MA50174 ADVANCED NUMERICAL METHODS – Part 1

I.G. Graham (heavily based on original notes by C.J.Budd)

Aims

1. To be an advanced level course in numerical methods and their applications;
2. To (introduce and) develop confidence in MATLAB (and UNIX);
3. To teach mathematical methods through computation;
4. To develop numerical methods in the context of case studies.

Objectives

1. To learn numerical methods for data analysis, optimisation,linear algebra and ODEs;
2. To learn MATLAB skills in numerical methods, programming and graphics;
3. To apply 1,2 to Mathematical problems and obtain solutions;
4. To present these solutions in a coherent manner for assessment.

Schedule 11 weeks of: 1 workshop in Lab;
1 lecture/problems class;
1 general lecture.

Books

- N and D Higham **"Matlab Guide"** SIAM
- Vettering et al **"Numerical Recipes"** CUP
- A Iserles **"A First Course in the Numerical Solution of DEs"**, CUP
- C.R.MacCluer **"Industrial Maths, Modelling in Industry, Science and Government"** Prentice Hall.
- L.L.Ascher and L.Petzold **"Computer methods for ordinary differential equations and differential algebraic equations"**, SIAM.
- C.B. Moler, Numerical Computing with MATLAB, SIAM.

Other Resources

- I.Graham, N.F. Britton and A. Robinson "MATLAB manual and Introductory tutorials."
- M.Willis, notes on UNIX
- Unix tutorial: http://www.isu.edu/departments/comcom/unix/workshop/unixindex.html
- Additional Unix tutorial: http://people.bath.ac.uk/mbr20/unix/

The website for the first part of the course is

http://www.maths.bath.ac.uk/~igg/ma50174

The website for the second part of the course is

http://www.maths.bath.ac.uk/~cjb/MA50174/MA50174.html

Email addresses of the lecturers:

I.G.Graham@bath.ac.uk

cjb@maths.bath.ac.uk

General Outline

The course will be based around four assignments, each of which is intended to take two weeks and which contribute 20% each to the final total. The assignments can be completed in your own time, although assistance will be given during the lab workshops and you can ask questions during the problem classes. There will also be a benchmark test during which you will be required to perform certain computational tasks in UNIX and MATLAB in a fixed time period. This will be given as a supervised lab session and will count for 20% of the course.
The assessed assignments are:

1. Data handling and function approximation;
2. Numerical Linear Algebra and applications;
3. Initial value ordinary differential equations, with applications to chemical engineering;
4. Boundary value problems.

Contents

Chapter 1

Introduction

1.1 Modern numerical methods for problems in industry

Most industrial problems require the computational solution of a variety of problems. The solution process generally involves three stages.

1. Front end: Problem description in easy manner.
   GUI - Java - Internet

2. Main computational effort. Various tasks including:

   Data input

   Data manipulation . Image and signal processing

   Linear Algebra

   Optimisation

   Solution of ordinary/partial DEs

   Approximation

3. Output: Typical 3D graphics. Often now animated.

Most "traditional" numerical courses concentrate on item 2 and teach this in isolation. They also don't look at software packages. This course will aim to teach computational mathematics and numerical methods in the overall context of 1,2,and 3 through:

- The use of the high level mathematical package MATLAB.
- Understanding of the mathematical principles behind how the various algorithms in MATLAB work and their limitations.
- Learning some ideas of structured programming through using the MATLAB language.
- Looking at all theory and methods in the context of case studies found from real applications.

This course is meant to give you some of the tools that you will need for further Msc courses and also for your project.In particular it links to:

The Modelling Course (MA50176)...	how do you recognise and formulate a problem?
The Scientific Computing Course ...	how do you apply computational
(MA50177 which uses	methods to the large scale
FORTRAN 95)	problems that you encounter in real applications
	when you must use a different programming
	language to get a solution quickly.

Further theory underlying the numerical methods used in this course can be found in the optional courses in this MSc.
In summary, the approach behind the implementation of a numerical method to an industrial problem can be summarised as follows:

- Solve the right problem ... make sure your model is right;
- Solve the problem right ... use the best methods;
- Don't reinvent the wheel ... Be aware of the existence of good software and be prepared to use it.
- Keep a healthy scepticism about your answers.

1.2 An example of this approach

We summarise with a short study of a famous system: the pendulum, which is described by the following second order ordinary differential equation

$$\frac{d^2\theta}{dt^2} + \sin(\theta) = 0 \tag{1.1}$$

subject to initial conditions (for example $\theta = \theta_0$ and $\frac{d\theta}{dt} = 0$ at $t = 0$). It is known that this problem has periodic solutions.

1. *The Traditional Mathematical Approach:*
   Change/simplify problem to
   $$\frac{d^2\theta}{dt^2} + \theta = 0 \ . \tag{1.2}$$
   Now solve this analytically to give
   $$\theta(t) = A\sin(t) + B\cos(t) \tag{1.3}$$
   This approach is fine for small swings, but is bad for large swings, where the solution is qualitatively wrong.

2. *The Traditional Numerical Method:*
   Now rewrite (1.1) as the system:
   $$d\theta/dt = v \tag{1.4}$$
   $$dv/dt = -\ sin(\theta) \tag{1.5}$$
   Divide time into equal steps $t_n = n \triangle t$ and approximate $\theta(t), v(t)$ by
   $$\theta_n \approx \theta(t_n),\ V_n \approx v(t_n) \tag{1.6}$$
   Now discretise the system (1.4), (1.5). For example by using the Euler method:
   $$\frac{\theta_{n+1} - \theta_n}{\triangle t} = V_n$$
   $$\frac{V_{n+1} - V_n}{\triangle t} = -\sin(\theta_n)$$
   Finally write a program (typically as a loop) that generates a sequence of values (θ_n, V_n) plot these and compare with reality. The problem with this approach is that we need to write a new program everytime we want to solve an ODE. Furthermore, any program that we will write will probably not be as accurate as one which has been written (hopefully) and will be tested by "experts". In the above example the values of θ_n and V_n quickly spiral out to infinity and all accuracy is lost.

3. *Software Approach:*
Instead of doing the discretisation "by hand" we can use a software package. We will now see how we use MATLAB to solve this.The approach considered is one that we will use a lot for the remainder of this course.

   (a) Firstly we create a function file **func.m** with the ODE coded into it
   This is the file

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                   %
%  MATH0174 : function file func.m                  %
%                                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This file sets up the equations for
%           theta'' + sin(theta) = 0
%
% To do this it takes thet(1) = theta and thet(2) = theta'
% It then sets up the vector thetprime so that
%
%  thetprime(1) = theta'
%  thetprime(2) = theta''
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  function func.m
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function thetprime=func(t,thet)
thetprime=[thet(2);-sin(thet(1))];
```

   It is a sequence of commands to describe the system:

$$\theta_1' = \theta_2, \;\; \theta_2' = -\sin(\theta_1)$$

   In this file anything preceded by a % is a comment for the reader (it is ignored by the computer). Only the last two lines do anything.

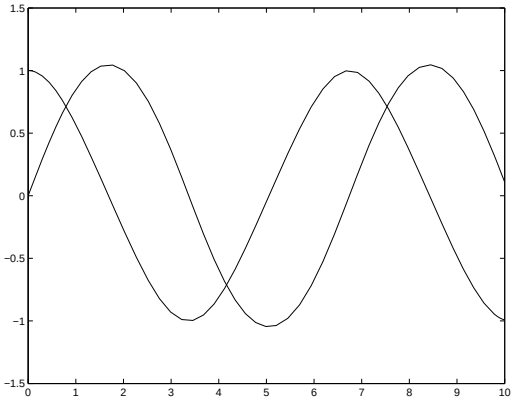
   (b) Now set the initial conditions, and the time span $t \in [0, 10]$ through the commands
   start= $[0; 1]$; tspan = $[0, 10]$

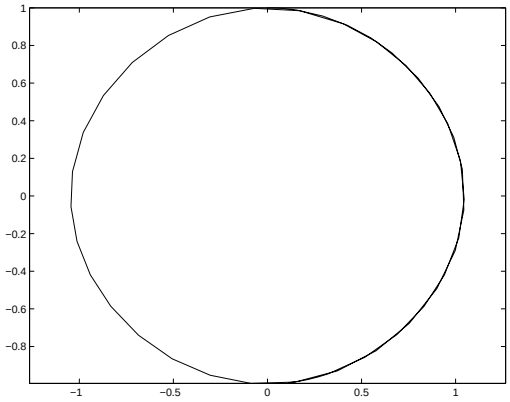
   (c) Next use the MATLAB command ode45 to solve the ODE over the time span; This implements a Runge-Kutta routine to solve (1.1) with a Dormand-Price error estimator. You can specify the error tolerance in advance

   (d) Finally plot the result. This will look like plot(t,thet)
   or plot(thet(:,1), thet(:,2))
   The code to implement this takes the following form:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
%  MATH0174:  Code to solve the pendulum equation given                        %
%             the function file func.m                                         %
%                                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
%  We specify an interval of t in [0,10]
%

tspan = [0 10];

%
%  We specify the initial conditions [thet,v] = [0 1]
%

start = [0;1];

%
% Options sets the tolerance to 1d-6
%

options = odeset('AbsTol',1d-6);

%
% Now solve the ODE using the Dormand-Price explicit Runge-Kutta
% method


[t,thet] = ode45('func',tspan,start,options);

%
% Now plot the solution
%

plot(t,thet)

pause

%
%  Now plot the phase plane
%

theta = thet(:,1);
v     = thet(:,2);

plot(theta,v)
axis('equal')
```

The output of from this code is the following plots. These plots are fairly crude and by using the plotting commands in MATLAB you can make them look a lot better. Over the short





time span we have considered the MATLAB code has given an accurate answer.

4. *The best Approach of all - use Geometry:* It is worth saying that even MATLAB is not perfect. Over a long period of time the MATLAB solution and the true solution of the pendulum will drift apart (see Assignment 3). In this case it is possible to prevent this drift by using a different method based upon the underlying geometry of the solution. An example of this is the Störmer - Verlet method which is given by

$$\begin{aligned} \theta_{n+\frac{1}{2}} &= \theta_n + \frac{\triangle t}{2} V_n \\ V_{n+1} &= V_n - \triangle t \sin(\theta_{n+\frac{1}{2}}) \\ \theta_{n+1} &= \theta_{n+\frac{1}{2}} + \frac{\triangle t}{2} \quad V_{n+1} \end{aligned}$$

In Assignment 3 you will investigate the advantages of using such a method, over the use of ode 45.

Chapter 2

A brief introduction to MATLAB

2.1 Introduction

The majority of this course will use MATLAB. This will be familiar to some and not others. MATLAB is written in C and fits on most PCs running under either Windows or Linux.We start by discussing some of its features. Then the course will teach various numerical analysis concepts through MATLAB.These notes are only meant to be a brief overview. For more detail you should read the MATLAB manual (by Graham, Britton and Robinson) and also the MATLAB guide by Higham & Higham

MATLAB has the following features

1. It has a high level code with many complex mathematical instructions coded as single instructions. e.g.to invert a matrix A you type **inv(A)**, to find its eigenvalues you type **eig(A)**.
2. Matrices and vectors are the basic data type.
3. Magnificent and easy to use graphics, including animation. In fact you often use MATLAB as a very high level plotting engine to produce jpeg, pdf,and postscript files from data sets that may be created from a calculation in another high-level language.
4. There are many specialist toolboxes → PDE
   → control
   → signal and image processing
   → SIMULINK
5. It is a flexible programming language.
6. Very good documentation is available.
7. An increasing amount of code is written in MATLAB both in universities and in industry.
8. It is an interpreted language which makes it flexible but it may be slower than a compiled language.

It is also rapidly getting faster. However it can't yet compete with FORTRAN for sheer speed, C^{++} for total flexibility or JAVA or Visual Basic for internet applications. We will have the opportunity to compare MATLAB with FORTRAN later on in this course.

2.2 General Points

MATLAB has essentially *one* data type. This is the *complex matrix*. It performs operations on matrices using state of the art numerical algorithms (written in C). For this course we will need to know something about the *speed* and *reliability* of these algorithms and their *suitability* for certain operations, but we

don't need to know the precise details of their implementation to use MATLAB. Later on you will write some of your own algorithms which you can compare with MATLAB ones.See other MSc courses for more detailed descriptions of the algorithms themselves..

- To give an example. Suppose A is a matrix eg.

$$A = \begin{pmatrix} 10 & 9 & 8 & 7 \\ 6 & 5 & 3 & 4 \\ 1 & 2 & 7 & 8 \\ 0 & 3 & 0 & 5 \end{pmatrix}$$

  We set this up in MATLAB via the command:

$$A = [\underbrace{10\ 9\ 8\ 7}_{\text{First row}}\ ;\ \overbrace{6\ 5\ 3\ 4}^{\text{Second row}}\ ;1\ 2\ 7\ 8\ ;0\ 3\ 0\ 5\ ];$$

  Here the semicolons between the groups of numbers are row separators. The last semi-colon suppresses the output.

- We can invert A to give the matrix B by
  B = inv (A).
  Usually MATLAB uses a direct method based on Gaussian elimination to do this inversion. This is good, but may not be optimal for certain problems. For these MATLAB has a suite of different other methods, mostly based on iterative techniques such as "gmres". You can also write your own routines and we will do this in assignment 4.

IMPORTANT POINT

MATLAB inverts a matrix numerically to within a certain precision. It can work with large matrices. Other packages such as MAPLE invert a matrix **exactly** using symbolic operations. This process takes a lot longer and can only be used for small matrices.

2.3 Manipulating Matrices and Vectors.

Most MATLAB commands are pretty obvious; *but* setting up and manipulating matrices and vectors needs a little care.
To set up a matrix such as:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

we type:

$$> A = [1\ 2\ ;3\ 4\ ].$$

To set up a new vector such as

$$x = (1\ 2\ 3)$$

we type:

$$> x = [1\ 2\ 3]$$

and to set up a column vector such as

$$x = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$$

we either type:

$$> x = [3;\ 4;\ 5]$$

or we type:

$$> x = [3\ 4\ 5]' \nwarrow \text{ transpose operator}$$

All matrices have a size. So if A is an $n \times m$ matrix (n rows, m columns), then the command `size(A)` returns the vector [n m]. Thus

$$\begin{array}{ll} > & \text{size}([4\ 8]) = [1\ 2] \\ & \text{size}([4\ 8]') = [2\ 1] \end{array}$$

If size$(A) = [n\ m]$ and size$(B) = [m\ k]$ then you can multiply A by B and get a $n \times k$ matrix, to give C via

$$> C = A * B.$$

If x and y are vectors then MATLAB has various facilities for setting them up and manipulating them.

1. We often want to set up a vector of the form

   $$x = [a, a + d, a + 2d, \ldots, b]$$

   This is useful for plotting points on a graph. MATLAB sets up x through the command:

   $$> x = [a\ :\ d\ :\ b];$$

2. We might want to plot a function of x eg. $\sin x$. To do this we firstly create a new vector y through the command

   $$> y = \sin(x);$$

   now we *plot* the output through the command

   $$> \text{plot}(x, y)$$

   The *plot* command gives you a basic graph. However, MATLAB plotting is very flexible ... you can change axes, label things, change the colour and the line type. For a detailed description of the use of these commands see Higham & Higham, Chapter 8. If you have 3 vectors x, y, z ; you can plot all three via the command
   $>$ plot3 (x, y, z).
   Later on we will look at *animating* these plots.

3. Given row vectors

   $$\begin{array}{rcl} \mathbf{y} & = & (y_1, \ldots y_n) \\ \mathbf{x} & = & (x_1, \ldots x_n) \end{array}$$

   We can formulate their scalar or dot product $y.x$ via the command $> x{*}y'$. (This is a matrix-vector multiplication of the row vector x with the column vector y'.)

   Alternatively we may be interested in the vector $z = (x_1 y_1 \ldots, x_n y_n)$ This is given by the command

   $$> z = x. * y$$

   Note the dot . makes this an operation on the components of the vectors. Similarly the vector $(x_1^2, \ldots, x_n^2)$ is obtained by typing $x.^2$ and the vector $\left(\frac{1}{x_1}, \ldots, \frac{1}{x_n}\right)$ is obtained by typing $1./x$.

4. You can access individual elements of the vector

   $$\mathbf{y} = (y_1, \ldots, y_k, \ldots, y_n)$$

   For example if you want to find y_k you type $y(k)$. Similarly, if you type $y(1 : 4)$ then you get y_1, y_2, y_3, y_4.

2.4 Programming in MATLAB

MATLAB has a powerful programming language which allows you to combine instructions in a flexible way. We will be expecting you to use this language when using MATLAB to solve the various case studies in the assignments. Full details of the language are given in the MATLAB manual and in Higham and Higham, and you should read these.

2.4.1 Loops

These allow you to repeat the use of commands. A simple loop takes the form :

```
for i = j : m : k
statements
end
```

which increases i from j to k in steps of m. You can also nest loops. An important Matrix in numerical analysis is the Hilbert matrix $A_{ij} = 1/(i+j)$. In MATLAB you could set this up by using a double loop in the manner:

```
for i = 1 : n
      for j = 1 : n
      A(i, j) = 1/(i + j);
      end
end
```

IMPORTANT NOTE. The vector and Matrix operations in MATLAB mean that many operations that would use loops in languages such as FORTRAN can be achieved by single instructions in MATLAB. This is both quicker and clearer than using a loop.

2.4.2 Making Decisions

Usually in any code we have to make decisions and then act on these decisions. This is generally achieved in one of two ways. The while statement takes the form

```
while (condition)
statements
end
```

This tests the condition and if it is true it repeats the statements until it is false.
The `if-then-else` statement takes the form

```
if (condition)
statements1
else
statements2
end
```

In this case it executes statements1 if the condition is true and statements2 if it is false.

> AWFUL WARNING TO LONG ESTABLISHED FORTRAN USERS: MATLAB has NO goto statement (You never really had to use it!)

2.4.3 Script and function files

We will be making extensive use of script and function files, which are both lists of instructions stored as `name.m`. A script file is just a list of instructions, run by typing `name`. Generally you would use a script file to do a complete task. In contrast a function file accomplishes a sub-task in the code which you may want to use repeatedly. Typically a function file has a set of inputs and a set of outputs. For example we used a function file to set up the derivatives of the differential equation describing the pendulum. The variables in a function file are local i.e. they do not change what is going on in the program which calls the function file. Very often we also want to specify variables as being global so that they take the same value in both the main code and the function file.

2.4.4 Input and output

You can input a variable directly.
x `= input (''Type in the variable x'')`
or (generally better) from a file. For example the file `data` may include a series of measurements x_i of a process at times t_i stored as a series of lines of the form t_i x_i (so that t and x are two columns of data in this file). To access this information type

```
load data
t = data (:,1);
x = data (:,2);
```

The vectors x and t are now in your MATLAB workspace, and you can work directly with them.

You can save variables x, t created in a MATLAB session by typing:
`save filename`x t

This stores x and t in the file `filename.mat`. To use these variables in future calculations type
`load filename`

MATLAB has a wide variety of formats for outputting data, from simple lists of numbers, through to plots or even movies stored as postscript, pdf, jpeg etc files. Full details are given in Higham and Higham

2.4.5 Structured Programming

You will be writing some programs in this course and more in the course on Scientific Computing. Programs are much easier to follow, and are much more likely to work, if they are written in a structured way. Some useful hints for doing this are as follows.

1. Break up the program into well defined tasks (which may be coded in function files) and test the code for each task separately. For example, it is useful to have a file which just handles input and another which just handles output.

2. Make extensive (but careful) use of comments.% In particular at the start of each script or function file you should state clearly what the file does and what variables it uses.

3. Give your variables names which clearly indicate what they represent.

4. Make use of blank lines and indenting to break up your code.

5. Don't use a loop if there is a MATLAB command which will do the job for you.

In the Scientific Computing course you will learn a lot more about structured programming in the context of large scale code development.

2.4.6 Help

The most useful command in MATLAB is help. This gives extensive help on all MATLAB commands. You will use this command regularly!

2.4.7 Recording a MATLAB session

I will ask you to make records of what you have done for the purposes of assessment. The best way to do this in a MATLAB session is to type:

```
diary filename
work session
diary off
```

This will record what you have done in the file given by `filename`. To print this in a MATLAB session type: `!lpr -P7 filename`

This will print the file into printer 7 in the MSc room. The ! command tells MATLAB that you are using a UNIX command. Similarly, if you have created a figure and you want to save it type:

```
print -dps filename.ps
```

This will save the figure to a file called filename as a postscript file. To print this then type:

```
!lpr -P7 filename.ps
```

Chapter 3

Operations on, and the analysis of, data.

3.1 Introduction

Numerical methods are concerned with the manipulation of both functions and of data stored within a computer. Usually the functions are complicated and the data set is large. We need to be able to handle both efficiently. Two closely related tasks are as follows:

1. How can we approximate a function ? For example should we take point values or should we represent the function in terms of simpler functions ?
2. The extraction of information from data. To give three examples of this we might:
   (a) Have a series of experimental data points and we want to fit a curve through them.
   (b) Want to synthesise the sound of an oboe. How do you determine what notes make up this sound?
   (c) Have an experiment which depends on parameters a and b. Can you work out a and b from the experimental data?

A closely related problem is that of *data compression* namely you have a complicated function or a long set of data. How can you convey the *same* (or at most slightly reduced set) of information using many fewer data points? This is important in computer graphics and in transmitting images over the Internet (jpeg files, for example, use data compression to transmit images). Also closely related is the question of reducing the *noise level* of a signal and/or removing the effects of distortion (such as blurring). MATLAB allows us to perform these operations relatively easily. Most data derived in applications has errors associated with it, and this should be considered when performing operations on it.

3.2 Curve fitting using regression and splines

Suppose that we have a data set and we wish to find a curve which models this set. How can we do this? The following criteria are appropriate in finding such a curve.

1. The curve should be as "simple" as possible. Later on we may wish to perform calculations on it such as integration and differentiation
2. The curve should be simple to compute
3. It should be robust to errors in the data
4. It should "look good to the eye!"

Two methods can be used to find such a curve:

(A) Interpolation. This is a good method for clean data and is used in computer graphics.

(B) Regression. ... This is better for noisy data and is used to interpret experiments.

3.2.1 Interpolation

Suppose that we have a underlying function $u(t)$, the values $u_1, \ldots, u_N$ of which are given exactly at the data points $t_1, \ldots, t_N$. Now suppose that we have a larger set of sample points $s_1, \ldots, s_M$ with $M \gg N$. The question is, knowing u at points t_i, how can we approximate it at the points s_i? Here we see the effects of data compression in that we need only transmit the N values $u(t_i)$ to recover the many more values of $u(s_i)$.
The simplest method is to approximate $u(t)$ by a piecewise linear function, where we draw straight lines between the data points. For example if

$$t = [1, 2, 3, 4, 5, 6] \quad \text{and} \quad u = [1, 2, -1, 0, -2, 4]$$

then the resulting interpolant has the following form:

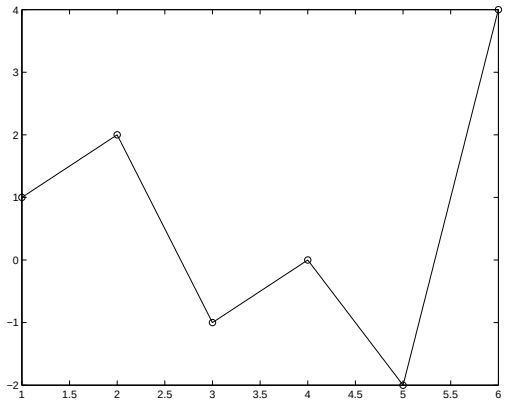


Figure 3.1: Linear interpolant

The resulting curve $p(t)$ is called the *linear interpolant*. It has the following properties:

1. $p(t)$ is continuous
2. $p(t_i) = u(t_i)$, so the curve exactly fits the given data
3. $p(t)$ is linear on each interval $[t_i, t_{i+1}]$

A nice way to implement piecewise linear interpolation in MATLAB is with the command `interp1`, selecting the option `'linear'`.

If $u(t)$ is not very smooth then the linear interpolant may be as good a representation of the data as you could reasonably expect. However, if $u(t)$ is a smooth function (i.e. many of its derivatives are continuous) then a better approximation of $u(t)$ might be possible with higher order polynomial interpolants. An example is the "cubic spline".

A cubic spline $p(t)$ interpolant to the data $(t_1, u(t_1)), (t_2, u(t_2)), \ldots, (t_N, u(t_N))$ is defined by the following criteria:

1. p is a cubic function on each interval $[t_i, t_{i+1}],\ i = 1, \ldots, N-1$.

2. p, p' and p'' are continuous on $[t_1, t_N]$.

3. $p(t_i) = u(t_i)$ i.e. p interpolates u at the points $t_i,\ \ i = 1, \ldots, N$.

Let us see whether this prescription can define p uniquely.

Since p is cubic in each $[t_i, t_{i+1}]\ , i = 1, \ldots, N-1$, that means that altogether p has $4(N-1)$ degrees of freedom which have to be determined.

The condition 2 imply that the values of p, p' and p'' have to agree from each side at the "break points" $t_i, i = 2, \ldots, N-1$, so this is $3(N-2)$ conditions. Condition 3 specifies the values of p at N points, so this is N conditions, so altogether we have $4N-6$ conditions, which is not enough. The two final conditions are what is called the "Natural" conditions at the end points:

4. $p''(t_1) = p''(t_N) = 0$.

For the above data set this looks like:

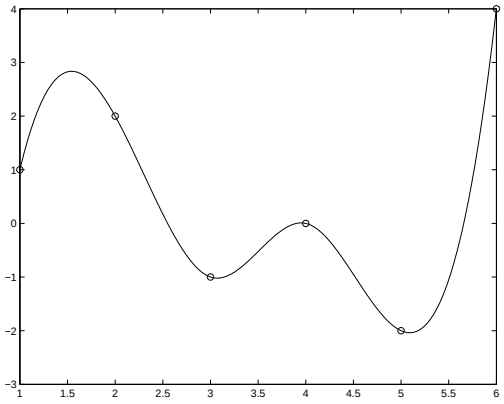


Figure 3.2: Cubic spline

Simple Example, $N = 3$**:** Then we have points t_1, t_2, t_3 and on (t_1, t_2)

$$p(t) = a_1 + b_1 t + c_1 t^2 + d_1 t^3,$$

with a similar representation on the interval $[t_2, t_3]$ with coefficients a_2, b_2, c_2, d_2. Then a_i, b_i, c_i, d_i must satisfy the following conditions.

(a) They must interpolate the data so that:

$$\begin{aligned} a_1 + b_1 t_1 + c_1 t_1^2 + d_1 t_1^3 &= u(t_1) \\ a_1 + b_1 t_2 + c_1 t_2^2 + d_1 t_2^3 &= u(t_2) \\ a_2 + b_2 t_2 + c_2 t_2^2 + d_2 t_2^3 &= u(t_2) \\ a_2 + b_2 t_3 + c_2, t_3^2 + d_2 t_3^3 &= u(t_3) \end{aligned}$$

(b) The continuity of $p'(t)$ and p'' requires that,

$$\begin{aligned} b_1 + 2c_1 t_2 + 3d_1 t_2^2 &= b_2 + 2c_2 t_2 + 3d_2 t_2^2 \\ 2c_1 + 6d_1 t_2 &= 2c_2 + 6d_2 t_2 \end{aligned}$$

This gives six equations for the eight unknowns $a_1, \ldots d_1, a_2 \ldots d_2$ and the two further conditions are given by the 'natural" condition 4.

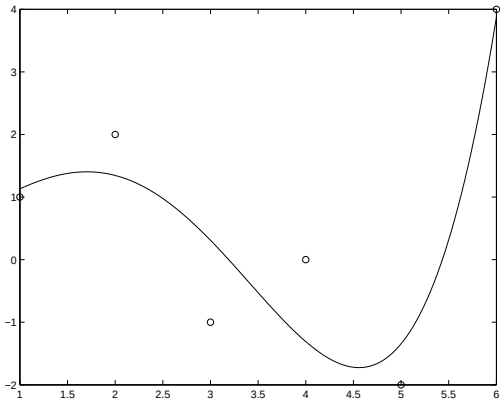
An advantage of using cubic splines is not only do they represent the function well, they are well-conditioned i.e. small changes in the data lead to small changes in the splines.
In MATLAB you can calculate cubic spline interpolation via the command: `spline` or else using `interp1` with option `spline`.

Cubic splines and their generalisations, such as NURBS, are *widely* used in the CAD industry.

```
>> help interp1

 INTERP1 1-D interpolation (table lookup).
    YI = INTERP1(X,Y,XI) interpolates to find YI, the values of
    the underlying function Y at the points in the vector XI.
    The vector X specifies the points at which the data Y is
    given. If Y is a matrix, then the interpolation is performed
    for each column of Y and YI will be length(XI)-by-size(Y,2).

    YI = INTERP1(Y,XI) assumes X = 1:N, where N is the length(Y)
    for vector Y or SIZE(Y,1) for matrix Y.

    Interpolation is the same operation as "table lookup".  Described in
    "table lookup" terms, the "table" is [X,Y] and INTERP1 "looks-up"
    the elements of XI in X, and, based upon their location, returns
    values YI interpolated within the elements of Y.

    YI = INTERP1(X,Y,XI,'method') specifies alternate methods.
    The default is linear interpolation.  Available methods are:

      'nearest'  - nearest neighbor interpolation
      'linear'   - linear interpolation
      'spline'   - piecewise cubic spline interpolation (SPLINE)
      'pchip'    - piecewise cubic Hermite interpolation (PCHIP)
      'cubic'    - same as 'pchip'
      'v5cubic'  - the cubic interpolation from MATLAB 5, which does not
                   extrapolate and uses 'spline' if X is not equally spaced.

    YI = INTERP1(X,Y,XI,'method','extrap') uses the specified method for
    extrapolation for any elements of XI outside the interval spanned by X.
    Alternatively, YI = INTERP1(X,Y,XI,'method',EXTRAPVAL) replaces
    these values with EXTRAPVAL.  NaN and 0 are often used for EXTRAPVAL.
    The default extrapolation behavior with four input arguments is 'extrap'
    for 'spline' and 'pchip' and EXTRAPVAL = NaN for the other methods.

    For example, generate a coarse sine curve and interpolate over a
    finer abscissa:
        x = 0:10; y = sin(x); xi = 0:.25:10;
        yi = interp1(x,y,xi); plot(x,y,'o',xi,yi)

    See also INTERP1Q, INTERPFT, SPLINE, INTERP2, INTERP3, INTERPN.
```

3.2.2 Regression

In regression, we approximate $u(t)$ by another (usually smooth) function. e.g a polynomial $p(t)$ and find the polynomial, which is "closest" to the data. For the data set above, the best fitting fourth order polynomial takes the form:



In general we take the polynomial

$$p(t) = p_{m+1} + p_m t + \ldots + p_1 t^m$$

and define its "distance" from the data by S where

$$S^2 = \sum_{i=1}^{n} |p(t_i) - u(t_i)|^2$$

We now find the set of values $\{p_1, \ldots, p_{m+1}\}$ which *minimises* S. Note that unlike the interpolant the values of $p(t)$ need *not* pass through the data points. This can be an *advantage* when the data is *noisy* (has errors) and we want to reduce the effects of the noise. To find such a polynomial $p(t)$ and to then evaluate it at the set of points $s_1, \ldots, s_M$. MATLAB uses the two commands

$$\left[\begin{array}{ll} pp & = \text{polyfit (t,u,m)} \\ p & = \text{polyval (pp,s)} \end{array} \right.$$

Here pp is the set of coefficients $p_1, \ldots, p_{m+1}$ and p is the polynomial evaluated at the data points.

3.2.3 Accuracy

In both the cases of interpolation and of regression the accuracy of the approximation *increases* as the number N of data points t_i *increases*. With regression, accuracy can be *improved* if the order m of the polynomial is increased. But this can lead to problems if m is too large as then p can be very oscillatory especially if the underlying function $u(t)$ has high derivatives. Often in any form of interpolation or regression we must make a compromise between the amount of work we must do to find the approximating function (measured by m) and the smoothness of the resulting approximation. This is a key area in the subject of approximation theory and is the subject of very active research as new approximations such as radial basis functions and wavelets are introduced.

3.3 The Discrete Fourier Transform

3.3.1 Overview

A very important tool in data analysis is the discrete version of the well-known Fourier transform, which is one of the key tools of applied mathematics. The discrete fourier transform allows a signal (represented by a vector in a finite dimensional space) to be represented as a superposition of discrete waves of different frequencies. By this mechanism the signal can be processed (for example its noisy components can be removced).

Most important to the success of this is a fast implementation of the discrete Fourier transform (called the FFT) which was published by Cooley and Tukey in 1965, and which has complexity $O(n \log n)$ operations to process a vector in $\mathbb{R}^n$ (whereas naive implementation takes $O(n^2)$ operations). This algorithm is also used a lot when studying both ordinary and partial differential equations using spectral methods. The FFT used to be almost the main tool for such analysis and it forms the backbone of modern digital electronics (including digital T.V.). It is now slowly being replaced by the more sophisticated wavelet transform.
The FFT allows us to express a data set as a linear combination of samples of periodic functions of different frequencies. Its many uses include:

1. noise reduction;
2. image compression;
3. convolution and deconvolution;
4. spectral analysis.
5. Wave form synthesis.
6. Prediction (for example calculating the tides).

The FFT is implemented in MATLAB with the commanf `fft`.

3.3.2 Definition

To motivate the definition of the discrete Fourier transform, recall the Fourier series for a $1-$ periodic function $u(t)$ (written in exponential form), which is discussed in many undergraduate courses:

$$u(t) = \sum_{n=-\infty}^{\infty} f(n) \exp(2\pi i n t) \ , \quad t \in [0,1] \ ,$$

where $f(n)$ is the **Fourier transform** of the function u:

$$f(n) = \int_0^1 u(t) \exp(-2\pi i n t) dt \ .$$

To find the discrete version of this, imagine that u is sampled only at N discrete points $u_k = u((k-1)/N)$, where $k = 1, \dots N$ and approximate f by the trapezoidal rule (remembering u is 1-periodic) to get

$$f(n) \approx \frac{1}{N} \sum_{k=1}^{N} u_k \exp(-2\pi i n (k-1)/N). \tag{3.1}$$

However since u is only a discrete vector we do not need to compute f for infinitely many n, in fact it is sufficient to sample (3.1) at integers $j-1$, for $j = 1, \dots, N$.

The Discrete Fourier transform (DFT) does exactly this (but typically leaves out the scaling factor $1/N$), hence defining:

$$f_j = \sum_{k=1}^{N} u_k \exp(-2\pi i (j-1)(k-1)/N) \quad , \quad j = 1, \dots, N. \tag{3.2}$$

In general this transform maps complex vectors $\mathbf{u}$ to complex vectors $\mathbf{f}$.

There is also an *inverse* DFT given by

$$u_k = \frac{1}{N} \sum_{j=1}^{N} f_j \exp(2\pi i (j-1)(k-1)/N) \ . \tag{3.3}$$

Exercise: Show that applying the DFT to **u**, followed by its inverse does indeed get **u** back again.

This expression is closely related to the Fourier decomposition of a function $u(t)$ for which u_k are sample data points. The MATLAB commands which implements these two operations using the FFT are simply

```
f= fft(u)
```

and

```
u= ifft(f)
```

The method performs fastest when N is a power of 2.

The commands `fft` and `ifft` are just fast ways of implementing the discrete Fourier transform and its inverse. The computational complexity of each transform grows with $\mathcal{O}(N \log N)$ as N increases, whereas a naive implementation usign matrix-vector multiplication would have complexity $\mathcal{O}(N^2)$. If you type `doc fft` in MATLAB you get the web help which has references to the important papers in this topic, which explain how it works.

3.3.3 An Interpretation of the DFT

The discrete values u_k can be interpreted as N *samples* of a function $u(t)$, e.g. $u_k = u((k-1)\Delta t)$, $k = 1, \ldots, N$, where Δt is the **sample time**. Then we can rewrite (3.3) as

$$u(t) = \frac{1}{N} \sum_{j=1}^{N} f_j \exp(i\omega_j t) \ , \quad \text{for} \ \ t = (k-1)\Delta t \ , \quad \text{and} k = 1, \ldots, N \ , \tag{3.4}$$

where

$$\omega_j = \frac{2\pi}{N\Delta t}(j-1). \tag{3.5}$$

That is, in (3.4), $u(t)$ is decomposed as a superposition of "waves" $\exp(i\omega_j t)$, each of frequency ω_j, and having coefficient f_j/N (and amplitude $|f_j|/N$). The "signal" $u(t)$ has been "analysed" into a sum of simple waves.

If u is *real*, then the vector **f** of is "symmetric" in the sense that

$$f_j = \bar{f}_{N+2-j} \ , \quad \text{and so} \quad |f_j| = |f_{N+2-j}| \ ,$$

where $\bar{x}$ denotes the complex conjugate of x. (Verify this as an exercise.)

We note that the frequency, of the component corresponding to the index at $N + 2 - j$ is given by

$$\begin{aligned} \omega_{N+2-j} &= \frac{2\pi}{N\triangle t}(N+1-j) \\ &= \frac{2\pi}{N\triangle t} \cdot N + \frac{2\pi}{N\triangle t}(1-j) \\ &= \frac{2\pi}{\triangle t} - \omega_j \end{aligned}$$

Thus the components with frequency $-\omega_j$ and ω_{N+2-j} are forced to have the same amplitude!

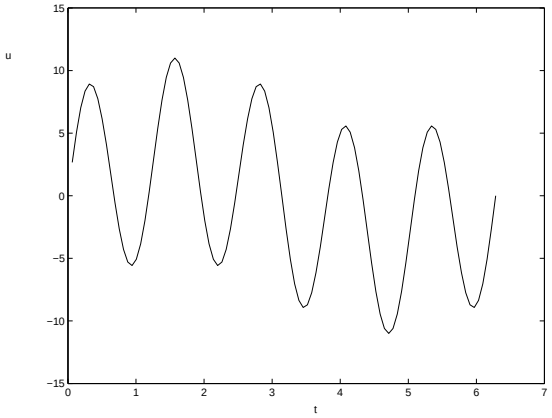
Since not all signals have this property, information has been lost by the sampling. The effect of sampling the signal has caused a loss of information. This is called *aliasing*. (It is the effect that we see when waggon wheels seem to go backwards in Westerns.) Aliasing is reduced by taking Δt small.

3.3.4 An example

Suppose, that $u(t)$ is a periodic function of the form:

$$u(t) = 3\sin(t) + 8\sin(5t)$$

illustrated below: This has a period of 2π. We can set $\Delta t = 2\pi/100$ and produce $N = 100$ samples u_k of u in MATLAB as follows:




```
t =  [1:1:100]*2*(pi/100);
u= 3*sin(t) + 8*sin(5*t);
```

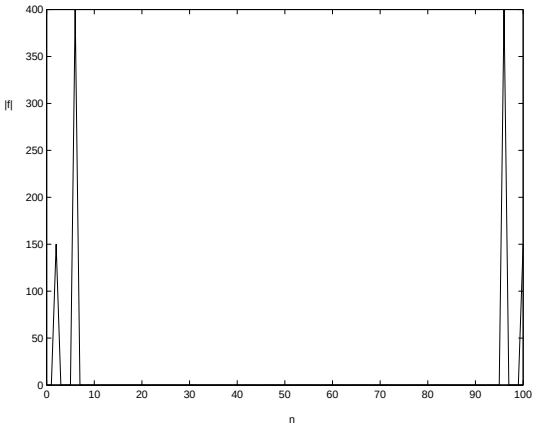
and calculate the FFT via,

```
f= fft(u);
```

The resulting vector of amplitudes is found by typing:

```
plot(abs(f))
```

which produces the picture:



The vector $|f_j|$ has four peaks at $j = 2, 6, 96, 100$ with corresponding amplitudes of $150, 400, 400$ and 150.

Note for example that $j = 6$ precisely corresponds to the term $8\sin(5t)$ in the decomposition of u. The reason for having four peaks is as follows. The function $\sin(5t)$ can be written as

$$\sin(5t) = \frac{1}{2i}(e^{i5t} - e^{-i5t})$$

thus it is composed of *two* signals, one of frequency 5 and the other of frequency -5. However, due to the effects of aliasing the signal of frequency -5 is "wrapped around" to a signal of frequency $\frac{2\pi}{\Delta t} - 5 = 95$, with a corresponding peak (as observed) at $j = 96$. The amplitude of this peak is given by $8 \times N \times \frac{1}{2} = 400$.

3.3.5 Denoising a signal

The DFT can be used to reduce the noise in a signal and/or to compress a signal. In general the contribution of noise to a signal appears as terms in the DFT centred on the index $N/2$. (This is because the least smooth components of u produce peaks near $j = N/2$.) The effects of noise can be

reduced by suppressing these compoents. To do this we multiply the DFT of a signal u by a symmetric vector which sets the central section of the DFT to zero. This vector is given by:

$$\begin{aligned} g_j &= 1 \quad j \leq M \\ g_j &= 1 \quad j \geq N + 2 - M \\ g_j &= 0 \quad \text{otherwise} \end{aligned}$$

We multiply f by g to give h, through the MATLAB command

```
h=g.*f
```

(note the use of `.*` for pointwise vector multiplication.)
The process of denoising a signal is then as follows:

(i) Choose M small enough to remove the noise but large enough to retain the details of u;

(ii) From u, calculate f, g and h as above;

(iii) Now transfer back to obtain a (hopefully smoother) approximation to u via the command

```
v=real (ifft(h))
```

The vector v is now a noise free version of u. This works (see Assignment One), but nowadays the "wavelet transform" is often used instead. Of course the choice of M has to be made by the user.

Chapter 4

Finding the roots of a nonlinear function and optimisation.

4.1 Introduction

An important task in numerical analysis is that of finding the root x of a simple function $f : \mathbb{R} \to \mathbb{R}$, so that $f(x) = 0$. An example of this is finding the solution of $f \equiv x^2 - 3x + 2 = 0$.

The multi-dimensional analogue is to find the roots $\mathbf{x}$ of a vector valued function

$$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n \quad \text{so that} \quad \mathbf{f}(\mathbf{x}) = 0,$$

where n could be large.

For example if $n = 2$ we might consider solving the problems:

$$\mathbf{f} : \begin{cases} x_1 - x_2^2 + 1 & = 0, \\ 3x_1 x_2 + x_2^3 & = 0. \end{cases}$$

It is important to realise that for many real industrial problems (e.g. discretisations of partial differential equations), n may be *large*. For example $n \sim 10000$ or even higher is not uncommon.

Closely related to both of these problems is the question of minimising a function $g(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$. Such a problem can take one of two forms:

1. *Unconstrained* optimisation: minimise $g(\mathbf{x})$.
2. *Constrained* optimisation : minimise $g(\mathbf{x})$ with an additional condition $\mathbf{f}(\mathbf{x}) = 0$ or possibly $f(\mathbf{x}) \geq 0$.

An example of a constrained minimisation problem might be to minimise the cost of producing a product in a factory subject to keeping the pollution caused in this production as low as possible.

Throughout we let either x^* or $\mathbf{x}^*$ denote the exact root of the nonlinear system which we are trying to compute.

4.2 Finding the zero of a single function of one variable

This is a well understood problem and can be solved relatively easily. Functions may have *several* roots and to find a root any algorithm requires an *initial guess* which guides the solution procedure. Finding such a guess is usually difficult and requires some knowledge of the form of solution. MATLAB also has a facility for specifying an interval which may contain the root.
Any method for solving a problem of the form $f(x) = 0$ or indeed the vector problem $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ should have three criteria:

1. It should be *fast* and find a root to a specified tolerance;
2. It should be *easy* to use ... preferably using only information on f, not on its derivatives.
3. It should be reliable i.e it should find a root close to an initial guess and not go off to infinity or become chaotic.

There is no ideal method and MATLAB uses a combination of methods to find the root.

1. **The method of bisection** The idea behind this method is to

   - Find an interval $[x_1, x_2]$ over which f changes sign and calculate $x_3 = (x_1 + x_2)/2$
   - Then f must change sign over one of the two intervals $[x_1, x_3]$ or $[x_3, x_2]$.
   - Replace $[x_1, x_2]$ by the bisected interval over which f changes sign.
   - Repeat until the interval is smaller than some specified tolerance.

   This is a reliable method as the interval over which the solution is known reduces in size by a factor of at least two at each iteration. However it is very slow, and its convergence is linear. This means that if x_i is an estimate for the root x of f and $e_i = x^* - x_i$ is the error (where x^* is the exact root), then if e_i is small

   $$|e_{i+1}| \approx K|e_i|$$

   where K is some constant with $0 < K < 1$. (For the bisection method $K = 1/2$.)

2. **The Secant Method** The idea behind this method is

   - Take two evaluations of f at the two points x_1, x_2;
   - Draw a straight line through the two points

     $$(x_1, f(x_1)); (x_2, f(x_2))$$

   - Find where this line has a zero x_3, so that $x_3 = \frac{x_1 f(x_2) - x_2 f(x_1)}{f(x_2) - f(x_1)}$.
   - Discard x_1 and repeat for the points x_2, x_3.
   - Continue to produce a set of approximations x_i to the root. Stop when either

     $$|f(x_i)| < \text{TOL} \quad \text{or when} \quad |x_{i+1} - x_i| < \text{TOL}$$

     where TOL is some specified tolerance.

   This method is fast and easy to use. No derivatives of f are needed. It is slightly unreliable but usually works. If $e_n = |x^* - x_n|$ is again the error then if $|e_n|$ is small there is a constant K such that

   $$|e_{n+1}| \approx K|e_n|^{1.61\ldots}.$$

   This is much faster than the linear convergence of the method of bisection and is called superlinear convergence.

3. **Newton's Method (sometimes called Newton-Raphson)**
   The idea behind this method is

   - Evaluate f and f' at x_1;
   - Approximate f by a line of slope f' through the point $(x_1, f(x))$;
   - Find the point x_2 where this line crosses zero so that

     $$x_2 = x_1 - f(x_1)/f'(x_1)$$

- Replace x_1 by x_2 and continue to generate a series of iterations x_i. Stop when

$$|f(x_i)| < \text{TOL} \quad \text{or when} \quad |x_{i+1} - x_i| < \text{TOL}.$$

This method is very fast and generalises to higher dimensions *BUT* it needs derivatives which may be hard to compute. Indeed we may simply *not* have derivatives of f, for example, if f is an experimental measurement. Newton's method often requires x_1 to be close to the root x^* to behave reliably. In this case $|e_{n+1}| \approx K|e_n|^2$ and the convergence is quadratic.

MATLAB combines these methods into the single instruction,

```
x = fzero('fun', xguess)
```

which finds the zero of the function `fun` close to the value `xguess`. In particular it uses the method of bisection to get close to a root and then the secant method (and a slightly more accurate variant called Inverse Quadratic Interpolation) to refine it. In the argument list `fun` is the function for which a zero is required and `xguess` is either a single guess of the solution or an interval which contains the solution.

For example if the problem is to find the zero of the function $f(x) = \sin(x) - x/2$ then the M-file (which I have called `fun1.m`) takes the form:

```
function output = fun1(x)
output = sin(x) - x/2;
```

Usually when finding the root x^* of a function we want to do this to find an approximation x_n so that either $|f(x_n)|$ or $|x^* - x_n|$ are smaller than a user specified tolerance. By default MATLAB sets the tolerance to be 10^{-6} but you may want to make this tolerance smaller or larger. It is possible to customise this function by changing the options using the *optimset* command e.g

$x =$ `fzero('fun', xguess, optimset('TolX', 1e-10))`

or

$x =$ `fzero(@fun, xguess, optimset('TolX', 1e-10))`

finds the zero to a tolerance of 1×10^{-10}. The optimset command is used in many other MATLAB routines.

Here is an example of a MATLAB program to make use of `fzero` (electronic version available from your lecturer's filespace):

```
% program nltest.m
% script to test nonlinear equation solver fzero
% The nonlinear function should be stored in the
% function file fun1.m

format long

xguess = input('type your initial guess:  ')
tol = input('type the tolerance for the approximate solution:  ')
options = optimset('Display','iter','TolX',tol);

fzero('fun1',xguess,options)
```

The function `fun1` has to be provided as a separate m-file. The choice of options allows a printout of the full convergence history of the method.

In the lectures we compared the behaviour of `nltest` with Newton's method, in particular we found that `nltest` is robust to poor choice of starting guess while Newton's method can behave unpredictably if the guess is a long way from the solution. Here's a programme to implement Newton's method:

```
% program newton.m
% implements Newton's method to find a  zero of  a function
% The function should be implemented in the function file fun1.m
% Its derivative should be in the function file dfun1.m


format long
x = input('type your initial guess:  ')
tol = input('type the tolerance for the approximate solution:  ')

icount = 0;       % counts Newton iterates
residual = fun1(x);   % Initial residual, used to check if
                              % convergence occurred

while(abs(residual) > tol)
    x = x - fun1(x)/dfun1(x);
    residual = fun1(x);
    icount = icount + 1;
    disp(['Newton iterate=', int2str(icount),'      solution=',num2str(x)])

end % while

disp('   ')
disp('****** Final Result**********')
disp('  ')

disp(['Newton iterates=', int2str(icount)])
fprintf('solution=   %16.10f', x)
```

4.2.1 Secant as an Approximate Newton Method

The general iteration step for the secant method is

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

A small amount of algebra shows the right hand side can be rearranged to give

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n) \ , \tag{4.1}$$

which is a simple approximation to the iteration

$$x_{n+1} = x_n - (f'(x_n))^{-1} f(x_n) \ , \tag{4.2}$$

which is well-known as Newton's method.

The secant method (4.1) is therefore an approximate version of Newton's method which makes use of evaluations of the function f and does not require evaluations of the derivative f'.

The disadvantage of the secant method is that it converges more slowly than Newton (although both methods are faster than linear).

To understand the convergence rate of Newton's method, subtract each side of (4.2) from the exact solution x^* to get

$$(x^* - x_{n+1}) = (x^* - x_n) + f'(x_n)^{-1}(f(x_n) - f(x^*))$$

(using $f(x^*) = 0$). Hence

$$(x^* - x_{n+1}) = f'(x_n)^{-1}\left[f(x_n) + f'(x_n)(x^* - x_n) - f(x^*)\right] \ .$$

By Taylor's theorem, the term in the square brackets is approximately $-\frac{1}{2}f''(x_n)(x^* - x_n)^2$, which shows that

$$|e_{n+1}| \ \approx \ \frac{1}{2}|f'(x_n)^{-1}f''(x_n)| \ |e_n|^2 \ , \quad \text{where} \quad e_n = x^* - x_n \ .$$

Obviously some mathematical analysis is needed to make this statement precise, but roughly speaking it shows that provided f, f' and f'' are continuous near x^*, $f'(x^*) \neq 0$ and x_0 is near enough to x^* then Newton's method converges quadratically, i.e.

$$|e_{n+1}| \leq K|e_n|^2 \quad , \quad \text{as} \quad n \to \infty.$$

4.3 Higher dimensional nonlinear systems

Methods for solving nonlinear systems of equations can be derived as generalisations of the scalar case.

Suppose we have to solve the system

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \ , \tag{4.3}$$

where $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ is a given vector-valued function of n variables $x_1, \ldots x_n$.

An example (for $n = 2$) is:

$$\begin{aligned} x_1^2 + x_2^2 - 1 &= 0 \\ x_1 - x_2 &= 0 \ , \end{aligned}$$

The solutions are where the given line meets the unit circle and are easily seen to be

$$\pm\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right) \ .$$

To write down Newton's method for (4.3), we write down the 'obvious" generalisation of the scalar case (4.2), i.e.

$$\mathbf{x}_{n+1} = \mathbf{x}_n - (J(\mathbf{x}_n))^{-1}\mathbf{f}(\mathbf{x}_n) \ . \tag{4.4}$$

where the role of the **derivative** of $\mathbf{f}$ is played by the Jacobian martrix:

$$J(\mathbf{x}) = \left(\frac{\partial f_i}{\partial x_j}(\mathbf{x})\right)_{i,j=1,\ldots,n} \ .$$

More realistically this should be written as

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{d}_n \ ,$$

where the Newton correction $\mathbf{d}_n$ is computed by solving the system of n linear equations:

$$(J(\mathbf{x}_n))\mathbf{d}_n = -\mathbf{f}(\mathbf{x}_n) \ .$$

Each step of Newton's method requires the solution of an n dimensional linear system and the matrix $J(\mathbf{x}_n)$ and right hand side $\mathbf{f}(\mathbf{x}_n)$ have to be recomputed at every step.

Note that the inverse of the Jacobian is not normally computed, it is not needed, all that is needed is the solution of a single linear system with coefficient matrix $J(\mathbf{x}_n)$, which can be done without actually computing the inverse of $J(\mathbf{x}_n)$.

More about this in the next chapter of the notes. The MATLAB command \ (type `help slash`) can be used to solve the linear systems arising in Newton's method.

An example of Newton's method for a simple nonlinear system in 2D illustrating the use of \ is available through your lecturer's filespace.

4.4 Unconstrained Minimisation

Here we are given a function $g : \mathbb{R}^n \to \mathbb{R}$ (i.e. a real valued function of several variables) and we have to seek an $\mathbf{x}^*$ such that

$$g(\mathbf{x}^*) = \min_{\mathbf{x}\in\mathbb{R}^n} g(\mathbf{x}) \ . \tag{4.5}$$

The minimisation is over all $\mathbf{x}$ in $\mathbb{R}^n$ without any constraints.

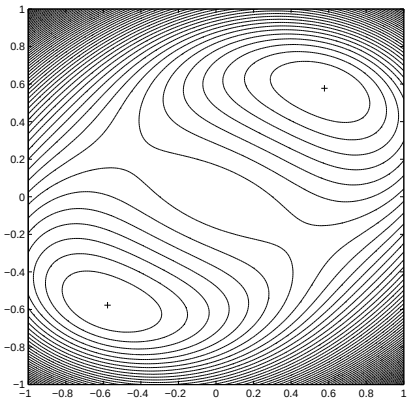
Most effective software only find local minima, i.e. they find solutions $\mathbf{x}^*$ to the equations

$$\nabla g(\mathbf{x}) = \begin{bmatrix} \partial g(\mathbf{x})/\partial x_1 \\ \partial g(\mathbf{x})/\partial x_2 \\ \cdot \\ \cdot \\ \cdot \\ \partial g(\mathbf{x})/\partial x_n \end{bmatrix} = 0 \tag{4.6}$$

(∇ denotes the "gradient"), while at the same time ensuring that $g(\mathbf{x}) \geq g(\mathbf{x}^*)$ for $\mathbf{x}$ "near" $\mathbf{x}^*$. Many good algorithms exist to do this.

The problem of finding a **global minimum** x^* so that $g(x) \geq g(x^*)$ for **all** x is much harder and no good generally available algorithms exist to do it.

A typical function $g(x)$ is given below. We represent this by a contour plot in which two local minima, at $(-0.6, -0.6)$ and $(0.6, 0.6)$ can be seen.



4.4.1 The Method of Steepest Descent

The simplest method for finding a local minimum is that of *steepest decent*. This method starts from the realisation that, at a point $\mathbf{x}_0 \in I\!R^n$, g decreases most rapidly in the direction $-\nabla g(\mathbf{x}_0)$. To see why this is, let us look for a unit direction $\hat{\mathbf{d}}$ such that

$$\frac{d}{dt}\left\{g(\mathbf{x}_0 + t\hat{\mathbf{d}})\right\}|_{t=0} \quad \text{is minimised} \ .$$

This implies (via the chain rule) that

$$(\nabla g)(\mathbf{x}_0 + t\hat{\mathbf{d}}).\hat{\mathbf{d}}|_{t=0} \quad \text{is minimised} \ ,$$

which in turn implies that $(\nabla g)(\mathbf{x}_0).\hat{\mathbf{d}}$ shoud be as negative as possible. Since (by the Cauchy-Schwarz inequality),

$$\nabla g(\mathbf{x}_0).\hat{\mathbf{d}} \ \leq \ \|\nabla g(\mathbf{x}_0)\|_2 \ \|\hat{\mathbf{d}}\|_2 \ = \ \|\nabla g(\mathbf{x}_0)\|_2 \ , \tag{4.7}$$

(where $\|\mathbf{y}\|_2 = \{\sum_i |y_i|^2\}$, the direction which gives the steepest descent is

$$\hat{\mathbf{d}} = \frac{-(\nabla g)(\mathbf{x}_0)}{\|(\nabla g)(\mathbf{x}_0)\|_2} \ ,$$

and the quantity on LHS of (4.7) is $-\|\nabla g(\mathbf{x}_0)\|_2$.

In the method of steepest descent, a series of steps is chosen, with each step taken in the direction $-\nabla g$. The iteration proceeds as follows

1. Start with a point $\mathbf{x}_0$,
2. Construct the vector $\mathbf{y} = \mathbf{x}_0 - t\nabla g(\mathbf{x}_0)$, for some $t > 0$,
3. Find the value of t which minimises $g(\mathbf{y})$
4. Set this value of $\mathbf{y}$ to be $\mathbf{x}_1$ and repeat from 2 until $g(\mathbf{y})$ cannot be reduced further.

Step 3 is a one dimensional minimisation problem. It involves minimising a function of a *single* variable t. This is conceptually an easy thing to do ... you just go downhill in one direction until you can go no further. There are many methods of doing this including the method of bisection and the (faster) golden search method.

MATLAB contains the command `fminbnd` which does a fast one-dimensional search

Advantages of steepest descent : Easy to use.
Disadvantages : The algorithm needs to calculate the derivative ∇g explicitly. Also it can be very slow. The reason that it is so slow is that the sequence of search directions are always orthogonal to each other, so that the algorithm is often having to make repeated searches in very different directions. If the contours of the solution are long and thin this can lead to a very large number of calculations.

To see why the search directions are orthogonal, note that $\mathbf{x}_{n+1} = \mathbf{x}_n - t_n\nabla g(\mathbf{x}_n)$ where t_n has the property that

$$\frac{d}{dt}\left\{g(\mathbf{x}_n - t\nabla g(\mathbf{x}_n))\right\}|_{t=t_n} = 0 \ .$$

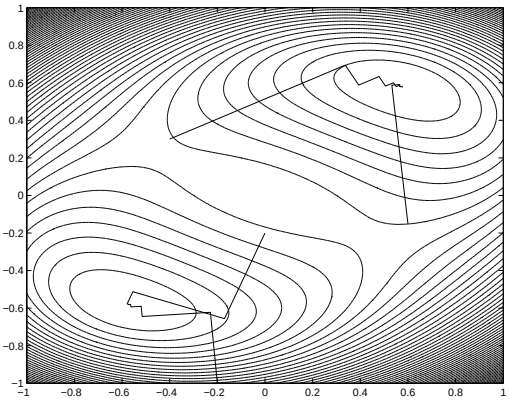
Hence (by the chain rule)

$$-\nabla g(\mathbf{x}_n - t_n\nabla g(\mathbf{x}_n)).\nabla g(\mathbf{x}_n) = 0 \ ,$$

and so $\nabla g(\mathbf{x}_{n+1}).\nabla g(\mathbf{x}_n) = 0$.

This implies a zig-zag approach to the minimum

We can see this in an application of the method of steepest descent to the previous example, in which four different calculations from different starting points are depicted below.



4.4.2 Variants of Newton's Method

Writing $\mathbf{f}(\mathbf{x}) = \nabla g(\mathbf{x})$ we see that (4.6) is a special case of (4.3). So we can consider applying Newton's method (or perhaps cheaper more reliable variants of it) to (4.6).

The Jacobian of $\nabla g(\mathbf{x})$ is the Hessian matrix:

$$\nabla^2 g(\mathbf{x}) = \left(\left(\frac{\partial^2 g_i}{\partial x_j \partial x_j} \right)(\mathbf{x}) \right)_{i,j=1,\ldots,n} \ ,$$

and Newton's method for (4.6) is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{d}_n \ , \quad \text{where } \left[(\nabla^2 g)(\mathbf{x}_n)\right] \mathbf{d}_n = -\nabla g(\mathbf{x}_n) \ .$$

More generally (and practically relevant) many methods approximate this by iterates of the form

$$\mathbf{x}_{n+1} = \mathbf{x}_n + t_n H_n^{-1} \mathbf{d}_n \ , \tag{4.8}$$

where t_n is some step length, $\mathbf{d}_n \approx -\nabla g(\mathbf{x}_n)$ is the approximate search direction and H_n^{-1} is some approximation to the true inverse Hessian at $\mathbf{x}_n$ such that its product with the current $\nabla g(\mathbf{x}_n)$

Note that the steepest descent method is one example of this general form (4.8), where t_n is the result of a line search, $\mathbf{d}_n = -\nabla g(\mathbf{x}_n)$, and H_n is the identity.

The DFP algorthm (Davidon Fletcher Powell) algorithm is another example of such a method. The DFP algorithm constructs a sequence of search directions and updates the matrix H_n^{-1} through a sequence of low rank updates. It is closely related to Broyden's method which is an approximate Newton method for solving the original problem (4.3). The DFP method can also be thought of as a higher dimensional version of the secant method.

Advantages

1. A fast algorithm with near quadratic convergence.
2. No derivative information is required, the method constructs an approximation to both $\nabla g(\mathbf{x}_n)$ and the Hessian $\nabla^2 g(\mathbf{x}_n)$.

Disadvantages Almost none, can be vulnerable to rounding error and its convergence is not "quite" quadratic. It also only finds a local minimum.

The DFP algorithm and its variants are very widely used.

MATLAB provides a function `fminsearch` for unconstrained minimisation which requires a function g and an initial guess x_0. The algorithm `fminsearch` then finds the local minimum of g which is close to $\mathbf{x_0}$. It uses a different "simplex" method for which there is very little theory (type `doc fminsearch` to see a reference to the literature).

4.4.3 Applications of minimisation methods

1. The minimisation of large systems. An interesting example of this arises in elliptic partial differential equations (for example problems in elasticity or electrostatics), where the solution minimises a function related to the energy of the system. Complicated engineering structures are designed by finding the local minima of a possible configuration as this represents a stable operating structure.
2. *Solving* **linear** systems of the form,
$$A\mathbf{x} = \mathbf{b}$$
where A is a symmetric positive definite matrix. An approach to do this is by *minimising* the function
$$g(x) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}.$$
This is the basis of the celebrated *conjugate gradient* algorithm and its derivatives for nonsymmetric matrices.
3. *Solving* **nonlinear** systems of the form
$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \ . \tag{4.9}$$

We can try to find a solution $\mathbf{x}^*$ by *minimising* the scalar valued function

$$g(\mathbf{x}) = \|\mathbf{f}(\mathbf{x})\|_2^2 = \sum_i |f_i(\mathbf{x})|^2$$

or more generally

$$g(\mathbf{x}) = \sum_i \alpha_i |f_i(\mathbf{x})|^2$$

where $\alpha_i > 0$ are suitably chosen weights. Beware, however, that to solve the system (4.9) requires finding a global minimum of g and unconstrained minimisation algorithms will only find a local minimum. Hopefully if the initial guess for the root is good enough, then the 'local' minimum of g near to this root will also be a 'global' minimum. You should always check to see if the 'minimum' located has $g \equiv 0$.

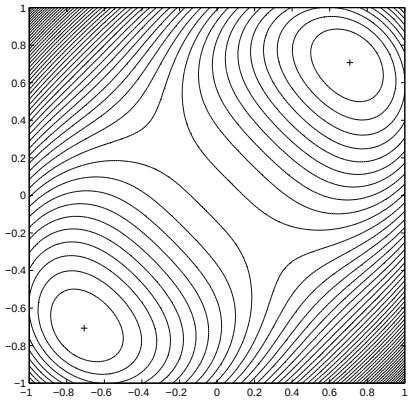
For example, if $f_1 = x - y$ and $f_2 = x^2 + y^2 - 1$ then the function $f = (f_1, f_2)$ has roots at the two points

$$(x, y) = \pm\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right).$$

we can set

$$g(x, y) = (x - y)^2 + (x^2 + y^2 - 1)^2$$

which has the contour plot below with minima at the roots and a saddle point at the origin. An



application of `fminsearch` to this function (described by the function file `gmin` gives the following result for three different initial guesses.

```
>> !more gmin.m
function f=gmin(xx)

x = xx(1);
y = xx(2);

f = (x-y)^2 + (x^2 + y^2 - 1)^2;

>> [x,r]=fminsearch('gmin',[1 2])

x =

    0.7071    0.7071


r =
```

```
   5.4074e-09

>> [x,r]=fminsearch('gmin',[1 -2])

x =

    0.7071    0.7071


r =

   1.7369e-09

>> [x,r]=fminsearch('gmin',[-1 -2])

x =

   -0.7071   -0.7071


r =

   5.4074e-09

>> diary off
```

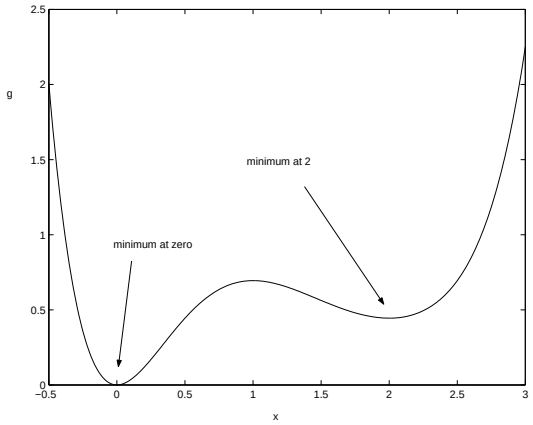
A problem with using this method for finding the root of $f(x)$ is that the associated function $g(x)$ may have several local minima. For example the function

$$f(x) = \frac{x^3}{3} - \frac{3}{2}x^2 + 2x$$

has a single root at $x = 0$, but the function

$$g(x) = f^2(x)$$

has local minima at $x = 0$ and $x = 2$ of which the second does not correspond to a root of f.



We can apply the routine `fminbnd` to find the minima of this one-dimensional function. The results of this are:

```
>> %
>> %  Find the minimum and the residual of the function g
>> %  using fminbnd. This requires as input an interval in
>> %  which the minimum occurs
>> %
>> [z,r]=fminbnd('(x^3/3-(3/2)*x^2+2*x)^2',-1,1)

z =

  -4.8334e-06


r =

   9.3447e-11

>> %
>> % Minimum is correct
>> %
>> [z,r]=fminbnd('(x^3/3-(3/2)*x^2+2*x)^2',1,3)

z =


    2.0000


r =

    0.4444

>> %
>> %  Another minimum but with a non-zero residual
>> %
>>
>> [z,r]=fminbnd('(x^3/3-(3/2)*x^2+2*x)^2',-1,10)

z =

    2.0000


r =

    0.4444

>> %
>> % The interval contains both minima .. and 2 is chosen
>> %
>>
>> diary off
```

Note that both minima are detected given different starts, however only one of these is the global minimum corresponding to zero. We can check this by looking at the value of g at the two minima.In

higher dimensions the problem is far worse, and `fminsearch` can easily be fooled into finding a "false" local minimum. More sophisticated routines for solving large nonlinear systems such as the Fortran code SNSQE make careful checks to avoid finding a false solution. There are many other excellent and sophisticated packages around for both unconstrained and constrained optimisation. A good example of such is the LANCELOT code developed by Professor N.Gould.

4.5 Global Minimisation

Finding the GLOBAL minimum of a general function $g(x)$ is a very hard task. Only recently have effective algorithms for this been developed. These algorithms include *simulated annealing* and *genetic algorithms*. MATLAB does not implement these, however they are now used a great deal in the new bioinformatic industries for tasks such as protein design, and by the power generating industry to schedule the on-off times of its power stations.

Chapter 5

Linear Algebra.

5.1 Introduction

A key feature of MATLAB is the ease with which it handles linear algebra problems. Linear algebra lies at the heart of nearly all large scale numerical computations. It is essential to do it efficiently and accurately.

1. Solving linear systems of the form $A\mathbf{x} = \mathbf{b}$ $\quad \mathbf{x} \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^n, A \in \mathbb{R}^{n\times n}$ and
2. Solving eigenvalue problems of the form $A\mathbf{x} = \lambda\mathbf{x}$ where $\lambda \in \mathbb{C}, \mathbf{x} \in \mathbb{C}^n$.

   We will also need to make sense of *overdetermined* problems of the form.
3. "Solve" $A\mathbf{x} = \mathbf{b}$ $\quad \mathbf{x} \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^m, A \in \mathbb{R}^{m\times n}$ where $m > n$.

The problem 3. often arises in fitting models to a large data sets where $\mathbf{x}$ represents parameters in the model and $\mathbf{b}$ the results of some experiments.

Linear algebra also roughly divides into "full" problems where most of the entries of A are non-zero. Usually the size of such matrices is moderate (e.g. $n \leq 10^4$). The alternative case arises when most of the entries of A are zero and n is large ($n \sim 10^6$ is not uncommon). These are called *sparse* problems. MATLAB can deal with these but we won't include them in this course. Details of sparse matrices and methods for dealing with them are given in the course on scientific computation. Sparse problems typically arise in discretisations of partial differential equations, using finite element or finite difference methods.

5.2 Vectors and Matrices

To make sense of linear algebra computations we need to be able to estimate the *size* of vectors and matrices. Let

$$\mathbf{x} = (x_1, \ldots, x_n)^T$$

The *norm* of $\mathbf{x}$ is a measure of how large it is. There are many such measures. Three important norms are given by

$$\| \mathbf{x} \|_1 = \sum_{i=1}^{n} |x_i|, \qquad \| \mathbf{x} \|_2 = \sqrt{x_1^2 + \cdots x_n^2} \quad \text{and} \quad \| \mathbf{x} \|_\infty = \max |x_i| \tag{5.1}$$

In MATLAB these norms can be calculated by typing
`norm(x,p)` where `p=1,2` or `inf` .

The norm of a matrix A is defined by:

$$\| A \|_p = \sup_{\mathbf{x}\in\mathbb{R}^n\setminus\{\mathbf{0}\}} \frac{\| A\mathbf{x} \|_p}{\| \mathbf{x} \|_p} \ . \tag{5.2}$$

It can be shown that this supremum exists and it follows from the definition that:

$$\| A\mathbf{x} \|_p \leq \| A \|_p \| \mathbf{x} \|_p \quad \text{for all} \quad \mathbf{x} \in \mathbb{R}^n.$$

To compute the norm of a matrix in MATLAB, you also use the command `norm`.

An explicit formula for the norm (5.2) is not obvious since it is defined as a supremum. However, fortunately there is a simple formula in the cases $p = 1$ and ∞. The formulae are:

$$\| A \|_1 = \max_j \sum_i |A_{ij}|$$

and

$$\| A \|_\infty = \max_i \sum_j |A_{ij}| \ .$$

(See, e.g. K.E. Atkinson "An Introduction to Numerical Analysis")

In the case $p = 2$ there is a very elegant formula for the norm. This is because, for any $\hat{\mathbf{x}} \in I\!R^n$ with $\|\hat{\mathbf{x}}\|_2 = 1$, we have

$$\|A\hat{\mathbf{x}}\|_2^2 \ = \ (A\hat{\mathbf{x}})^T(A\hat{\mathbf{x}}) \ = \ \hat{\mathbf{x}}^T A^T A\hat{\mathbf{x}}^T$$

and in §5.5 we will show that the maximum value of the right-hand side is the maximum eigenvalue of $A^T A$. Hence for any vector $\mathbf{x} \neq \mathbf{0}$ we have

$$\frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} \ = \ \left\| \frac{1}{\|\mathbf{x}\|_2} A\mathbf{x} \right\|_2 = \|A\hat{\mathbf{x}}\|_2 \ , \quad \text{where} \quad \hat{\mathbf{x}} = \mathbf{x}/\|\mathbf{x}\|_2 \ .$$

Hence

$$\|A\|_2 = \sup_{\mathbf{x}\neq\mathbf{0}} \frac{\|A\mathbf{x}\|_2}{\|x\|_2} \ = \left\{ \text{max eigenvalue of } A^T A \right\}^{1/2} \ .$$

This is a neat formula but unfortunately it requires the solution of an eigenvalue problem before the norm can be written down. If A is symmetric, the formula is slightly simpler:

$$\|A\|_2 = \{\max|\lambda| : \lambda \text{ is an eigenvalue of } A\} \quad \text{when} \quad A^T = A.$$

A measure of how sensitive a matrix is to perturbations in data is provided by the *condition number* $\kappa_p(A)$ .This is defined by the expression

$$\kappa_p(A) = \| A \|_p \| A^{-1} \|_p$$

Roughly speaking $\kappa_p(A)$ is proportional to the largest eigenvalue in modulus of A divided by its smallest eigenvalue in modulus. In MATLAB the condition number is found by typing

```
cond(A,p)
```

The condition number has the following properties:

$$\begin{aligned} \kappa_p(I) &= 1 \\ \kappa_p(A) &\geq 1 \qquad \text{for all matrices } A \\ \kappa_p(O) &= 1 \qquad \text{if } O \text{ is orthogonal} \\ \kappa_p(A) &= \infty \qquad \text{if } A \text{ is singular.} \end{aligned}$$

Its unfortunately expensive to compute the condition number since in principle this involves finding the inverse of A. MATLAB provides two condition number estimates: `condest` and `rcond` .

Let's have a look at how the condition number arises in some problems in Linear Algebra. Suppose that we want to solve the linear system

$$A\mathbf{x} = \mathbf{b}.$$

In a real applications $\mathbf{b}$ may not be known exactly, and in a computer calculation there will always be rounding errors. The condition number then tells us how errors in $\mathbf{b}$ may feed into to errors in $\mathbf{x}$. In particular if we make an error $\delta\mathbf{b}$ in $\mathbf{b}$ then we have an error $\delta\mathbf{x}$ in $\mathbf{x}$, that is

$$A(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \ .$$

Since $A\mathbf{x} = \mathbf{b}$, subtracting gives

$$A\delta\mathbf{x} = \delta\mathbf{b} \ ,$$

and so

$$\|\delta\mathbf{x}\|_p \ = \ \|A^{-1}\delta\mathbf{b}\|_p \ \leq \ \|A^{-1}\|_p\|\delta\mathbf{b}\|_p \ .$$

Hence

$$\frac{\|\delta\mathbf{x}\|_p}{\|\mathbf{x}\|_p} \ \leq \ \kappa_p(A)\frac{\|\delta\mathbf{b}\|_p}{\|A\|_p\|\mathbf{x}\|_p} \ .$$

Also $\|\mathbf{b}\|_p = \|A\mathbf{x}\|_p \leq \|A\|_p\|\mathbf{x}\|_p$, so we have :

$$\frac{\| \delta\mathbf{x} \|_p}{\| \mathbf{x} \|_p} \leq \kappa_p(A)\frac{\| \delta\mathbf{b} \|_p}{\| \mathbf{b} \|_p}. \tag{5.3}$$

Similarly if an error δA is also committed in A then the result is, for sufficiently small δA,

$$\frac{\| \delta\mathbf{x} \|_p}{\| \mathbf{x} \|_p} \leq \frac{\kappa_p(A)}{1 - \frac{\|\delta A\|_p}{\|A\|_p}} \left\{ \frac{\| \delta\mathbf{b} \|_p}{\| \mathbf{b} \|_p} + \frac{\|\delta A\|_p}{\|A\|_p} \right\}. \tag{5.4}$$

In problems for which A arises as a discretisation matrix of a partial differential equation the condition number of A can be very large (and increases rapdily as the number of mesh points increases). For example we typically have (for PDEs in 2D) $\kappa_2(A) = O(N)$, where N is the number of points, and it is not uncommon nowadays to have $N \approx 10^6 - 10^8$. In this case errors in $\mathbf{b}$ may be multiplied enormously in the solution process. Thus if $\kappa_p(A)$ is large we have difficulties in solving the system reliably, a problem which plagues calculations with partial differential equations.

Unfortunately it gets worse. If A is large then we usually solve the problem $A\mathbf{x} = \mathbf{b}$ by using *iterative methods*. In these methods a sequence $\mathbf{x}_m$ of approximations to $\mathbf{x}$ is constructed and the method works by ensuring that each $\mathbf{x}_m$ is easy to calculate and that $\mathbf{x}_m$ rapidly tends to $\mathbf{x}$ as $m \to \infty$. The number of iterations that have to be made so that $\| \mathbf{x}_m - \mathbf{x} \|_p$ is less than some tolerance, increases rapidly as $\kappa_p(A)$ increases, often being proportional to $\kappa_p(A)$ or even to $\kappa_p(A)^2$. Thus not only do errors in $\mathbf{x}$ accumulate for large $\kappa_p(A)$ but the amount of computation we must do to find $\mathbf{x}$ increases as well.

5.3 Solving Linear Systems when A is a square matrix

5.3.1 Direct Methods

Suppose we have to solve a linear system:

$$A\mathbf{x} = \mathbf{b} \tag{5.5}$$

where A, an $n \times n$ real matrix, and $\mathbf{b} \in \mathbb{R}^n$ are given and $\mathbf{x} \in \mathbb{R}^n$ is to be found.

Forgetting for a moment about $\mathbf{b}$, lets revise the standard Gaussian elimination process. At the first step consider the matrix $A^{(1)} = A$, i.e.

$$A \ = A^{(1)} \ = \begin{bmatrix} A_{11}^{(1)} & . & . & . & . & . & . & . & . & A_{1n}^{(1)} \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ A_{n1}^{(1)} & . & . & . & . & . & . & . & . & A_{nn}^{(1)} \end{bmatrix} .$$

Assuming that $A_{11} \neq 0$, the first step is to create zeros in the first column below the diagonal. To achieve this, we introduce a vector M of multipliers

$$M_{i1} = A_{i1}^{(1)}/A_{11}^{(1)} \quad i = 2, \ldots, n \ .$$

Now if we multiply the first row of $A^{(1)}$ by M_{i1} and subtract the result from row i we create the zeros we want. We call this next matrix $A^{(2)}$ and the process of getting $A^{(2)}$ from $A^{(1)}$ is illustrated as follows:

$$\underbrace{\begin{bmatrix} 1 & & & & & & & \\ -M_{21} & 1 & & & & & & \\ . & & 1 & & & & & \\ . & & & 1 & & & & \\ . & & & & 1 & & & \\ . & & & & & 1 & & \\ -M_{N1} & & & & & & & 1 \end{bmatrix}}_{=:M^{(1)}} \begin{bmatrix} A_{11}^{(1)} & . & . & . & . & . & . & . & . & A_{1N}^{(1)} \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ A_{N1}^{(1)} & . & . & . & . & . & . & . & . & A_{NN}^{(1)} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}^{(2)} & . & . & . & . & . & . & . & . & A_{1N}^{(2)} \\ 0 & A_{22}^{(2)} & . & . & . & . & . & . & . & A_{2N}^{(2)} \\ 0 & . & . & . & . & . & . & . & . & . \\ 0 & . & . & . & . & . & . & . & . & . \\ 0 & . & . & . & . & . & . & . & . & . \\ 0 & . & . & . & . & . & . & . & . & . \\ 0 & A_{N2}^{(2)} & . & . & . & . & . & . & . & A_{NN}^{(2)} \end{bmatrix} = A^{(2)} \ .$$

Another way to write this (using MATLAB notation) is that the first row of $A^{(2)}$ is just same as the first row of $A^{(1)}$, the first column of $A^{(2)}$ below the diagonal is zero (and doesn't have to be stored) and the remaining bottom right $n-1 \times n-1$ block is obtained by

$$A^{(2)}(2:n, 2:n) = A(2:n, 2:n) - M(2:n) * A(1, 2:n) \ .$$

This formula shows that $A^{(2)}$ is obtained from $A^{(1)}$ by an update which involves multiplication of a column vector $M(2:n)$ by the row vector $A(1, 2:n)$. You'll see much more on this in the Scientific computing course.

The number of uperations in this updating process is $(n-1)^2$ multiplications and $(n-1)^2$ subtractions. This is repeated for rows $2, 3, \ldots, n-1$. It is easily possible to count the operations involved and the result is that this process takes

$$\text{approximately} \quad \frac{2}{3}n^3 \quad \textit{operations}$$

for large n, when A is an $n \times n$ full matrix.

Continuing the method above, and assuming the diagonal entries of A (the 'pivots") do not vanish, we end up with a sequence of lower triangular matrices $M^{(1)}, \ldots, M^{(n-1)}$ (determined by the multipliers) such that

$$M^{(n-1)} M^{(n-2)} \ldots M^{(1)} A = U \ ,$$

where U is upper triangular. Each $M^{(i)}$ has an inverse which is lower traingular and the products of these inverses is also lower triangular and thus we have

$$A = LU \qquad \text{the LU decomposition}$$

where L is a lower triangular matrix and U is an upper triangular matrix. In fact U is the end result of the Gaussian elimination, while L has $1s$ on the diagional and the multipliers below the diagonal.

The $O(n^3)$ growth is a very fast growth of complexity. For example if $n = 10^6$ then even on a fast computer with a 1ns time for a floating point operation, $\frac{2}{3}n^3$ operations take 21 years. A lot of modern numerical linear algebra is based on trying to reduce it. For example if a matrix is banded (i.e. only a bounded number of diagonals contain non-zero entries), then the complexity of Gaussian elimination is $0(n^2)$.

In MATLAB the LU decomposition is computed by the command:

```
[L,U,P] = lu(A)
```

This in fact gives L, U and a permutation matrix P such that

$$PA = LU \ .$$

(The permutation matrix P appears because row swops may be needed in practice to avoid zero pivots.) computed to avoid conditioning problems).

Thus if

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

we have

$$LU\mathbf{x} = P\mathbf{b}$$

and we solve this system with two steps:

$$(i)\ L\mathbf{y} = P\mathbf{b}, \quad \text{and} \quad (ii)U\mathbf{x} = \mathbf{y} \ .$$

Because of the triangular form of the matrices L and U, these two solves (i), (ii) can be computed in a total of n^2 operations (by forward and back substitution), which is much cheaper than the actual LU decomposition itself.

- The MATLAB command

  ```
  x= A\b
  ```

  will solve the equation $A\mathbf{x} = \mathbf{b}$, by performing an LU decomposition of A and then the forward and back substitutions. This is one of MATLAB's most important commands and the single instruction hides a very great deal of careful programming.

- If you repeatedly solve the problems of the form $A\mathbf{x}_i = \mathbf{b}_i$ with many different right-hand sides $\mathbf{b}_i$, but with A fixed, then only one LU decomposition is needed and the cost is only the repeated forward and back substitutions.

5.3.2 Iterative Methods

These will be discussed in the second half of the course.

As described earlier, when solving $A\mathbf{x} = \mathbf{b}$ using an *iterative* method a sequence of approximations $\mathbf{x}_m$ to $\mathbf{x}$ are computed. The method stops when the iterates differ by a pre specified tolerance `tol` or when $\| A\mathbf{x}_m - \mathbf{b} \| < tol$. Iterative methods are used for large problems and when storage is a problem.

Typically such methods are used when A is sparse (arising, for example, in the discretisation of a PDE.). Iterative methods are most effective when A is *symmetric* so that

$$A = A^T,$$

and *positive definite*, so that

$$\mathbf{x}^T A\mathbf{x} > 0$$

for all non-zero vectors $\mathbf{x}$.

MATLAB has many inbuilt iterative methods. These include the conjugate gradient algorithms `gmres`, `cgs`, and `bicg`. For details about these commands, see the courses on numerical linear algebra and scientific computing. Usually these methods need *preconditioning* to work well in which instead of solving the problem $A\mathbf{x} = \mathbf{b}$ you solve either the problem

$$M_1^{-1}AM_2^{-1} \;\; M_2\mathbf{x} = M_1^{-1}\mathbf{b}$$

or the problem

$$M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}.$$

Here M, M_1, M_2 are the *preconditioning* matrices chosen so that the new matrices $M^{-1}A$ or $M_1^{-1}AM_2^{-1}$ are "close" to the identity matrix. In this case the condition number of the new matrix is closer to 1 and the inversion process is then faster and more accurate.
A common example of preconditioning is to take M to be the matrix formed from the diagonals of A.

$$M = \begin{bmatrix} A_{11} & & & 0 \\ & A_{22} & & \\ & & \ddots & \\ 0 & & & A_{nn} \end{bmatrix}.$$

This procedure is called "diagonal" preconditioning. Choosing a good preconditioner is not easy and depends strongly on the problem.There is always a trade-off between how much work you do in preconditioning and how much you do in then solving the system. Determining an effective preconditioner for a given class of problems (e.g. problems arising in fluid mechanics) is an important area of research in modern numerical analysis. The variety of iterative methods is also confusing. None is "best" in all cases and it may be worth trying out several on the same problem.

An example of the use of the `cgs` (conjugate gradient squared) algorithm in MATLAB with a diagonal preconditioning matrix is given by:

```
> cgs (A,b, tol, num, diag(diag(A)))
```

Here `tol` is the tolerance of the method, `num` is the maximum permitted number of iterations and `diag(diag(A))` is the preconditioning matrix.
Another example of an iterative method is the Jacobi method. In this we set

$$A = D + C$$

where D is the matrix of the diagonals of A.
Then

$$A\mathbf{x} = \mathbf{b} \Rightarrow D\mathbf{x} + C\mathbf{x} = \mathbf{b} \quad \text{so that} \quad \mathbf{x} = D^{-1}(\mathbf{b} - C\mathbf{x})$$

We then generate the sequence

$$\mathbf{x}_{n+1} = D^{-1}(\mathbf{b} - C\mathbf{x}_n)$$

starting with $\mathbf{x}_0 = 0$.

The Jacobi iteration is cheap (as D^{-1} is easy to calculate) and converges provided $\| D^{-1}C \|_p < 1$ for some p.
A MATLAB function file to implement this algorithm and to compare its time of execution with that of the cgs algorithm is given as follows:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                    %
%  Function file to solve the system Ax = b          %
%  using the Jacobi algorithm, to time this          %
%  and compare with the cgs algorithm                %
%                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [x,t1,y,t2] = jacobi(A,b,tol)

%
%  A  : matrix
%  b  : right hand side
%  x  : solution to tolerence tol using Jacobi
% tol : tolerence
% t1  : time of Jacobi
%  y  : solution to tolerence tol using cgs
% t2  : time of cgs
%


t0 = cputime;
D  = diag(diag(A));
C  = A - D;
d  = diag(A);

res = norm(b,1);
xn   = 0.*b;

  while res > tol

  z = b - C*xn;
  xn = z./d;
  res = norm(A*xn-b,1);

  end

x  = xn;
t1 = cputime - t0;

%
% Compare with cgs using diagonal preconditioning
%

t0 = cputime;
y  = cgs(A,b,tol,100,diag(diag(A)));
t2 = cputime - t0;
```

The Jacobi method is especially fast on parallel computers where the terms of $\mathbf{x}_m$ can be updated simultaneously.

At present the most powerful general purpose iterative methods are the conjugate gradient methods as implemented in MATLAB. A comparison of the accuracy speed of the Jacobi and cgs algorithms on two different problems is given below with t_1 being the time of the Jacobi calculation and t_2 the time

of the cgs calculation.

```
%
% Set up a diagonally dominant matrix A
%



A=[10 2 3 4;2 13 4 5;0 0 4 1;1 2 3 12]

A =

    10     2     3     4
     2    13     4     5
     0     0     4     1
     1     2     3    12

%
% and a right hand side
%

b=[1 2 3 4]'

b =

     1
     2
     3
     4

%
% Invert to tolerance 1e-8
%


[x,t1,y,t2]=jacobi(A,b,1e-8)
cgs converged at iteration 4 to a solution with relative residual 1.1e-16

x =

  -0.16467236452955
  -0.10997150983932
   0.70256410260600
   0.18974358982986


t1 =

   0.00999999999999


y =

  -0.16467236467236
```

```
  -0.10997150997151
   0.70256410256410
   0.18974358974359


t2 =

   0.00999999999999


%
% Note that cgs and jacobi take similar time
% the diagonal dominance means that the
% the jacobi algorithm performs well in this example.
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% Set up another matrix
%

A=[5 3 2 1;1 5 3 2;4 6 10 1;1 2 3 4]

A =

     5     3     2     1
     1     5     3     2
     4     6    10     1
     1     2     3     4

%
% and a right hand side
%

b=[1 2 3 4]';

%
% Now invert
%

[x,t1,y,t2]=jacobi(A,b,1e-6)
cgs converged at iteration 4 to a solution with relative residual 2.5e-14

x =

   Inf
   Inf
   NaN
   Inf
```

```
t1 =

   0.65000000000003


y =

  -0.02020202020202
  -0.10505050505051
   0.28686868686868
   0.84242424242424


t2 =

   0.00999999999999

%
% Here we see that the Jacobi iteration has
% failed to converge .. cgs had no trouble
% Lets see why
%

D=diag(diag(A));

norm(inv(D)*(A-D),1)

ans =

   1.75000000000000

%
% The culprit is revealed - the matrix fails
% the convergence condition
%
```

For certain problems, in particular those arising in discretisations of elliptic partial differential equations, there are even more powerful *multi-grid* methods, which are the fastest possible iterative methods. MATLAB does not implement multigrid methods directly, but they are extremely important in modern scientific computing.

5.4 The QR decomposition of the matrix A

The LU decomposition of A is described above.

Another useful decomposition of an $m \times n$ matrix A is to express it in the form

$$A = QR; \quad Q^{-1} = Q^T$$

where Q is an $m \times m$ orthogonal matrix and R is an $m \times n$ upper triangular matrix.

This is precisely the process of Gram-Schmidt orthogonalisation which forms an orthogonal set of vectors (which make up Q) from the columns of A. It takes about twice as long as the LU decomposition.

If $A = QR$ and if $A\mathbf{x} = \mathbf{b}$ we have

$$R\mathbf{x} = Q^T\mathbf{b}$$

The vector $\mathbf{x}$ can then be found quickly by inverting R.

It is important that this decomposition does not only work for square matrices but can be applied to a rectangular matrix $A \in I\!R^{m\times n}$. The QR decomposition is especially useful when finding the solution to over determined problems. Suppose that we want to "solve"

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

where $A \in I\!R^{m\times n}, x \in I\!R^n, b \in I\!R^m$ and $m > n$. In general, this problem does not have a well defined solution so that $A\mathbf{x}$ is exactly equal to $\mathbf{b}$. Instead we find the vector $\mathbf{x}$ so that the quantity

$$\| \mathbf{r} \|_2 = \| A\mathbf{x} - \mathbf{b} \|_2$$

is minimised over all possible choices. The vector $\mathbf{x}$ is called the "least squares fit to the data" and this is a very special optimisation problem called quadratic minimisation. Now suppose that $A = QR$. It follows that $Ax - b = QRx - b = Q(Rx - Q^T b)$. Thus

$$\| \mathbf{r} \|_2 = \| A\mathbf{x} - \mathbf{b} \|_2 = \| Q(R\mathbf{x} - Q^T\mathbf{b}) \|_2 = \| R\mathbf{x} - Q^T\mathbf{b} \|_2 \ .$$

The latter result follows from the fact that Q is an orthogonal matrix.
Now in many problems of estimating n parameters in a process with m experimental data points we have $m \ll n$. The problem of minimising $\| R\mathbf{x} - Q^T\mathbf{b} \|_2$ over all values of $\mathbf{x}$ is then a *much smaller* problem than that of minimising $\| A\mathbf{x} - \mathbf{b} \|_2$ This can be done directly. In particular, if $Q^T\mathbf{b} = \mathbf{y}$ we have

$$R\mathbf{x} - Q^T\mathbf{b} = \begin{bmatrix} R_{11} & R_{12} & R_{1n} \\ O & R_{22} & R_{2n} \\ O & \dots & R_{nn} \\ O & \dots & O \\ O & \dots & O \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} - \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_m \end{bmatrix} \equiv \mathbf{a} + \mathbf{c}$$

where $\mathbf{a}$ and $\mathbf{c}$ are othogonal vectors given by

$$\mathbf{a} = \begin{bmatrix} s_1 \\ \cdot \\ \cdot \\ \cdot \\ s_n \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} \quad \text{with} \quad \mathbf{s} = \begin{bmatrix} R_{11} & \dots & R_{1n} \\ & \ddots & \\ 0 & & R_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

and

$$\mathbf{c} = - \begin{bmatrix} 0 \\ \cdot \\ \cdot \\ \cdot \\ y_{n+1} \\ \cdot \\ \cdot \\ \cdot \\ y_m \end{bmatrix}$$

It follows that

$$\| \mathbf{r} \|_2^2 = \| R\mathbf{x} - Q^T\mathbf{b} \|_2^2 = \| \mathbf{a} \|_2^2 + \| \mathbf{c} \|_2^2 = \| \mathbf{s} \|_2^2 + \| \mathbf{c} \|_2^2$$

Thus $\| \mathbf{r} \|_2$ is minimised over all values of $\mathbf{x}$ if $\| s \|_2 = 0$. So we have:

$$\mathbf{x} = \begin{bmatrix} R_{11} & \dots & R_{1n} \\ & \ddots & \\ 0 & & R_{nn} \end{bmatrix}^{-1} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

as the best least squares fit. The residual $\| \mathbf{r} \|_2 = \| R\mathbf{x} - \mathbf{y} \|_2$ is then given by

$$\| \mathbf{r} \|_2 = \| \mathbf{c} \|_2 = \| (y_{n+1}, \dots, y_m) \|_2,$$

so that $\| \mathbf{c} \|$ gives an estimate for how good the best fit is.

5.5 Eigenvalue calculations

Often we need to solve the eigenvalue equation

$$A\mathbf{v} = \lambda \mathbf{v} \ . \tag{5.6}$$

Here A is an $n \times n$ real matrix and λ and $\mathbf{v} \neq \mathbf{0}$ have to be found. Unfortunately, even though A is real, λ and $\mathbf{v}$ may be complex.

Example

$$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \ .$$

It is easily found that

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ 1 \end{bmatrix} = i \begin{bmatrix} i \\ 1 \end{bmatrix}$$

and

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ i \end{bmatrix} = -i \begin{bmatrix} 1 \\ i \end{bmatrix} \ .$$

So A has complex eigenvalues and eigenvectors in this case.

It can be shown (Fundamental Theorem of Algebra) that every $n \times n$ matrix has at least one eigenvalue.

In fact "usually" an $n \times n$ matrix A has n eigenvectors (say $\mathbf{v}_1, \dots \mathbf{v}_n$) and these are linearly independent and so form a basis for $\mathbb{C}^n$. Denoting the corresponding eigenvalues $\lambda_1 \ , \dots, \lambda_n$ (these may not all be distinct), set $V = [\mathbf{v}_1, \dots \mathbf{v}_n]$ and let Λ be the diagonal matrix with $\lambda_1, \dots, \lambda_n$ on the diagonal. Then, gathering the definitions of each λ_i and $\mathbf{v}_i$, we have

$$AV = V\Lambda \quad .$$

When the $\mathbf{v}_1, \dots, \mathbf{v}_n$ are linearly independent, V is nonsingular and so

$$A = V\Lambda V^{-1} \ . \tag{5.7}$$

This is the **eigenvalue decomposition** of A.

Example There are many applications of the eigenvalue decomposition. A simple one involves the analysis of the Fibonacci numbers

$$F_0 = 0; \quad F_1 = 1; \quad F_{n+1} = F_n + F_{n-1} \ , \quad n \geq 1 \ .$$

Question: What is the ratio F_{n+1}/F_n for very large n?

Solution: Writing

$$\mathbf{u}_n = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} ,$$

we have $\mathbf{u}_n = A\mathbf{u}_{n-1}$, where

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} .$$

So $\mathbf{u}_n = A^n\mathbf{u}_0$ and $\mathbf{u}_0 = [1,0]^T$.

The eigenvalues of A are

$$\lambda_1 = \frac{1+\sqrt{5}}{2} , \quad \lambda_2 = \frac{1-\sqrt{5}}{2} .$$

The corresponding eigenvectors are $\mathbf{v}_i = [\lambda_i, 1]^T$, $i = 1, 2$. So the eigenvector decomposition is

$$A = V \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} V^{-1}$$

where V contains $\mathbf{v}_1, \mathbf{v}_2$ as columns. Hence

$$\mathbf{u}_n = A^n\mathbf{u}_0 = V \begin{bmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{bmatrix} V^{-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = V \begin{bmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{bmatrix} \begin{bmatrix} 1/(\lambda_1-\lambda_2) \\ -1/(\lambda_1-\lambda_2) \end{bmatrix} =: \mathbf{c} .$$

Hence

$$\mathbf{u}_n = \begin{bmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \lambda_1^n c_1 \\ \lambda_2^n c_2 \end{bmatrix} = \lambda_1^n \begin{bmatrix} \lambda_1 \\ 1 \end{bmatrix} c_1 + \lambda_2^n \begin{bmatrix} \lambda_2 \\ 1 \end{bmatrix} c_2 .$$

Thus

$$\frac{F_{n+1}}{F_n} = \frac{\lambda_1^{n+1}c_1 + \lambda_2^{n+1}c_2}{\lambda_1^n c_1 + \lambda_2^n c_2}$$

Since $|\lambda_2/\lambda_1| < 1$ it is easily seen that

$$\frac{F_{n+1}}{F_n} \to \lambda_1 , \quad \text{as} \quad n \to \infty .$$

λ_1 **is known as the golden ratio**.

In MATLAB the command to compute eigenvalues and eigenvectors is `eig`.

The command

```
e = eig(A)
```

puts the eigenvalues of `A` in `e`. The command

```
[V,Lambda] = eig(A)
```

computes the matrix `V` of eigenvectors and the diagonal matrix `Lambda` of eigenvalues in the eigenvalue decomposition.

The general method for computing the eigenvalue decomposition (5.7) is the "QR method" which is based on the QR decomposition described in the previous section.
It takes the form:

Set $A_0 = A$
Given A_n compute the QR decomposition $A_n = QR$
then form the product $A_{n+1} = RQ$

The amazing thing is that A_n converges to an upper tringular matrix. Also

$$A_{n+1} = Q^T A_n Q = Q^{-1} A_n Q ,$$

which means that for all n, A_n has same eigenvalues as $A_0 = A$ so the diagonal entries of A_n get closer and closer to the required eigenvalues of A as $n \to \infty$.

There are various tricks for making this go fast, which are especially effective in the symmetric case.

From now on for a general complex matrix B, let B^* denote the Hermitian transpose matrix: $(B^*) = \overline{B^T}$, (i.e. we take the transpose and the complex conjugate). When B is real and symmetric we have $B^* = B^T = B$.

The eigenvalue problem is much simpler in the case when A is real symmetric. This is because all eigenvalues are real. To see why, let λ and $\mathbf{v}$ be an eigenpair, and write

$$\lambda \mathbf{v}^*\mathbf{v} = \mathbf{v}^*(A\mathbf{v}) = \mathbf{v}^*(A^*\mathbf{v}) = (A\mathbf{v})^*\mathbf{v} = (\lambda\mathbf{v})^*\mathbf{v} = \overline{\lambda}\mathbf{v}^*\mathbf{v} \ .$$

Since $\mathbf{v}^*\mathbf{v} = \|\mathbf{v}\|_2^2$, we have $\lambda = \overline{\lambda}$ and since $A\mathbf{v} = \lambda\mathbf{v}$, with A real, then $\mathbf{v}$ can be chosen real too.

Going further, the eigenvectors corresponding to distinct eigenvalues of real symmetric A are orthogonal, for suppose $\lambda, \mathbf{v}$ and $\mu, \mathbf{u}$ are eigenpairs (all real), then

$$\mu \mathbf{v}^T\mathbf{u} = \mathbf{v}^T(A\mathbf{u}) = (A\mathbf{v})^T\mathbf{u} = \lambda\mathbf{v}^T\mathbf{u} \ .$$

So if $\lambda \neq \mu$, then $\mathbf{v}^T\mathbf{u} = 0$.

A less obvious fact is that when A is symmetric, then A is always diagonalisable and in fact the eigenvectors of A can be chosen orthonormal. That means the columns of the matrix V in the eigenvalue decomposition (5.7) satisfy $\mathbf{v}_i^T\mathbf{v}_j = \delta_{i,j}$. In other words $V^TV = I$ and the eigenvalue decomposition (5.7) becomes

$$A = V\Lambda V^T. \tag{5.8}$$

This is amazingly useful, since it tells us that for eigenvectors $\mathbf{v}_i, \mathbf{v}_j$, we have two relations

$$\mathbf{v}_i^T\mathbf{v}_j = \delta_{i,j} \quad \text{and} \quad \mathbf{v}_i^TA\mathbf{v}_j \ = \delta_{i,j}\lambda_i \ .$$

Thus if $\mathbf{x}$ is any vector in $\mathbb{R}^n$ with $\|\mathbf{x}\|_2 = 1$, we can write $\mathbf{x} = \sum_j a_j\mathbf{v}_j$, fos some real numbers a_j. Hence

$$1 = \mathbf{x}^T\mathbf{x} = \sum_j a_j^2 \quad \text{and} \quad \mathbf{x}^TA\mathbf{x} = \sum_j \lambda_j a_j^2 \ .$$

It then follows easily that

$$\min_j \lambda_j \ = \ \min_{\|\mathbf{x}\|_2=1} \mathbf{x}^TA\mathbf{x} \ \leq \ \max_{\|\mathbf{x}\|_2=1} \mathbf{x}^TA\mathbf{x} \ = \ \max_j \lambda_j \ .$$

That is the eigenvalue decomposition solves certain types of constrained optimisation problem!

If A is <u>not</u> symmetric an alternative to the eigenvalue decompositon is the singular value decomposition (SVD). Here one starts from realising that A^TA is symmetric and so its eigenvalue decomposition gives:

$$A^TA = V\Lambda V^T \ .$$

Moreover the eigenvalues of A^TA are always non-negative since if $A^TA\mathbf{v} = \lambda\mathbf{v}$ then $\lambda\mathbf{v}^T\mathbf{v} = \mathbf{v}^TA^TA\mathbf{v} = \|A\mathbf{v}\|_2^2 \geq 0$.

Supposing for the moment that all the eigenvalues of A^TA are positive, then writing $D = \Lambda^{1/2}$, we have

$$D^{-1}V^TA^TAVD^{-1} = I$$

Now setting $U = AVD^{-1}$, we see that

$$U^TU = I \quad \text{ and moreover } \quad V^TAU = D \ .$$

The argument extends easily to the case where D may contain some zero diagonal entries.

This leads to the SVD: **Every matrix A has a Singular Value Decomposition**

$$A = VDU^T$$

where D is a diagonal matrix containing the **singular values** and U and V are both **orthogonal**. The singular values of A are the square roots of the eigenvalues of A^TA.

The SVD even extends to rectangular matrices.

The SVD is used a lot in optimisation and in signal-processing. See Higham and Higham and also G.Golub and C.Van Loan 'Matrix Computations'.

To compute the SVD in MATLAB:

```
[U,Lambda,V]= svd(A)
```

5.6 Functions of a matrix

MATLAB can compute various functions of matrices. One of the most important of these is the *matrix exponential* which is a map from a Lie algebra to a Lie group. It is defined by the infinite sum

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!} \tag{5.9}$$

If we express A as $A = U\Lambda U^{-1}$ then

$$e^A = U \sum \frac{\Lambda^n}{n!} U^{-1} = U e^{\Lambda} U^{-1}$$

where if

$$\Lambda = \begin{bmatrix} \lambda_1 & & \\ & \dots & \\ & & \lambda_k \end{bmatrix} \quad \text{then} \quad e^{\Lambda} = \begin{bmatrix} e^{\lambda_1} & & \\ & \dots & \\ & & e^{\lambda_n} \end{bmatrix}.$$

The matrix exponential is used to solve differential equations with *constant* coefficients. Suppose that A is a constant matrix and that $y(t)$ satisfies the matrix differential equation

$$\frac{dy}{dt} = Ay, \quad y(0) = y_0$$

Then this has the *exact* solution

$$y = e^{At} y_0$$

(which you can check by differentiating the right hand side).
If y satisfies the slightly more complex equation

$$\frac{dy}{dt} = Ay + b, \quad y(0) = y_0, \tag{5.10}$$

Then this has the particular solution $p = -A^{-1}b$. So if $y = z + p$ then

$$\frac{dz}{dt} = Az \quad \text{and} \quad z(0) = y_0 + A^{-1}b.$$

Hence the solution of (5.10) is given by

$$y = e^{At}(y_0 + A^{-1}b) - A^{-1}b$$

In MATLAB, the exponential is calculated (using an algorithm based on the Padé approximant) by the command
`>expm(A)`.
We can thus solve (5.10) 'exactly' via the command

```
>y = expm(A * t) * (yo + A\b) - A\b
```

Another important matrix map is the *Cayley transform*.

$$\text{cay}(A) = (I - A)^{-1}(I + A)$$

which is very easy to calculate for an $n \times n$ matrix via the command

$$c = \text{inv}(\text{eye}(n) - A) * (\text{eye}(n) + A).$$

The Cayley Transform is used to help cluster the eigenvalues of a matrix, or to act as a map from a skew symmetric matrix A into an orthogonal matrix C.