab recognises i and j to mean complex i unless they are user-defined first (i.e. at the start of a loop). Therefore, standard Matlab practise is to use $1i$ as complex i and allow you to change the value of i in a loop. If you want to input a complex number in Matlab, you should write it like this

```
>>> 2 + 1i
```

- Inside an if statement you can use several identities to check for conditions. The main ones are
  - == (is equal to)
  - $\sim$= (is not equal to)
  - >=, > (is greater than or equal to, is greater than, similar for less than)
  - You can combine conditions with && (and) and || (or).
- There is another type of loop called a *while* loop. These are similar to for loops but are particularly advantageous when you don't know how many iterations you need to end the loop. An example is below and one will need to be used in the tasks that follow.

```
j = 1; %initialsise the variable j to have value 1 – why?
while j <= 10 %loop until j reaches 10
    disp(j) %output j at each step
    j = j + 1; %add one to j and loop again
end
```

Task

Write a script that will loop through the numbers 1 to N and print out all numbers that are a multiple of 3.

Hint: You may find the following useful.

```
>> help mod
```

Extension: Extend the program so that it reproduces the game *Fizzbuzz*. Have the code display the word *Fizz* when it encounters a multiple of 3, *Buzz* on a multiple of 5 and *FizzBuzz* on a multiple of both 3 and 5.

Task

Write a script that will check the Collatz[a] conjecture for some number N.

[a] This is an open problem - no one has proved it for all $N \in \mathbb{N}$

The Collatz conjecture goes like this:

1. *Pick a number.*
2. *If the number is even, halve it.*
3. *If the number is odd, multiply it by* 3 *and add* 1.
4. *Conjecture: the sequence will always reach* 1 *in a finite number of steps.*

Plotting

- There are an overwhelming amount of possibilities that Matlab offers with regard to plotting functions - just describing it could be a course of its own. Therefore, in the interest of concision, I will only outline the absolute basic way of plotting here. You will pick up more points as you play around, but for now just see the following:

```
>> x = linspace(0,1,10); %generates grid of 10 points in [0,1]
>> y = sin(x); %evaluates sin(x) at each of the 10 points
>> plot(x,y) %plots
```

Task - Quick

Plot the graph of $y = \cos(3x)$. Experiment with changing the value of the number of points plotted across. Plot the graph with circles at each point instead of a line using

```
>> plot(x,y,'o')
```

Vectors & Arrays

- Matlab stores vectors in a very intuitive way and there are several ways to create them.

```
>> u = [1 3 6]; %gives row vector containing 1, 3 and 6
>> v = 1:4; %gives row vector containing integers 1,2,3,4
>> w = 1:2:9; %gives row vector containing 1,3,5,7,9
>> for j = 1:4
        p(j) = j+1; %gives row vector containing 2,3,4,5
   end
```

- The elements of the vector can be accessed by the notation $v(j)$, where v is your variable name and j is the number of the index you want to access. Note that Matlab indexing *starts* at 1. It is never possible to access $v(0)$.

- Therefore, we can get the value 9 from w by typing

```
>> w(5)
ans = 9
```

- Column vectors can be defined as transposes of row vectors or with the notation

```
>> c = [1; 3; 5]; %note semi-colons. Row vectors are r = [1 3 5];
```

- The transpose operation is

```
>> c' %this will trasnform a row into column and vice-versa
```

Task

Write a script that will print out the first N Fibonacci numbers.

Extension: Have a play around with the following command:

```
%Display a header line. num2str changes with the value of N.
disp(['The first ' num2str(N) ' terms are:']);
```

to format your output so that it's easily readable.

Task

Use your previous Fibonacci program to plot the convergence of the ratio of successive terms to the golden ratio.

It can be shown that $\frac{F_{n+1}}{F_n} \to \frac{1+\sqrt{5}}{2} =: \varphi$ as $n \to \infty$. You program should plot both the ratio calculated and the value φ on the same axis.

Task

Use your previous Collatz program to plot the route of the sequence to its final value 1.

Task - Difficult

Write a script that will count and list the number of primes less than a given number N. This is called $\pi(N)$.

Extension: Compare your answer with the estimates $\pi(N) \approx N/log(N)$ and $\pi(N) \approx Li(N) := \int_0^N \frac{dt}{ln(t)}$. *Hint: Matlab has a built in function for Li called logint. Hint: To check that a number P is prime, you only need to check divisibility by numbers up to $\sqrt{P}$*

Matlab Specific Coding Ideas

Functions

- Until now, everything you have written has been in the form of a *script*. While that is useful, many programs will require the use of *functions*.
- A function contains parts of a code that need to be called in several places, to save you having to repeat code several times. The syntax and an example is below

```
function c = myfunction(a,b) %function inputs a,b – outputs c
    c = a*b; %multiplies a and b – stores in c
end
```

- The function is then called from the command prompt as

```
>> c = myfunction(3,2)
c =
    6
```

- This function should be written in a script and saved with the *same name as the function* in the current directory.

Task

Go back through your programs and change them into functions, with N as an input wherever possible.

Matrices

- One of Matlab's huge advantages over its competitors is the way it handles operations with matrices.
- There are some immensely powerful tools that make many matrix calculations simple in Matlab that are very difficult to implement in some other programming languages.
- For example, Matlab has inbuilt commands for matrix multiplication, inversion, calculating eigenvalues, solving systems of equations, creating diagonal matrices, determinants and many many more. Before you go any further, think of what it would take to write a program for any of these things. Could you write a code that calculates the eigenvalues of any given matrix?
- Matrices are formed in Matlab to a very similar way to vectors, and simply take two indices in an intuitive way

```
for i = 1:N
    for j = 1:N
        A(i,j) = %some condition on the entries of A
    end
end
```

- The elements of A are then accessed similarly to vectors. For example, $A(1,3)$ will give you the entry in the 1st row and 3rd column of A.
- Some standard commands are explained below. If there is a command which I've missed that you think should exist - chances are it probably does. Google is the answer!

```
>> det(A); %determinant
>> inv(A); %inverse
>> A'; %transpose
>> A*B; %multiplication (note #columns of A = #rows of B)
>> A.*B; %Hadamard (entrywise) multiplication
>> ones(10); %creates a 10x10 matrix of ones
>> zeros(3,4); %creates a 3x4 matrix of zeros
>> diag(1:4); %a 4x4 diagonal matrix with 1 to 4 on the diagonal
>> eye(3); %the 3x3 identity matrix
>> rand(3,2); %a 3x2 matrix of random numbers
```

Task - Quick

Write a script that will create an $N \times N$ identity matrix using for loops and if statements. Check against eye(N).

Task (Matrix Multiplication Algorithm)

This is an exercise that will enable you to multiply two $N \times N$ matrices i.e. for a given matrix $A = a_{ij}$ and $B = b_{ij}$ write a code that computes $C = A \times B$

(i) Using the *randi* function generate two $N \times N$ matrices A and B and initialize a third $N \times N$ matrix $C1$ using *zeros*

(ii) Write a *for* loop to compute $A \times B$ **without** using MATLAB's matrix multiplication function

Systems of Linear Equations

- Matlab has a particularly powerful tool for solving systems of equations called *mldivide*, more commonly known simply as *backslash*.

- Given an matrix A and a right hand side vector b, the solution vector x which solves $Ax = b$ is calculated simply by

```
>> A = rand(10); b = rand(10,1);
>> x = A\b;
```

- Of course for this example the following would be equivalent

```
>> A = rand(10); b = rand(10,1);
>> x = inv(A)*b;
```

  but this is considered bad practise as there is no need to calculate the inverse to solve the problem - leading to an increase in computational time. If you try to solve the problem in this way, Matlab will shout a warning.

- Backslash can also be used to solve over- and under- determined systems of equations in a similar way and uses least-squares fitting to do so. It is a very powerful tool which has made Matlab very popular for matrix operations.

- The command

```
>> help mldivide
```

  will give a good explanation of the methods used and the best practice usage of this operator.

Task

Solve the system of equations

$$3x + 4y + 7z = 32$$

$$x + 5y + 2z = 17$$

$$2x + y + z = 7$$

and compare the time taken between backslash and inverse multiplication.

To measure the time taken by a script (in this case called *test.m*), use the following

```
>> tic, test; toc
```

- Hopefully the following example will convince you to use backslash!

```
>> A = diag(rand(5000,1)); b = rand(5000,1);
>> tic, x = inv(A)*b; toc
Elapsed time is 30.292078 seconds.
>> tic, x = A\b; toc
Elapsed time is 0.040788 seconds.
```

Eigenvalues & Eigenvectors

- Another important feature of Matlab's matrix capabilities is its ability to calculate eigenvalues and eigenvectors of matrices.
- Given a matrix A, the eigenvalues and eigenvectors can be calculated simply

using

```
>> [V,D] = eig(A);
```

- This effectively solves the problem

$$Av = \lambda v$$

and stores the vectors v in V and the corresponding eigenvalues λ along the diagonals of the diagonal matrix D.

- Time-permitting, we may revisit this later in the course, but for now it is useful to just know that it exists.

Some More Advanced Ideas - with Applications in Applied Mathematics

- The following section will be organised into four/five distinct parts. We may not have time to cover them all, but they will be included for reference anyway.

  1. Numerical integration via the trapezium rule.
  2. Solving linear ODEs with inbuilt functions.
  3. Newton's method for root-finding.
  4. Simple finite difference solutions to ODEs.
  5. A brief introduction to spectral methods.

- Note that this course is NOT intended to be about the mathematical justification of the methods involved. We will likely be sloppy with assumptions

and not get too heavily involved in the analysis. Take me at my word for now that these methods are valid - the MAGIC course on Numerical Analysis as well as the MMath course on fluid dynamics will reinforce the ideas presented here if the interested student would like to know more.

Numerical Integration

- Matlab has a useful inbuilt function for calculating integrals via the trapezium rule called *trapz*. Example usage can be found via

```
>> help trapz
```

Task

Write a function which will integrate some input function y over a user-defined interval $[a, b]$. Your function should compare the results with the exact answer and make it simple to alter the function being integrated if required.

Experiment with the accuracy obtained by altering the number of grid points taken.

Solving linear ODEs

- To solve an ODE numerically, we can use Matlab's built in tools called *ode23* and *ode45*. These can be used to solve equations of the form

$$\frac{dy}{dt} = f(t); \quad y(y_0) = t_0$$

or even higher order linear ODEs and systems of ODES. Typing

```
>> help ode45
```

will give you plenty of information about the implementation of this routine, but I have included a full working example below to illustrate the usage. The code below solves the equation

$$\frac{dy}{dt} = \cos(t); \quad y(0) = 2$$

on the interval $[0, 2\pi]$.

```
%Initialise solution parameters
a = 0; b = 2*pi; %Interval end points
y0 = 2; %Initial condition

[t23,y23] = ode23(@func,[a b],y0); %Solve using ode23

function f = func(t,y)
    f = cos(t); %right hand side.
end
```

Task

Implement the routine above and also include the solution from *ode45*. Compare the solutions to the exact solution $y = 2 + \sin(t)$. What do you notice? Which is more accurate?

- You can also use *ode23/45* to solve second or higher order linear ODEs if you reformulate them as linear systems of ODEs first. Assume we have

$$y'' + \exp(t)y' + ty(t) = 3\sin(2t)$$
$$y(0) = 2; \quad y'(0) = 8$$

in the interval $[0, 4]$. Then we can let $x_1 = y$ and $x_2 = y'$ and write this as

$$\begin{aligned} x_1' &= x_2 \\ x_2' &= -tx_1 - \exp(t)x_1 + 3\sin(2t) \\ x_1(0) &= 2; \quad x_2(0) = 8 \end{aligned}$$

- We can then solve this system in a similar way to the first order case except now we will get a *vector* of solutions. A full working script is below

```
function ode

%Initialise solution parameters
a = 0; b = 4; %Interval end points
x0 = 2; x1 = 8; %Initial conditions

[t23,x23] = ode23(@func,[a b],[x0,x1]); %Solve using ode23

%Plots
plot(t23,x23(:,1),'b');
xlabel('t'); ylabel('y(t)'); xlim([min(t23) max(t23)]);

end

function f = func(t,x)
    f = zeros(2,1); % since output must be a column vector
    f(1) = x(2);
    f(2) = -t*x(1)-exp(t)*x(2)+3*sin(2*t);
end
```

- Note that the solution y is stored in $x(:, 1)$, while y' would be stored in $x(:, 2)$.

Task

Implement the routine above for *ode45*. Plot both solutions on the same axis and compare.

Newton's method

- Given an equation $f(x) = 0$, where f is differentiable, we can iterate towards the root x_r by means of Newton's method in the following way

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

for some initial guess x_0. Provided we have a good initial guess, this will produce approximate solutions to any degree of accuracy required. In some cases, Newton's method requires the guess x_0 to be *fairly close* to the true solution - something which is not always easy to do.

Task

Write a code that implements Newton's method for $f(x) = 0$ in the following cases:

- $f(x) = x^2 - 5$ with starting guess $x_0 = 2$.
- $f(x) = \sin(x) + x\cos(x)$ with starting guess $x_0 = 1.5$. Try the same function with $x_0 = 1$. What do you notice? Is this valid?

Your code should be independent of your choice of function. Try to write the Newton method as its own function, and call your input functions separately. You should produce a table of the error in successive iterations and a plot of the convergence. In the first case, plot also the exact solution $x_r = \sqrt{5}$ and check your results. You can assume you have convergence when $|x_{n+1} - x_n| < 10^{-8}$

Finite Differences

- Finite difference methods are a class of solution methods used widely in applications to solve complicated systems of ODEs.

- By the process of discretising an ODE, we essentially replace the problem of finding the exact, continuous solution with the problem of finding the values of the solution at specified discrete points in space and time by deriving and solving an appropriate set of algebraic equations. This discrete problem is only an approximation to the problem, which improves as we increase the number of points we use.

- A crucial step in the finite difference method (FDM) is the approximation of derivatives by finite differences (FD), i.e. the replacement of derivatives by some algebraic formula.

- The method relies on Taylor series expansions and I won't go into too much detail here. However, it does not take much thought to arrive at the following approximation for the derivative of a function F.

$$F'(x_n) \approx \frac{F(x_{n+1}) - F(x_n)}{x_{n+1} - x_n}$$

  which is basically just the GCSE formula $m = \frac{\Delta y}{\Delta x}$. This is called the *forward-difference formula* and is *first-order accurate*. This means that if we increase the resolution by $\frac{1}{10}$ then the accuracy of the solution also increases by $\frac{1}{10}$. This is not considered very accurate, and as such is rarely used.

- The expression can be similarly derived by Taylor series expansions around x_n and it is not much of a stretch to imagine a formula in the opposite direction, namely

$$F'(x_n) \approx \frac{F(x_n) - F(x_{n-1})}{x_n - x_{n-1}}$$

This is called the *backward-difference formula* and is also first-order accurate. These two can be combined into a single, second-order accurate formula

$$F'(x_n) \approx \frac{F(x_{n+1}) - F(x_{n-1})}{2h}$$

where $h := x_n - x_{n-1}$. This is called the *central difference formula.*

- Similar formulae exist for higher order derivatives which we won't derive, but we state the second-order accurate central difference approximation to the second derivative as

$$F''(x_n) \approx \frac{F(x_{n+1}) - 2F(x_n) + F(x_{n-1})}{h^2}$$

An example with Matlab implementation

- Suppose that we want to solve the boundary value problem

$$u'' = f(x), \quad x \in (0,1)$$

where we set

$$f(x) = \pi^2 \sin(\pi x)$$
$$u(0) = 0; \quad u(1) = 1$$

This has exact solution $u_e = x - \sin(\pi x)$ which we will use later to compare results.

- We set up the grid in the x direction in the usual way

```
N = 10;
x = linspace(0,1,N+1); %N+1 domain points (gives spacing h = 1/N)
h = x(2)-x(1); %gives nodal spacing
```

- At each *interior* point x_j, $(j = 1, \ldots, N-1)$ in the domain, we can approximate the derivative u'' using our central difference formula as follows

$$\frac{U_{i+1} - 2U_i + U_{i-1}}{h^2} = f(x_i) \implies U_{i+1} - 2U_i + U_{i-1} = h^2 f(x_i)$$

Thus we can write the resulting algebraic system as (convince yourself this is correct!)

$$\begin{pmatrix} 1 & -2 & 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 0 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} U_0 \\ U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_{N-3} \\ U_{N-2} \\ U_{N-1} \\ U_N \end{pmatrix} = h^2 \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-3} \\ f_{N-2} \\ f_{N-1} \end{pmatrix}$$

- However, we already know the solutions at nodes $U_0 = u(0) = 0$ and $U_N = u(1) = 1$ so the first and last columns may be transferred to the right hand side to give

$$\begin{pmatrix} -2 & 1 & 0 & 0 & \ldots & 0 & 0 \\ 1 & -2 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & -2 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & & & & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & -2 & 1 & 0 \\ 0 & 0 & \ldots & 0 & 1 & -2 & 1 \\ 0 & 0 & \ldots & 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_{N-3} \\ U_{N-2} \\ U_{N-1} \end{pmatrix} = h^2 \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-3} \\ f_{N-2} \\ f_{N-1} \end{pmatrix} - u(0) \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \end{pmatrix} - u(1) \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

- By doing so, we isolate the unknown nodal values U_i, for $i = 1, \ldots, N-1$, on the left hand side, casting the global system of equations in the form $Au = g$.

- To convert this into a MATLAB script, the main difficulty is in the construction of the matrix A. To do this we will use the *diag* command which we introduced earlier. You could also do it with for loops, although there is no need here.

```
% A diagonal matrix with -2 on the diagonal
D0 = -2*diag(ones(N-1,1));

% A matrix with 1 along the 1st superdiagonal
D1 = diag(ones(N-2,1),1); %

% A matrix with 1 along the 1st subdiagonal
D2 = diag(ones(N-2,1),-1);

% Construct the full matrix
A = D0 + D1 + D2;

% The right hand side of the BVP (x(2:N) gives interior nodes)
f = pi^2*sin(pi*x(2:N));

% The right hand side of the linear system
g = h^2*f;
g(1) = g(1) - 0; %not necessary, but illustrative
g(N-1) = g(N-1) - 1; %from boundary conditions
g = g'; %transpose for backslash solver
```

- Finally we are left to solve and plot the solution

```
u(1) = 0; u(N+1) = 1; %sets boundary conditions
u(2:N) = A\g; %solves the system Au = g
plot(x,u)
```

Task

Implement the above code in Matlab. Plot the result against the exact solution and quantify the error. Experiment with altering the value of N. Plot both exact solution and approximate solution on the same axes - use circles to denote the points at which the finite difference approximation is taken.

Task

Use finite difference methods to solve the *convection-diffusion equation*

$$-\epsilon u''(x) + \beta u'(x) = 0; \quad u(0) = 1, \quad u(1) = 0$$

and compare against the exact solution $u(x) = \left(\frac{\exp\left(\frac{\beta}{\epsilon}\right) - \exp\left(\frac{\beta}{\epsilon}\right)}{\exp\left(\frac{\beta}{\epsilon}\right) - 1} \right) x$

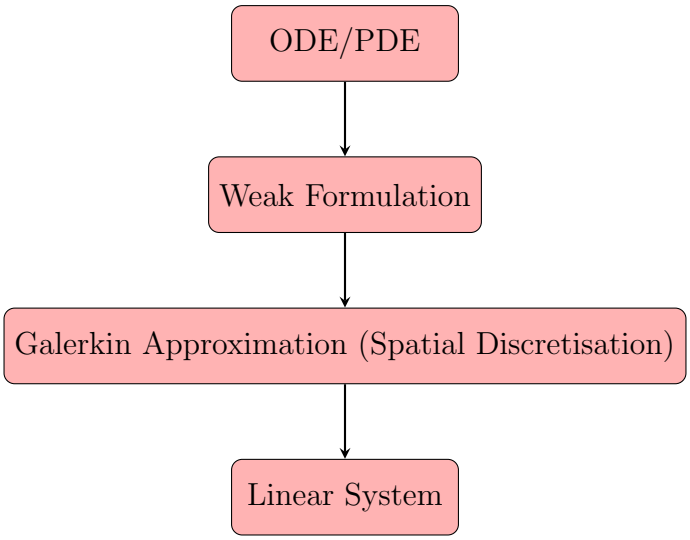
Experiment with changing the values of ϵ and β. Start with $N = 10$ and $\epsilon = 0.1$, $\beta = 0.5$. Then try $\epsilon = 0.01$, $\beta = 0.5$. What do you notice? Can you work out when & why this happens? *Hint: The ratio* $\frac{\beta h}{\epsilon}$ *is important.*

The Finite Element Method

Overview

- The Finite Element Method (FEM) is another numerical method that is used to approximate solutions to boundary value problems (ODEs/PDEs).
- Similar to Finite Differences, the Finite Element method solves boundary value problems by converting differential equations into systems of linear equations
- The solution of the linear systems provide a discrete set of points that the approximate the values of the function at those points
- FEM is extensively used in industrial modelling and engineering and is one of the most well known numerical methods for solving differential equations
- The spatial discretisation permitted by Finite Element modelling is especially useful when solving (2D) problems in irregular shaped domains that are cannot be discretized easily by finite differences)

Finite Elemnt Method Implementation



The Finite Element method can be broken down into 4 discrete stages

- **ODE/PDE** - The Boundary Value Problem needing to be solved
- **Weak Formulation** - Re-writing the problem in terms of *inner products* using *test functions*
- **Galerkin Approximation** - Restrict the weak problem to *piecewise linear* functions
- **Linear System** - Construct the Linear system of equations that represents the Galerkin (linear) approximation

Note: Finite Element Analysis is difficult and requires extensive knowledge of functional analysis and linear algebra. In the context of this course we will avoid going into detail about some of the more technical aspects. If you want to find out more about FEA please let us know.

An Example of 1D FEM implementation

Consider the same problem as discussed in sections ...

$$u''(x) = f(x) \qquad \text{for} \qquad -1 < x < 1 \tag{1}$$

$$u(-1) = 1 = u(1) \qquad f(x) = \pi^2 \sin(\pi x)$$

It is easy to verify that the solution to the BVP (1) is given by $u(x) = 1 - \sin(\pi x)$.

Weak Formulation

- The weak formulation of a BVP is an alternate way of writing our problem using *bilinear forms*
- It requires the use of *test functions* that belong to certain function spaces
- The weak form of the differential equation given by (1) requires the following function spaces

$$V = \left\{ v(x) : \quad \int_{-1}^{+1} |v(x)|^2 + |v'(x)|^2 \, dx < \infty, \quad v(\pm 1) = 0 \right\} \tag{2}$$

$$W = \left\{ w(x) : \quad \int_{-1}^{+1} |w(x)|^2 \, dx < \infty, \quad w(\pm 1) = 0 \right\} \tag{3}$$

First multiply (1) by a test function $v \in V$ and integrate over $[-1, 1]$

$$\int_{-1}^{+1} u''(x) v(x) \, dx = \int_{-1}^{+1} f(x) v(x) \, dx \tag{4}$$

Using *integration by parts* the LHS

$$[u(x)v(x)]_{-1}^{1} - \int_{-1}^{+1} u'(x)v'(x)\,dx = \int_{-1}^{+1} f(x)v(x)\,dx \tag{5}$$

Using the fact that $v(\pm 1) = 0$

$$\int_{-1}^{+1} u'(x)v'(x)\,dx = -\int_{-1}^{+1} f(x)v(x)\,dx \tag{6}$$

Eq. (6) is known as the *weak form* of (1). Using inner product notation $\int_\Omega f g\,d\Omega = (f, g)$

$$(u', v') = -(f, v) \tag{7}$$

The weak form of the equations is then: Find $u \in W$ such that

$$a(u, v) = -l(v) \qquad \forall v \in V \tag{8}$$

where $a(u, v) = \int_{-1}^{+1} u'(x)v'(x)\,dx$ and $l(v) = \int_{-1}^{+1} f(x)v(x)\,dx$.

Galerkin Approximation

- The objective now is to find *piecewise linear* functions that approximate the solution to (8).
- The trial space, V, and test space, W, in the weak formulation are replaced by subspaces comprising of polynomials of degree 1. These subspaces are denoted by V^h and W^h , respectively, i.e. $V^h \subset V$ and $W^h \subset W$.
- As with the case for finite differences divide the interval $[-1, 1]$ into N subintervals (elements) $[x_{k-1}, x_k]$, $k = 1, ..., N$ with $x_0 = -1$ and $x_N = 1$.

Consider a piecewise linear approximation to $u(x)$ and $f(x)$ given by

$$u(x) \approx u^h(x) = U_0\phi_0(x) + ... + U_N\phi_N(x) \tag{9}$$

$$f(x) \approx f_0\phi_0(x) + ... + f_N\phi_N(x) \tag{10}$$

where $\phi_i(x)$ are piecewise linear basis functions (hat functions)

- As in the case of finite differences, the goal is to construct a (linear) system of equations to solve for the unknown nodal values of u^h: U_i $(i = 1, .., N-1)$

Basis Functions

In the kth element there are two local basis functions

$$\psi_{1,k} = \frac{x_k - x_{k-1}}{h} \qquad \psi_{2,k} = \frac{x_{k+1} - x_k}{h}$$

The basis functions $\phi_j(x)$ are then defined

$$\phi_j(x) = \begin{cases} \psi_{j,2}(x) & x_{j-1} \le x \le x_j \\ \psi_{j+1,1}(x) & x_j \le x \le x_{j+1} \end{cases}$$

for interior nodes $j = 1, 2, ..N-1$ and at the boundary nodes x_0 and x_N

$$\phi_1(x) = \begin{cases} \psi_{1,1}(x) & x_0 \le x \le x_1 \\ 0 & \text{otherwise} \end{cases}$$

$$\phi_N(x) = \begin{cases} \psi_{2,N}(x) & x_{N-1} \le x \le x_N \\ 0 & \text{otherwise} \end{cases}$$

- Substituting the approximations (9) and (10) into (8) we get the following

$$a(u^h, v^h) = -l(v^h) \tag{11}$$

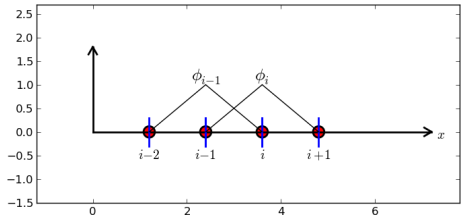


Figure 1: The basis function ϕ_k at $k = i - 1$ and $k = i$. The basis functions are such that $\phi_i(x_i) = 1$ and $\phi_i(x_j) = 0$ $j \neq i$

$$a\left(\sum_{j=0}^{N} U_j \phi_j(x), \phi_i(x)\right) = -\left(\sum_{j=0}^{N} f_j \phi_j(x), \phi_i(x)\right) \tag{12}$$

$$\sum_{j=0}^{N} U_j a\left(\phi_j(x), \phi_i(x)\right) = -\sum_{j=0}^{N} f_j \left(\phi_j(x), \phi_i(x)\right) \tag{13}$$

$$\sum_{j=0}^{N} U_j A_{ij} = -\sum_{j=0}^{N} M_{ij} f_j \tag{14}$$

Matrix Assembly

Equation (14) represents a *linear system of equations* of the form $A\mathbf{u} = -M\mathbf{f}$. We now have to construct the matrices A and M in order to

Local Matrix Construction

- In this 1D example, this means that we have to only consider the interactions of the two element basis functions that are local to the kth element, $\psi_{1,k}(x)$ and $\psi_{2,k}(x)$.
- For a 1D problem the stiffness matrix local to the kth element is the 2×2 matrix defined in terms of the local basis functions by

$$A^{(k)}_{\alpha,\beta} = \int_{x_{k-1}}^{x_k} \psi'_{\alpha,k}\psi'_{k,\beta}\,dx = \frac{(-1)^{\alpha+\beta}}{h} \qquad \alpha,\beta = 1,2 \tag{15}$$

Similarly the local mass matrix is given by

$$M^{(k)}_{\alpha,\beta} = \int_{x_{k-1}}^{x_k} \psi_{\alpha,k}\psi_{k,\beta}\,dx \qquad \alpha,\beta = 1,2 \tag{16}$$

$$\begin{aligned} M^{(k)}_{1,1} &= \frac{1}{h^2}\int_{x_{k-1}}^{x_k} (x_k - x)^2\,dx = \frac{h}{3} \qquad M^{(k)}_{2,2} = \frac{1}{h^2}\int_{x_{k-1}}^{x_k} (x - x_{k-1})^2\,dx = \frac{h}{3} \\ M^{(k)}_{1,2} &= M^{(k)}_{2,1} = \frac{1}{h^2}\int_{x_{k-1}}^{x_k} (x_k - x)(x - x_{k-1})\,dx = \frac{h}{6} \end{aligned} \tag{17}$$

Thus the

$$A^{(k)} = \frac{1}{h}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

$$M^{(k)} = \frac{h}{6}\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

Global Matrix Assembly

- Global Matrices are constructed by cutting and pasting the local 2×2 matrices into $(N+1) \times (N+1)$ arrays using the following stamping procedure

Stamping procedure

$$A_s = \begin{pmatrix} a^{(1)}_{1,1} & a^{(1)}_{1,2} & 0 & \cdots & 0 & 0 \\ a^{(1)}_{2,1} & a^{(1)}_{2,2} + a^{(2)}_{1,1} & a^{(2)}_{1,2} & \cdots & 0 & 0 \\ 0 & a^{(2)}_{2,1} & a^{(2)}_{2,2} + a^{(3)}_{1,1} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a^{(N-1)}_{2,2} + a^{(N)}_{1,1} & a^{(N)}_{1,2} \\ 0 & 0 & 0 & \cdots & a^{(N)}_{2,1} & a^{(N)}_{2,2} \end{pmatrix}$$

where $A^{(k)} = a_{ij}^{(k)}$, $i, j = 1, 2 \quad k = 1, ..., N$

$$M_s = \begin{pmatrix} m_{1,1}^{(1)} & m_{1,2}^{(1)} & 0 & \cdots & 0 & 0 \\ m_{2,1}^{(1)} & m_{2,2}^{(1)} + m_{1,1}^{(2)} & m_{1,2}^{(2)} & \cdots & 0 & 0 \\ 0 & m_{2,1}^{(2)} & m_{2,2}^{(2)} + m_{1,1}^{(3)} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & m_{2,2}^{(N-1)} + m_{1,1}^{(N)} & m_{1,2}^{(N)} \\ 0 & 0 & 0 & \cdots & m_{2,1}^{(N)} & m_{2,2}^{(N)} \end{pmatrix}$$

Similarly $M^{(k)} = m_{ij}^{(k)}$. The global stiffness matrix A is constructed by taking the $N-1 \times N-1$ **interior** matrix of A_s

$$A = \begin{pmatrix} a_{2,2}^{(1)} + a_{1,1}^{(2)} & a_{1,2}^{(2)} & 0 & \cdots & 0 \\ a_{2,1}^{(2)} & a_{2,2}^{(2)} + a_{1,1}^{(3)} & a_{1,2}^{(3)} & \cdots & 0 \\ 0 & a_{2,1}^{(3)} & a_{2,2}^{(3)} + a_{1,1}^{(4)} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{2,2}^{(N-1)} + a_{1,1}^{(N)} \end{pmatrix} = \frac{1}{h} \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 2 \end{pmatrix}$$

M is constructed by taking the $N-1 \times N+1$ interior matrix of M_s

$$M = \begin{pmatrix} m_{2,1}^{(1)} & m_{2,2}^{(1)} + m_{1,1}^{(2)} & m_{1,2}^{(2)} & 0 & \cdots & 0 & 0 \\ 0 & m_{2,1}^{(2)} & m_{2,2}^{(2)} + m_{1,1}^{(3)} & m_{1,2}^{(3)} & \cdots & 0 & 0 \\ 0 & 0 & m_{2,1}^{(3)} & m_{2,2}^{(3)} + m_{1,1}^{(4)} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & m_{2,2}^{(N-1)} + m_{1,1}^{(N)} & m_{1,2}^{(N)} \end{pmatrix}$$

$$= \frac{h}{6} \begin{pmatrix} 1 & 4 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 4 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 4 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 4 & 1 \end{pmatrix}$$

The Galerkin Approximation to the equation is given by the linear system

$$A\mathbf{u} = -M\mathbf{f} + \mathbf{g} \tag{18}$$

$\mathbf{u}$ and $\mathbf{g}$ are the $N-1$ vectors

$$\mathbf{u} = \begin{pmatrix} U_1 \\ U_2 \\ \vdots \\ U_{N-2} \\ U_{N-1} \end{pmatrix}, \qquad \mathbf{g} = \begin{pmatrix} \frac{1}{h}u_L \\ 0 \\ \vdots \\ 0 \\ \frac{1}{h}u_R \end{pmatrix}$$

and $\mathbf{f}$ is the $N+1$ vector

$$\mathbf{f} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}$$

In this particular example $u_L = u(-1) = 1$ and $u_R = u(1) = 1$. The solution is then given by

$$\mathbf{u} = -A^{-1}M\mathbf{f} + A^{-1}\mathbf{g} \tag{19}$$

Implementation in MATLAB

Implementating FEM in MATLAB

1. Initialization (mesh, initial/boundary conditions, variables etc.)
2. Matrix assembly: Form the stiffness and mass matrices by accounting for the contributions coming from
   - the boundary elements
   - the interior elements
3. Solve the resulting system of equations using the backslash operator

- Initialise the variables
- Set boundary values, number of elements and nodes using *linspace*

```
pi = 3.14159265359 % Pi to 11 decimal places
N = 5              % Number of elements
U_L = 1;           % BC at x = -1
U_R = 1;           % BC at x = +1
x = linspace(-1,1,N+1); % Create set of discrete points x_0,x_1,...,x_N
f = pi^2*sin(pi*x);     % The f-vector
```

- Initialise global matrices $\mathbf{A}$, $\mathbf{M}$ and vector $\mathbf{g}$ using *zeros(m,n)*

```
% Initialise Global Matrices and g vector

A = zeros(N-1,N-1); % Initialise the stiffness and mass matrices
M = zeros(N-1,N+1); % and the g-vector
g = zeros(N-1,1);
```

- Specify the outer entries of the stifness and mass matrices
- Use a *for* loop to stamp the local matrices together as shown in Sec.

```
% Global Matrix Contruction

h = diff(x)  % Step length

A(1,1) = 1/h(1);
M(1,1) = h(1)/6;
M(1,2) = h(1)/3;
g(1) = U_L/h(1);



A(N-1,N-1)= 1/h(N);
M(N-1,N) = h(N)/6;
M(N-1,N+1)= h(N)/3;
g(N-1) = U_R/h(N);



% Local matrices without the prefactor h

Ae = [1 -1; -1 1];
Me = [2 1;1 2]/6;

% Loop through elements 2 to N-1 (Stamping procedure)
for k = 2:N-1
    A(k-1:k,k-1:k) = A(k-1:k,k-1:k) + Ae/h(k);
    M(k-1:k,k:k+1) = M(k-1:k,k:k+1) + h(k)*Me;
end
```

- Solve the system for an $N+1$ vector U

```
% Solve the system of equations
U = [U_L;A\(-M*f+g);U_R];  % Solves the linear system A*U = -M*f + g
                           % with boundary conditions U(-1) = U_L,
                           %                          U(1) = U_R
```

- Plot solution data against analytic solution using *plot*
- Create a finer resolution for analytic solution plot (if desired) by defining new set of discrete points t

```
% Analytic solution
t=linspace(-1,1,10000);    % High resolution
u_exact = 1-sin(pi*t);     % Analytic Solution

% Plot the solution
plot(x,U,'ro-');  % Finite Elment Solution
xlabel('x')
ylabel('y')
hold on                % Plot two functions in the same figure
plot(t,u_exact,'b-')   % Plot analytic solution
legend('FEM Solution: u^h(x)','Analytical Solution: u(x)')
```

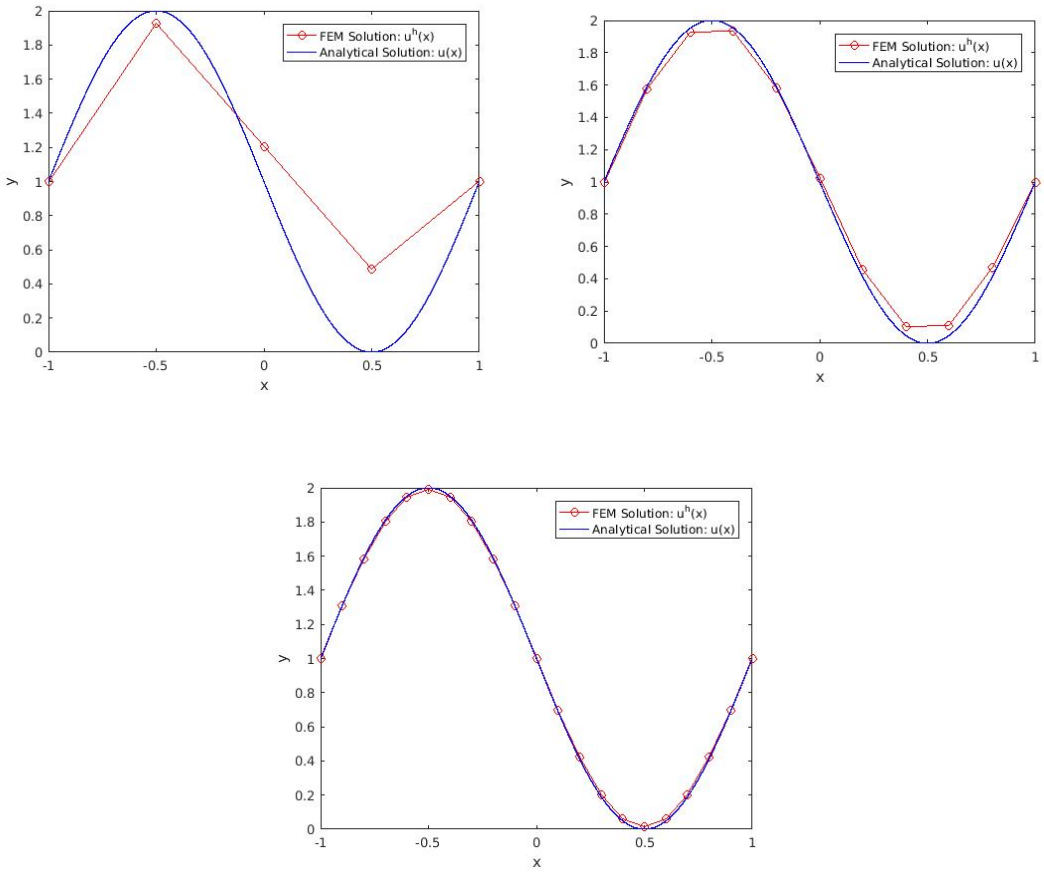


Figure 2: Finite Element method solution (red) for $N = 4$, $N = 10$ and $N = 20$ compared with the true solution (blue) $u(x) = 1 - sin(\pi x)$

Task

By using the example above correct the following FEM code for the solution to the problem

$$\begin{aligned} u''(x) &= f(x) \qquad \text{for} \qquad 0 < x < 2 \\ u(0) &= 1 \quad u(2) = 0 \\ f(x) &= -x^2 \end{aligned} \tag{20}$$

Copy and paste the following code into a MATLAB script and **debug** the code. At each stage errors have deliberately included

```
N = 10;              % Number of piecewise linear elements
U_L = 1;             % BC at x = 0
U_R = 0;             % BC at x = 2
x = linspace(0,2,N+1)';    % Create set of discrete
                           % points x_0,x_1,...,x_N
f = x*x;           % The f-vector

A = zeros(N-1,N-1); % Initialise the stiffness and mass matrices
M = zeros(N-1,N+1); % and the g-vector
g = zeros(N,1);
h = diff(x)  % Step length

% Global Matrix Contruction

A(1,1) = 1/h(1);
M(1,1) = h(1)/6;
M(1,2) = h(1)/3;
g(1) = U_L/h(1);



A(N-1,N-1)= 1/h(N);
M(N-1,N) = h(N)/6;
M(N-1,N+1)= h(N)/3;
g(N) = U_R/h(N);

% Element matrices without the prefactor h

Ae = [1 -1; -1 1];
Me = [2 1;1 2]/6;

% Loop through elements 2 to N-1
for k = 2:N-1
```

```
    A(k-1:k,k-1:k) = A(k-1:k,k-1:k) + Ae/h(k);
    M(k-1:k,k:k+1) = M(k-1:k,k:k+1) + h(k)*Me;
end

% Solve the system of equations
U = [U_L;A\(M*f+g);U_R];    % Solves the system A*U = M*f + g
                            % with boundary conditions U(-1) = U_L,
                            %                          U(1) = U_R

% Analytic solution
t=linspace(-1,1,10000);   % High resolution
u = 1+t.*(1/6.0)-(t.^4/12.0);

% Plot the solution
plot(x,U,'ro-');  % Finite Elment Solution
xlabel('x')
ylabel('y')
hold on           % Plot two functions in the same figure
plot(t,u,'b-')    % Analytic Solution
legend('FEM Solution: u^h(x)','Analytical Solution: u(x)')
```

Task (Difficult)

Write a code to find the FEM solution to the BVP

$$\begin{aligned} u''(x) + u(x) &= f(x) \qquad \text{for} \qquad 0 < x < 1 \\ f(x) &= 2e^x \end{aligned} \tag{21}$$

With boundary conditions

$$u(0) = 1 \qquad u(1) = e$$

Hints: The linear (Galerkin) approximation of $\int_0^1 u(x)v(x)\,dx$ is