AMAT3122

Mathematical Methods for Differential Equations

Numerical Methods and Techniques using MATLAB

Preface

These notes comprise the first half of the numerical part of the course AMAT3122 Mathematical Methods for Differential Equations. They are not a complete set of notes. Extra material and examples may also be presented in the lectures and tutorials.

Using the electronic version of these notes

These notes are hyperlinked. All green text is a link to somewhere else within this document. For example the contents page links to the appropriate page in the text, the numbers in the index link to the page reference, the word Index in the header of most pages links to the index and the page numbers in the header on each page link back to the contents page. There are also some internal linked words that take you to the relevant text. Links to external web pages are red in colour. Provided your PDF reader (eg Adobe Acrobat Reader) is set up correctly these links should open the appropriate page in your web browser (eg Microsoft Internet Explorer, Netscape, Mozilla).

MATLAB code

The MATLAB codes used in these notes are available at the course web site
http://www.ma.adfa.edu.au/Teaching/ThirdYear/mmde.html .
Many of the codes are also linked directly from within the text to the appropriate web page.

MATLAB code in the text is usually shown in purple and the resulting output in blue.

If you have any difficulties with any part of the course do not hesitate in contacting me to sort them out.

Geoff Mercer
Room G12 Science South
School of Physical, Environmental and Mathematical Sciences
Phone: 6268 8734
Email:g.mercer@adfa.edu.au

Contents

1 Introduction to MATLAB

MATLAB (MATrix LABoratory) is a very powerful package designed for numerical analysis, matrix calculations and visualisation (as well as many other applications). One of MATLABs strengths is that it can be used as a simple 'calculator' and also is easy to program more complicated tasks. Many mathematical algorithms are already built in to MATLAB and many more are easily added.

1.1 Access to MATLAB

University College has a limited user licence for the full version of MATLAB, this is available via octarine. Alternatively you can purchase MATLAB (from the Co-op Bookshop), this is a student release. The most recent version of MATLAB is 6.5, Release 13. Older version of MATLAB had a student edition which is more than adequate for this course (and is often all that is ever needed for many applications).

1.2 Using MATLAB

MATLAB is an interactive program so you can enter commands at the prompt or read in files with lines of MATLAB commands. For relatively simple tasks it is acceptable to enter commands at the prompt but for more complicated tasks it is better to store the commands in a file and then read in the file. This makes it easy to change the commands and to store them. In fact all inbuilt MATLAB commands are written as files of commands. For this reason there are vast numbers of MATLAB files (called M-files) written for various purposes available on the internet.

For this course you should do your work using M-files as these will be handed up as part of your assignments along with the relevant output.

All M-files given in the notes will be available from the course web page at
http://www.ma.adfa.edu.au/Teaching/ThirdYear/mmde.html.
Many of the assignment questions will be a modification of these files.

1.3 A few simple examples

1. **Matrix Manipulation**

   Enter a 2×2 matrix find its inverse, and eigenvalues and eigenvectors. Check that its inverse times itself is the identity.

   MATLAB code

```
% any thing after a % sign is a comment
A=[1,2;3,4]     % enter the 2x2 matrix and call it A
B=inv(A)        % calculate the inverse of A and call it B
eig(A)          % calculate the eigenvalues of A
[V,D]=eig(A)    % calc the eigenvectors (columns of matrix V)
                % diagonal matrix D (eigenvalues on diagonal)
check=B*A
```

   If these commands are stored in an M-file called `example1.m` then to run it type `example1` (without the `.m`) from within MATLAB. The MATLAB output from these commands is the following

```
A =

     1     2
     3     4

B =

   -2.0000    1.0000
    1.5000   -0.5000

ans =

   -0.3723
    5.3723

V =

   -0.8246   -0.4160
    0.5658   -0.9094

D =

   -0.3723         0
         0    5.3723

check =

    1.0000         0
    0.0000    1.0000
```

2. **Plotting**

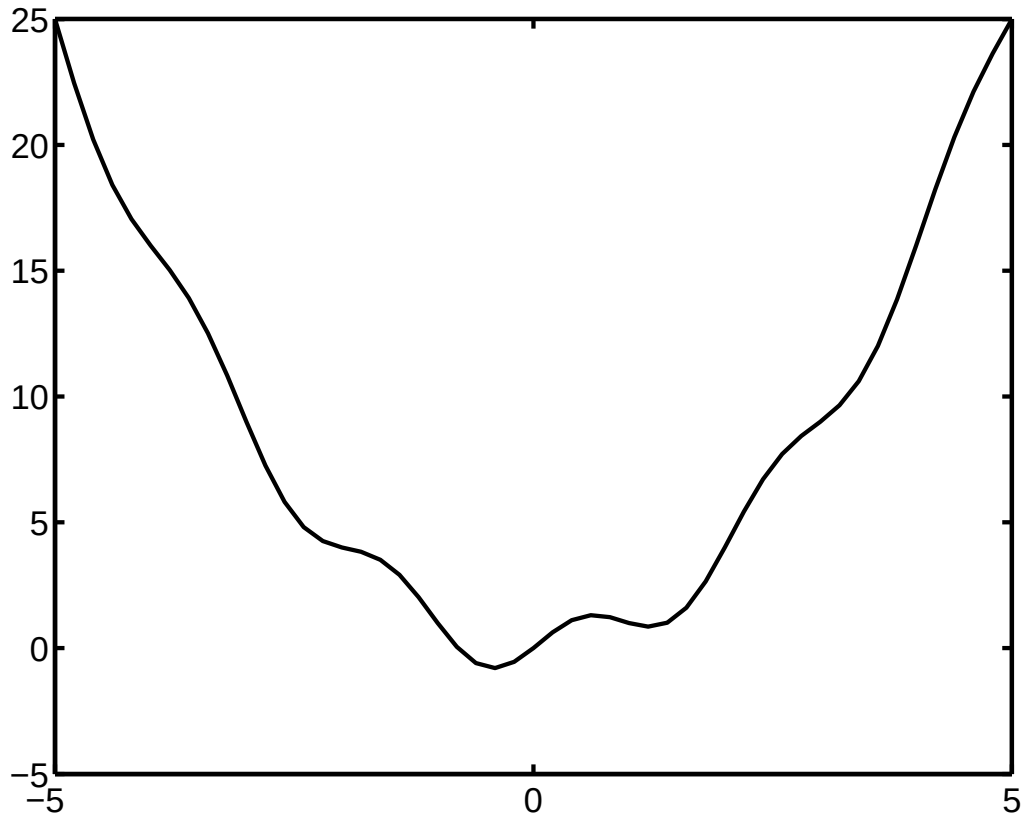
MATLAB has very good graphics capability. The easiest method for simple plots is to define a vector (for example x) and then perform a function on each element of that vector (eg $y(x)$) and then plot $y(x)$ versus x.

Defining a vector is straightforward. The MATLAB command `x=a:b:c` defines x to be a vector of values starting at a increasing in steps of b to c. For example `x=-4:3:8` results in x being the vector $[-4,-1,2,5,8]$.

Example. Plot $y = x^2 + \sin(x\pi)$.

MATLAB code

```
x=-5:0.2:5;        % define a vector x with values
                   % from -5 to 5 in steps of 0.2
                   % the semicolon suppresses the output
y=x.*x+sin(x*pi); % calculate y=x*x+sin(x*pi), note the .*
                   % this is because x is a vector so we need
                   % to multiple 'component by component'
plot(x,y)
print -deps example2  % send the plot output to a
                      % postscript file called example2.eps
```

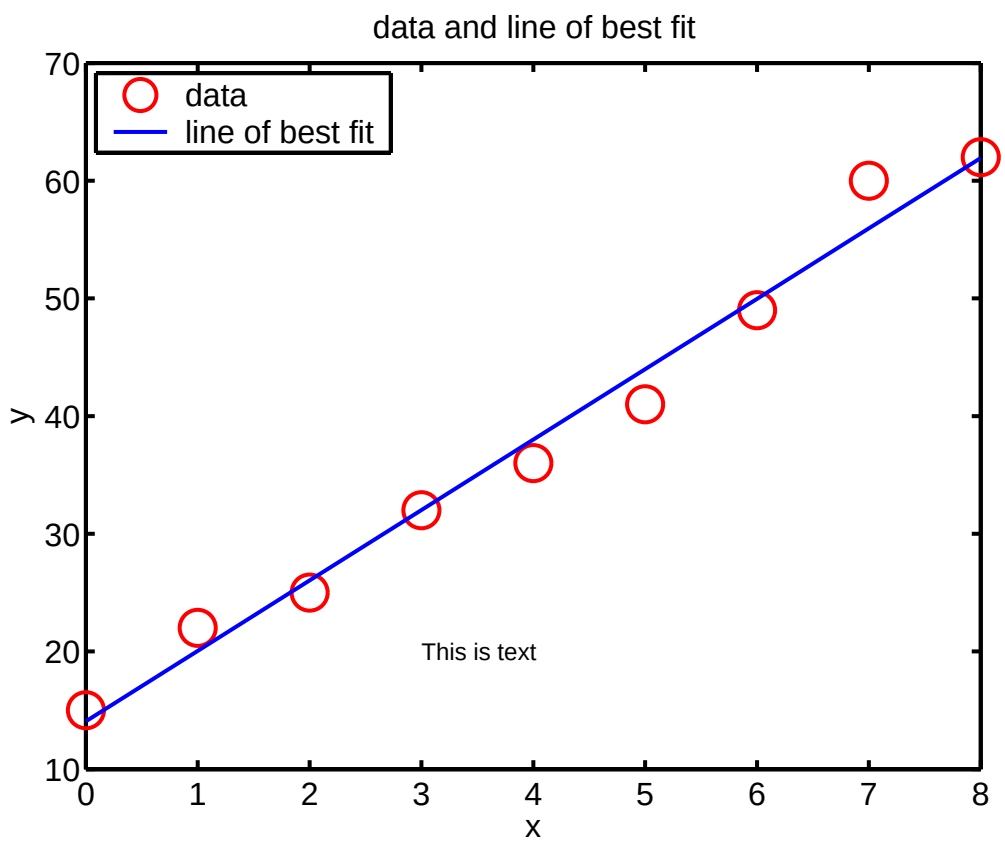


3. **Plotting data**

It is very easy to read data in from a file, manipulate it and plot it. The command `load` is used to load data from an external file. For example read data from a file called datafile, calculate the line of best fit to the data and add some titles, legend and text to the plot.

MATLAB code

```
% read data from the file datafile.  This puts it into
% a variable matrix called datafile by default
load datafile
x=datafile(:,1)   % put first column into variable x
y=datafile(:,2)   % put second column into variable y

% Fit a linear regression line to the data
t=[ones(size(x)) x];
coeffofregn=t\y
straightline=[ones(size(x)) x]*coeffofregn;

% Plot the data and line and add titles and labels
plot(x,y,'ro',x,straightline,'b-')
title('data and line of best fit')
xlabel('x')
ylabel('y')

% add some text to the plot and a legend (the number is
% where to put the legend,2 is the top left corner)
text(3,20,'This is text')
legend('data','line of best fit',2)
```



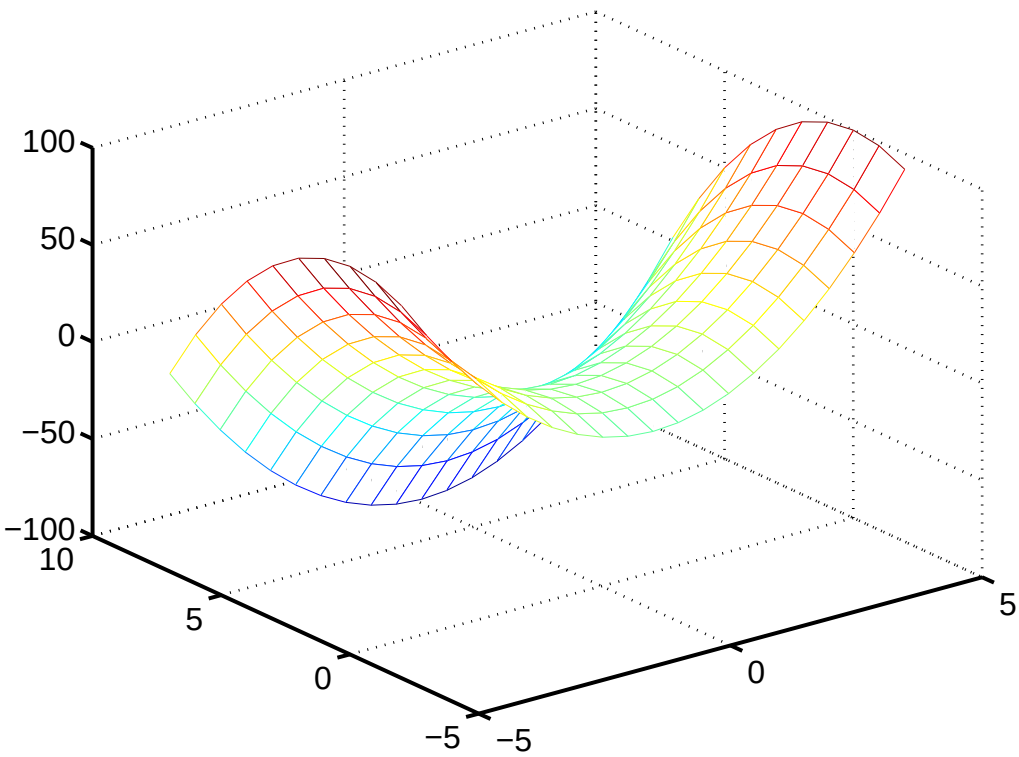
4. **3D Plotting**

MATLAB has very good 3D graphics capability. The easiest method is do define the domain using the `meshgrid` command, then define the function and then plot it using `mesh` or `surf` commands.

Example. Plot $z = 4x^2 - 2y^2$.

MATLAB code

```
[x,y]=meshgrid(-5:0.5:5,-2:1.0:7);
   % define the mesh (x,y)  the domain of plotting
   % x from -5 to 5 in steps of 0.5
   % y from -2 to 7 in steps of 1.
z=4*x.*x-2*y.^2;
   % calculate z=4x^2-2y^2
   % note the .* and .^ this is because x and y are
   % vectors so we need to multiple 'component by component'
mesh(x,y,z)
   % draw the surface as a mesh
   % or use  surf(x,y,z)  to draw it as a surface
print -depsc example2a
   % send the plot output to a colour postscript file
```



5. **Solving Linear Equations**

The original reason for developing MATLAB was to perform matrix calculations. Due to this it has many convenient ways of dealing with matrices. For example to solve a linear equation system $Ax = b$ there is a command known as 'backslash divide'.

MATLAB code

```
A=[1,2,3;4,5,6;7,8,10];  % define the matrix A
b=[1,1,1]';              % note the ' to take transpose
                         % or could do b=[1;1;1]
x=A\b                    % solve for x
```

```
x =

   -1.0000
    1.0000
    0.0000
```

6. **Extracting Segments of Matrices**

Often you may want to extract sections of a matrix. For example all of the first row or the second column.

MATLAB code

```
A=[1,2,3;4,5,6;7,8,10];
c=A(1,:)          % extract all of the first row of A
d=A(:,2)          % extract all of the second column of A
```

```
c =
     1     2     3


d =
     2
     5
     8

diary off
```

1.4 Running MATLAB scripts

One of the easiest and most transportable ways to use MATLAB is to enter your MATLAB commands into a file and then run that file. These types of files are called **M-files** and have the extension .m after their name. To run an M-file from MATLAB make sure you are in the directory where the file is located and then just type the name of the file without the extension (i.e. without the .m). MATLAB will then process all the commands that are in that file.

Example

1. Create a file called example1.m
2. In that file put the following lines

```
format compact        % this outputs in a compact form
A=[1,2;3,-4]          % this enters the 2x2 matrix A
C=A^10                % this finds the tenth power of A
eig(C)                % this finds the eigenvalues of C
```

3. From within MATLAB type

   example1

   that will run the commands in example1.m
4. The following will appear on the screen

```
A =
     1     2
     3    -4
C =
     1395967    -2789886
    -4184829     8370682
ans =
        1024
     9765625
```

1.5 Sending output to a file

Often you want to keep the output of a MATLAB command. The **diary** commands does that for you. By typing

diary example1.out

all subsequent output will sent to the file example1.out as well as to the screen.
To turn the diary command off use

diary off

1.6 MATLAB help facility and lookfor command

MATLAB has a comprehensive help facility. It is accessed by typing help followed by the name of the command you require help on. For example

help diary

will return the following

```
DIARY Save text of MATLAB session.
   DIARY filename causes a copy of all subsequent command window input
   and most of the resulting command window output to be appended to the
   named file.  If no file is specified, the file 'diary' is used.

   DIARY OFF suspends it.
   DIARY ON turns it back on.
   DIARY, by itself, toggles the diary state.

   Use the functional form of DIARY, such as DIARY('file'),
   when the file name is stored in a string.
```

Note that the help facility uses capitals for the command name, capitals should NOT be used when actually using the commands.

If you don't know the name of the function then use

lookfor keyword

This will find all occurrences of functions or commands with the specified keyword.
For example

lookfor differentiate

returns all functions that mention differentiate

```
lookfor differentiate
POLYDER Differentiate polynomial.
FNDER Differentiate a function.
DIFF   Differentiate.
```

1.7 Functions in MATLAB

Sometimes you will want to write your own function in MATLAB. For example an M-file that takes some kind of input and returns some calculations on that input.

For example write a function that calculates

$$f(x) = x^3 + 4x^2 + 3$$

Create a file called calcf.m that looks like the following

MATLAB code

```
function f=calcf(x)
% this file is named calcf.m
% this function calculates f(x) and
% returns the answers in the variables f
%
f = x^3 + 4*x^2 + 3;
return
```

From within MATLAB type

calcf(3)

to calculate $f(3)$ this results in the following output

```
calcf(3)

ans =

    66
```

1.8 Functions in MATLAB – vector input

Since MATLAB deals with vectors of numbers it is more useful to write functions so that they can handle vectors rather than an individual number. The only change needed to achieve this is to ensure that the calculations within the function are done on an element-by-element basis by using the `.*` notation instead of the `*` notation (and equivalently `.^`, `./` etc).

For example write a function that calculates

$$f(x) = x^3 + 4x^2 + 3$$

for an input vector x. That is, it should calculate $f(x)$ for each value of a vector x and return the answer in a vector.

MATLAB code

```
function f=calcfvec(x)
% this file is named calcfvec.m
% this function calculates f(x) for a vector x and
% returns the answers in the variable f
%
f = x.^3 + 4.*x.^2 + 3;
return
```

From within MATLAB type

y=0:0.2:1.0
calcfvec(y)

to calculate $f(y)$ this results in the following output

```
y=0:0.2:1.0
y =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
calcfvec(y)
ans =
    3.0000    3.1680    3.7040    4.6560    6.0720    8.0000
```

1.9 Passing function names

Often in MATLAB you need to pass the name of a function to another function. For example if you want to numerically calculate the integral of a function MATLAB has an inbuilt function that does this but somehow you need to tell it what function to integrate. Hence you need to pass the functions name.

MATLAB has two inbuilt functions for calculating integrals
`[A,count]=quad('funcname',a,b,tol,trace)`
`[A,count]=quad8('funcname',a,b,tol,trace)`

`quad` uses Simpson's rule, `quad8` uses a degree 8 polynomial. `tol` is the tolerance for the convergence using a comparison between successive approximations. `trace` is an additional feature for plotting a trace of the integrand used. The answer is returned in `A` and `count` is a count of how many function evaluations have been performed.

funcname is the name of the function to be passed. For example if integrating the function defined in the M-file quadexamplefunc.m then use quadexamplefunc where ever you see funcname.

Important Note: The function referenced by 'funcname' **must** be written to take a vector input and give a vector output.

Example
Calculate

$$f(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

For $x = 1$. This function is known as the error function (`erf(x)` in MATLAB)

MATLAB code

```
% quadexample.m
format long     % use lots of decimal places
% the 'exact' answer using MATLABs erf function
errorfunc=erf(1)
% numerically integrate using Simpsons rule (quad)
integral=quad('quadexamplefunc',0,1,1e-6)
% now do it again using an 8th order polynomial (quadl)
integral=quadl('quadexamplefunc',0,1)
% count the number of function calls using the trace switch
trace=1;
[integral,count]=quad('quadexamplefunc',0,1,1e-6,trace)
```

MATLAB code

```
function f=quadexamplefunc(x)
% quadexamplefunc.m
% the function to be integrated
f=(2/sqrt(pi))*exp(-x.^2);
return
```

```
>> quadexample

errorfunc =

   0.84270079294971


integral =

   0.84270079342046


integral =

   0.84270079427671

       5     0.0000000000     1.00000000e+00     0.8427115995
       7     0.0000000000     5.00000000e-01     0.5204995573
       9     0.0000000000     2.50000000e-01     0.2763263864
      11     0.2500000000     2.50000000e-01     0.2441734868
      13     0.5000000000     5.00000000e-01     0.3222012775
      15     0.5000000000     2.50000000e-01     0.1906557580
      17     0.7500000000     2.50000000e-01     0.1315451622

integral =

   0.84270079342046


count =

    17

diary off
```

1.10 An Introduction to solving DEs numerically

MATLAB has many ways of numerically solving Differential Equations. Here we will present one method, the details of the method and the mathematics behind it will be explained in later lectures. This will give you enough now to be able to numerically solve many of the DEs you will encounter early in this course. Consider a first order ODE

$$\frac{dy}{dt} = f(t,y)$$

subject to some initial condition (at time $t = 0$)

$$y(0) = y_0$$

The MATLAB function `ode45` will solve these kind of problems. Here is an excerpt from the help on `ode45`

```
ODE45  Solve non-stiff differential equations, medium order method.
   [T,Y] = ODE45(ODEFUN,TSPAN,Y0) with TSPAN = [T0 TFINAL]
   integrates the system of differential equations y' = f(t,y) from
   time T0 to TFINAL with initial conditions Y0. Function ODEFUN(T,Y)
   must return a column vector corresponding to f(t,y). Each row in
   the solution array Y corresponds to a time returned in the column
   vector T. To obtain solutions at specific times T0,T1,...,TFINAL
   (all increasing or all decreasing), use TSPAN = [T0 T1 ... TFINAL].

   Example
      [t,y]=ode45(@vdp1,[0 20],[2 0]);
      plot(t,y(:,1));
   solves the system y' = vdp1(t,y), using the default relative error
   tolerance 1e-3 and the default absolute tolerance of 1e-6 for each
   component, and plots the first component of the solution.
```

There are 3 things you need to define to use MATLAB to solve this type of DE.

1. The range of t you wish to calculate the solution over (`TSPAN=[T0,TFINAL]`)
2. The initial condition on y (`Y0`)
3. An M-file with the function $f(t,y)$ (`ODEFUN`)

Notes

1. t is returned as a vector of all the times where the solution was determined.
2. Because `ode45` is set up to solve systems of equations y is a matrix. The first dimension corresponds to t, that is solutions at the times in t, the second dimension is the number of equations which in this case is just 1. This has an impact on how the `ODEFUN` function is written. See the next example to see how this works.

Example
Solve

$$\frac{dy}{dt} = y - 2\pi\cos t \quad \text{subject to} \quad y(0) = 3$$

over the range $t = [0,5]$.

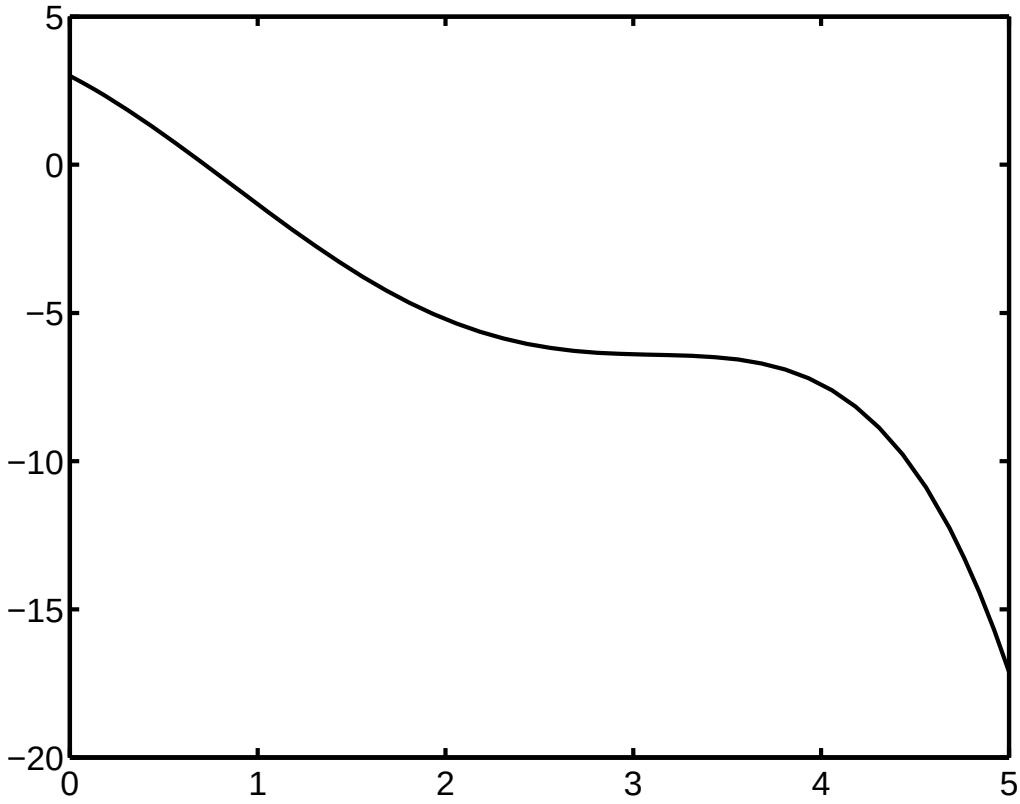
Note that you could solve this particular DE using the Integrating Factor method. The numerical solution works for DEs that can not be solved analytically. This is the point of developing numerical methods for solving DEs, they let you get solutions that can not be obtained analytically.

MATLAB code

```
% odeex1.m
% solves y'= y-2*pi*cos(t)  subject to   y(0)=3
% over the range t=[0,5]
%
tspan=[0,5]                          % sets up the time span
y0=3                                 % the initial condition
% solve the DE and return solution in t and y
[t,y]=ode45('odeex1fun',tspan,y0)
% now plot the solution, note that t is a vector of all the
% points in time where the solution was calculated and y is a
% matrix whose first dimension corresponds to t and second
% dimension is the number of equations (here just 1)
plot(t,y(:,1))
% now send the plot to a file called odeex1.eps to be saved
print -deps odeex1
```

MATLAB code

```
function f=odeex1fun(t,y)
% odeex1fun.m
% the RHS of the system of DEs
% note using f(1) and y(1) as these can be vectors
% of longer than 1 if you have more than 1 equation
f(1)=y(1) - 2*pi*cos(t);
f=f(:);            % forces f to be a column vector
return
```



The same files without all the extraneous comments

MATLAB code

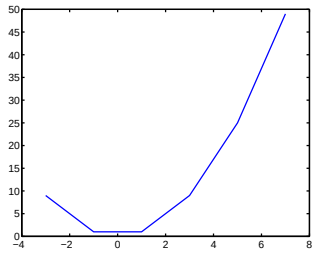
```
% odeex1.m
tspan=[0,5]
y0=3
[t,y]=ode45('odeex1fun',tspan,y0)
plot(t,y(:,1))
print -deps odeex1
```

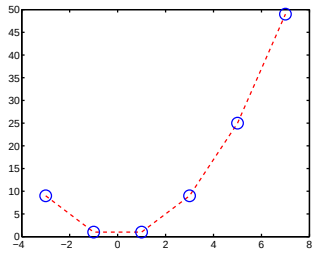
MATLAB code

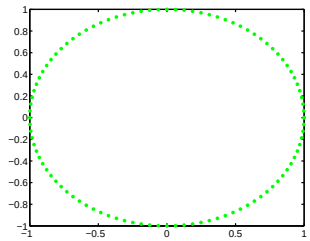
```
function f=odeex1fun(t,y)
% odeex1fun.m
f(1)=y(1) - 2*pi*cos(t);
f=f(:);
return
```

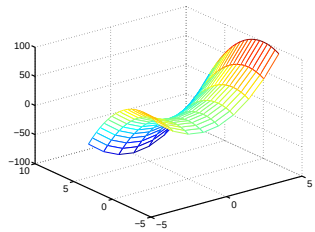
To solve other single equation DEs simply alter the equation (f(1)=. . .), the time range (tspan) or the initial condition (y0).

1.11 Some Elementary MATLAB Commands

Input	Output	Comment
`x = [-3:2:7]`	`x = -3 -1 1 3 5 7`	x is a vector that starts at –3 and goes up in steps of 2 to 7
`x(2)`	`-1`	the second element of the vector x
`y = x.^2`	`y = 9 1 1 9 25 49`	using 'dot' notation square the vector x 'component by component'
`plot(x,y)`		plot y versus x

`plot(x,y,'r--')`
`hold on`
`plot(x,y,'bo')`
		plot y vs x with a red dash line
		hold the plot in place
		add blue circles to the plot

`t = 0:pi/50:2*pi;`
`a=cos(t);`
`b=sin(t);`
`plot(a,b,'g.')`
		t is a vector 0 to 2π in steps of $\pi/50$
		Semicolon supressess the output.
		
		The plot is a parametric plot of a circle with green dots

`c=[-3:1.0:5];`
`d=[-2:0.5:7];`
`[x,y]=meshgrid(c,d)`
`z=4*x.^2-2*y.^2;`
`mesh(x,y,z)`
		3D plotting
		Set up the domain in c and d
		Make a mesh of values in x and y
		The function $z = 4x^2 - 2y^2$
		Plot the function in 3D

`title('text here')`		add a title to a plot
`xlabel('text here')`		add x axis label to a plot
`axis([0 5 -3 8])`		set the x axis to be from 0 to 5 and the y axis from -3 to 8
`text(1,20,'blah')`		place some text on the plot at the point (1,20)

Input	Output	Comment
`x=[1 2 3]`	`x = 1 2 3`	define a vector
`length(x)`	`3`	calculate the size of the vector

`y=x'`	`y = 1`
	`2`
	`3`
		y is the transpose of x

`z=x*x`		error: dimensions dont agree
`z=x*y`	`z = 14`	vector multiplication
`dot(x,y)`	`14`	dot product of vectors x and y
`cross(x,y)`	`0 0 0`	cross product of vectors x and y
`A=[1,2;3,4]`	`A = 1 2` `3 4`	enter the matrix A
`eig(A)`	`-0.3723` `5.3723`	find the eigenvalues of A
`inv(A)`	`-2.0000 1.0000` `1.5000 -0.5000`	find the inverse of A
`det(A)`	`-2.0000`	the determinant of A
`A^3`	`37 54` `81 118`	A cubed (A*A*A)
`A.^3`	`1 8` `27 64`	each component of A cubed
`A(1,:)`	`1 2`	all of the first row of matrix A

`A(:,2)`	`2`
	
	`4`
		all of the second column of matrix A

`b=[1;4]`	`b = 1` `4`	b is a column vector
`c=A\b`	`c = 2.0000` `-0.5000`	Solve Ac=b using backslash divide
`zeros(2)`	`0 0` `0 0`	a 2x2 matrix of zeros
`ones(2)`	`1 1` `1 1`	a 2x2 matrix of ones
`help plot`		help on the plot command
`lookfor sinh`		find all references to the word sinh
`diary file.txt`		send all output to the file file.txt
`hold on`		hold the plot on so that more can be added to it
`hold off`		turn the hold off so that a new plot is made
`load xdata`		read the contents of the file xdata and store the results in a variable called xdata

2 Initial Value Problems

2.1 Introduction

An Initial Value Problem (**IVP**) is a differential equation of nth order where the function value and the first $n-1$ derivatives are all specified at **one** given point known as the initial point.

For examples

$$y' = 2y^2 \quad \text{with} \quad y(1) = 3$$

$$y''' + 3y'' + 7y' - 3y = 0 \quad \text{with} \quad y''(2) = 2 \quad y'(2) = -1 \quad y(2) = 0.5$$

Also it can be a **system** of first order differential equations with each function given a value at the **same** point.

For example

$$y''' + 3y'' + 7y' - 3y = 0 \quad \text{with} \quad y''(2) = 2 \quad y'(2) = -1 \quad y(2) = 0.5$$

can be written as

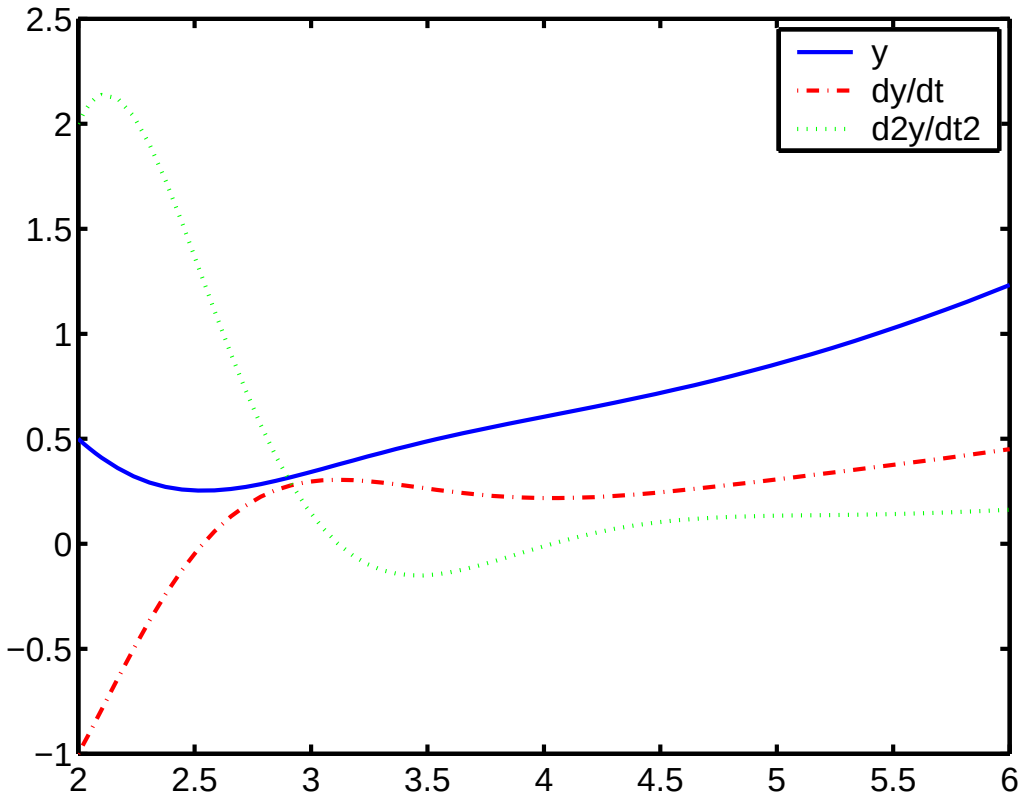
A numerical solution to this system of equations over $t = [2,6]$ using MATLAB is

MATLAB code

```
% system1.m
tspan=[2,6]         % note starts at t=2 now
y0=[0.5,-1,2]       % note its a vector now
[t,y]=ode45('system1fun',tspan,y0)
% plot all y, y' and y'' on one graph and add a legend
plot(t,y(:,1),t,y(:,2),'r-.',t,y(:,3),'g:')
legend('y','dy/dt','d2y/dt2')
print -depsc system1
```

MATLAB code

```
function f=system1fun(t,y)
% system1fun.m
f(1)=y(2);
f(2)=y(3);
f(3)=-3*y(3)-7*y(2)+3*y(1);
f=f(:);
return
```



2.2 Numerical approach

For an IVP because all the derivatives are known at the one given initial point a Taylor's Series can be used to expand about that point to get an approximation to the solution at neighbouring points. So for instance if you know the about the function at the point a then information at a point x is given by

$$f(x) = f(a) + f'(a)(x-a) + \frac{1}{2!}f''(a)(x-a)^2 + \dots$$

If you know something about $f'(a)$, $f''(a)$,... then you can approximate the solution at the point x.

The process for solving IVPs numerically is to start at the initial point and step forward using information provided by the governing differential equation. This can be done in a variety of ways. The simplest is Euler's method.

2.3 Euler's method

(Revision from first year)

Euler's method is the simplest (and least accurate) method for solving IVPs numerically. It consists of approximating the function by a straight line at each point that you step forward at. Recall the definition of the derivative as

$$\frac{dy}{dx} \approx \frac{y(x+\Delta x) - y(x)}{\Delta x}$$

If we are solving the IVP

$$\frac{dy}{dx} = f(x,y) \quad \text{with} \quad y(x_0) = y_0$$

we start at the position x_0 and hence $y(x_0) = y_0$ is known then

$$\frac{dy}{dx}(x_0,y_0) = f(x_0,y_0) \approx \frac{y(x_0+\Delta x) - y(x_0)}{\Delta x}$$

which rearranging gives the solution at the next space step $(x_0 + \Delta x)$ to be

$$y(x_0+\Delta x) = y(x_0) + f(x_0,y_0)\Delta x$$

This is then repeated treating the new point as known. This gives

$$y_{i+1} = y_i + f_i \Delta x$$

where y_i is the approximation at the point $x_i = x_0 + i\Delta x$ and $f_i = f(x_i,y_i)$.

Euler's method is equivalent to approximating the solution by using the first two terms in the Taylor Series (the function value and the first derivative) and ignoring the higher order terms. The first ignored term is

$$\frac{1}{2!}f''(a)(x-a)^2 = \frac{1}{2!}f''(a)(\Delta x)^2$$

so each step in the process has an error of the order of $(\Delta x)^2$.

2.3.1 Graphically

2.3.2 Errors and order of the method

Assuming we are calculating the solution from $x = a$ to $x = b$ using N steps then $\Delta x = (b-a)/N$. The error in each step is proportional to $(\Delta x)^2$ as this is the first term neglected in the Taylor Series so then **total** error over the entire range of solution is proportional to

$$N(\Delta x)^2 = N\Delta x\Delta x = (b-a)\Delta x$$

Hence the total error is proportional to $(\Delta x)^1$ so this is known as a **first order** method.

A pth order method is one whose error is proportional to $(\Delta x)^p$.

The higher the order the method the more accurate it is. This usually comes at a price of being more difficult to program.

First order methods are usually very slow if you want any reasonable amount of accuracy. Why is this?

Example

Solve

$$y' = y \quad \text{with} \quad y(0) = 1$$

over the range $x = [0,1]$. This has exact solution $y = e^x$.

MATLAB code

```
% eulerexample.m
%
% solve y'=y subject to y(0)=1
% find the result at x=1 and calculate the error.
%
clear all;
N=100;            % number of steps used
a=0;  b=1;        % endpoints of solution
dx=(b-a)/N;       % step size
% save x and y values in vectors so we can plot solution
x(1)=a; y(1)=1;  % initial x and y values
for i=1:N
  f=y(i);         % calculate the function value at (xi,yi)
  y(i+1)=y(i)+dx*f;
  x(i+1)=x(i)+dx;
end
plot(x,y,'b-')
axis([0 1 1 2.8])
hold on
exact=exp(x);
plot(x,exact,'r--');
legend('Euler','Exact',2)
hold off
print -depsc eulerexample
error=abs(exact(N+1)-y(N+1))
```

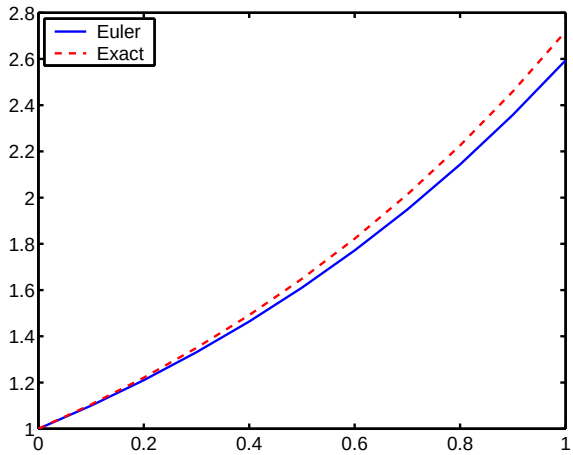
Results of running the above code for different number of points ($N = 10, 100, 1000, 10000$) and hence different step sizes

N	Δx	y_N	error
10	0.1	2.59374246010000	0.12453936835904
100	0.01	2.70481382942153	0.01346799903752
1000	0.001	2.71692393223590	0.00135789622315
10000	0.0001	2.71814592682523	0.00013590163356

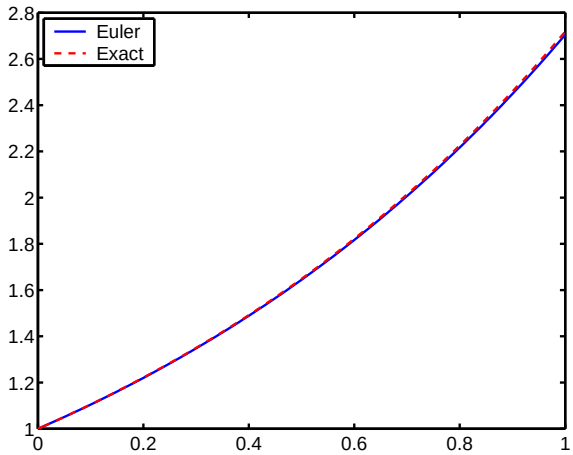
What do you notice about how the error decreases as the step size decreases? How is this related to the order of the method?

Exact solution and Euler's method with $N = 10$ and $N = 100$ applied to

$$y' = y \quad \text{with} \quad y(0) = 1$$



$N = 10$, $\Delta x = 0.1$



$N = 100$, $\Delta x = 0.01$

2.3.3 Adjusting the step size

How do you know how accurate the solution is? When should you use a smaller step size to get a more accurate solution at the cost of increased computing time?

All you know about Euler's method is that the error is proportional to the step size (Δx) since it is a first order method, but you have no idea of what that constant of proportionality is. It can change for each different problem and even within the one problem.

What would be ideal if we had some way of changing the step size so that in areas where the solution changes rapidly we choose a small step size and in areas where it changes slowly we can take a larger step size so that the error is about the same at each step. This is something we would want to be able to do for any method not just Euler's method.

All well written numerical ODE solvers will use some kind of adjustable step size that takes account of how fast the solution is varying and the size of the errors involved.

Algorithm for adjusting the step size

1. For *each step* compare the result of using one step of size Δx with the result of using **two** steps of size $\frac{\Delta x}{2}$. Call the difference between the solutions Δy. If the method you are using is a pth order method then the error from a **single** step is $(\Delta x)^{p+1}$ so

$$\begin{aligned}
\Delta y &= y(x_i+\Delta x) - y(x_i + 2\frac{\Delta x}{2}) \\
&\approx k(\Delta x)^{p+1} - \left(k\left(\frac{\Delta x}{2}\right)^{p+1} + k\left(\frac{\Delta x}{2}\right)^{p+1}\right) \\
&= k(\Delta x)^{p+1} - \frac{k(\Delta x)^{p+1}}{2^p} \\
&= k(\Delta x)^{p+1}\left(1-\frac{1}{2^p}\right) \\
&= k'(\Delta x)^{p+1}
\end{aligned}$$

2. If you wish to make your solution accurate to some predetermined accuracy (call it ε) then $|\Delta y/y| < \varepsilon$. This corresponds to some ideal step size which we will call δx. Then

$$|\Delta y/y| = \varepsilon = k(\delta x)^{p+1} \quad \text{so} \quad \delta x = \left(\frac{\varepsilon}{k}\right)^{1/(p+1)}$$

in reality we used a step size Δx which had

$$|\Delta y/y| = k(\Delta x)^{p+1} \quad \text{so} \quad \Delta x = \left(\frac{1}{k}|\Delta y/y|\right)^{1/(p+1)}$$

The ratio of the step sizes (R) is then a measure of how good our step size is compared to the ideal step size

$$R = \frac{\delta x}{\Delta x} = \left(\frac{\varepsilon y}{\Delta y}\right)^{1/(p+1)}$$

3. If $R > 1$ then step size is too small so it can be increased at the **next** step.
If $R < 1$ then step size is too big so it must be decreased at the **current** step and the process repeated until $R > 1$.
Use the size of R as a *guide* to how much to increase or decrease the step size.
For example let

$$\Delta x_{new} = mR\Delta x_{old}$$

where m is a tuning parameter usually taken to be a little less than one (eg $m = 0.95$ or thereabouts).

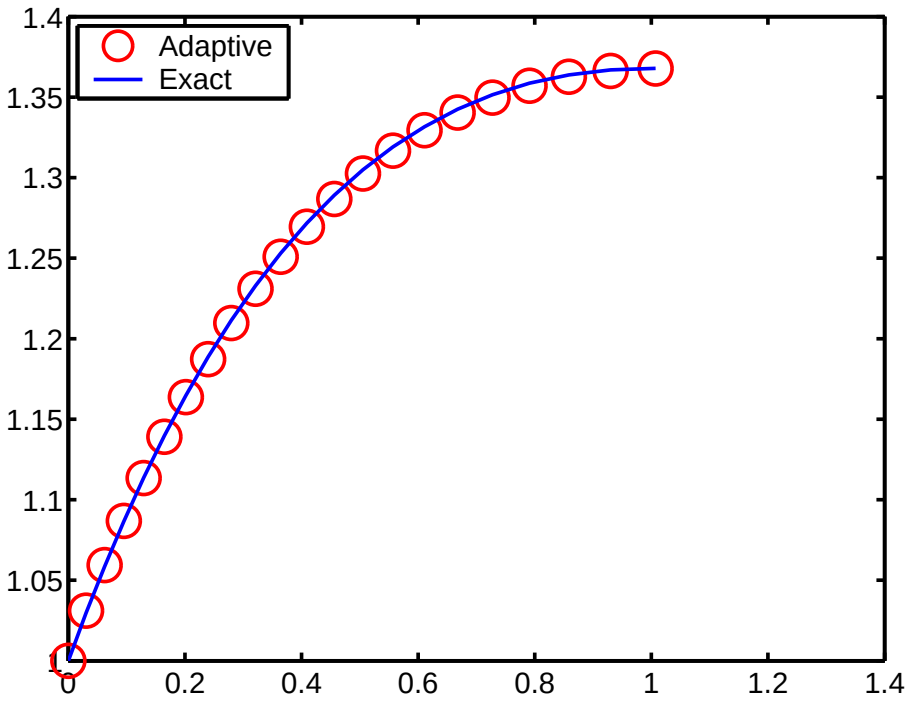
MATLAB code

```
% euleradaptive.m
% Euler method with adaptive time stepping
%
% dy/dx + y = exp(-x) + 1   with   y(0)=1
% which has exact solution y = xexp(-x) + 1
%
clear all;
p=1;  m=0.95;  eps=5.e-4;
a=0;  b=1.0;        % endpoints of solution
dx(1)=0.1;          % initial step size
% save x,y and dx values in vectors so we can plot solution
x(1)=a; y(1)=a*exp(-a)+1;  % initial x and y values
i=1;
while x(i) < b   % keep going until you reach the end point
  R=0;
  while R < 1
    % calculate solution in one step
    yone=y(i)+dx(i)*feval('eulerf',x(i),y(i));
    % calculate solution in two steps
    yhalf=y(i)+dx(i)/2*feval('eulerf',x(i),y(i));
    ytwo=yhalf+dx(i)/2*feval('eulerf',x(i)+dx(i)/2,yhalf);
    R=(abs(eps*ytwo/(yone-ytwo)))^(1/(p+1));
    dx(i)=m*R*dx(i);
  end
  y(i+1)=ytwo;  x(i+1)=x(i)+dx(i);
  dx(i+1)=dx(i);  i=i+1;
end
for j=1:i
  exact(j)=x(j)*exp(-x(j))+1;
end
plot(x,y,'ro')
hold on
plot(x,exact,'b-');
legend('Adaptive','Exact',2)
hold off
print -depsc euleradaptive
plot(x,dx,'ro')
print -depsc euleradaptivedx
```

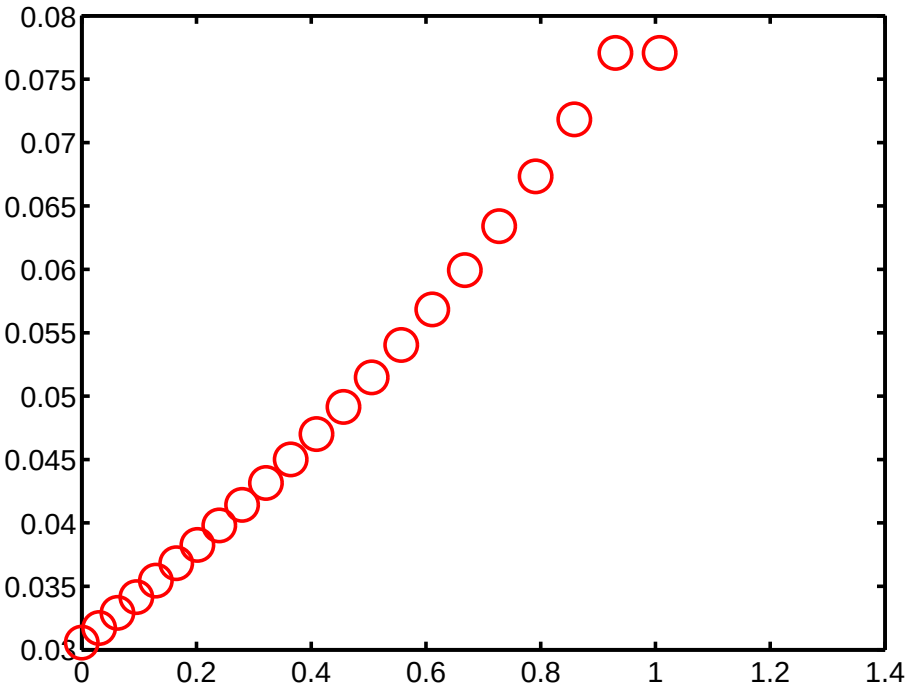
MATLAB code

```
function fxy=eulerf(xx,yy)
fxy=exp(-xx)-yy+1;
return
```

Below are plots of the Euler method and adaptive Euler method and also a plot showing the step size used. Notice that the step size increases as the solution flattens as larger steps can be taken for the same error.



Exact, Euler and Adaptive Euler solution



Showing how the step size changes

2.4 Improvements to Euler's method - midpoint method

Instead of using the slope at the starting point use the slope at the midpoint. This in general will be a better approximation to the function.

Graphically

First estimate the value of y at the **midpoint** using Euler's method

$$y_{i+1/2} = y_i + f(x_i, y_i)\Delta x/2$$

The slope at the midpoint is then approximated by $f(x_{i+1/2}, y_{i+1/2})$ so use this in an Euler like formulation to get the formula

$$y_{i+1} = y_i + f(x_{i+1/2}, y_{i+1/2})\Delta x$$

This is known as the **midpoint method**.

It can be shown that the midpoint method is a **second** order method (as opposed to Euler's method that is first order) but this comes at the price of having to calculate twice the number of function evaluations (since we calculate the function at the midpoints as well).

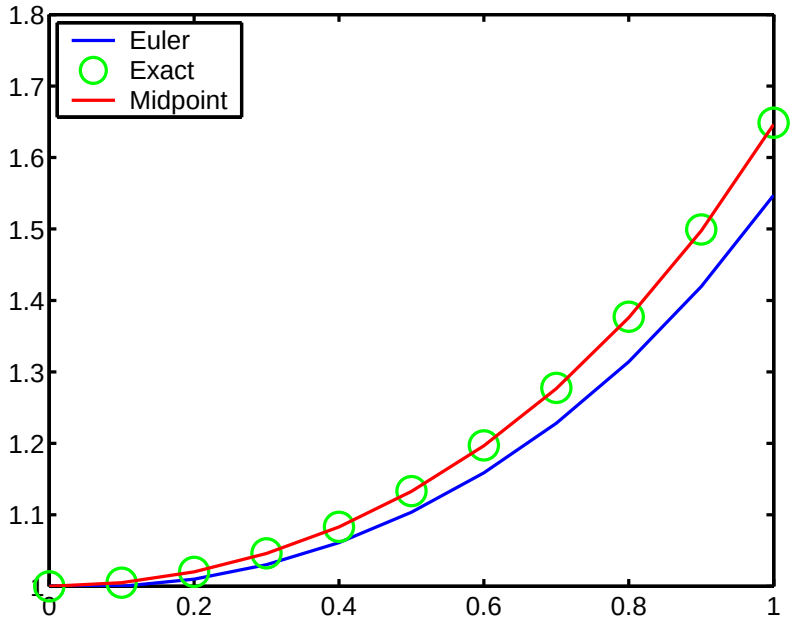
How will this effect the methods efficiency? That is, how will the extra function evaluations be offset by the improvement in the calculation as it is a higher order method?

Example

Solve

$$y' = xy \quad \text{with} \quad y(0) = 1$$

on $[0,1]$. This has exact solution $y = e^{x^2/2}$.



Midpoint method and Euler method with $N = 10$, $\Delta x = 0.1$

Below are the results of calculation with the Euler and Midpoint methods for various step sizes. Compare the size of the errors for the same number of flops*.

N	Δx	error Euler	fbps * Euler	error Midpoint	fbps * Midpoint
10	0.1	0.10161087269011	60	0.00257111404563	120
100	0.01	0.01090081247050	600	0.00002730016798	1200
1000	0.001	0.00109823249901	6000	0.00000027460827	12000
20	0.05	0.05277960435958	120	0.00066472539344	240
200	0.005	0.00547295587046	1200	0.00000684735089	2400

* A fbp is a measure of how many operations (addition, multiplication) where necessary for the calculation.

The Midpoint method is a second order method. That is the total error is proportional to $(\Delta x)^2$. Hence for a 10 fold decrease in the step size (Δx) you get an approximate 100 fold decrease in the error.

For the same amount of computing effort (flops) the Midpoint method is far superior to Euler's method. Just compare the errors for the Euler 120 flops calculation ($N = 20$, error≈ 0.05278) with the Midpoint 120 flops calculation ($N = 10$, error≈ 0.00257).

MATLAB code

```
% midpoint.m
%
% solve y'=xy subject to y(0)=1
% find the result at x=1
% for Euler and Midpoint methods and count flops.
%
clear all;
N=10;                        % number of points,
a=0;  b=1;                   % initial and end points
dx=(b-a)/N;                  % step size
x(1)=a; y(1)=1;              % initial x, y(Euler) values
z(1)=y(1);                   % initial z(Midpoint) value
flops(0)
for i=1:N              % Euler method
  y(i+1)=y(i)+dx*feval('midf',x(i),y(i));
  x(i+1)=x(i)+dx;
end
eulerflops=flops
flops(0);
for i=1:N                % Midpoint method
  zhalf=z(i)+dx/2*feval('midf',x(i),z(i));
  z(i+1)=z(i)+dx*feval('midf',x(i)+dx/2,zhalf);
  x(i+1)=x(i)+dx;
end
midptflops=flops
plot(x,y,'b-')
hold on
exact=exp(0.5.*x.^2);
plot(x,exact,'go');
plot(x,z,'r-')
legend('Euler','Exact','Midpoint',2)
hold off
print -depsc midptexample
erroreuler=abs(exact(N+1)-y(N+1))
errormidpt=abs(exact(N+1)-z(N+1))
```

MATLAB code

```
function f=midf(xx,yy)
% midf.m
f=xx*yy;
return
```

2.5 Second order Runge-Kutta method

Named after 2 German Mathematicians, Carl David Tolme **Runge** (1856–1927) and Wilhelm **Kutta** (1867–1944)

The previous methods (Euler, Midpoint) have all used some approximation to the slope.
Euler: the slope at the left point
Midpoint: the slope at the midpoint.

The Runge-Kutta method uses a *weighted* average of the the slope of the left point and some as yet unknown intermediate point. So the general formula is

$$y_{i+1} = y_i + \Delta x f_{avg}$$

where f_{avg} is a weighted average given by

$$f_{avg} = a f_i + b f_{i'}$$

where $f_i = f(x_i, y_i)$ and $f_{i'} = f(x_{i'}, y_{i'})$ is the function evaluated at some undetermined point given by

$$\begin{aligned} x_{i'} &= x_i + \alpha \Delta x \\ y_{i'} &= y_i + \beta f_i \Delta x \end{aligned}$$

The method consists of finding the values of the weights a, b and the position of the intermediate point given by α and β.

Recall the Taylor Series expansion for $y_{i+1} = y(x_i + \Delta x)$ is

$$\begin{aligned} y_{i+1} &= y_i + \Delta x \frac{dy}{dx}|_i + \frac{(\Delta x)^2}{2} \frac{d^2 y}{dx^2}|_i + \dots \\ &= y_i + \Delta x f_i + \frac{(\Delta x)^2}{2} \frac{df}{dx}|_i + \dots \\ &= y_i + \Delta x f_i + \frac{(\Delta x)^2}{2} \left(\frac{\partial f}{\partial x}|_i + \frac{\partial f}{\partial y} \frac{dy}{dx}|_i \right) + \dots \\ &= y_i + \Delta x f_i + \frac{(\Delta x)^2}{2} \left(\frac{\partial f}{\partial x}|_i + \frac{\partial f}{\partial y} f|_i \right) + \dots \end{aligned}$$

But expanding $f_{i'}$ in a 2D Taylor Series gives

$$\begin{aligned} f_{i'} &= f(x_{i'}, y_{i'}) \\ &= f(x_i + \alpha \Delta x, y_i + \beta f_i \Delta x) \\ &= f(x_i, y_i) + \alpha \Delta x \frac{\partial f}{\partial x}|_i + \beta f_i \Delta x \frac{\partial f}{\partial y}|_i + \dots \end{aligned}$$

So the Runge-Kutta formulation gives

$$\begin{aligned} y_{i+1} &= y_i + \Delta x f_{avg} \\ &= y_i + \Delta x \left(a f_i + b f_{i'} \right) \\ &\approx y_i + \Delta x \left(a f_i + b \left[f(x_i, y_i) + \alpha \Delta x \frac{\partial f}{\partial x}|_i + \beta f_i \Delta x \frac{\partial f}{\partial y}|_i \right] \right) \\ &= y_i + \Delta x \left(a f_i + b f_i + b \alpha \Delta x \frac{\partial f}{\partial x}|_i + b \beta f_i \Delta x \frac{\partial f}{\partial y}|_i \right) \end{aligned}$$

Comparing this with the Taylor Series expansion for y_{i+1} gives

$$a+b=1 \qquad b\alpha=\frac{1}{2} \qquad b\beta=\frac{1}{2}$$

Which is 3 equations in the 4 unknowns a, b, α and β so there is still some degree of freedom of choice.

For example setting

$$a=0 \qquad b=1 \qquad \alpha=\beta=\frac{1}{2}$$

gives the midpoint method.

These methods are second order. Because there is still a degree of freedom in the equations for specific problems it is possible to make a choice of a, b, α and β that will make the method third order. This method has been precoded in MATLAB and is used with a call to `ode23`

To solve the IVP

$$\frac{dy}{dt}=odefunc(t,y) \quad \text{with} \quad y(a)=c$$

over the interval $[a,b]$ a call to `ode23` would be like

```
y0=[c];
tspan=[a b];
[t,y] = ode23('odefunc',tspan,y0);
```

Use `help ode23` to find out more about the MATLAB command and how to use it. In particular how to choose the accuracy.

`ode23` also uses an adaptive step size as well if it needs to find the solution more accurately in a given region of the interval.

Example

Solve

$$\frac{dy}{dt} = ty \quad \text{with} \quad y(0) = 1$$

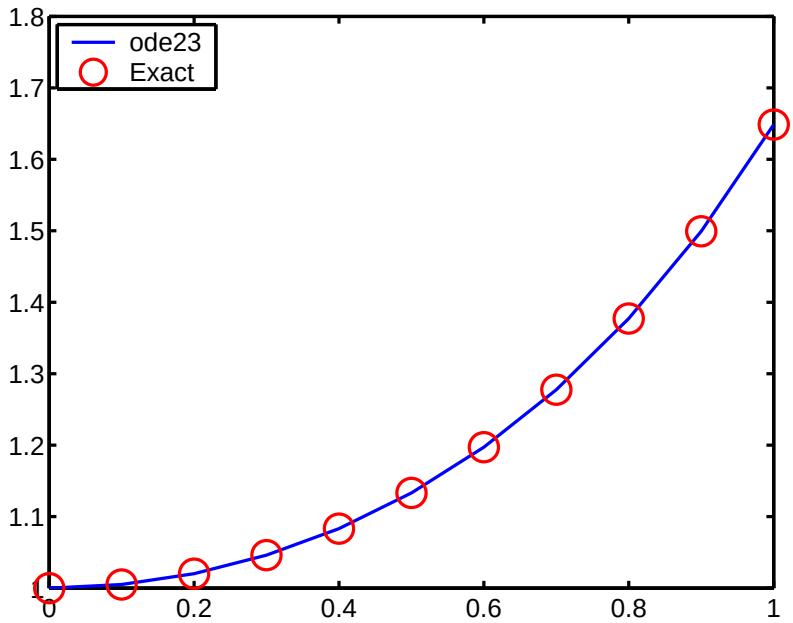
on $[0,1]$. This has exact solution $y = e^{t^2/2}$.

MATLAB code

```
% o23example.m
% solve dy/dt=odefunc(t,y) on [a,b] given y(a)=c
a=0; b=1; c=1;                    % interval and initial value
y0=[c];                           % initial condition
tspan=[a b];                      % interval of integration
[t,y]=ode23('odefunc',tspan,y0);% soln in vectors t and y
plot(t,y)
hold on
exact=exp(0.5.*t.^2);
plot(t,exact,'ro')
legend('ode23','Exact',2)
print -depsc o23example
```

MATLAB code

```
function f=odefunc(t,y)
% odefunc.m
% the rhs of the IVP dy/dt=f(t,y)
f=t*y;
return
```



2.6 Fourth order Runge-Kutta method

In exactly the same way as the second order Runge-Kutta method was developed higher order methods can be developed. By including more sampling points in the interval greater accuracy can be obtained. The second order method used the left-hand point and one other point to determine an approximation to the solution. Hence two function evaluations were required for each step. The Runge-Kutta fourth order method uses the left-hand point and 3 other points to hence four function evaluations are required for each step. The method is fourth order accurate that is the total error is proportional to $(\Delta x)^4$.

$$y_{i+1} = y_i + \frac{1}{6}\left(z_1 + 2z_2 + 2z_3 + z_4\right)$$

where

$$\begin{aligned}
z_1 &= f(x_i, y_i)\Delta x \\
z_2 &= f(x_i + \frac{1}{2}\Delta x, y_i + \frac{1}{2}z_1)\Delta x \\
z_3 &= f(x_i + \frac{1}{2}\Delta x, y_i + \frac{1}{2}z_2)\Delta x \\
z_4 &= f(x_{i+1}, y_i + z_3)\Delta x
\end{aligned}$$

The MATLAB command for using this method is `ode45` and is used exactly the same way as `ode23`. The fourth order Runge-Kutta method is by far the most commonly used method for solving IVPs.

2.7 ode45

```
>> help ode45

 ODE45  Solve non-stiff differential equations, medium order method.
    [T,Y] = ODE45('F',TSPAN,Y0) with TSPAN = [T0 TFINAL] integrates
    the system of differential equations y' = F(t,y) from time T0
    to TFINAL with initial conditions Y0.  'F' is a string
    containing the name of an ODE file.  Function F(T,Y) must
    return a column vector.  Each row in solution array Y
    corresponds to a time returned in column vector T.  To obtain
    solutions at specific times T0, T1, ..., TFINAL (all increasing
    or all decreasing), use TSPAN = [T0 T1 ... TFINAL].

    [T,Y] = ODE45('F',TSPAN,Y0,OPTIONS) solves as above with
    default integration parameters replaced by values in OPTIONS,
    an argument created with the ODESET function.  See ODESET for
    details.  Commonly used options are scalar relative error
    tolerance 'RelTol' (1e-3 by default) and vector of absolute
    error tolerances 'AbsTol' (all components 1e-6 by default).

    [T,Y] = ODE45('F',TSPAN,Y0,OPTIONS,P1,P2,...) passes the
    additional parameters P1,P2,... to the ODE file as
    F(T,Y,FLAG,P1,P2,...) (see ODEFILE).  Use OPTIONS = [] as
    a place holder if no options are set.

    It is possible to specify TSPAN, Y0 and OPTIONS in the ODE
    file (see ODEFILE).  If TSPAN or Y0 is empty, then ODE45 calls
    the ODE file [TSPAN,Y0,OPTIONS] = F([],[],'init') to obtain
    any values not supplied in the ODE45 argument list.  Empty
    arguments at the end of the call list may be omitted, e.g.
    ODE45('F').

    As an example, the commands

        options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
        ode45('rigidode',[0 12],[0 1 1],options);

    solve the system y' = rigidode(t,y) with relative error
    tolerance 1e-4 and absolute tolerances of 1e-4 for the first
    two components and 1e-5 for the third.  When called with no
    output arguments, as in this example, ODE45 calls the default
    output function ODEPLOT to plot the solution as it is computed.

    See also ODEFILE and
        other ODE solvers:  ODE23, ODE113, ODE15S, ODE23S, ODE23T
        options handling:   ODESET, ODEGET
        output functions:   ODEPLOT, ODEPHAS2, ODEPHAS3, ODEPRINT
        odefile examples:   ORBITODE, ORBT2ODE, RIGIDODE, VDPODE
```

Example
Solve

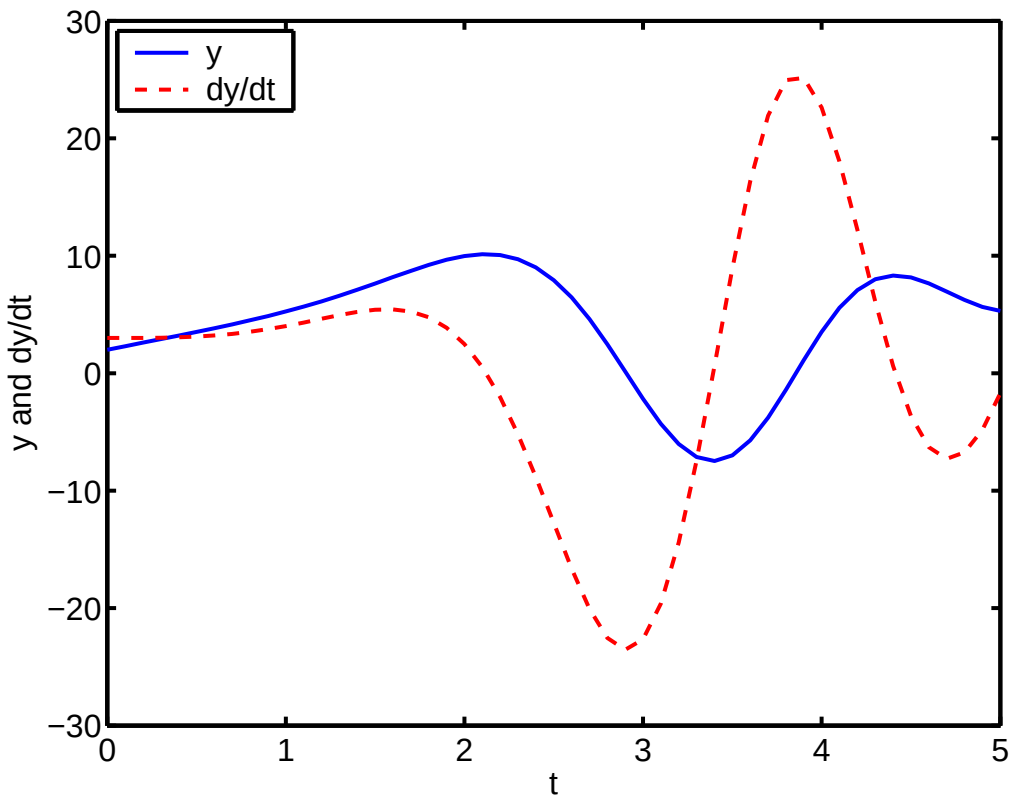
$$y'' = t^2 \cos(t) y \quad \text{s.t. } y(0) = 2 \text{ and } y'(0) = 3$$

MATLAB code

```
% o45ex1.m
% solve y''=t^2*y*cos(t)on [0,5] given y(0)=2, y'(0)=3
% this can be written as a system of first order DEs
% y1'=y2                    y1(0)=2
% y2'=t^2*y1*cos(t)         y2(0)=3
a=0; b=5;                          % interval
y0=[2 3];                          % initial condition
tspan=[a:.1:b];                    % interval of integration
[t,y]=ode23('o45ex1func',tspan,y0);% soln in vectors t and y
plot(t,y(:,1),t,y(:,2),'r--')      % plot y1 and y2 versus t
legend('y','dy/dt',2)
xlabel('t'); ylabel('y and dy/dt')
print -depsc o45ex1                 % output to postscript file
```

MATLAB code

```
function f=o45ex1func(t,y)
% o45ex1func.m
% the RHS of the system of DEs
f(1)=y(2);
f(2)=t^2.*y(1).*cos(t);
f=f(:);           % forces f to be a column vector
return
```



The MATLAB function `ode45` has a vast array of options that can be used.

Finding solutions at given points.

use tspan = [T0 T1 ... TFINAL]

eg `tspan=[0.1 0.2 0.3 0.4 0.5]` will return solutions at $t = 0.1, 0.2$ etc.

Error tolerances

There is an extra parameter that can be added to the parameter list called 'options'. The call to `ode45` is then

```
ode45('func',tspan,y0,options)
```

The options parameter is set using the command `odeset`. For example

```
options = odeset('RelTol',1e-3,'AbsTol',[1e-4 1e-4 1e-5]);
```

Would set the relative error tolerance to 1×10^{-3} and the absolute error tolerance to 1×10^{-4} for the first two components and 1×10^{-5} for the third. This is useful if you know (or expect) that some of the variables will be substantially different is magnitude than others.

Stopping the integration

It is possible to stop the integration when certain events occur. For example when any of the variables pass through a given value.

```
options = odeset('Events','on');
```

```
[t,y] = ode45('func',tspan,y0,options)
```

For this to work the file func.m must return appropriate information. func.m is coded so that if ode45 passes it a flag with a value of 'events' it determines the .m file that tells it information about the stopping criteria. If the flag is empty it determines the .m file that is actually used to calculate the integration (the DEs).

2.8 Projectile example

Consider the problem of a projectile fired from a fixed position. If you know the initial velocity and angle of the projectile it is possible to determine the trajectory of the projectile. Other factors such as drag on the projectile and lift can also be taken into account. A system of differential that governs the motion of the projectile is given by

$$\begin{aligned} m\frac{dv}{dt} &= -mg\sin\phi - \frac{1}{2}\rho A C_D v^2 \\ mv\frac{d\phi}{dt} &= -mg\cos\phi + \frac{1}{2}\rho A C_L v^2 \end{aligned}$$

where ϕ = angle from horizontal, v = velocity, m = mass. ρ = density, A = area, C_D drag coefficient, C_L lift coefficient, g gravity.

The trajectory of the projectile is governed by

$$\begin{aligned} \frac{dx}{dt} &= v\cos\phi \\ \frac{dy}{dt} &= v\sin\phi \end{aligned}$$

If the initial position (x,y coordinates), initial velocity (v) and angle (ϕ) are known then the trajectory of the projectile can be easily determined.

This is a system of 4 *nonlinear* differential equations in the 4 unknowns x, y, v and ϕ. Solving these equations analytically is not possible but solving them numerically is very easy using MATLAB. Rearrange the equations into a standard form where the left hand side is just the derivative of each variable.

$$\begin{aligned} \frac{dv}{dt} &= -g\sin\phi - \frac{\rho A C_D v^2}{2m} \\ \frac{d\phi}{dt} &= \frac{-g\cos\phi}{v} + \frac{\rho A C_L v}{2m} \\ \frac{dx}{dt} &= v\cos\phi \\ \frac{dy}{dt} &= v\sin\phi \end{aligned}$$

Then just numerically integrate these equations forward in time form the known initial position, velocity and angle.

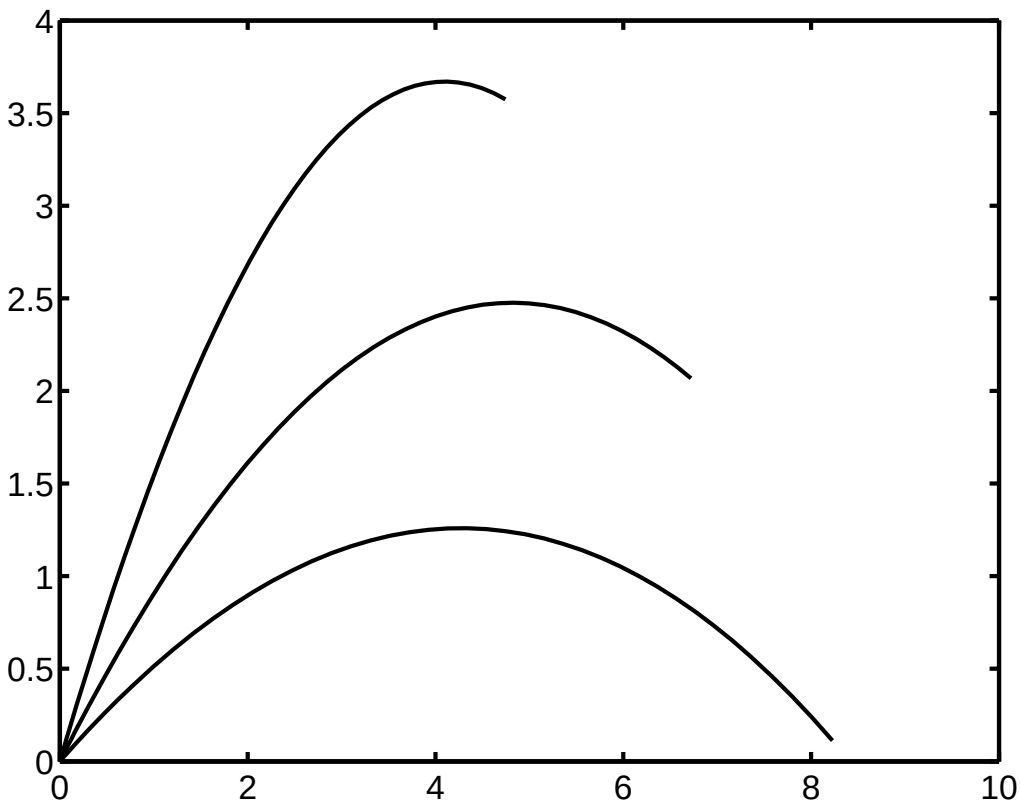
MATLAB code

```
% This file is projectile.m
% It calculates the trajectory of a projectile
% with a predefined initial velocity and an
% inputted angle (theta, in degrees).
%
% This file calls the other file
% projectiledes.m   the governing differential equations for
%                   the velocity, angle, x and y
%
format compact
v0=10;                      % initial velocity
theta=0;
while theta >= 0
  theta=input('enter theta, use < 0 to stop...')
  if theta<0
    break
  else
    % initial point [velocity angle x y]
    y0=[v0 theta*pi/180 0 0];
    % make the time span large enough that it gets to the target
    tspan=[0 1];
    [t,y]=ode45('projectiledes',tspan,y0);
    plot(y(:,3),y(:,4))        % plot the trajectory
    hold on
    % determine how long the solution vector is
    lgthy=length(y);
    range=y(lgthy,3)           % the range is the last x value
  end
end
hold off
print -deps projectile
```

MATLAB code

```
function f=projectiledes(t,y)
% projectiledes.m    the governing DEs for the projectile
% y(1) is velocity, y(2) is angle, y(3) is x, y(4) is y.
% values for a spinning soccer ball
g=9.8; rho=1.23; A=0.038; m=0.42; D=0.22; Cd=0.20; Cl=0.05;
v=y(1);  phi=y(2);
f(1)=(-g.*sin(phi)-rho.*A.*Cd./(2..*m).*v.^2); % velocity
f(2)=(-g.*cos(phi)./v + rho.*A.*Cl./(2.*m).*v);% angle
f(3)=v.*cos(phi);                              % x coordinate
f(4)=v.*sin(phi);                              % y coordinate
f=f(:);             % force f to be a column vector
return
```

```
>> projectile
enter theta, use < 0 to stop...30
theta =
    30
range =
    8.2272
enter theta, use < 0 to stop...45
theta =
    45
range =
    6.7203
enter theta, use < 0 to stop...60
theta =
    60
range =
    4.7443
enter theta, use < 0 to stop...-1
theta =
    -1
>> diary off
```



MATLAB can be set up to integrate the DE until a specific event occurs, for example here that might be when $y = 0$ (ie when the projectile hits the ground). You can then use `fzero` to find the initial angle that gives a specific range (ie hits the target).

3 Phase Plane Analysis

3.1 Introduction

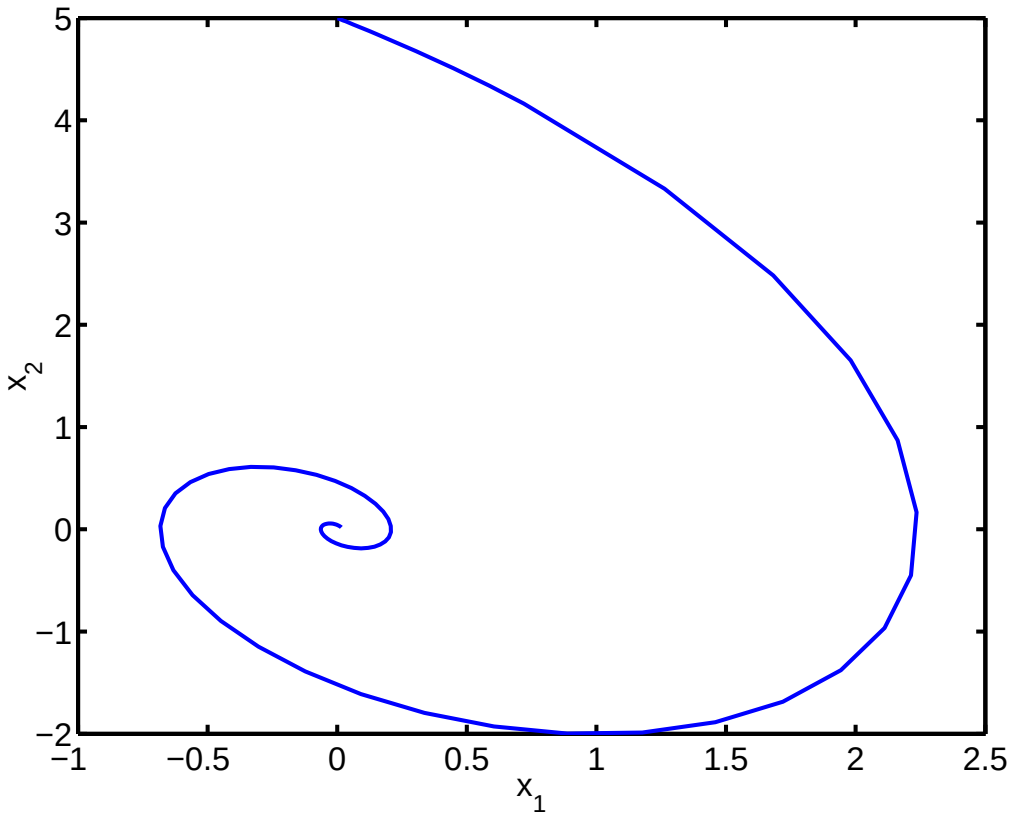
Phase plane analysis is useful in determining the behaviour of solutions to systems of differential equations without necessarily solving the system analytically. More importantly this technique is not limited to studying linear equations.

For systems of 2 autonomous first order differential equations it is possible to plot solutions of one dependent variable against the other dependent variable with the time dependence varying along the curves. This is known as a **phase plane**. The goal is to sketch enough solution curves so that the general behaviour of the system can be summarised by looking at the graph.

The system

$$\begin{aligned} \frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= -2x_1 - x_2 \end{aligned}$$

has phase plane consisting of spirals converging to the origin.



MATLAB code

```
% ppexample.m
% drawing phase plane for
% x1'=x2
% x2'=-2x1-x2
%
tend=10;
tspan=[0 tend];
y0=[0 5];
[t,y]=ode45('ppexamplef',tspan,y0);
plot(y(:,1),y(:,2))
xlabel('x_1'); ylabel('x_2');
print -depsc ppexample
```

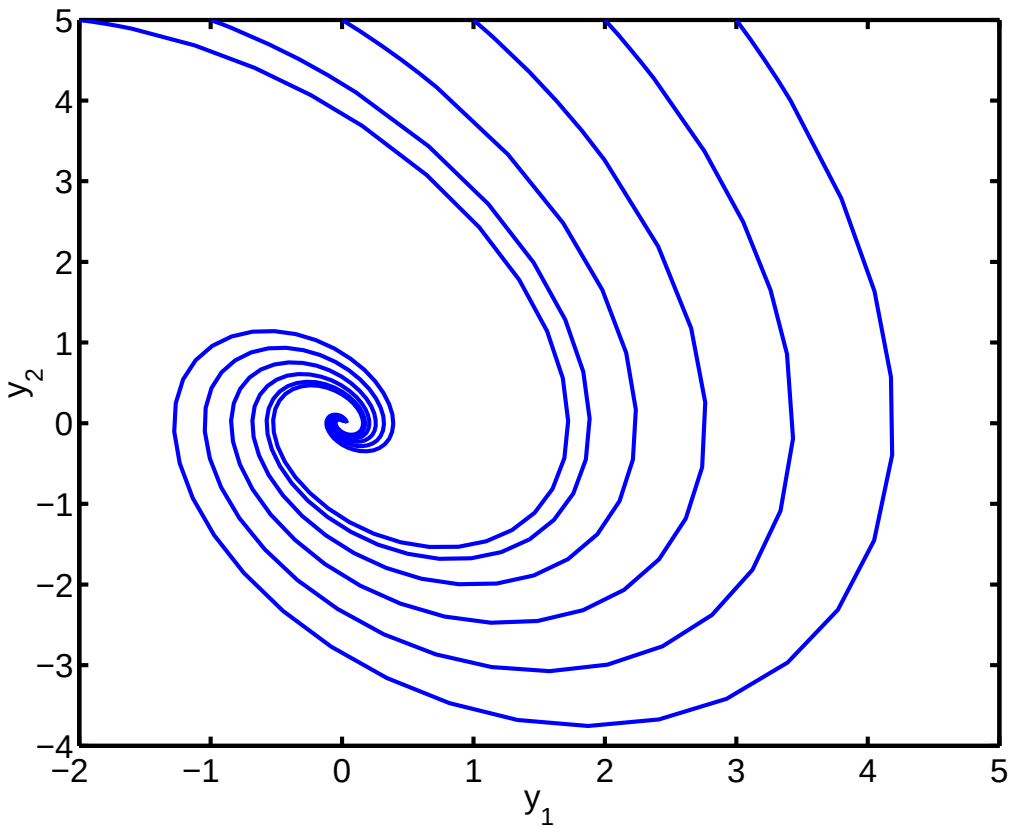
MATLAB code

```
function f=ppexamplef(t,y)
% shootingexamplef.m
f(1)=y(2);
f(2)=-2*y(1)-y(2);
f=f(:);
return
```

Or with multiple initial points put the whole thing into a `for` loop and update the initial points as you go through the loop.

MATLAB code

```
% ppexample2.m
% drawing phase plane for
% x1'=x2
% x2'=-2x1-x2
%
tend=10;
tspan=[0 tend];
for s=-2:1:3
  y0=[s 5];
  [t,y]=ode45('ppexamplef',tspan,y0);
  plot(y(:,1),y(:,2))
  xlabel('y_1'); ylabel('y_2');
  hold on
end
hold off
print -depsc ppexample2
```



3.1.1 Quiver plots

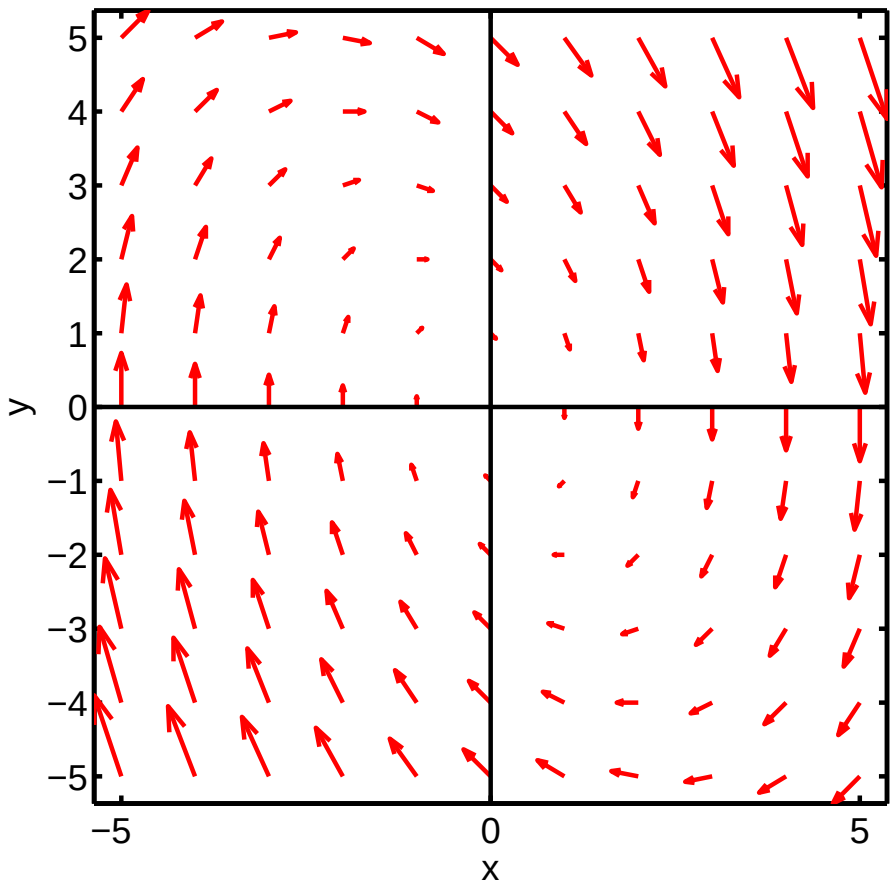
A very simple way in MATLAB to quickly get a feel for the solutions of a system of DEs is to draw the velocity field.

The MATLAB command `quiver` is designed to do precisely this.

```
help quiver

 QUIVER Quiver plot.
    QUIVER(X,Y,U,V) plots velocity vectors as arrows with
    components (u,v) at the points (x,y).  The matrices X,Y,U,V
    must all be the same size and contain corresponding position
    and velocity components (X and Y can also be vectors to
    specify a uniform grid).  QUIVER automatically scales the
    arrows to fit within the grid.

    QUIVER(U,V) plots velocity vectors at equally spaced points
    in the x-y plane.

    QUIVER(U,V,S) or QUIVER(X,Y,U,V,S) automatically scales the
    arrows to fit within the grid and then stretches them by S.
    Use S=0 to plot the arrows without the automatic scaling.

    QUIVER(...,LINESPEC) uses the plot linestyle specified for
    the velocity vectors.  Any marker in LINESPEC is drawn at
    the base instead of an arrow on the tip.  Use a marker of
    '.' to specify no marker at all.  See PLOT for other
    possibilities.

    QUIVER(...,'filled') fills any markers specified.

    H = QUIVER(...) returns a vector of line handles.

    Example:
       [x,y] = meshgrid(-2:.2:2,-1:.15:1);
       z = x.*exp(-x.^2 - y.^2);
       [px,py] = gradient(z,.2,.15);
       contour(x,y,z), hold on
       quiver(x,y,px,py), hold off, axis image

    See also FEATHER, QUIVER3, PLOT.

diary off
```

MATLAB code

```
% vfield.m
%
% plots velocity vectors for the following system
% dx/dt = y
% dy/dt = -2x-y
%
a=-5;  b=5;                    % size of the grid
[x,y]=meshgrid(a:1:b,a:1:b);% a square grid (a to b)x(a to b)
dx=y;                          % the DEs
dy=-2*x-y;                     %
quiver(x,y,dx,dy,'r')          % plot the vector field
axis square                    % make the plot square
axis tight                     % make axis tight on data
hold on
w=[a-1 b+1];
plot(w,0*w,'k-',0*w,w,'k-')   % adds x and y  axes to plot
xlabel('x'); ylabel('y');
hold off
% make an eps version of the plot called vfield.eps
print -depsc vfield
% if you dont want the arrow heads then just use
% quiver(x,y,dx,dy,'.');
% the arrows are automatically scaled to fit the plot space
% if you want bigger or smaller arrows use a scaling
% quiver(x,y,dx,dy,3)   % makes the arrow 3 times longer
% for more help type    help quiver
```



3.2 Predator-Prey model

Predator-Prey models are used to model the interaction between a predator and its prey. The simplest is known as the Lotka-Volterra Model. For example the number of rabbits (prey) and foxes (predator). Let y_1 be the number of predators and y_2 be the number of prey then the governing differential equations are

$$\begin{aligned} \frac{dy_1}{dt} &= -ay_1 + by_1y_2 \\ \frac{dy_2}{dt} &= -cy_1y_2 + dy_2 \end{aligned}$$

Here the terms are explained as

$-ay_1$	If there is no prey ($y_2 = 0$) the number of predators must decline.
dy_2	Number of prey increases if there are no predators
by_1y_2	Number of predators increases if it meets (eats) a prey.
$-cy_1y_2$	Number of prey decreases if it meets (and is eaten) by a predator.

Critical points

Set derivatives equal to zero and solve for y_1 and y_2.

This gives the critical points as $(y_1, y_2) = (0,0)$ and $(y_1, y_2) = (d/c, a/b)$

Calculate the Jacobian and classify each of the critical points. The Jacobian is

$$\begin{aligned} J(y_1, y_2) &= \begin{bmatrix} \frac{\partial}{\partial y_1}(-ay_1 + by_1y_2) & \frac{\partial}{\partial y_2}(-ay_1 + by_1y_2) \\ \frac{\partial}{\partial y_1}(-cy_1y_2 + dy_2) & \frac{\partial}{\partial y_2}(-cy_1y_2 + dy_2) \end{bmatrix} \\ &= \begin{bmatrix} -a + by_2 & by_1 \\ -cy_2 & -cy_1 + d \end{bmatrix} \end{aligned}$$

So substituting each critical point gives

$$J(0,0) = \begin{bmatrix} -a & 0 \\ 0 & d \end{bmatrix}$$

Which is a diagonal matrix so the eigenvalues are $\lambda_1 = -a$ and $\lambda_2 = d$ hence $(0,0)$ is a saddle.

$$J(d/c, a/b) = \begin{bmatrix} 0 & bd/c \\ -ac/b & 0 \end{bmatrix}$$

Calculating the eigenvalues gives $\lambda = \pm i\sqrt{ad}$ and so $(d/c, a/b)$ is (probably) a centre.

How does the population of both the predator and prey species vary with time?

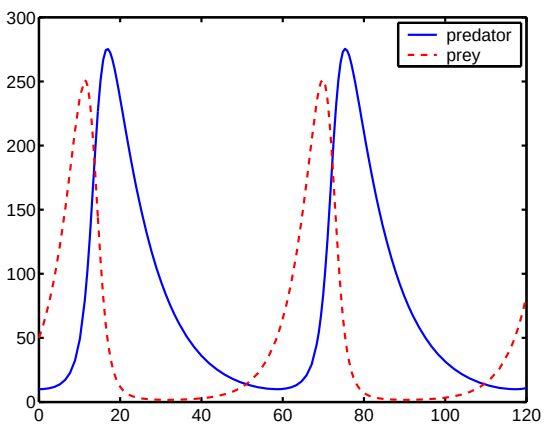
MATLAB

MATLAB code

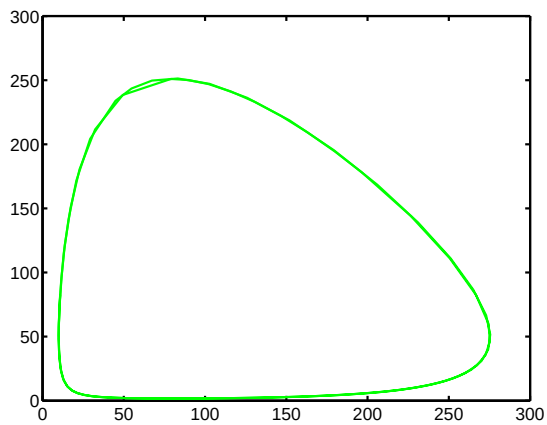
```
% predprey.m
% Predator-Prey system
%
clear all
global a b c d            % variables global to all functions
a=0.1;  b=0.002;  c=0.0025;  d=0.2;
y0=[10 50];               % initial point
tspan=[0 120];            % range to integrate over
[t,y]=ode45('predpreyf',tspan,y0);
plot(t,y(:,1),'b-')       % plot predator popoluation vs time
hold on                   % hold the plot
plot(t,y(:,2),'r--')      % plot prey popoluation vs time
legend('predator','prey')
hold off                  % turn off plot hold
print -depsc predprey     % send plot to postscript file
plot(y(:,1),y(:,2),'g-') % plot trajectories in (y1,y2)space
print -depsc predpreytraj% send plot to postscript file
```

MATLAB code

```
function f=predpreyf(t,y)
% predpreyf.m
global a b c d
f(1)=-a*y(1) + b*y(1)*y(2);
f(2)=-c*y(1)*y(2) + d*y(2);
f=f(:);
return
```



Time plot



A trajectory in (y_1, y_2) space

3.3 Competition model

Competition models are used to model the competition between two (or more) species (companies, people) who are competing for the same resource.

$$\begin{aligned}\frac{dy_1}{dt} &= \frac{r_1}{K_1}y_1\left(K_1 - y_1 - ay_2\right)\\ \frac{dy_2}{dt} &= \frac{r_2}{K_2}y_2\left(K_2 - y_2 - by_1\right)\end{aligned}$$

If there is no second species ($y_2 = 0$) then the first population grows until it approaches the steady population K_1. Similarly for the other species.

Critical points

Set derivatives equal to zero and solve for y_1 and y_2.

$(y_1, y_2) = (0, 0)$
$(y_1, y_2) = (K_1, 0)$
$(y_1, y_2) = (0, K_2)$
and a fourth point is located at the intersection of the two lines $K_1 - y_1 - ay_2 = 0$ and $K_2 - y_2 - by_1 = 0$.

Look at the Jacobian to classify each critical point.

MATLAB code

```
% compjacobian.m
%
format compact
clear all
a=0.75;  b=3.0;  r1=0.2;  r2=0.1;  K1=50;  K2=100;

% store critical points in a vector
y1=[0 K1 0 20];
y2=[0 0 K2 40];
for i=1:4
  fprintf('\nFor critical point (%5.2f %5.2f)\n',y1(i),y2(i))
  Jac=[r1-2*r1*y1(i)/K1-a*r1/K1*y2(i) -a*r1*y1(i)/K1;...
       -b*r2/K2*y2(i) r2-r2*2*y2(i)/K2-r2*b*y1(i)/K2]
  [V,D]=eig(Jac)
end
```

```
>> compjacobian

For critical point ( 0.00  0.00)
Jac =
    0.2000         0
         0    0.1000
V =
     1     0
     0     1
D =
    0.2000         0
         0    0.1000

For critical point (50.00  0.00)
Jac =
   -0.2000   -0.1500
         0   -0.0500
V =
    1.0000   -0.7071
         0    0.7071
D =
   -0.2000         0
         0   -0.0500

For critical point ( 0.00 100.00)
Jac =
   -0.1000         0
   -0.3000   -0.1000
V =
         0    0.0000
    1.0000    1.0000
D =
   -0.1000         0
         0   -0.1000

For critical point (20.00 40.00)
Jac =
   -0.0800   -0.0600
   -0.1200   -0.0400
V =
   -0.6661    0.4885
   -0.7458   -0.8726
D =
   -0.1472         0
         0    0.0272
>> diary off
```

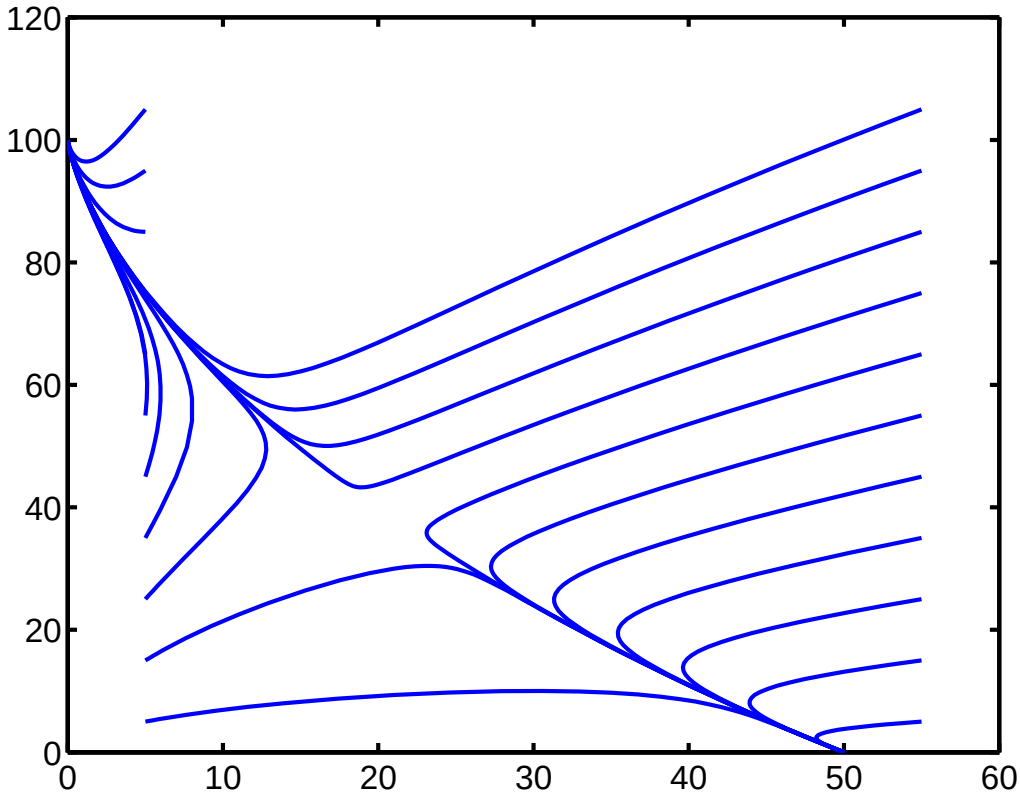
How does the population of both the species vary with time? Does one species dominate the other?

MATLAB code

```
% competition.m
%
clear all
global a b r1 r2 K1 K2   % variables global to all functions
a=0.75;  b=3.0;  r1=0.2;  r2=0.1;  K1=50;  K2=100;
for i=0:10               % loop through various initial points
  for j=0:1
    y0=[j*50+5 i*10+5];  % initial points
    tspan=[0 200];       % range to integrate over
    [t,y]=ode45('competitionf',tspan,y0);
    plot(y(:,1),y(:,2))  % plot trajectories in (y1,y2)space
    hold on
  end
end
print -depsc competitiontraj   % send plot to postscript file
hold off                      % turn off plot hold
```

MATLAB code

```
function f=competitionf(t,y)
% competitionf.m
global a b r1 r2 K1 K2  % variables global to all functions
f(1)=r1/K1*y(1)*(K1 - y(1) - a*y(2));
f(2)=r2/K2*y(2)*(K2 - y(2) - b*y(1));
f=f(:);
return
```



3.4 SIR disease model

Disease models can give great insight into how diseases spread through a population. They can be used to determine suitable strategies for inoculation, such as what percentage of the population need to be inoculated to prevent the spread of a disease. Or simply as a guide to how widespread a disease will become if left alone.

Consider a fixed population of size K. A fixed population size is a reasonable assumption if the birth rate is approximately the same as the death rate and the disease does not actually kill the person. For instance measles in the developed world is rarely a fatal disease and has no impact on the overall birth or death rate.

Consider three distinct classes of people:

Susceptibles (S):	those people who are susceptible to catching the disease
Infectives (I):	those people who have the disease and can pass it on
Recovered (R):	those people who have had the disease and recovered from it. They are no longer susceptible as they have immunity now.

This is (for obvious reasons) known as an SIR model. The total population is fixed so that

$$S+R+I=K$$

The general birth rate of people (who are all born susceptible) is μ. Because the population is assumed to be constant the death rate must also be μ but any of susceptibles, infectives or recovereds can die. The disease is assumed not to alter the death rate.

Consider how the population of susceptibles can change.
Susceptibles catching the disease and becoming infective, births of susceptibles, or dying. To catch the disease a susceptible must come in contact with an infective hence the chance of catching the disease is proportional (with constant β) to the product of the number of susceptibles and infectives. The DE that models the susceptible population is therefore

$$\frac{dS}{dt} = \underbrace{-\beta SI}_{\text{contact}} + \underbrace{\mu K}_{\text{birth}} - \underbrace{\mu S}_{\text{death}}$$

Consider how the population of infectives can change.
They can be a susceptibles who becomes infectives (as described above), they can recover (with rate γ) or they can die (with rate μ as described above). The DE that models the infective population is therefore

$$\frac{dI}{dt} = \underbrace{\beta SI}_{\text{contact}} - \underbrace{\gamma I}_{\text{recover}} - \underbrace{\mu I}_{\text{death}}$$

Consider how the population of recovereds can change.
They are infectives who have recovered, or they can die. The DE that models the recovered population is therefore

$$\frac{dR}{dt} = \underbrace{\gamma I}_{\text{recovered}} - \underbrace{\mu R}_{\text{death}}$$

Note that the recovered equation does not effect the susceptible or infective equations and in fact since the population is assumed constant the number of recovereds can always be determined from the number of susceptibles and the number of infectives. This last equation is not needed in the model.

So we have two first oder nonlinear differential equations in two unknowns (S and I).

$$\begin{aligned} \frac{dS}{dt} &= -\beta SI + \mu K - \mu S \\ \frac{dI}{dt} &= \beta SI - \gamma I - \mu I \end{aligned}$$

These are nonlinear equations hence we need to calculate the critical points and analyse the behaviour near those critical points.

Critical points

Set derivatives equal to zero and solve for S and I.

$$\begin{aligned} 0 &= -\beta SI + \mu K - \mu S \\ 0 &= \beta SI - \gamma I - \mu I \end{aligned}$$

Look at the Jacobian to classify each critical point.

$$\begin{aligned} J(S,I) &= \begin{bmatrix} \frac{\partial}{\partial S}(-\beta SI+\mu K-\mu S) & \frac{\partial}{\partial I}(-\beta SI+\mu K-\mu S) \\ \frac{\partial}{\partial S}(\beta SI-\gamma I-\mu I) & \frac{\partial}{\partial I}(\beta SI-\gamma I-\mu I) \end{bmatrix} \\ &= \begin{bmatrix} -\beta I-\mu & -\beta S \\ \beta I & \beta S-\gamma-\mu \end{bmatrix} \end{aligned}$$

So for critical point $(S,I)=(K,0)$

$$J(K,0)=\begin{bmatrix} -\mu & -\beta K \\ 0 & \beta K-\gamma-\mu \end{bmatrix}$$

Which is a triangular matrix so the eigenvalues are $\lambda=-\mu<0$ and $\lambda=\beta K-\gamma-\mu$

If $\beta K-\gamma-\mu<0$ then the critical point $(S,I)=(K,0)$ is a stable node and hence the $S\to K$, $I\to 0$, that is the disease dies out.

If $\beta K-\gamma-\mu>0$ then the critical point $(S,I)=(K,0)$ is a saddle and hence $(K,0)$ is not the final state. So what is the final state in his case? Presumably it must be the other critical point since the model is bounded. Why is the model bounded?

Example
Consider the case with $K=100$, $\beta=0.001$, $\mu=0.02$, $\gamma=0.03$

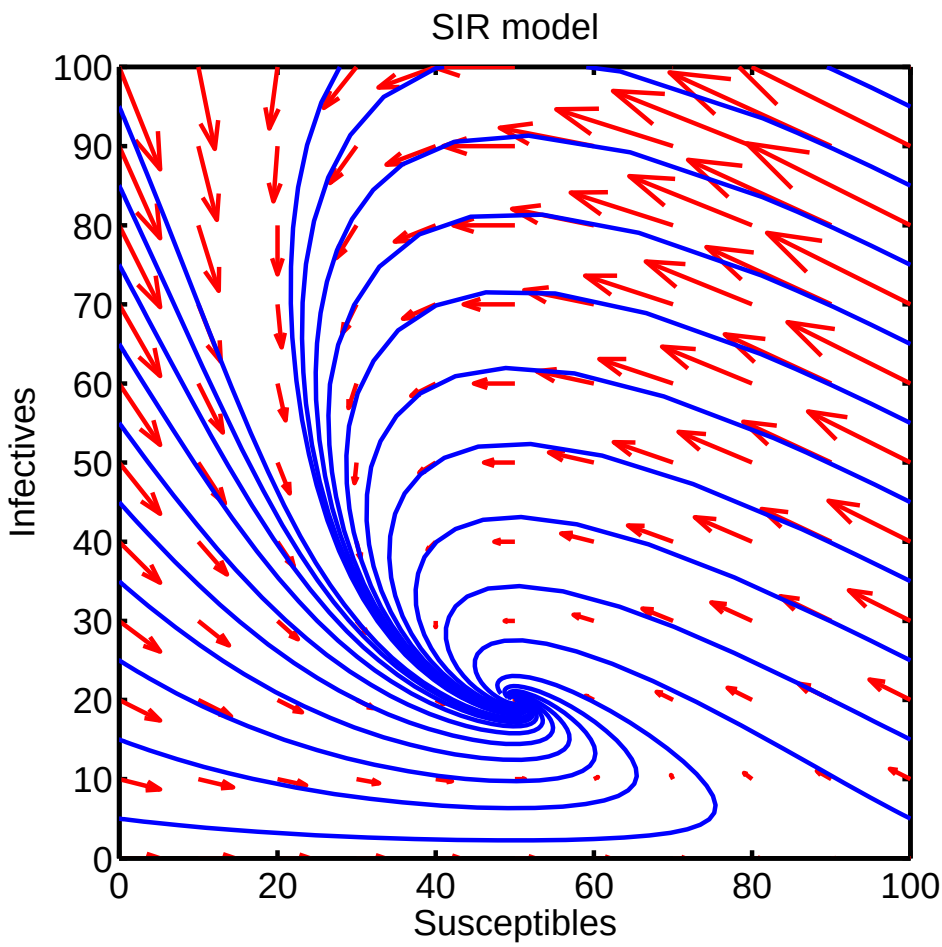
MATLAB code

```
% sir.m
% this M-file plots velocity vectors for the SIR model, plots
% some trajectories, calculates e'values and e'vectors of Jacobian
%
global beta K mu gamma        % makes these global variables
format compact
beta=0.001;  K=100;   mu=0.02;   gamma=0.03;
%
% use quiver to plot the vector field
a=0;  b=100;                  % size of the grid
[S,I]=meshgrid(a:10:b,a:10:b);% a square grid (a to b)x(a to b)
dS=-beta*S.*I+mu*K-mu*S;      % the DEs
dI=beta*S.*I-gamma*I-mu*I;    %
quiver(S,I,dS,dI,2.5,'r')         % plot vector field scaled by 2.5
axis square                   % make the plot square
axis([a b a b])               % restrict the axes to be a to b
hold on
w=[a b];
plot(w,0*w,'k-',0*w,w,'k-')   % adds x and y  axes to plot
xlabel('Susceptibles'); ylabel('Infectives'); title('SIR model')
%
% now add some trajectories by numerically solving the DE
tspan=[0 250];
for icS=0:100:100
  for icI=5:10:95
    initcond=[icS icI];
    [t,y]=ode45('sirfunc',tspan,initcond);
    plot(y(:,1),y(:,2),'b-')
  end
end
hold off
print -depsc sirtraj
%
% now set up the critical points and calculate the Jacobian
% and the eigenvalues and eigenvectors so you can classify the
% critical points
cpS=[K (gamma+mu)/beta];
cpI=[0 mu*K/(gamma+mu)-mu/beta];
for i=1:1:2
  fprintf('\nCritical point number %d is (%5.2f,%5.2f) \n',...
      i,cpS(i),cpI(i))
  Jacobian=[-beta*cpI(i)-mu -beta*cpS(i);...
      beta*cpI(i) beta*cpS(i)-gamma-mu]
  [Jeigvec, Jeigval]=eig(Jacobian)
end;
```

MATLAB code

```
function f=sirfunc(t,y)
% sirfunc.m
% the DEs for the SIR model
global beta K mu gamma
S=y(1);  I=y(2);
f(1)=-beta*S.*I+mu*K-mu*S;
f(2)=beta*S.*I-gamma*I-mu*I;
f=f(:);
return
```

```
sir

Critical point number 1 is (100.00, 0.00)
Jacobian =
   -0.0200   -0.1000
         0    0.0500
Jeigvec =
    1.0000   -0.8192
         0    0.5735
Jeigval =
   -0.0200         0
         0    0.0500

Critical point number 2 is (50.00,20.00)
Jacobian =
   -0.0400   -0.0500
    0.0200    0.0000
Jeigvec =
   0.8452             0.8452
  -0.3381 - 0.4140i  -0.3381 + 0.4140i
Jeigval =
  -0.0200 + 0.0245i        0
        0            -0.0200 - 0.0245i
diary off
```



4 Boundary Value Problems

4.1 Introduction

In contrast to Initial Value Problems where all function values and derivatives are given at one point **Boundary Value Problems (BVP)** have function values and derivatives at two points. Hence it is not possible to construct a Taylor Series about one point and use this to build up the solution as was done with IVPs.

Examples

$$y'' + 3y' + 2y = x \quad \text{where} \quad y(a) = 1 \quad y(b) = 3$$

$$y'''' + y'' + 2y = \sin x \quad \text{where} \quad y(a) = 1 \quad y'(a) = 3 \quad y''(a) = 6 \quad y(b) = 3$$

BVPs do not necessarily have unique solutions . For example

$$y'' + \omega^2 y = 0 \quad \text{where} \quad y(0) = 0 \quad y(\pi) = 0$$

There are two general methods for solving BVPs numerically, **shooting method** and **relaxation method**.

4.2 Shooting method

4.2.1 Shooting method using guesses

Example

How might you go about numerically solving

$$y'' + 3y' + 2y = 2x \quad \text{where} \quad y(0) = 1 \quad y(2) = 3$$

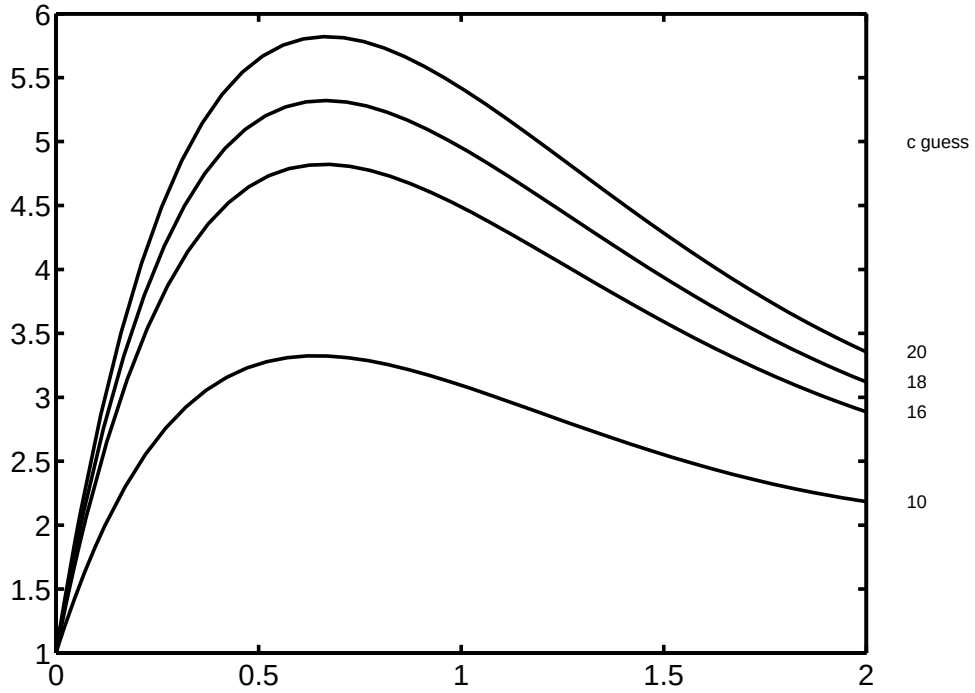
1. Guess a value for $y'(0)$. Now the problem is an IVP.
2. Integrate forward to $x = 2$ using any of the previous methods (Euler, midpoint, R-K, ode45, ...)
3. If $|y(2) - 3| > \varepsilon$ make a new guess and repeat from 2.

MATLAB code

```
% shootingexample.m
% trying to solve y'' + 3y' + 2y = 2x   y(0)=1  y(2)=3
clear all
c=1;                          % the first guess for y'(0)
tspan=[0 2];
while c > -999                % loop through different c values
  c=input('enter c, -1000 to stop  ');
  if c==-1000 break; end      % if c=-1000 jump out of loop
  y0=[1 c];                   % initial condition [y(0) y'(0)]
  [t,y]=ode45('shootingexamplef',tspan,y0);
  plot(t,y(:,1))              % plot current solution
  hold on                     % keep the plot
  lgthy=length(y);            % find the length of the vector y
  disp([y(lgthy,1)])          % show y(x=2) value
  cc=num2str(c);              % convert value of c to a string variable
  text(2.1,y(lgthy,1),cc)     % write value of c on rhs of graph
end
text(2.1,5,'c guess')         % write  c guess  on the plot at (2.1,5)
hold off
print -deps shootingexample
```

MATLAB code

```
function f=shootingexamplef(t,y)
% shootingexamplef.m
% the DE y'' + 3y' + 2y = 2x    written as a system
f(1)=y(2);
f(2)=-3*y(2)-2*y(1)+2*t;
f=f(:);
return
```



Hence there is a solution that has $y(2) = 3$ somewhere between $y'(0) = 16$ and $y'(0) = 18$.

How do you go about finding this value of $y'(0)$ that gives the correct value of $y(2)$?

Treat the problem as a zero finding problem. That is find the value of $y'(0) = c$ that results in $f(c) = y(2) - 3 = 0$. So use one of MATLABs methods for finding zeros of functions.

4.2.2 MATLAB finding zeros

MATLAB has an inbuilt function for finding zeros of a function called `fzero`. Its use is
`r=fzero('funcname',x0)`

searches for a region with a zero of the function defined in funcname.m then uses bisection method with linear or quadratic interpolation to find the zero.

```
>> help fzero

 FZERO  Scalar nonlinear zero finding.
    X = FZERO(FUN,X0) tries to find a zero of FUN near X0. FUN
    (usually an M-file): FUN.M should take a scalar real value
    and return a real scalar value when called with feval:
    F=feval(FUN,X). The value X returned by FZERO is near a point
    where FUN changes sign, or NaN if the search fails.

    X = FZERO(FUN,X0), where X is a vector of length 2, assumes
    X0 is an interval where the sign of FUN(X0(1)) differs from
    the sign of FUN(X0(2)). An error occurs if this is not true.
    Calling FZERO with an interval guarantees FZERO will return
    a value near a point where FUN changes sign.

    X = FZERO(FUN,X0), where X0 is a scalar value, uses X0 as a
    starting guess. FZERO looks for an interval containing a sign
    change for FUN and containing X0.  If no such interval is found,
    NaN is returned.  In this case, the search terminates when the
    search interval is expanded until an Inf, NaN, or complex value
    is found.

    See also ROOTS.

>> diary off
```

Example

Find a zero of $x^3 - 2x^2 - 9 = 0$

```
>> fzero('findzerof',4)
Zero found in the interval: [2.72, 4.9051].

ans =

    3.0000

>> diary off
```

MATLAB code

```
function f=findzerof(x)
% findzerof.m
f=x.^3-2*x.^2-9;
return
```

4.2.3 Shooting method using fzero

So back to the problem of solving

$$y'' + 3y' + 2y = 2x \quad \text{where} \quad y(0) = 1 \quad y(2) = 3$$

Set it up so that there is a MATLAB function that takes as input the guess for the initial slope and returns the error in the right hand boundary condition. Then call this function from `fzero` to find the correct slope that integrates to the correct right hand boundary condition.

MATLAB code

```
% shootingexamplemain.m
% the main M-file for using the shooting method to solve the
% DE defined in shootingexamplef.m
% Uses fzero to solve the equation y'(0)=3 where y'(0) is
% determined in shootingfunc.m by shooting forward using ode45
%
myguess=4;              % a guess at the value of y'(0)
actualvalue=fzero('shootingfunc',myguess)
% use ode45 to get correct solution using actualvalue and plot
tspan=[0 2];    y0=[1 actualvalue];
[t,y]=ode45('shootingexamplef',tspan,y0);
plot(t,y(:,1))
print -deps shootingexampleplot
```

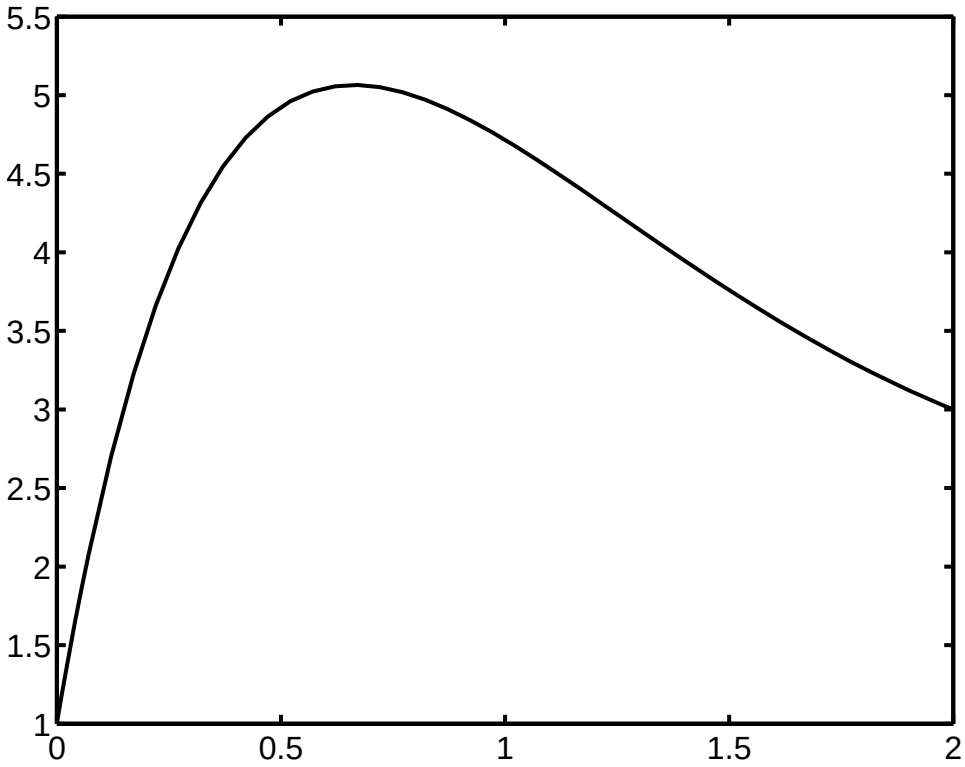
MATLAB code

```
function f=shootingfunc(c)
% set up as a function, the input (c) is the initial slope
% y'(0) and the output (f) is the difference between the value
% of y at x=2 and the number 3 (since y(2)=3 for the correct
% solution.  This function is called by MATLABs fzero
%
% shootingfunc.m
% trying to solve y'' + 3y' + 2y = 2x   y(0)=1  y(2)=3
% using y(0)=1 and y'(0)=c
tspan=[0 2];  y0=[1 c];
[t,y]=ode45('shootingexamplef',tspan,y0);
f=y(length(y),1)-3;  % set function value to y(2)-3
return
```

MATLAB code

```
function f=shootingexamplef(t,y)
% shootingexamplef.m
% the DE y'' + 3y' + 2y = 2x    written as a system
f(1)=y(2);
f(2)=-3*y(2)-2*y(1)+2*t;
f=f(:);
return
```

```
>> shootingexamplemain

actualvalue = 16.9727
```



4.3 Relaxation methods

Relaxation methods differ substantially from the previous shooting methods. The function is not integrated but rather an approximate solution is made that fits the governing differential equation. This results in a system of equations where the unknowns are the values of the function at the interior points. To do this we need to determine some approximations to the derivatives in any given differential equation.

4.3.1 Finite differences

Finite differences is the procedure where we replace any derivatives by a **finite difference** approximation. For example recall the definition of the derivative

$$\frac{dy}{dx} \approx \frac{y(x+\Delta x)-y(x)}{\Delta x}$$

alternatively this could be written

$$\begin{aligned} \left.\frac{dy}{dx}\right|_n &\approx \frac{y_{n+1}-y_n}{x_{n+1}-x_n} \\ &= \frac{y_{n+1}-y_n}{h} \end{aligned}$$

where $y_n = y(x_n)$ and h is the difference in x steps or just the space between x points. This is known as a **forward approximation** to the first derivative as it uses the current point (n) and the next point $(n+1)$. There is also a **backward approximation** that uses the current point (n) and the previous point $(n-1)$. This is given by

$$\left.\frac{dy}{dx}\right|_n \approx \frac{y_n - y_{n-1}}{h}$$

Both the forward and backward approximations are **first order accurate**, that is the first neglected term is order Δx.

There is in fact a second order accurate approximation known as the **central approximation** given by

$$\left.\frac{dy}{dx}\right|_n \approx \frac{y_{n+1} - y_{n-1}}{2h}$$

These can be used to find approximations to the higher derivatives.

$$\left.\frac{d^2y}{dx^2}\right|_n \approx \frac{y_{n+1} - 2y_n + y_{n-1}}{h^2}$$

This is the central approximation to the second derivative and is second order accurate.

4.3.2 Using finite differenes to solve a BVP

Finite differences can be used to solve a BVP. What results is simply solving a system of linear equations which is simple in MATLAB.

Algorithm

1. Divide up the region into N equal subintervals. Note that the differential equation must be true at each grid point in the region.
2. Replace the derivatives in the differential equation by thier finite difference approximations at each grid point in the region.
3. If possible use the boundary conditions to replace some of the values. This is usually only possible for the equations at each end of the interval.
4. Rearrange the equations so that they are written as a matrix system of equations. The unknowns are the value of the solution at each grid point.
5. Solve the matrix system of equations. This then gives the solution at each grid point.

Example

Numerically solve

$$y'' + 3y' + 2y = 2x \quad \text{where} \quad y(0) = 1 \quad y(2) = 3$$

1. First divide up the region into N subintervals of width h where

$$h = \frac{b-a}{N}$$

so

$$x_n = a + hn \quad \text{for} \quad n = 0 \ldots N$$

2. Replace each derivative by its approximate value at each **interior** point.

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{h^2} + 3\,\frac{y_{n+1} - y_{n-1}}{2h} + 2y_n \approx 2x_n \quad \text{for} \quad n = 1 \ldots N-1$$

This results in $N-1$ equations in the $N-1$ unknowns $y_1, y_2, \ldots y_{N-1}$.

3. The values at the edge points are known ($y_0 = y(a) = 1$, $y_N = y(b) = 3$ in this case) The equations for $n = 1$ and $n = N-1$ are slightly different to the others ($n = 2, \ldots N-2$) as they involve y_0 and Y_N respectively which are known as they are just the boundary values.

$$\frac{y_2 - 2y_1 + y_0}{h^2} + 3\,\frac{y_2 - y_0}{2h} + 2y_1 \approx 2x_1 \quad \text{for} \quad n = 1$$

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{h^2} + 3\,\frac{y_{n+1} - y_{n-1}}{2h} + 2y_n \approx 2x_n \quad \text{for} \quad n = 2 \ldots N-2$$

$$\frac{y_N - 2y_{N-1} + y_{N-2}}{h^2} + 3\,\frac{y_N - y_{N-2}}{2h} + 2y_{N-1} \approx 2x_{N-1} \quad \text{for} \quad n = N-1$$

4. Mutiply by h^2 to simplify, collect all terms with the same unknown value and written out in full we have

$$
\begin{aligned}
(1-3h/2)y_0+(2h^2-2)y_1+(1+3h/2)y_2 &= 2h^2x_1 \\
(1-3h/2)y_1+(2h^2-2)y_2+(1+3h/2)y_3 &= 2h^2x_2 \\
(1-3h/2)y_2+(2h^2-2)y_3+(1+3h/2)y_4 &= 2h^2x_3 \\
\vdots \qquad\qquad \vdots \qquad\qquad \vdots & \\
(1-3h/2)y_{N-3}+(2h^2-2)y_{N-2}+(1+3h/2)y_{N-1} &= 2h^2x_{N-2} \\
(1-3h/2)y_{N-2}+(2h^2-2)y_{N-1}+(1+3h/2)y_N &= 2h^2x_{N-1}
\end{aligned}
$$

Now y_0 and y_N are known (since they are the boundary conditions) so put them on the right hand side with all the other known parts gives

$$
\begin{aligned}
(2h^2-2)y_1+(1+3h/2)y_2 &= 2h^2x_1-(1-3h/2)y_0 \\
(1-3h/2)y_{n-1}+(2h^2-2)y_n+(1+3h/2)y_{n+1} &= 2h^2x_n \qquad \text{for} \quad n=2\ldots N-2 \\
(1-3h/2)y_{N-2}+(2h^2-2)y_{N-1} &= 2h^2x_{N-1}-(1+3h/2)y_N
\end{aligned}
$$

5. This is a system of $N-1$ equations in the $N-1$ unknowns $y_1, y_2, \ldots y_{N-1}$. Substituting the known end points ($y_0=1$ and $y_N=3$) and writing in matrix form this can be easily solved in MATLAB using any number of solution methods depending on the size of the matrix (direct inversion, iterations etc.). In matrix form it can be written

$$AY=b$$

where the matrix A has coefficients as above, Y is the vectors of unknowns $Y=(y_1,y_2,\ldots,y_{N-1})^T$ and b is the known right hand side vector.

$$A = \begin{bmatrix} 2h^2-2 & 1-3h/2 & 0 & 0 & 0 & \dots & 0 \\ 1-3h/2 & 2h^2-2 & 1+3h/2 & 0 & 0 & \dots & 0 \\ 0 & 1-3h/2 & 2h^2-2 & 1+3h/2 & 0 & \dots & 0 \\ 0 & 0 & 1-3h/2 & 2h^2-2 & 1+3h/2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & 0 & 1-3h/2 & 2h^2-2 & 1+3h/2 \\ 0 & \dots & 0 & 0 & 0 & 1-3h/2 & 2h^2-2 \end{bmatrix}$$

$$b = \begin{bmatrix} 2h^2x_1 - (1-3h/2)y_0 \\ 2h^2x_2 \\ 2h^2x_3 \\ 2h^2x_4 \\ \vdots \\ 2h^2x_{N-2} \\ 2h^2x_{N-1} - (1+3h/2)y_N \end{bmatrix}$$

What do you notice about the matrix A? You will see that it is 'tri-diagonal' (only 3 diagonals are non-zero) and each diagonal has the same value. This makes it particularly easy to program in MATLAB using the `diag` command.

Once you have coded A, and b it is a simple matter to solve the system $AY = b$ to get the solution Y. This is then the approximation to the solution of the original ODE.

Coding the diagonal matrix A is relatively easy in MATLAB as there is a command `diag` that is used to enter values into a diagonal matrix. The `ones` command is also useful for making vectors of a certain length with the same value in each position.

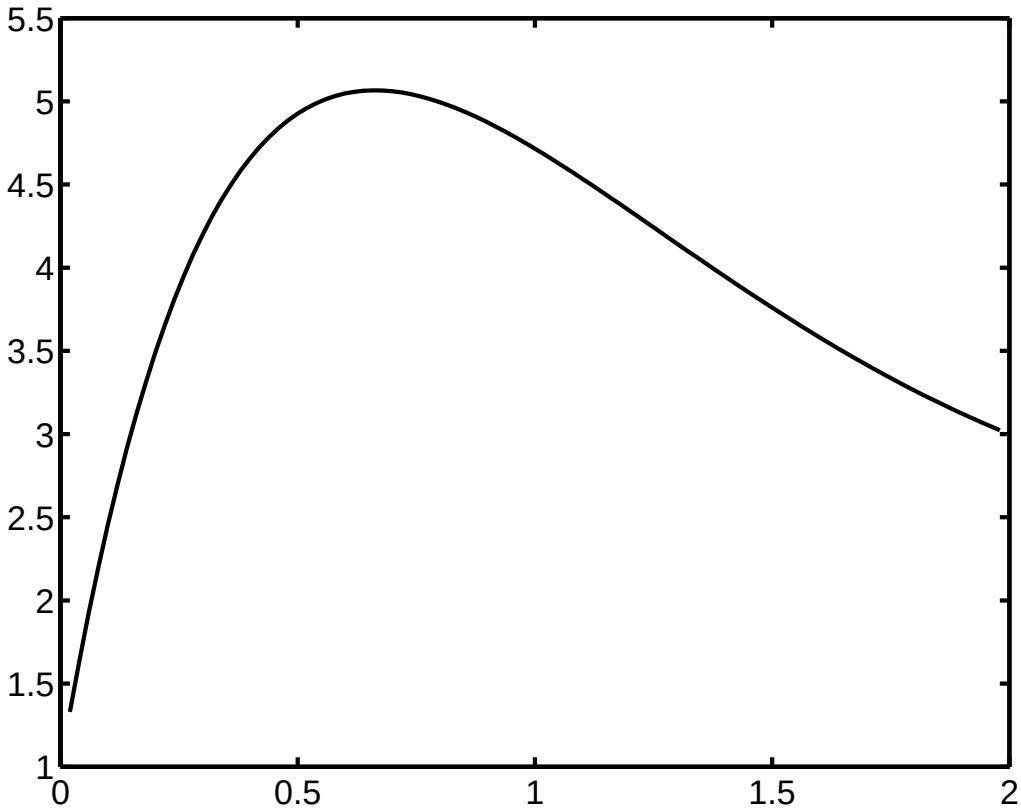
Solving the system $AY = b$ is easy using the 'backslash divide' command `y=A\b` .

```
>> help diag

 DIAG Diagonal matrices and diagonals of a matrix.
    DIAG(V,K) when V is a vector with N components is a square
    matrix of order N+ABS(K) with the elements of V on the K-th
    diagonal. K = 0 is the main diagonal, K > 0 is above the main
    diagonal and K < 0 is below the main diagonal.

    DIAG(V) is the same as DIAG(V,0) and puts V on the main
    diagonal.

    DIAG(X,K) when X is a matrix is a column vector formed from
    the elements of the K-th diagonal of X.

    DIAG(X) is the main diagonal of X. DIAG(DIAG(X)) is a diagonal
    matrix.

    Example
       m = 5;
       diag(-m:m) + diag(ones(2*m,1),1) + diag(ones(2*m,1),-1)
    produces a tridiagonal matrix of order 2*m+1.

    See also SPDIAGS, TRIU, TRIL.

 Overloaded methods
    help sym/diag.m

>> diary off
```

```
>> help ones

 ONES   Ones array.
    ONES(N) is an N-by-N matrix of ones.
    ONES(M,N) or ONES([M,N]) is an M-by-N matrix of ones.
    ONES(M,N,P,...) or ONES([M N P ...]) is an
    M-by-N-by-P-by-...  array of ones.
    ONES(SIZE(A)) is the same size as A and all ones.

    See also ZEROS.

>> diary off
```

MATLAB code

```
% fdexample.m
%
% solve y'' + 3y' + 2y = 2x  y(0)=1  y(2)=3
% using finite differences
a=0; b=2;        % endpoints
y0=1; yN=3;      % y values at endpoints
N=100;           % number of points
h=(b-a)/N;       % x step size
x=a+h:h:b-h;     % set up vector of x points (interior points)
% set up the matrix and solve Ay=RHS for y
A=(2*h^2-2)*diag(ones(1,N-1));      % main diagonal elements
A=A+(1-3*h/2)*diag(ones(1,N-2),-1); % one below main diagonal
A=A+(1+3*h/2)*diag(ones(1,N-2),1);  % one above main diagonal
RHS=2*h^2*x;
RHS(1)=RHS(1)-(1-3*h/2)*y0;
RHS(N-1)=RHS(N-1)-(1+3*h/2)*yN;
y=A\RHS';
plot(x,y)
```



Example

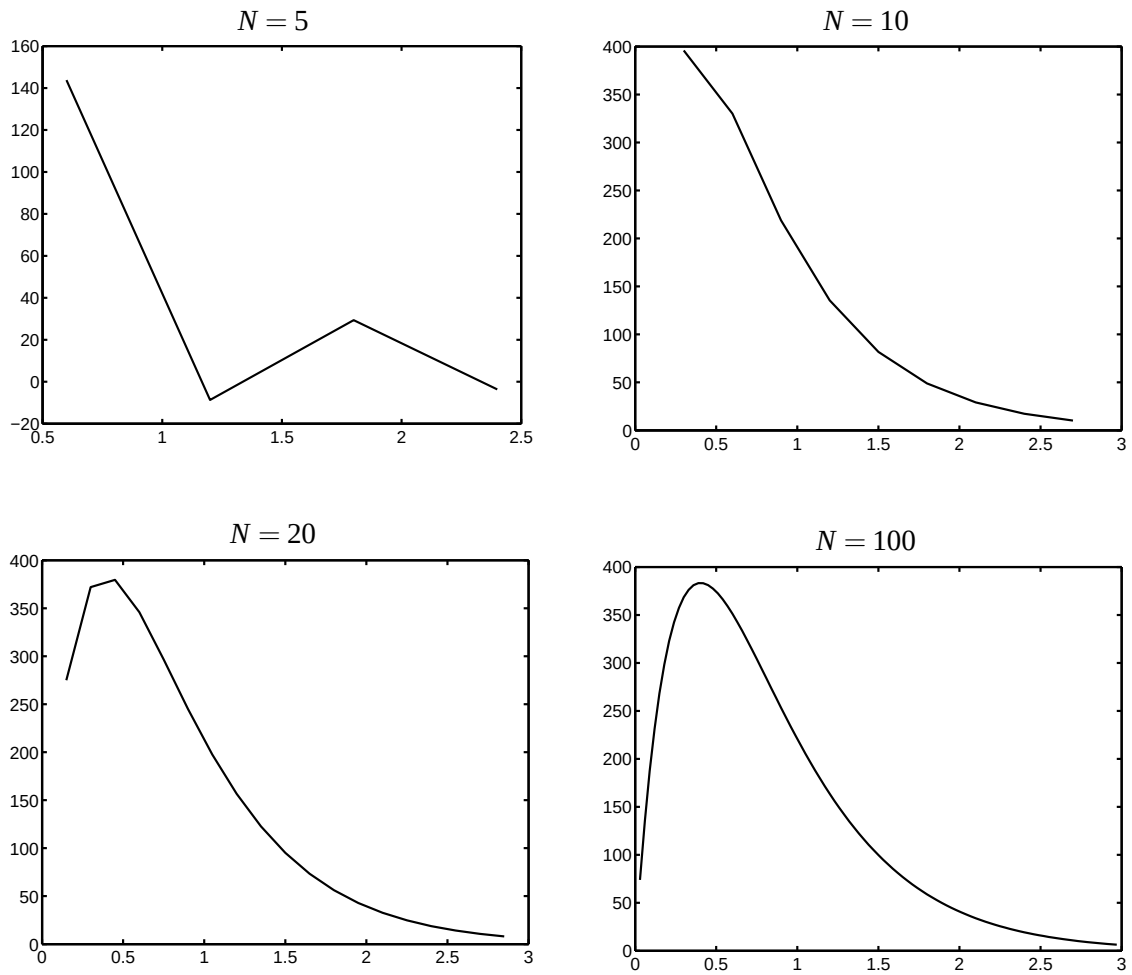
Solve

$$y'' + 5y' + 6y = \cos x \quad \text{where} \quad y(0) = 2 \quad y(3) = 6$$

using finite differences.

MATLAB code

```
% fdexample2.m
%
% solve y'' + 5y' + 6y = cos x  y(0)=2  y(3)=6
% using finite differences
a=0; b=3;        % endpoints
ya=2; yb=6;      % y values at endpoints
N=100;          % number of points
h=(b-a)/N;       % x step size
x=a+h:h:b-h;     % set up vector of x points (interior points)
% set up the matrix and solve Ay=RHS for y
A=(6*h^2-2)*diag(ones(1,N-1));      % main diagonal elements
A=A+(1-5*h/2)*diag(ones(1,N-2),-1); % one below main diagonal
A=A+(1+5*h/2)*diag(ones(1,N-2),1);  % one above main diagonal
RHS=h^2*cos(x);
RHS(1)=RHS(1)-(1-5*h/2)*ya;
RHS(N-1)=RHS(N-1)-(1+5*h/2)*yb;
y=A\RHS';
plot(x,y)
```



4.3.3 Comments on solving matrix equations

There are many numerical problems with solving the matrix equation $A\mathbf{x} = \mathbf{b}$

The matrix equations that result from the finite difference method are **tridiagonal**. That is they have entries down the 3 central diagonals and zeros elsewhere. These are relatively efficient to solve $A\mathbf{x} = \mathbf{b}$ for.

Suggested Solution Methods

1. **Inverse** This can be very slow for large matrices.
   What if inverse doesn't exist?

2. **Iteration**
   Make a guess and use the equations to refine that guess until some convergence criteria is met.
   Usually the best method for large matrices (bigger than 25×25).
   Most common method is known as the Gauss-Seidel Iteration method

3. **Row reduction then back-substitution**

4.3.4 Finite elements

Finite differences are only one of the many method of solving BVPs. Another very popular method is known as the **finite element** method. The concept underlying finite difference methods is that approximations are made to the derivatives in the differential equation. In contrast to this for finite element methods the solution is approximated by a sequence of model functions and the error in using these functions is minimized.

For finite element methods it is more convenient to work on a BVP that has *homogeneous* boundary conditions. That is the value of the function at the end points is zero. Fortunately we can make a change of variable to ensure this. For example consider the problem

$$y'' + p(x)y' + q(x) = r(x) \qquad y(x_L) = y_L \quad y(x_R) = y_R$$

we can transform this to a problem with homogeneous boundary conditions by the transformation

$$y(x) = Y(x) + ax + b$$

Which can be shown to give

$$a = \frac{1}{x_L - x_R}(y_L - y_R) \qquad b = \frac{1}{x_L - x_R}(-x_R y_L + x_L y_R)$$

Example

What does

$$y'' + 3y' + 2y = 5x^2 \qquad y(0) = 2 \quad y(2) = 3$$

become when it is transformed to have homogeneous boundary conditions

4.3.5 Symbolic form of a BVP

Consider an ordinary differential equation with homogeneous boundary conditions this can be written in sysmbolic form as

$$\mathscr{L}y = r(x) \qquad y(x_L) = 0 \quad y(x_R) = 0$$

where $\mathscr{L}$ is a sysmbol that represents all the differential operators in the differential equation.

Example

The ODE

$$y'' + 3x^2 y'' + 2e^x y = 5x^2$$

can be represented as

$$\mathscr{L}y = 5x^2 \qquad \text{where} \quad \mathscr{L} \equiv \frac{d^2}{dx^2} + 3x^2 \frac{d}{dx} + 2e^x$$

4.3.6 Finite element theory

For the BVP given by

$$\mathscr{L}y = r(x) \qquad y(x_L) = 0 \quad y(x_R) = 0$$

Algorithm

1. Select a set of n independent expansion functions. Call these $\phi_i(x)$ for $i = 1, \ldots, n$. The exact form of these functions will be specified later. The crux of the finite element method is that the approximate solution (denoted $Y(x)$) is written as some linear combination of these expansion functions, namely

   $$y(x) \approx Y(x) = \sum_{i=1}^{n} c_i \phi_i(x)$$

   The aim is to find the coefficients c_i to get the best solution possible.

2. Construct what is known as a residual function (denoted $\Delta(x)$) which is a measure of the error of the approximate solution

   $$\begin{aligned} \Delta(x) &= \mathscr{L}Y(x) - r(x) \\ &= \mathscr{L}\sum_{i=1}^{n} c_i \phi_i(x) - r(x) \\ &= \sum_{i=1}^{n} c_i \mathscr{L}\phi_i(x) - r(x) \end{aligned}$$

3. Establish a set of conditions that allow us to find the coefficients c_i. To do this choose a set of weight functions $w_i(x)$ for $i = 1, \ldots, n$. Again these are as yet unspecified. These weight functions have the following properties

   (a) They are normalised so that

$$\int_{x_L}^{x_R} w_j(x)dx = 1 \qquad \text{for} \quad j = 1, \ldots, n$$

   (b) The weighted residual for each fucntion is zero

$$\int_{x_L}^{x_R} w_j(x)\Delta(x)dx = 0 \qquad \text{for} \quad j = 1, \ldots, n$$

4. It is this last condition 3(b) that gives conditions that enable the coefficients c_i to be found. By substituting the definition of $\Delta(x)$ above into this expression we get for each $j = 1, \ldots, n$

$$\begin{aligned}
\int_{x_L}^{x_R} w_j(x)\Delta(x)dx &= 0 \\
\int_{x_L}^{x_R} w_j(x)\left[\sum_{i=1}^{n} c_i \mathscr{L}\phi_i(x) - r(x)\right] dx &= 0 \\
\int_{x_L}^{x_R} w_j(x)\sum_{i=1}^{n} c_i \mathscr{L}\phi_i(x)dx - \int_{x_L}^{x_R} w_j(x)r(x)dx &= 0 \\
\sum_{i=1}^{n} c_i \int_{x_L}^{x_R} w_j(x)\mathscr{L}\phi_i(x)dx &= \int_{x_L}^{x_R} w_j(x)r(x)dx
\end{aligned}$$

If the weight functions $(w_j(x))$ and the expansion functions $(\phi_j(x))$ are specified then the right hand side of the last expression is known.

$$b_j = \int_{x_L}^{x_R} w_j(x)r(x)dx$$

and the part

$$\int_{x_L}^{x_R} w_j(x)\mathscr{L}\phi_i(x)dx = M_{ji}$$

is also known. Hence the equation reduces to

$$\sum_{i=1}^{n} c_i M_{ji} = b_j \qquad \text{for} \quad j = 1, \ldots, n$$

This is just the matrix equation

$$Mc = b$$

So provided $w_j(x)$ and $\phi_j(x)$ are specified then the coefficients to use as the linear combination of the expansion functions to give the approximate solution are given by solving $Mc = b$. This gives the c_i in

$$y(x) \approx Y(x) = \sum_{i=1}^{n} c_i \phi_i(x)$$

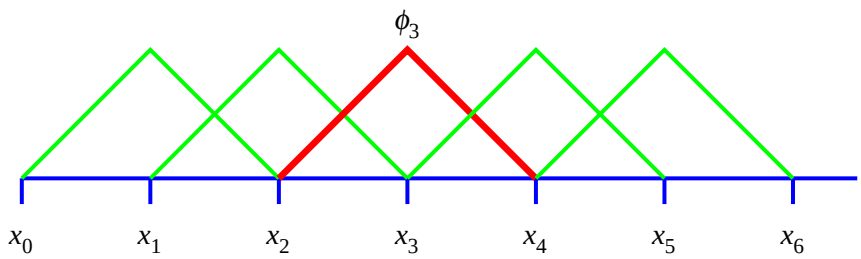
and so the approximate solution has been found.

Choice of weight and expansion functions

There are many choices that can be made for the weight functions ($w_j(x)$) and the expansion functions ($\phi_j(x)$). Different choices give different types of methods. By far the most common and what has come to be known as the collocation method is to choose them equal, namely $w_j(x) = \phi_j(x)$. The **finite element** method uses **"triangular hat"** functions for both the weight functions ($w_i(x)$) and the expansion functions ($\phi_i(x)$).

$$\phi_i(x) = \begin{cases} \frac{1}{h}\left(x - x_{i-1}\right) & \text{for} \quad x_{i-1} < x < x_i \\ \frac{1}{h}\left(x_{i+1} - x\right) & \text{for} \quad x_i < x < x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

These functions are triangles that span 3 grid points and are zero everwhere else. For example ϕ_3 is a triangle that begins at zero at x_2 increases to a maximum at x_3 and then decreases to zero at x_4.



The benefit of these functions is that they are *local*. That it is only nonzero in the immediate neighbourhood of the central point. This guarantees that the resulting matrix eqaution will be sparse (have many zero entries) and hence can be solved effeciently using iterative methods.

These triangular hat functions are useful as they have some special properties that make calculating the M matrix easier.

$F(x)$	$\int_{x_L}^{x_R} \phi_i F(x)\, dx$	
ϕ_i	$2h/3$	
$\phi_{i\pm1}$	$h/6$	(zero for all other indices)
$\phi'_{i\pm1}$	$\pm1/2$	(zero for all other indices)
ϕ''_i	$-2/h$	
$\phi''_{i\pm1}$	$1/h$	(zero for all other indices)
1	h	
x	ih^2	
x^2	$(i^2+1/6)h^3$	
x^3	$i(i^2+1/2)h^4$	

4.3.7 Finite element example

Solve

$$y'' + y = 1 \qquad y(0) = 0 \quad y(\pi/2) = 0$$

Now $\mathscr{L} \equiv \frac{d^2}{dx^2} + 1$ so to calculate M_{ji} break it down into each term such that

$$\begin{aligned} M_{ji} &= \int_0^{\pi/2} \phi_j(x) \mathscr{L} \phi_i(x) dx \\ &= \int_0^{\pi/2} \phi_j(x) \left[\frac{d^2}{dx^2} + 1 \right] \phi_i(x) dx \\ &= \int_0^{\pi/2} \phi_j(x) \frac{d^2 \phi_i(x)}{dx^2} dx + \int_0^{\pi/2} \phi_j(x) \phi_i(x) dx \end{aligned}$$

and so from the table of special properties of the triangular functions

$$M = \begin{bmatrix} -2/h & 1/h & 0 & \dots & 0 & 0 \\ 1/h & -2/h & 1/h & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -2/h & 1/h \\ 0 & 0 & 0 & \dots & 1/h & -2/h \end{bmatrix} + \begin{bmatrix} 2h/3 & h/6 & 0 & \dots & 0 & 0 \\ h/6 & 2h/3 & h/6 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 2h/3 & h/6 \\ 0 & 0 & 0 & \dots & h/6 & 2h/3 \end{bmatrix}$$

Calculating the right hand side gives

$$\begin{aligned} b_j &= \int_0^{\pi/2} w_j(x) r(x) dx \\ &= \int_0^{\pi/2} \phi_j(x) \; 1 \; dx \end{aligned}$$

and so from the tables

$$b = \begin{bmatrix} h \\ h \\ h \\ \vdots \\ h \\ h \end{bmatrix}$$

Now find the coefficients c_i in

$$y(x) \approx Y(x) = \sum_{i=1}^{n} c_i \phi_i(x)$$

by solving the matrix equation

$$Mc = b$$

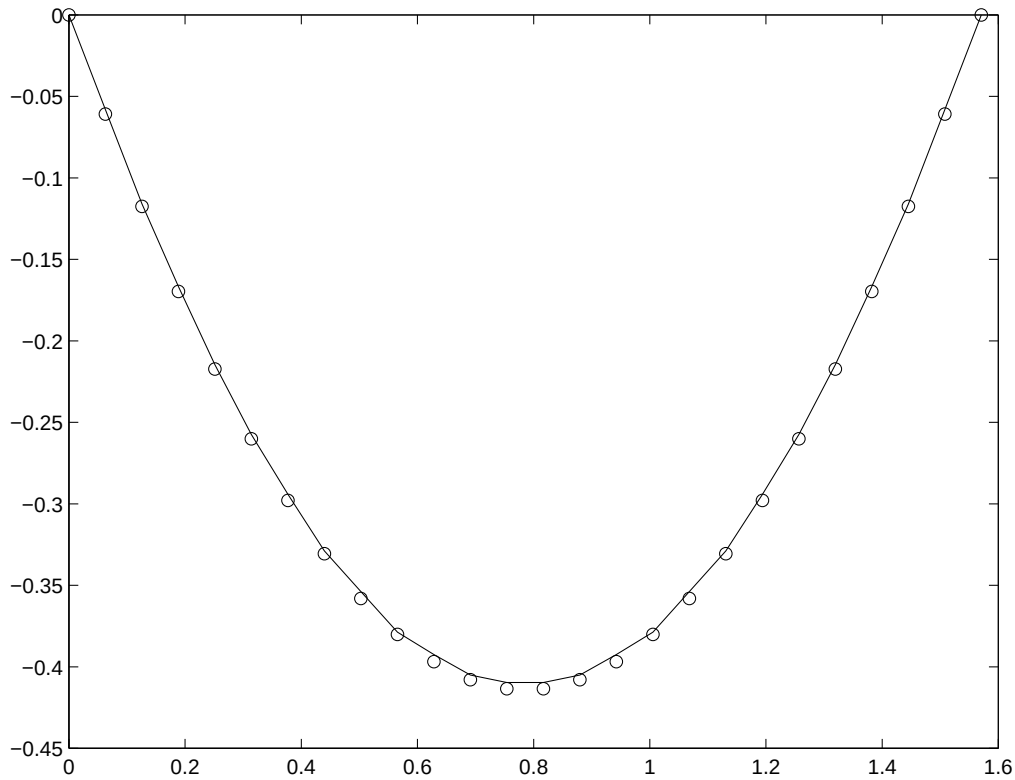
with M and b above.

MATLAB code

```
% feexample.m
%
% solving L{y} = y'' + y = 1   with  y(0)=0  y(pi/2)=0
% using finite element method
% Exact solution is  y(x) = -sin(x) - cos(x) + 1
%
clear all
global a b N
a=0;  b=pi/2;  N=10;  % N interior points
h=(b-a)/(N+1);
xk=a+h:h:b-h;        % interior points
% set up and solve Mc=RHS
M=(-2+2/3*h^2)*diag(ones(1,N));      % main diagonal elements
M=M+(1+h^2/6)*diag(ones(1,N-1),-1);  % one below main diagonal
M=M+(1+h^2/6)*diag(ones(1,N-1),1);   % one above main diagonal
M=M/h;                               % multiplicative factor
RHS=h*ones(N,1);                     % right hand side vector
c=M\RHS                              % solve for c the coefficients
% plot numerical solution and exact solution and calculate error
npts=26;
x=a:(b-a)/(npts-1):b;          % vector to calculate solution at
% call feexamplef.m which is a function that calculates the solution
% at a given point i.e. it works out y(x)=sum(c(i)*phi_i(x))
% where the phi_i are the 'triangular hat' functions
for k=1:npts
  y(k)=feval('feexamplef',c,x(k));
end
plot(x,y)
hold on
exact=-sin(x)-cos(x)+1;
plot(x,exact,'ro')
ylabel('exact and numerical')
error=abs(exact-y);
maxerror=max(error)
hold off
```

MATLAB code

```
function f=feexamplef(c,x)
% feexamplef.m
% function that calculates y(x)=sum(c(i)*phi_i(x))
% where the phi_i are the 'triangular hat' functions centered around x_i
%
global a b N
h=(b-a)/(N+1);
xk=a+h:h:b-h;
if (x <= xk(1))       % left most function
  f=c(1)*(x-a)/h;
end
for i=1:N-1    % all interior functions that have left and right parts
  if (x >= xk(i)) & (x <= xk(i+1))
    right=c(i)*(xk(i+1)-x)/h;
    left =c(i+1)*(x-xk(i))/h;
    f=left+right;
  end
end
if (x >= xk(N))      % right most function
  f=c(N)*(b-x)/h;
end
return
```



4.3.8 Finite element summary

For the BVP

$$\mathscr{L}y = r(x) \qquad y(x_L) = 0 \quad y(x_R) = 0$$

1. Choose n expansion functions $\phi_i(x)$ for $i = 1, 2, \ldots, n$.
2. Choose n weight functions $w_i(x)$ for $i = 1, 2, \ldots, n$.
3. Let

$$w_i(x) = \phi_i(x) = \begin{cases} \frac{1}{h}\left(x - x_{i-1}\right) & \text{for} \quad x_{i-1} < x < x_i \\ \frac{1}{h}\left(x_{i+1} - x\right) & \text{for} \quad x_i < x < x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

4. Numerical solution $Y(x) = \sum_{i=1}^{n} c_i \phi_i(x)$
5. Solve $M\mathbf{c} = \mathbf{b}$ for the c_i's where

$$M_{ji} = \int_{x_0}^{x_f} w_j(x) \mathscr{L} \phi_i(x)\, dx$$

$$b_j = \int_{x_0}^{x_f} w_j(x) r(x)\, dx$$

6. Properties of triangular functions

$F(x)$	$\int_{x_0}^{x_f} \phi_i F(x)\, dx$	
ϕ_i	$2h/3$	
$\phi_{i\pm1}$	$h/6$	(zero for all other indices)
$\phi'_{i\pm1}$	$\pm 1/2$	(zero for all other indices)
ϕ''_i	$-2/h$	
$\phi''_{i\pm1}$	$1/h$	(zero for all other indices)
1	h	
x	ih^2	
x^2	$(i^2 + 1/6)h^3$	
x^3	$i(i^2 + 1/2)h^4$	

4.4 Differential eigenvalue equations

Recall the example

$$y'' + \omega^2 y = 0 \quad \text{where} \quad y(0) = 0 \quad y(\pi) = 0$$

which has solution many solutions each one of which is

$$y_n = A_n \sin \omega_n x$$

where A_n is an unknown constant and $\omega_n = n = 0, \pm 1, \pm 2, \ldots$

This equation is known as a **Differential Eigenvalue Equation** as there are only particular values (eigenvalues) for which a solution exists.

How would you solve this numerically given you don't know the eigenvalues ω ?

Treat the eigenvalues as one of the unknowns of the problem. But how ?

We know that the eigenvalue is a constant so we can write a differential equation for it as

$$\frac{d\omega}{dx} = 0$$

but then we also need an extra boundary condition.

$$\omega(0) = \omega_0$$

where ω_0 is the eigenvalue we need to find. So putting $y = y_1$, $dy/dx = y_2$ and $\omega = y_3$ the system to date is

$$\begin{aligned} \frac{dy_1}{dx} &= y_2 \\ \frac{dy_2}{dx} &= -(y_3)^2 y_1 \\ \frac{dy_3}{dx} &= 0 \end{aligned}$$

subject to

$$y_1(0) = 0 \quad y_1(\pi) = 0 \quad y_3(0) = \omega_0$$

Note that now that the system is nonlinear (the second equation).

How do you solve this numerically we have 3 boundary conditions but one of them is unknown (ω_0) and one of them is at $x = \pi$ not $x = 0$. To be able to integrate forward from $x = 0$ we need three initial conditions at $x = 0$ and we also need three known conditions in total for the problem to have a unique solution.

To use a shooting method you need 3 boundary conditions at the same point so introduce a new boundary condition

$$y_2(0) = c$$

where c is a value that has to be determined to satisfy the boundary conditions at the other end and ω_0 is a value that also has to be determined.

But there is still not enough information to find the solution as there are only 2 **known** boundary conditions ($y_1(0) = 0 \quad y_1(\pi) = 0$) so we must introduce another boundary condition that is considered to be known. Looking at the analytic solutions that were found earlier

$$y_n = A_n \sin \omega_n x$$

note that they are only defined to within some arbitrary constant (A_n). To specify a specific solution another condition would also have to be used. This condition is known as the **normalising** condition and consists of choosing some given value for one of the unknown values. So in fact we are free to choose the value of $y_2(0) = c$ to be any (non zero) value as this just fixes the unknown constant.

The system of equations is now

$$\begin{aligned} \frac{dy_1}{dx} &= y_2 \\ \frac{dy_2}{dx} &= -(y_3)^2 y_1 \\ \frac{dy_3}{dx} &= 0 \end{aligned}$$

subject to

$$y_1(0) = 0 \quad y_2(0) = c \ (known) \quad y_3(0) = \omega_0 \ (unknown) \quad y_1(\pi) = 0$$

So there are the required 3 known boundary conditions but 1 unknown one at $x = 0$ that has to be determined to satisfy the boundary conditions at $x = \pi$.

So choose a value for ω_0 integrate forward to $x = \pi$ and check the value of $y_1(\pi)$ if it is equal to zero it is the correct ω_0 if it is not zero then change the ω_0 guess. This can be done using the MATLAB command `fzero`.

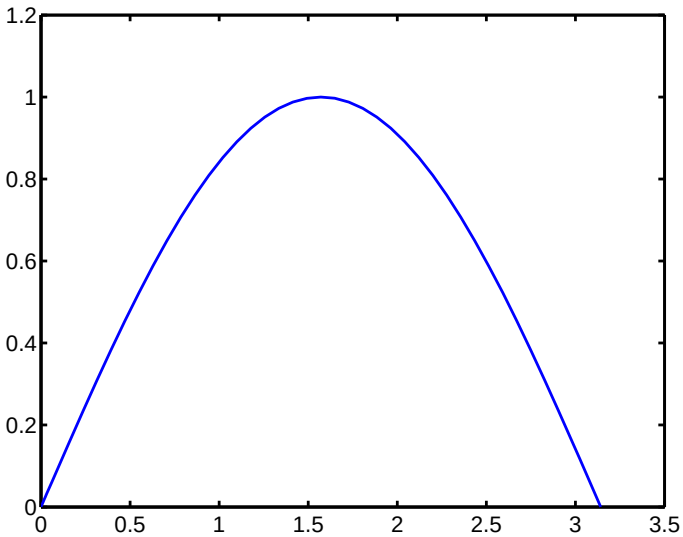
MATLAB code

```
function f=deig(omega0)
% deig.m
%
% solving   y'' + omega^2 y = 0    y(0)=0  y(pi)=0
% This function takes as input the value of omega0 and returns
% the value at y(pi).  If this equals zero then that value of
% omega0 gave a valid solution to the ODE.
% Note the system of equations becomes 3rd order!
% This function is called by fzero to find the value of omega0
% that gives the correct solution.
%
format compact
tspan=[0 pi];
y0=[0 1 omega0];   % y_1(0)=0, y_2(0)=1, y_3(0)=omega0
[t,y]=ode45('deigf',tspan,y0);
plot(t,y(:,1));
i=length(y);       % determine length of solution vector
f=y(i,1);          % set function value = y_1(pi)
return
```

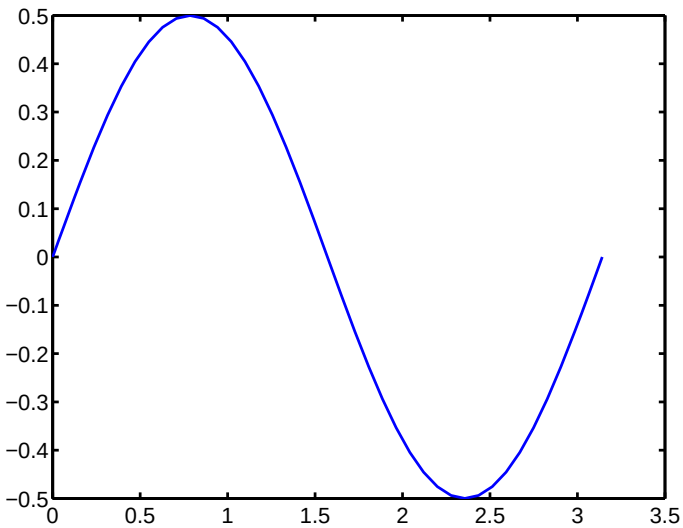
MATLAB code

```
function f=deigf(t,y)
% deigf.m
f(1)=y(2);
f(2)=-y(3).^2.*y(1);
f(3)=0;
f=f(:);           % forces f to be a column vector
return
```

```
>> omega=fzero('deig',1.4,0.0001)

omega = 1.0000
```




```
>> omega=fzero('deig',1.6,0.0001)

omega = 2.0000
```



Example

Consider the differential equation

$$\frac{dz}{dt} = 3\sqrt{z} - \alpha t^2 \quad \text{with} \quad z(0) = 2 \quad z(1) = 4$$

There is one DE but two boundary conditions. Is the system overdetermined?

No since there is a parameter α. This is like an eigenvalue in that there might be a particular value (or values) that make the equation satisfied.

How do you find that particular value?

MATLAB code

```
% alpha.m
% get the correct value of alpha by calling fzero
% fzero in turn calls findalpha.m which is the function that
% integrates the DEs with a guess for alpha.
% The actual DEs are in alphaf.m
%
format compact
format long
initialguess=11
correctalpha=fzero('findalpha',initialguess)
% now use that value of alpha to integrate the
% equations and plot the solution
y0=[2 correctalpha];           % values at t=0
tspan=[0 1];            % integrate equation over t=(0,1)
[t,y]=ode45('alphaf',tspan,y0);
plot(t,y(:,1))          % plot y vs t
print -depsc alpha
```

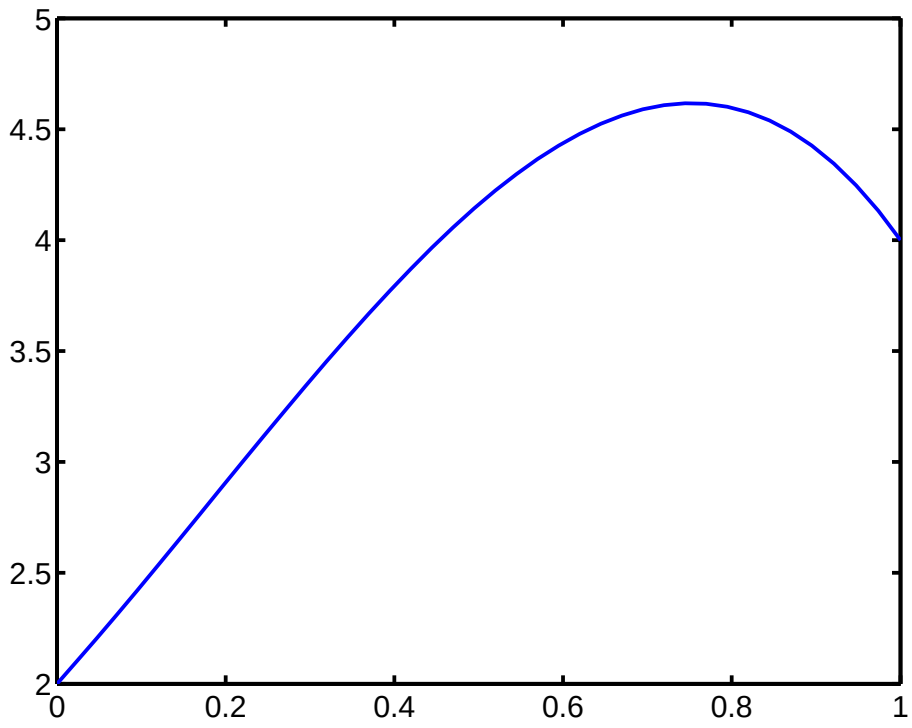
MATLAB code

```
function f=findalpha(alphaguess)
% findalpha.f
% this function takes as input a guess for the constant
% alpha and returns y(1)-4
% if this is close to zero then its a good guess
y0=[2 alphaguess];  % values at t=0
tspan=[0 1];        % integrate equation over t=(0,1)
[t,y]=ode45('alphaf',tspan,y0);
lgthy=length(y);    % determine how long the vector is
% f is a measure of how good the solution matches the desired
% boundary condition which is y(1)=4
% if f is close to zero then this is a good solution.
currentguess=alphaguess ;
f=y(lgthy,1)-4;
sprintf('Current guess for alpha =%10.8f y(1)-4 = %10.8f'...
  ,currentguess,f)
return ;
```

MATLAB code

```
function f=alphaf(t,y)
% alphaf.m    the governing DEs the alpha
% eigenvalue problem
%
% y(1) is z, y(2) is alpha
%
z=y(1);   alpha=y(2);
% set up the functions
f(1)=3*sqrt(z)-alpha*t.^2;  % the DE
f(2)=0;                      % since alpha is a constant
f=f(:);                    % force f to be a column vector
return ;
```

```
alpha
initialguess =
    11
ans =
Current guess for alpha =11.00000000    y(1)-4 = 0.15451914
ans =
Current guess for alpha =10.68887302    y(1)-4 = 0.27958057
ans =
Current guess for alpha =11.31112698    y(1)-4 = 0.02927189
ans =
Current guess for alpha =10.56000000    y(1)-4 = 0.33132966
ans =
Current guess for alpha =11.44000000    y(1)-4 = -0.02266307
ans =
Current guess for alpha =11.38366126    y(1)-4 = 0.00004514
ans =
Current guess for alpha =11.38377326    y(1)-4 = 0.00000001
ans =
Current guess for alpha =11.38377328    y(1)-4 = 0.00000000
ans =
Current guess for alpha =11.38377328    y(1)-4 = 0.00000000
correctalpha =
  11.38377327554915
diary off
```



Index