# Computational Modelling

by Prof Simon J. Cox (sjc@soton.ac.uk)

# 1 What is Computational Modelling?

Computational modelling and the use of associated numerical methods is about choosing the right algorithm or technique for the job in hand. In these notes we will discuss some important methods and the general tips about possible problems which might be encountered, efficiencies of different methods, and stability of techniques are applicable to other numerical techniques.

## *1.1 Practical Software Design*

Practical software design for computational modelling requires a balance between the time spent choosing the correct algorithm for a computation, performing the computation and analysing the results. Python or Matlab can be used for each of these tasks and often people use C or Fortran for larger or more complex cases.

| Algorithm | Computation | Results |
|---|---|---|
| Matlab/ Python provides a high-level and simple way to design and check algorithms | Matlab/ Python can be used to check small test cases. Consider translating/ compiling to C, C++ or Fortran for larger cases. | The results from computational simulations can be analysed and post-processed with Matlab/ Python. |

At the end of these notes there is a short appendix on Matlab for reference. For the Python examples in these notes, we use the Enthought Python build and IPython Console. We also use Visual Studio with the free Python Tools for Visual Studio plug-in.

## *1.2 Python notes*

In Python we assume that the following modules have been imported:

```
» import scipy
» import numpy
» from scipy import linalg
```

See the links at the end for more information on Python.

# 2 Linear Equations

## *2.1 Examples*

### 2.1.1 Electrical Circuits

In the analysis of electrical circuits, the mesh-current method is often used to compute the currents in a circuit

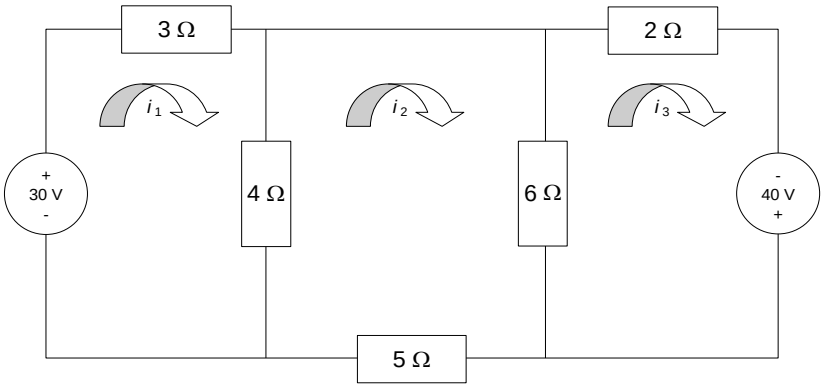



Figure 1 Circuit diagram [Etter,Matlab Intro, p512]

The voltage drops (or increases) across each element in a mesh are summed and set equal to zero to form an equation for each mesh. The resulting set of simultaneous equations is then solved for the currents within each loop. For Figure 1 the mesh equations become

$$\begin{aligned} -30+3i_1+4(i_1-i_2)&=0\\ 4(i_2-i_1)+6(i_2-i_3)+5i_2&=0\\ 6(i_3-i_2)+2i_3-40&=0 \end{aligned} \tag{2.1}$$

Combining and rearranging these equations yields a system of simultaneous equations:

$$\begin{aligned} 7i_1-4i_2+0i_3&=30\\ -4i_1+15i_2-6i_3&=0\\ 0i_1-6i_2+8i_3&=40 \end{aligned} \tag{2.2}$$

This can be written in matrix form:

$$A\,x=b \tag{2.3}$$

thus

$$\begin{pmatrix} 7 & -4 & 0\\ -4 & 15 & -6\\ 0 & -6 & 8 \end{pmatrix} \bullet \begin{pmatrix} i_1\\ i_2\\ i_3 \end{pmatrix} = \begin{pmatrix} 30\\ 0\\ 40 \end{pmatrix} \tag{2.4}$$

### 2.1.2 Others

Systems of simultaneous equations occur in the solution of differential equations, such as the heat equation which governs diffusion of heat in a material. An example of this is the flow of heat in a Pentium Processor by conduction.

## *2.2 Norms and Notation*

In this section we will solve the system of equations $Ax = b$, with $A$ a real $N \times N$ matrix, $b$ is the known right hand side and $x$ is a vector of $N$ unknowns.

The equations

$$A\,x = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix} \bullet \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix} = b \qquad (2.5)$$

only have a solution if the determinant of $A \neq 0$. This can be checked in Matlab using `det(A)`. The solution can then be written

$$x = A^{-1}b\,. \qquad (2.6)$$

However finding the inverse of $A$ is numerically unstable, expensive (in terms of solution time) and usually unnecessary. We therefore use a variety of different methods for solving systems of equations.

### 2.2.1 Norms

A norm is a single number which summarises the information in a matrix or vector. There are several frequently used norms for vectors

$$\text{1-Norm } \|x\|_1 = |x_1| + |x_2| + |x_3| + \ldots + |x_N| \qquad (2.7)$$

$$\text{2-Norm } \|x\|_2 = \sqrt{x_1^2 + x_2^2 + x_3^2 + \ldots x_N^2} \qquad (2.8)$$

$$\infty\text{-Norm } \|x\|_\infty = \text{Max}\{|x_1|, |x_2|, |x_3|, \ldots, |x_N|\}. \qquad (2.9)$$

There are analogous definitions for matrices

$$\text{1-Norm } \|A\|_1 = \underset{1\le k\le N}{\text{Max}} \sum_{i=1}^{N} |\, a_{ik} \,| = \text{Maximum Absolute Column Sum} \qquad (2.10)$$

$$\text{2-Norm } \|A\|_2 = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N} a_{ij}^2} \qquad (2.11)$$

$$\infty\text{-Norm} \|A\|_\infty = \underset{1\le k\le N}{\text{Max}} \sum_{i=1}^{N} |\, a_{ki} \,| = \text{Maximum Absolute Row Sum.} \qquad (2.12)$$

## *2.3 Ill-conditioning and Poor Scaling*

The condition number of a matrix, $\kappa(A)$ is defined as:

$$\kappa(A) = \|A\| \times \|A^{-1}\| = \texttt{cond(A)}, \qquad (2.13)$$

where any norm can be used. If this is large and you are solving the system $Ax = b$, a small change in the vector $b$ can result in a large change in the solution $x$. See section 1.9.2 of the Matlab hand-out for an example of this.

Poor scaling is a different problem and is caused by the matrix elements varying over perhaps several orders of magnitude. Numerical rounding errors cause a loss of accuracy in the final solution. Techniques such as pivotting (which we discuss later) help to reduce these errors.

## *2.4 Back and Forward Substitution*

If a set of equations is upper triangular, we can find the solution easily by 'back substitution.'

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix} \bullet \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{2.14}$$

$$\begin{aligned} 6x_3 = 3 &\Rightarrow x_3 = \tfrac{1}{2} \\ 4x_2 + 5x_3 = 2 &\Rightarrow x_2 = (2 - \tfrac{5}{2})/4 = -\tfrac{1}{8} \\ x_1 + 2x_2 + 3x_3 = 1 &\Rightarrow x_1 = 1 + \tfrac{2}{8} - \tfrac{3}{2} = -\tfrac{1}{4} \end{aligned} \tag{2.15}$$

Solution in Matlab follows using:

```
» a = [1,2,3;0,4,5;0,0,6];
» b=[1;2;3];
» a\b

ans =
   -0.2500
   -0.1250
    0.5000
```

In Python this is:

```
» a = [ [1,2,3], [0,4,5],[0,0,6]]
» b=[[1],[2],[3]]
» solve(a,b)

array([[-0.25 ],
       [-0.125],
       [ 0.5  ]])
```

If the equations are in lower triangular form:

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{pmatrix} \bullet \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{2.16}$$

then solution proceeds from the top down ('forward substitution')

## *2.5 Gaussian Elimination*

To perform Gaussian elimination, we perform row and column operations to transform the matrix into upper triangular form so that back substitution can be used. Consider

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \bullet \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 24 \\ 63 \\ 57 \end{pmatrix} \tag{2.17}$$

It is usual to perform the working using the following abbreviated notation to store $A$ and $b$.

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 24 \\ 4 & 5 & 6 & 63 \\ 7 & 8 & 0 & 57 \end{array}\right) \tag{2.18}$$

The steps of the reduction are (i) get first column into upper triangular form.

$R2 \rightarrow R2\text{-}4\times R1$
$R3 \rightarrow R3\text{-}7\times R1$

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 24 \\ 0 & 5-4\times 2 & 6-4\times 3 & 63-4\times 24 \\ 0 & 8-7\times 2 & 0-7\times 3 & 57-7\times 24 \end{array}\right) = \left(\begin{array}{ccc|c} 1 & 2 & 3 & 24 \\ 0 & -3 & -6 & -33 \\ 0 & -6 & -21 & -111 \end{array}\right) \tag{2.19}$$

(ii) Get second column into upper triangular form (and so on for a larger matrix)

$R3 \rightarrow R3\text{-}2\times R2$

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 24 \\ 0 & -3 & -6 & -33 \\ 0 & 0 & -21-2(-6) & -111-2(-33) \end{array}\right) = \left(\begin{array}{ccc|c} 1 & 2 & 3 & 24 \\ 0 & -3 & -6 & -33 \\ 0 & 0 & -9 & -45 \end{array}\right) \tag{2.20}$$

In Matlab the solution is given by

```
» a = [1,2,3;4,5,6;7,8,0];
» b=[24;63;57];
» a\b

ans =
    7.0000
    1.0000
    5.0000
```

In Python this is:

```
» a = [[1,2,3],[4,5,6],[7,8,0]]
» b=[[24],[63],[57]]
» solve(a,b)
array([[ 7.],
       [ 1.],
       [ 5.]])
```

For an $N\times N$ matrix the work required for the elimination is $O(N^3)$

## 2.5.1 Partial and Total Pivoting

In general we replace rows thus:

$$\text{Row } j \rightarrow \text{Row } j - \varepsilon \times \text{Row } i, \tag{2.21}$$

where $\varepsilon$ is as small as possible. We can ensure that $\varepsilon$ is as small as possible (and in general it will be $\leq 1$) by re-arranging the set of equations so that we are eliminating using the largest 'pivot' possible. Without pivoting Gaussian elimination is numerically unstable. Clearly if the pivot was zero, it would not be possible to proceed with the calculation. In general we pick the largest element to be the pivot, which makes $\varepsilon$ as small as possible. It is found that this reduces rounding errors – Stoer (2010) has a detailed treatment of this topic including the desirability of having an equilibriated matrix before proceeding with the calculation which further improves stability. Partial pivoting for the above example proceeds thus:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 24 \\ 4 & 5 & 6 & 63 \\ \underbrace{7}_{\text{largest}} & 8 & 0 & 57 \end{array}\right) \tag{2.22}$$

We swap rows 3 and 1 and eliminate using the pivot 7.

$\text{R1} \leftrightarrow \text{R3}$

$$\left(\begin{array}{ccc|c} 7 & 8 & 0 & 57 \\ 4 & 5 & 6 & 63 \\ 1 & 2 & 3 & 24 \end{array}\right) \tag{2.23}$$

$\text{R2} \rightarrow \text{R2-4/7}\times\text{R1}$
$\text{R3} \rightarrow \text{R3-1/7}\times\text{R1}$

$$\left(\begin{array}{ccc|c} 7 & 8 & 0 & 57 \\ 0 & 5-\tfrac{4}{7}\times 8 & 6-\tfrac{4}{7}\times 0 & 63-\tfrac{4}{7}\times 57 \\ 0 & 2-\tfrac{1}{7}\times 8 & 3-\tfrac{1}{7}\times 0 & 24-\tfrac{1}{7}\times 57 \end{array}\right) = \left(\begin{array}{ccc|c} 7 & 8 & 0 & 57 \\ 0 & \tfrac{3}{7} & 6 & 30\tfrac{3}{7} \\ 0 & \tfrac{6}{7} & 3 & 15\tfrac{6}{7} \end{array}\right) \tag{2.24}$$

The largest element in the second column is now the $\tfrac{6}{7}$, so we swap the last two rows before eliminating again.

$\text{R2} \leftrightarrow \text{R3}$

$$\left(\begin{array}{ccc|c} 7 & 8 & 0 & 57 \\ 0 & \tfrac{6}{7} & 3 & 15\tfrac{6}{7} \\ 0 & \tfrac{3}{7} & 6 & 30\tfrac{3}{7} \end{array}\right) \tag{2.25}$$

$\text{R3} \rightarrow \text{R3-1/2}\times\text{R2}$

$$\left(\begin{array}{ccc|c} 7 & 8 & 0 & 57 \\ 0 & \tfrac{6}{7} & 3 & 15\tfrac{6}{7} \\ 0 & \tfrac{3}{7}-\tfrac{1}{2}\times\tfrac{6}{7} & 6-\tfrac{1}{2}\times 3 & 30\tfrac{3}{7}-\tfrac{1}{2}\times 15\tfrac{6}{7} \end{array}\right) = \left(\begin{array}{ccc|c} 7 & 8 & 0 & 57 \\ 0 & \tfrac{6}{7} & 3 & 15\tfrac{6}{7} \\ 0 & 0 & 4\tfrac{1}{2} & 22\tfrac{1}{2} \end{array}\right) \tag{2.26}$$

Once again backsubstitution yields the solution $(7, 1, 5)^T$.

An even better result may be obtained using total pivoting, where instead of picking the largest element in the column as the pivot, we select the largest element left in the rest of the matrix:

$$\mathrm{A}_{\mathrm{Pivot}} = \mathrm{Pivot} = \underset{1\le i,j\le N}{\mathrm{Max}}\left\{\left|a_{ij}\right|\right\} \tag{2.27}$$

Then retaining the *i*th equation, eliminate the coefficient of $x_j$. At each stage we are now eliminating using the largest pivot possible (so $\varepsilon$ is always as small as possible). Total pivoting is generally harder to code than partial pivoting, since it is necessary to take account of the variable order as the elimination proceeds. If at any stage the largest pivot remaining is 0, then the matrix is singular (i.e. its determinant is zero).

## *2.6 LU decomposition*

When using Gaussian elimination the matrix, *A*, and the right hand side, *b*, are treated together. If, as may be the case in the electrical circuit example, we would like to try a number of different right hand sides, then each one is 'just like the first' and we have to repeat all of the steps for the full matrix from scratch. LU decomposition is a method of performing a splitting of the matrix once and for all, which allows new right hand sides to be employed with little additional effort. It is, in fact, identical to Gaussian elimination with the operations re-ordered.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix} =$$
$$\begin{pmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ L_{N,1} & \cdots & L_{N,N-1} & L_{NN} \end{pmatrix} \bullet \begin{pmatrix} U_{11} & U_{12} & \cdots & U_{1N} \\ 0 & U_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & U_{N-1,N} \\ 0 & \cdots & 0 & U_{NN} \end{pmatrix} = L \bullet U \tag{2.28}$$

By convention we set $L_{ii}$=1 ('Doolittle's methods') so *L* is unit lower triangular. Setting $U_{ii}$ = 1 is known as Crout's method- and *U* is unit upper triangular. How does this help to solve *Ax* = *b*? Once the factorization is performed we can write:

$$\begin{aligned} & LUx = b, \text{ let } y = Ux \\ & \Rightarrow Ly = b \end{aligned} . \tag{2.29}$$

The steps for the solution of the system are (i) find the factorization, (ii) find *y* by forward substitution, (iii) solve *Ux* = *y* by back-substitution.

Consider solve the following equations:

$$\begin{pmatrix} 2 & 4 & 6 \\ 1 & 10 & 10 \\ 1 & 10 & 7 \end{pmatrix} \bullet \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -2 \\ -7 \\ -1 \end{pmatrix} \tag{2.30}$$

(i) Perform the factorization

$$\begin{pmatrix} 1 & 0 & 0 \\ \overset{(4)}{L_{21}} & 1 & 0 \\ \overset{(5)}{L_{31}} & \overset{(8)}{L_{32}} & 1 \end{pmatrix} \bullet \begin{pmatrix} \overset{(1)}{U_{11}} & \overset{(2)}{U_{12}} & \overset{(3)}{U_{13}} \\ 0 & \overset{(6)}{U_{22}} & \overset{(7)}{U_{23}} \\ 0 & 0 & \overset{(9)}{U_{33}} \end{pmatrix} = \begin{pmatrix} 2 & 4 & 6 \\ 1 & 10 & 10 \\ 1 & 10 & 7 \end{pmatrix} \tag{2.31}$$

The order in which we determine the unknowns is marked and ensures that we only have one unknown at each stage.

$$\begin{aligned}
(1) \Rightarrow \quad & U_{11} = 2 \\
(2) \Rightarrow \quad & U_{12} = 4 \\
(3) \Rightarrow \quad & U_{13} = 6 \\
(4) \Rightarrow \quad & L_{21} \times U_{11} = 1 \\
\Rightarrow \quad & L_{21} \times 2 = 1 \Rightarrow L_{21} = \tfrac{1}{2} \\
(5) \Rightarrow \quad & L_{31} \times U_{11} = 1 \\
\Rightarrow \quad & L_{31} \times 2 = 1 \Rightarrow L_{31} = \tfrac{1}{2} \\
(6) \Rightarrow \quad & L_{21} \times U_{12} + U_{22} = 10 \\
\Rightarrow \quad & U_{22} = 10 - \tfrac{1}{2} \times 4 = 8 \\
(7) \Rightarrow \quad & L_{21} \times U_{13} + U_{23} = 10 \\
\Rightarrow \quad & U_{23} = 10 - \tfrac{1}{2} \times 6 = 7 \\
(8) \Rightarrow \quad & L_{31} \times U_{12} + L_{32} \times U_{22} = 10 \\
\Rightarrow \quad & L_{32} = (10 - \tfrac{1}{2} \times 4)/8 = 1 \\
(9) \Rightarrow \quad & L_{31} \times U_{13} + L_{32} \times U_{23} + U_{33} = 7 \\
\Rightarrow \quad & U_{33} = 7 - \tfrac{1}{2} \times 6 - 1 \times 7 = -3
\end{aligned} \tag{2.32}$$

The factorisation is

$$L \bullet U = \begin{pmatrix} 1 & 0 & 0 \\ \tfrac{1}{2} & 1 & 0 \\ \tfrac{1}{2} & 1 & 1 \end{pmatrix} \bullet \begin{pmatrix} 2 & 4 & 6 \\ 0 & 8 & 7 \\ 0 & 0 & -3 \end{pmatrix} \tag{2.33}$$

(ii) Solve $Ly = b$ to find $y$ by forward substitution

$$Ly = \begin{pmatrix} 1 & 0 & 0 \\ \tfrac{1}{2} & 1 & 0 \\ \tfrac{1}{2} & 1 & 1 \end{pmatrix} \bullet \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} -2 \\ -7 \\ -1 \end{pmatrix} = b \tag{2.34}$$

$$\begin{aligned}
& y_1 = -2 \\
& \tfrac{1}{2} y_1 + y_2 = -7 \Rightarrow y_2 = -7 - \tfrac{1}{2} \times (-2) = -6 \\
& \tfrac{1}{2} y_1 + y_2 + y_3 = -1 \Rightarrow y_3 = -1 - \tfrac{1}{2} \times (-2) - 1 \times (-6) = 6
\end{aligned} \tag{2.35}$$

(iii) Solve $Ux = y$ by back substitution

$$Ux = \begin{pmatrix} 2 & 4 & 6 \\ 0 & 8 & 7 \\ 0 & 0 & -3 \end{pmatrix} \bullet \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -2 \\ -6 \\ 6 \end{pmatrix} = y \qquad (2.36)$$

$$\begin{aligned} -3x_3 &= 6 \Rightarrow x_3 = -2 \\ 8x_2 + 7x_3 &= -6 \Rightarrow x_2 = \{-6 - 7 \times (-2)\}/8 = 1 \\ 2x_1 + 4x_2 + 6x_3 &= -2 \Rightarrow x_1 = \{-2 - 4 \times 1 - 6 \times (-2)\}/2 = 3 \end{aligned} \qquad (2.37)$$

Thus the solution is $x = (3, 1, -2)^T$.

It is possible to solve the equations as above using Matlab

```
» a = [2,4,6;1,10,10;1,10,7]
a =
     2     4     6
     1    10    10
     1    10     7

» b = [-2;-7;-1]
b =
    -2
    -7
    -1

» [l,u]=lu(a)
l =
    1.0000         0         0
    0.5000    1.0000         0
    0.5000    1.0000    1.0000
u =
     2     4     6
     0     8     7
     0     0    -3

» y=l\b
y =
    -2
    -6
     6
» x=u\y
x =
     3
     1
    -2
```

We can also use Python:

```
» from scipy import linalg
» a = [[2,4,6],[1,10,10],[1,10,7]]
» b = [[-2],[-7],[-1]]
```

```
» l,p = linalg.lu_factor(a)
» disp(l)
 [[ 2.   4.   6. ]
 [ 0.5  8.   7. ]
 [ 0.5  1.  -3. ]]
```

From which the lower and upper triangular parts can be seen (the '1's along the leading diagonal are assumed in Python). We solve using

```
» linalg.lu_solve((l,p),b)
array([[ 3.],
       [ 1.],
       [-2.]])
```

[The *p* is a pivot matrix.]

In general the LU factorisation is performed using partial pivoting to avoid numerical rounding errors. Using `[L, U, P] = lu(A)` in Matlab returns a permutation matrix, *p*, such that $PA = LU$. It is shown above in Python also. This takes account of the partial pivoting in the algorithm.

If the matrix *A* is symmetric, then we may write $A = L\,L^{T}$. i.e. the upper triangular part of the decomposition is just the transpose of the lower triangular part. In this case we perform a 'Cholesky decomposition', and we do not need to set the elements $L_{ii}$ (as we would have with the methods of Crout or Doolittle.)

## *2.7 Sparse Systems and fill-in*

If a matrix has a large number of zero entries then the matrix is said to be 'sparse.' By only storing the non-zero elements of the matrix you (i) save memory in the computer (ii) save time by avoiding null operations (such as $0 \times x = 0$). It is possible to retain the sparsity of a matrix when performing operations such as Gaussian elimination or LU decomposition. However, sometimes in the course of performing e.g. Gaussian elimination a non-sparse matrix will be produced: this is known as 'fill-in'. Consider the following matrix

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & N \\ 2 & 1 & 0 & \cdots & 0 \\ 3 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ N & 0 & \cdots & 0 & 1 \end{pmatrix}. \qquad (2.38)$$

This matrix fills in completely using Gaussian elimination. Interchange of the first and last row yields a matrix which does not fill-in. More general techniques exist for re-arranging sparse matrices to minimise fill-in.

# 3 Interpolation

[For Python information on interpolation, see Chapter 16 of Prof Fangohr's notes at:

http://www.soton.ac.uk/~sesg2006/textbook/Python-for-Computational-Science-and-Engineering.pdf]

So, you have some points from an experiment or by sampling a function by a computational experiment. What next? How do you assign a value for the function at the points you did not measure. For example, you measure the function at the points [1.6, 2.2 , 3.3, 4.8] – what is the value of the function at 3.5? If this is between the range of values you measured (i.e. 1.6 and 4.8), it is known as "interpolation"; if you are trying to use those samples to estimate the value outside that range (at, say -10.4 or 7.3), it is known as extrapolation. In this section we will concentrate on interpolation, and although many of the same principles apply for extrapolation, it can be a far more dangerous and risky procedure.

More generally we might be trying to approximate a complicated function by a simpler one. We may also wish to represent a set of discrete points by a continuous function (or set of continuous functions). This too is the realm of interpolation and curve-fitting.

Let us consider a few of the things you have to take into account. If I sample a function at [1, 2, 3, 4], then is it meaningful to ask "What is the value at 3.5?" –if the sample points represent discrete variables, like food eaten by *n* children in a family, then is meaningful to ask "What if I had 2.5 children?" ?- and the answer is "Well, it might be reasonable to ask this". Other things to consider are in the following sections, but let us now start with an example and we will consider MATLAB's built-in `humps` function, which has interesting properties.

### 3.1.1 Curve Fitting: Least Squares

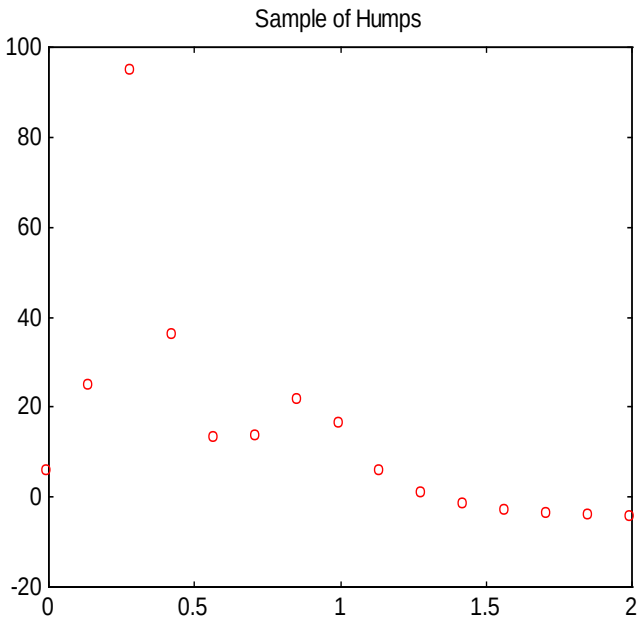
Given a discrete sample of points from some function, what kind of curve could we draw which best represents the trends in the data. Naturally there are lots of possible curves, an infinite number in fact, so which one can we choose? When we fit a curve, we do not expect that the curve will pass through each of the data points.

One way to fit the curve is to minimize the sum of the squares of the discrepancies (or "residuals") between the data and the values predicted by the hypothetical function. We adjust the parameters of the function to minimise this sum. This is known as least squares fitting of data.

MATLAB provides `polyfit(x, y, o)` to perform a least squares fit of a polynomial of order `o` to the data in `x` and `y`. The function `polyval(p, x)`, evaluates the polynomial defined by `p` at the set of points `x`.

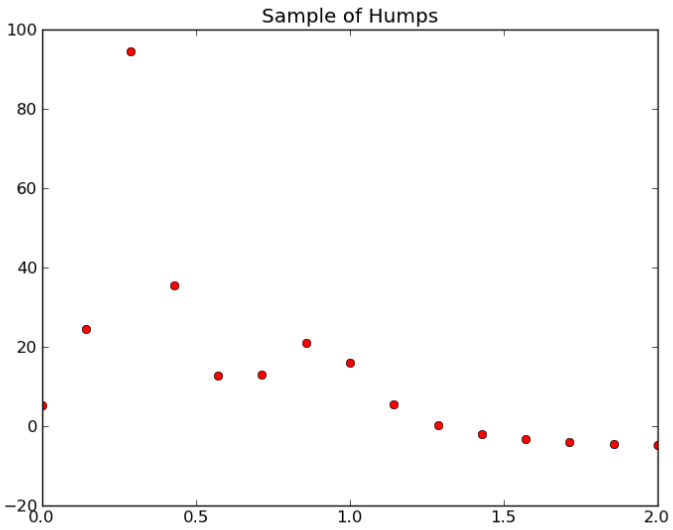
Let us take 15 equally spaced points from the function humps, and consider qualitatively the effect of plotting polynomials of various order through these points.

```
x = linspace(0,2,15);
y = humps(x);
plot(x,y,'ro')
title('Sample of Humps')
```



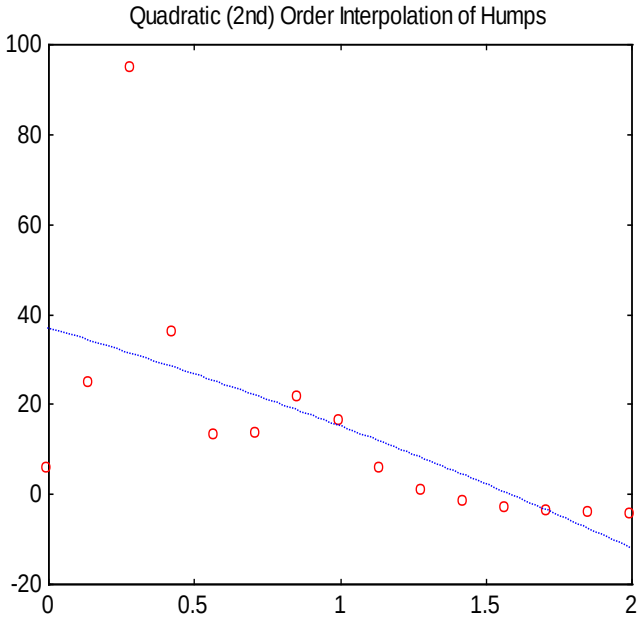

In Python this becomes

```
x = numpy.linspace(0, 2, num=15)
y = 1 / ((x-.3)**2 + .01) + 1 / ((x-.9)**2 + .04) - 6
plt.plot(x,y,'ro')
plt.title('Sample of Humps')
plt.show()
```




With just this sample of points, it is hard to know what sort of curve to fit. We could try a second order polynomial as follows

```
order_2 = polyfit(x,y,2);
x_2 = linspace(0,2,100);
y_2 = polyval(order_2, x_2);
plot(x, y, 'ro', x_2, y_2, 'b:')
title('Quadratic (2nd) Order Interpolation of Humps')
```

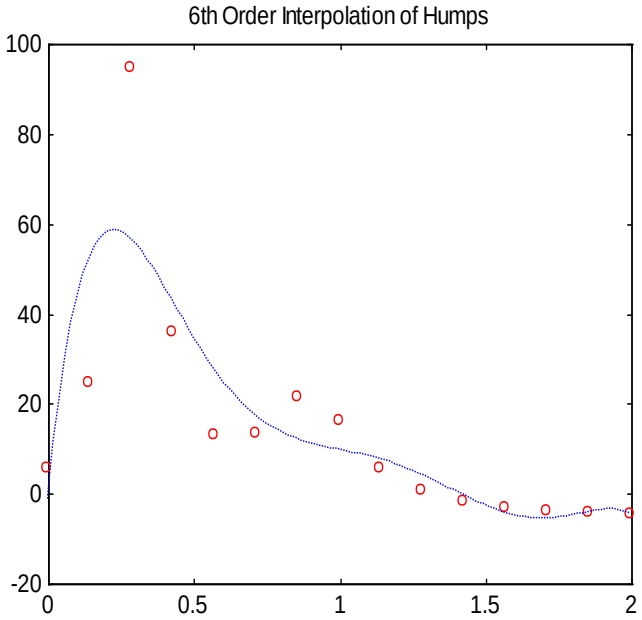



In Python this is:

```
import numpy
x = linspace(0, 2, num=15)
y = 1 / ((x-.3)**2 + .01) + 1 / ((x-.9)**2 + .04) - 6
order_2 = numpy.polyfit(x,y,2)
x_2 = linspace(0,2,num=100)
y_2 = polyval(order_2, x_2)
pylab.plot(x, y, 'ro', x_2 , y_2 , 'b:',linewidth=5)
pylab.title('Quadratic (2nd) Order Interpolation of Humps')
```

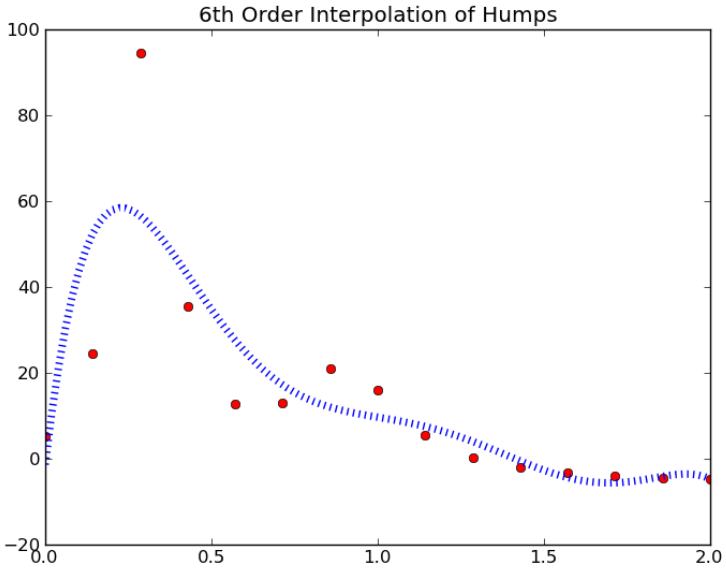
The fit does not look very impressive. Perhaps a higher order polynomial will do better:

```
order_6 = polyfit(x,y,6); x_6 = linspace(0,2,100);
y_6 = polyval(order_6, x_6);
plot(x, y, 'ro', x_6, y_6, 'b:')
title('6th Order Interpolation of Humps')
```




In Python this is:

```
import numpy
x = linspace(0, 2, num=15)
y = 1 / ((x-.3)**2 + .01) + 1 / ((x-.9)**2 + .04) - 6
order_6 = numpy.polyfit(x,y,6)
x_6 = linspace(0,2,num=100)
y_6 = polyval(order_6, x_6)
pylab.plot(x, y, 'ro', x_6 , y_6 , 'b:',linewidth=5)
pylab.title('6th Order Interpolation of Humps')
```



This fit looks more reasonable. Using `polyval`, we can evaluate the function at each of the sample points, and determine the residual (the difference between the estimate given by our function, and the actual function value).

```
y_s = polyval(order_6, x);
residual = y_s - y;
results = [x; y; y_s; residual; residual/std(residual)]'
```

```
results =
         0    5.1765   -1.3549   -6.5314   -0.4710
    0.1429   24.4541   52.4369   27.9827    2.0177
    0.2857   94.3961   56.5751  -37.8210   -2.7271
    0.4286   35.5055   42.7865    7.2809    0.5250
    0.5714   12.7098   27.5492   14.8393    1.0700
    0.7143   12.9303   17.1772    4.2469    0.3062
    0.8571   21.0235   12.0047   -9.0187   -0.6503
    1.0000   16.0000    9.6716   -6.3284   -0.4563
    1.1429    5.4912    7.5085    2.0173    0.1455
    1.2857    0.3160    4.0234    3.7074    0.2673
    1.4286   -2.0900   -0.5119    1.5781    0.1138
    1.5714   -3.3478   -4.3748   -1.0270   -0.0741
    1.7143   -4.0802   -5.6036   -1.5235   -0.1099
    1.8571   -4.5434   -4.1103    0.4331    0.0312
    2.0000   -4.8552   -4.6910    0.1642    0.0118
```

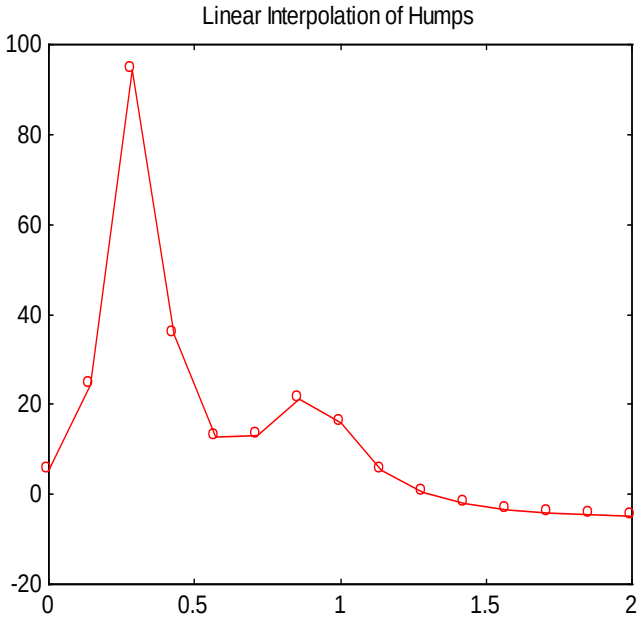
The table above gives the `x` and `y` values, our estimate of the `y` value using a 6th order polynomial, the residual and the normalized residual. A normalized residual of more than 2 is surprising, so if the humps function were a set of experimental results, we would be suspicious of the y values of 52.4369 and 56.5751, but we would probably not reject them.

### 3.1.2 Interpolation I: Linear

Given a discrete sample of points from some function, how can we determine the function value between those points. This is like curve fitting, but the curve must pass through every data point.

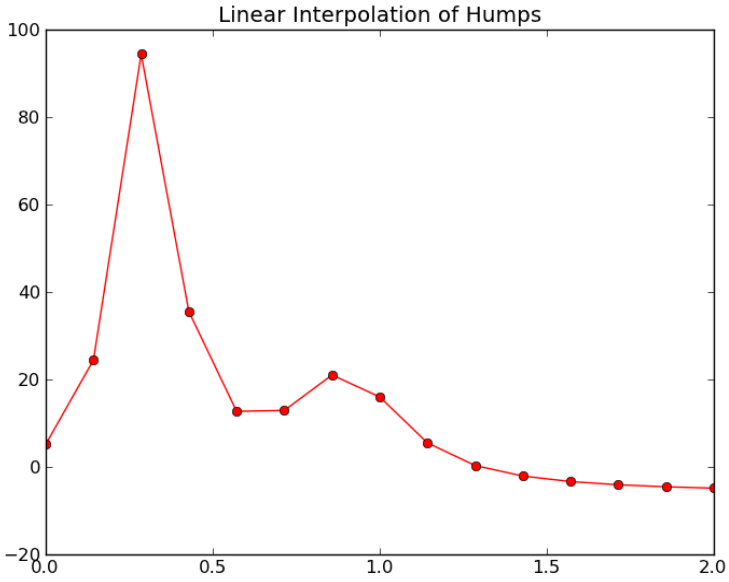
The simplest type of interpolation is the linear interpolation used by MATLAB when plotting a curve; the points are joined up by straight lines.

```
x = linspace(0, 2, 15);
y = humps(x);
plot(x, y,'r-', x, y, 'ro');
title('Linear Interpolation of Humps')
```




In Python this becomes

```
import numpy
x = linspace(0, 2, num=15)
y = 1 / ((x-.3)**2 + .01) + 1 / ((x-.9)**2 + .04) - 6
pylab.plot(x,y,'ro-')
pylab.title('Linear Interpolation of Humps')
pylab.show()
```

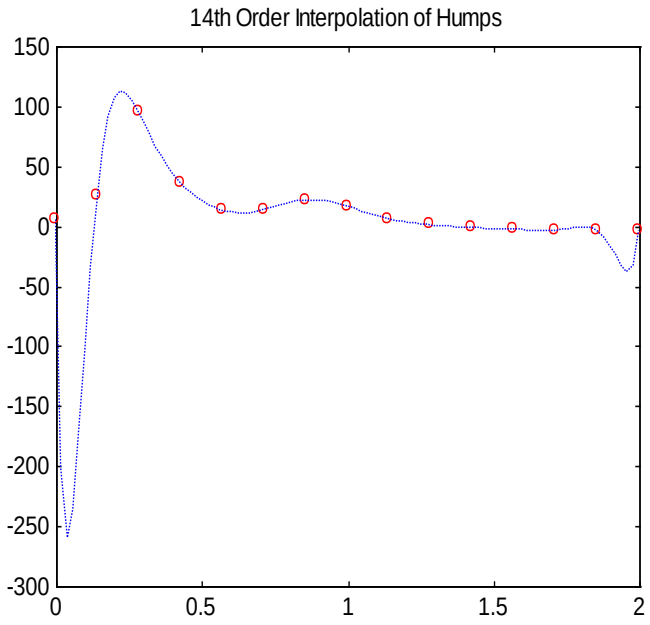



As the number of sample points increases and the distance between them decreases, linear interpolation becomes more accurate. However, if we only have a fixed number of samples, then we want to do the best job fitting the points we have.

## 3.1.3 Interpolation II: Polynomials

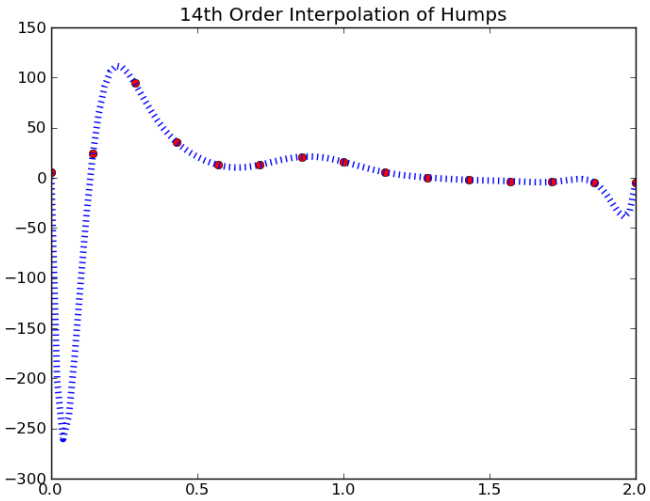
For `n` data points, a polynomial of order `n-1` will pass through each data point.

```
order_14 = polyfit(x,y,14);
x_14 = linspace(0,2,100); y_14 = polyval(order_14, x_14);
plot(x, y, 'ro', x_14, y_14, 'b:')
title('14th Order Interpolation of Humps')
```



In Python this is

```
import numpy
x = linspace(0, 2, num=15)
y = 1 / ((x-.3)**2 + .01) + 1 / ((x-.9)**2 + .04) - 6
order_14 = numpy.polyfit(x,y,14)
x_14 = linspace(0,2,num=100)
y_14 = polyval(order_14, x_14)
pylab.plot(x, y, 'ro', x_14, y_14, 'b:',linewidth=5)
pylab.title('14th Order Interpolation of Humps')
```




The curve fits through every data points, but it oscillates wildly- especially between the first pair and last pair of points. There is no evidence in the data for this oscillatory behaviour, and yet our attempt to plot a curve through all the points has produced it. If we have more data points, we require an even higher order polynomial. The oscillatory behaviour will be worse, and evaluating the polynomial may become very time-consuming. Matlab even gives the following helpful error that there may be a problem with what we are trying to do

```
Warning: Polynomial is badly conditioned. Add points with distinct X
         values, reduce the degree of the polynomial, or try centering
         and scaling as described in HELP POLYFIT.
```

### 3.1.4 Interpolation III: Splines

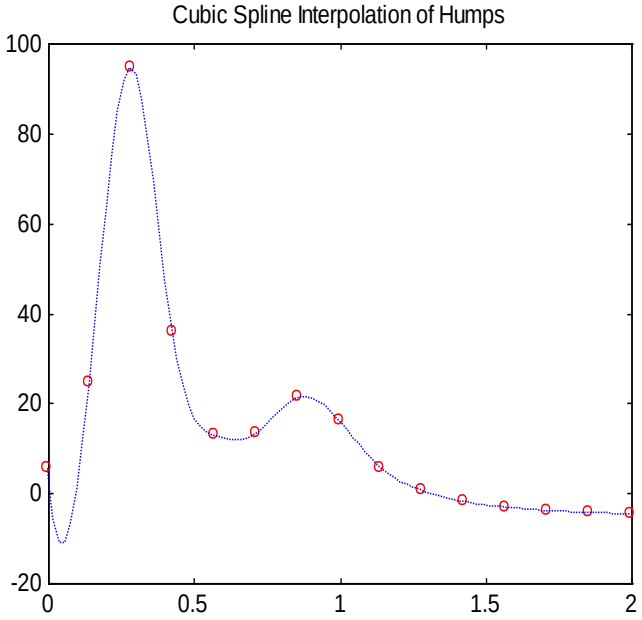
Fortunately, there is a rigorous way to draw a smooth curve between a set of points. The basis of this method is to fit sections of curve between sets of points, using the 'spline' method. Many cars are designed using splines to ensure 'smoothness' of the chassis. MATLAB provides the function `interp1(x, y, xi, 'method')` to perform various types of interpolation between points `x` and `y` and return the interpolated value at the points in `xi`:

- `linear` fits a straight line between pairs of points. This is the default if no method is specified, and is the method used by MATLAB when joining adjacent points on a plot.

- `spline` fits cubic splines between adjacent points.
- `cubic` fits cubic polynomials between sets of 4 points. The `x` points must be uniformly spaced

In all cases, `x` must be monotonic (it must either increase or decrease over the range). None of the values in `xi` can lie outside the range of the `x` values supplied- `interp1` will not perform extrapolation.

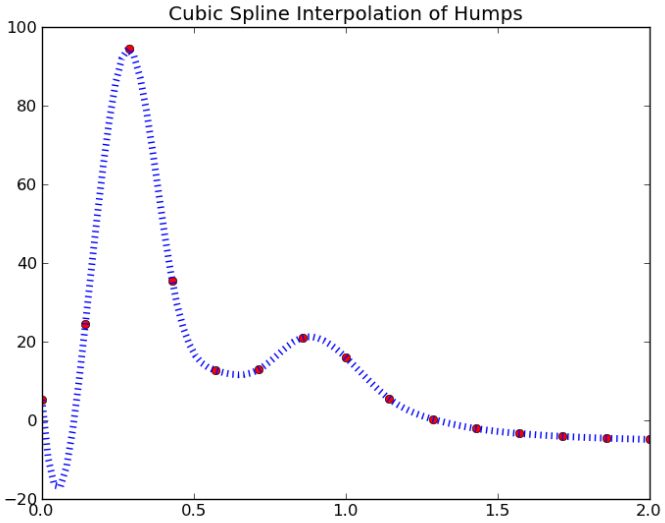
```
spline_x = linspace(0,2,100);
cubic_spline = interp1(x,y,spline_x,'spline');
plot(x, y, 'ro', spline_x, cubic_spline, 'b:')
title('Cubic Spline Interpolation of Humps')
```




We now have a smooth curve between the points.

In Python, this becomes

```
import numpy
x = linspace(0, 2, num=15)
y = 1 / ((x-.3)**2 + .01) + 1 / ((x-.9)**2 + .04) - 6
import scipy.interpolate
cubic_spline = scipy.interpolate.UnivariateSpline(x,y,s=1)
spline_x = linspace(0,2,100)
spline_y=cubic_spline(spline_x)
pylab.plot(x, y, 'ro', spline_x, spline_y, 'b:')
pylab.title('Cubic Spline Interpolation of Humps')
```

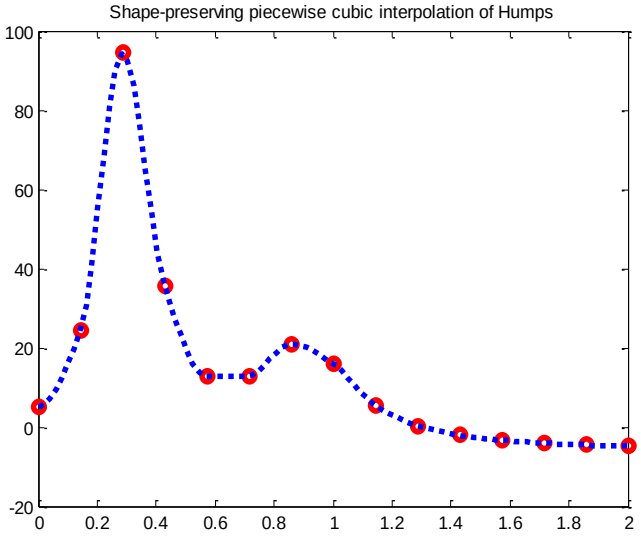



This may also be written as

```
import numpy
x = linspace(0, 2, num=15)
y = 1 / ((x-.3)**2 + .01) + 1 / ((x-.9)**2 + .04) - 6
import scipy.interpolate
spline_x = linspace(0,2,100)
cubic_spline = scipy.interpolate.interp1d(x,y,kind='cubic');
spline_y=cubic_spline(spline_x)
pylab.plot(x, y, 'ro', spline_x, spline_y, 'b:')
pylab.title('Cubic Spline Interpolation of Humps')
```

Finally we can use a (shape-preserving) piecewise cubic interpolation to get

```
x = linspace(0,2,15);
y = humps(x);
pchip_x = linspace(0,2,100);
pchip_y = interp1(x,y, pchip_x,'pchip');
plot(x, y, 'ro', pchip_x, pchip_y, 'b:','LineWidth',3)
title('Shape-preserving piecewise cubic interpolation of Humps')
```

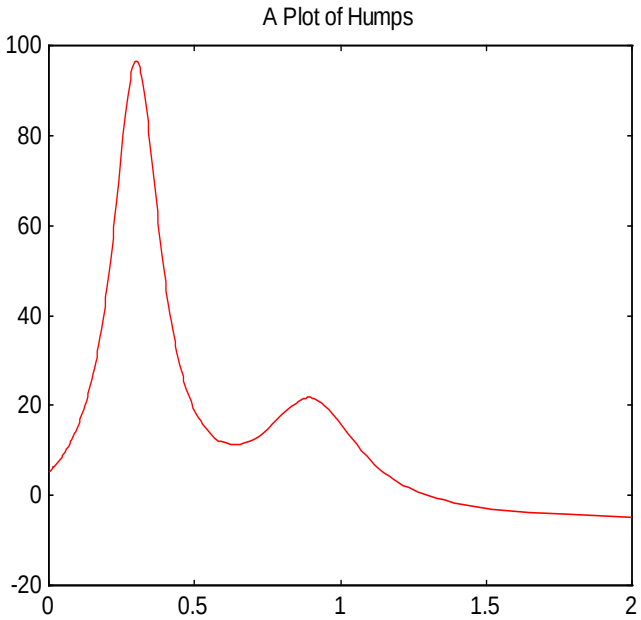



In this case, the '**pchip**' method provides a smooth curve with the least oscillations between points. In the next section, we reveal what the function looks like. There is no direct analogy of this built into Python, though code exists on the web to perform this method.

### 3.1.5 The Humps function

Here is a plot of the humps function.

```
fplot('humps',[0 2]);
title('A Plot of Humps')
```




The cubic interpolation reproduced a similar plot using only a sample of ten points across the range. For more information about interpolation, see one of the numerical analysis books in the bibliography.

## 3.1.6 Interpolation III: Surface Splines

(This section includes some 3D plots, so you might like to look briefly at the Matlab functions `mesh` and `meshgrid` first.)

Interpolating over a surface is an extension of interpolating between two data points. In MATLAB we can use `interp2(X, Y, Z, XI, ZI, 'method')` to interpolate at the points `(XI, YI)` the function whose value is `Z at X, Y`.

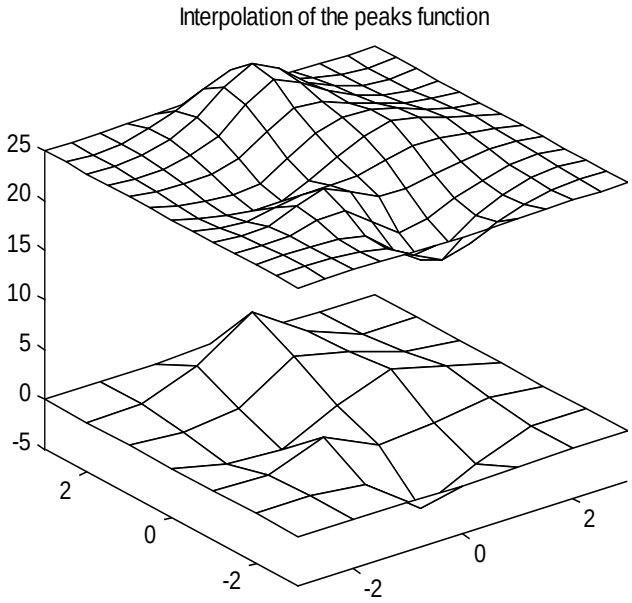
Two methods that can be used

- `linear` linear interpolation. This is the default if no method is specified.
- `cubic` cubic interpolation. The `X` and `Y` points must be uniformly spaced.

In all cases, `X` and `Y` must be monotonic. None of the values in `XI` can lie outside the range of the `X` or `Y` values supplied- `interp2` will not perform extrapolation.

In this example, we use cubic interpolation to produce a smooth mesh from samples of MATLAB's `peaks` function:

```
[X, Y] = meshgrid(-3:1:3);
Z = peaks(X, Y);
[XI, YI]= meshgrid(-3:0.5:3);
ZI_linear = interp2(X, Y, Z, XI, YI, 'linear');
ZI_cubic = interp2(X, Y, Z, XI, YI, 'cubic');
mesh(XI, YI, ZI_cubic+25);
hold on
mesh(X, Y, Z);
colormap([0,0,0]);
hold off
axis([-3, 3, -3, 3, -5, 25]);
title('Interpolation of the peaks function')
```



The bottom part of the figure shows the coarse samples, and the top shows the interpolated grid. We have displaced the interpolated grid upwards.

## *3.2 Lagrange Interpolation Polynomial*

### 3.2.1 Linear interpolation

Let us first consider linear interpolation between two points. The Lagrange form of the equation for a straight line passing through $(x_1, y_1)$ and $(x_2, y_2)$ is:

$$p(x) = \frac{(x - x_2)}{(x_1 - x_2)} y_1 + \frac{(x - x_1)}{(x_2 - x_1)} y_2 \tag{3.39}$$

Where does this come from? Consider the normal formula "$y = m\ x + c$" for the equation of a straight line shown in Figure 2.

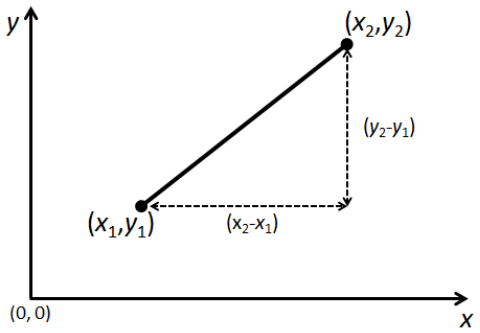



Figure 2 Straight line passing through $(x_1, y_1)$ and $(x_2, y_2)$

$$y = p(x) = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) + y_1 \tag{3.40}$$

Putting over the common denominator $(x_2 - x_1)$

$$y = p(x) = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) + y_1 \frac{(x_2 - x_1)}{(x_2 - x_1)}. \tag{3.41}$$

Collecting common terms in $y_2$ and $y_1$:

$$y = p(x) = \frac{y_2(x - x_1) + y_1(-x + x_1 + x_2 - x_1)}{(x_2 - x_1)}. \tag{3.42}$$

Grouping terms in $y_1$ and $y_2$:

$$p(x) = \frac{(-x + x_2)}{(x_2 - x_1)} y_1 + \frac{(x - x_1)}{(x_2 - x_1)} y_2. \tag{3.43}$$

Changing the sign of the $y_1$ part yields:

$$p(x) = \frac{(x - x_2)}{(x_1 - x_2)} y_1 + \frac{(x - x_1)}{(x_2 - x_1)} y_2 \tag{3.44}$$

which is the same as (3.39). Q.E.D. We can check the equation by noting that if $x = x_1$, then $y=y_1$ and if $x=x_2$, then $y=y_2$ (this is not, of course, a proof!)

## 3.2.2 Quadratic interpolation

The Lagrange form of the equation of the parabola passing through three points $(x_1, y_1)$, $(x_2, y_2)$, and $(x_3, y_3)$ is:

$$p(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} y_1 + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} y_2 + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} y_3 \qquad (3.45)$$

Let us consider the quadratic passing through

$$(x_1, y_1) = (-2, 4),\ (x_2, y_2) = (0, 2), \text{ and } (x_3, y_3) = (2, 8). \qquad (3.46)$$

Using the formula above gives

$$p(x) = \frac{(x-0)(x-2)}{(-2-0)(-2-2)} 4 + \frac{(x-(-2))(x-2)}{(0-(-2))(0-2)} 2 + \frac{(x-(-2))(x-0)}{(2-(-2))(2-0)} 8 \qquad (3.47)$$

Simplifying yields

$$p(x) = \frac{x(x-2)}{8} 4 + \frac{(x+2)(x-2)}{-4} 2 + \frac{x(x+2)}{8} 8 \qquad (3.48)$$

$$p(x) = x^2 + x + 2 \qquad (3.49)$$

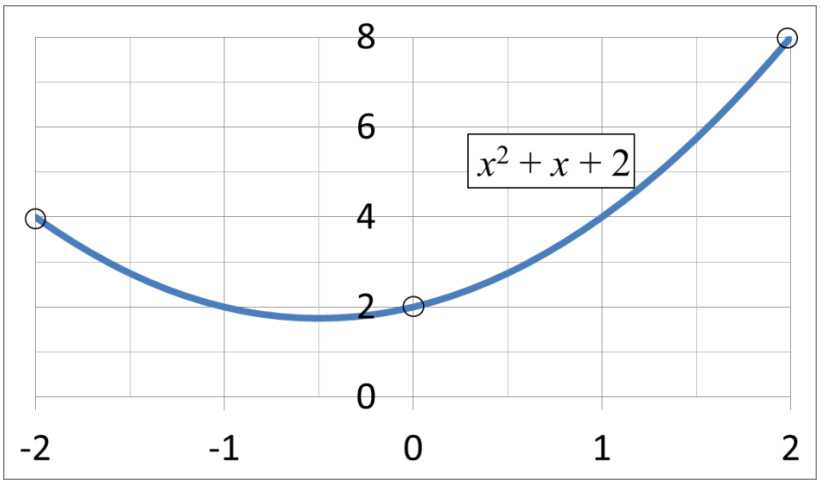



Figure 3 Quadratic $x^2 + x + 2$ passing through (-2, 4), (0, 2) and (2, 8). Control points marked with circles

We can check this in Matlab

```
x=[-2,0,2];
y=[4,2,8];
p=polyfit(x,y,2)
p =
    1.0000    1.0000    2.0000
```

We can also check this in Python

```
x=array([-2,0,2]);
y=array([4,2,8]);
p=pylab.polyfit(x,y,2)
print p
[ 1.  1.  2.]
```

These `p` coefficients are, respectively, the terms from $x^2$, $x$, constant which yields the polynomial $p(x) = x^2 + x + 2$.

### 3.2.3 General form

The general form for Lagrangian interpolation of an $n^{th}$ order polynomial through ($n$+1) points is

$$p(x) = L_1 y_1 + L_2 y_2 + \ldots L_n y_n \text{, where}$$

$$L_k(x) = \frac{(x-x_1)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)}{(x_k-x_1)\ldots(x_k-x_{k-1})(x_k-x_{k+1})\ldots(x_k-x_n)}. \tag{3.50}$$

We note that $L_k(x_k)=1$ and $L_k(x)=0$ when $x=x_j$ for $j \neq k$.

Whilst there is nothing wrong with using this formula directly to compute the interpolating polynomial, there is a more computationally efficient way to set up the calculation, which we will now describe.

## *3.3 Newton Interpolation Polynomial and Divided Differences*

Newton's form of the equation for a straight line passing through ($x_1$, $y_1$) and ($x_2$, $y_2$) is:

$$p(x) = a_1 + a_2(x-x_1). \tag{3.51}$$

For a parabola passing through three points ($x_1$, $y_1$), ($x_2$, $y_2$), and ($x_3$, $y_3$) it is:

$$p(x) = a_1 + a_2(x-x_1) + a_3(x-x_1)(x-x_2). \tag{3.52}$$

The general Newton form of an $n^{th}$ order polynomial through ($n$+1) points is:

$$p(x) = a_1 + a_2(x-x_1) + a_3(x-x_1)(x-x_2) + \ldots + a_n(x-x_1)\ldots(x-x_{n-1}). \tag{3.53}$$

This is just a different way of writing the polynomial that fits between two points. Let us reconsider the previous example of interpolating between

$$(x_1, y_1) = (-2, 4),\ (x_2, y_2) = (0, 2), \text{ and } (x_3, y_3) = (2, 8). \tag{3.54}$$

We have

$$p(x) = a_1 + a_2(x-(-2)) + a_3(x-(-2))(x-0). \tag{3.55}$$

At $x$ = -2, $y = p(x) = 4$ so:

$$a_1 = y_1 = 4, \tag{3.56}$$

And

$$a_2 = \frac{y_2 - y_1}{x_2 - x_1} = \frac{2-4}{0-(-2)} = -1, \tag{3.57}$$

and

$$a_3 = \frac{\dfrac{y_3 - y_2}{x_3 - x_2} - \dfrac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1} = \frac{\dfrac{8-2}{2-0} - \dfrac{2-4}{0-(-2)}}{2-(-2)} = 1. \tag{3.58}$$

Thus we have

$$p(x) = 4 - (x+2) + x(x+2) = x^2 + x + 2 \tag{3.59}$$

As in the previous section. There is another way to calculate this, by noting how the working in (3.58) builds on values calculated in (3.57)

| $x_i$ | $y_i$ | $d_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$ | $dd_i = \frac{d_{i+1} - d_i}{x_{i+2} - x_i}$ |
|---|---|---|---|
| -2 | **4** | $\frac{y_2 - y_1}{x_2 - x_1} = \frac{2-4}{0-(-2)} = $ **-1** | |
| 0 | 2 | $\frac{y_3 - y_2}{x_3 - x_2} = \frac{8-2}{2-0} = 3$ | $\frac{d_2 - d_1}{x_3 - x_1} = \frac{3-(-1)}{2-(-2)} = $ **1** |
| 2 | 8 | | |

Reading the boxed numbers gives $a_1 = 4$, $a_2 = -1$ and $a_3 = 1$, as above.

There is a related method called "Neville's Algorithm" which uses a similar divided difference table to efficiently compute the value of the polynomial at a given point.

## *3.4 Splines*

### 3.4.1 Linear Splines

As illustrated in 3.1.3 interpolating an $(n\text{-}1)^{th}$ order polynomial to fit through $n$ data points leads to oscillations in the fitting function that are generally not representative of the underlying function. Splines are a way of fitting a line between successive groups of points so that we have a function that interpolates "smoothly" between the data values.

Let us consider first fitting a straight line between each of the pairs of points in turn. Recall equation (3.39)

$$p(x) = \frac{(x - x_2)}{(x_1 - x_2)} y_1 + \frac{(x - x_1)}{(x_2 - x_1)} y_2 \qquad (3.60)$$

Consider that we have four points to interpolate between: $(x_1, y_1)$, $(x_2, y_2)$, $(x_3, y_3)$, and $(x_4, y_4)$. This defines three intervals on the $x$-axis where we will fit three straight lines using (3.39)

$$p(x)=\begin{cases}\dfrac{(x-x_2)}{(x_1-x_2)}y_1+\dfrac{(x-x_1)}{(x_2-x_1)}y_2 & x_1\le x\le x_2\\ \dfrac{(x-x_3)}{(x_2-x_3)}y_2+\dfrac{(x-x_2)}{(x_3-x_2)}y_3 & x_2\le x\le x_3\\ \dfrac{(x-x_4)}{(x_3-x_4)}y_3+\dfrac{(x-x_3)}{(x_4-x_3)}y_4 & x_3\le x\le x_4\end{cases}. \qquad (3.61)$$

If we have these points as (0,0), (1,1), (2,4) and (3,3). This yields the following equations for the straight lines

$$p(x)=\begin{cases}\dfrac{(x-1)}{(0-1)}0+\dfrac{(x-0)}{(1-0)}1 = x & 0\le x\le 1\\ \dfrac{(x-2)}{(1-2)}1+\dfrac{(x-1)}{(2-1)}4 = 3x-2 & 1\le x\le 2\\ \dfrac{(x-3)}{(2-3)}4+\dfrac{(x-2)}{(3-2)}3 = 6-x & 2\le x\le 3\end{cases}. \qquad (3.62)$$

We also can use Matlab to perform the interpolation.

```
x=[0 , 1, 2, 3];
y=[0, 1, 4, 3];
spline_linear_x = linspace(0,3,100);
linear_spline = interp1(x,y,spline_linear_x,'linear');
plot(x, y, 'ro', spline_linear_x, linear_spline, 'b:')
title('Linear spline interpolation of Data')
grid on
```

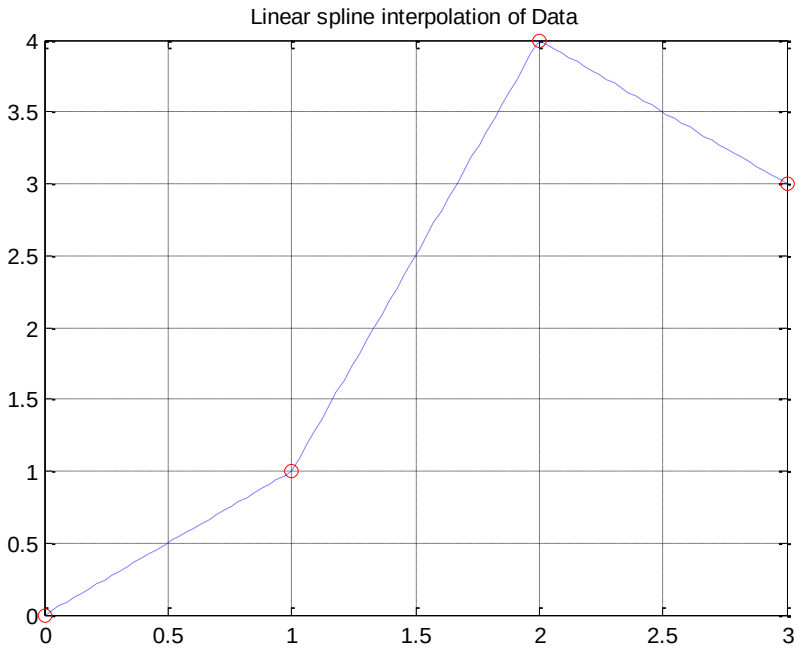



Figure 4 Piecewise Linear Interpolation between 4 points. Within each interval we just join the points up with a straight line and the equation of the line is given by the Lagrange form for linear interpolation.

Whilst we have achieved the aim of fitting a line between the data points and avoided the oscillatory nature of using a polynomial to achieve this, the result is not "smooth" in the sense that the derivative of the function has a discontinuity at the fixed interpolation points (and its second derivative is undefined). We now turn out attention to addressing this.

### 3.4.2 Cubic Splines

It is possible to construct a smooth curve between data points such that the resulting curve has continuous first and second derivatives across the whole interval. The continuity of the first derivative means that the result graph does not have "sharp corners" and continuity of the second derivative means that, effectively, a radius of curvature is defined at each point.

Suppose we have $n$ points $(x_1, y_1)$, $(x_2, y_2)$, …, $(x_n,y_n)$ in an interval $[a, b]$ such that $a = x_1 < x_2 < \ldots < x_n = b$. We do not require that the spacing between these points is even, so it is convenient to define $h_i = x_{i+1} - x_i$. We are looking for a spline function

$$S(x) = \begin{cases} P_1(x) & x_1 \le x \le x_2 \\ P_i(x) & x_i \le x \le x_{i+1} \\ P_{n-1}(x) & x_{n-1} \le x \le x_n \end{cases}, \qquad (3.63)$$

that is a piecewise cubic with continuous derivatives up to order 2. For $i = 1, 2, \ldots, n\text{-}1$ we can write:

$$P_i(x) = a_{i-1}\frac{(x_{i+1}-x)^3}{6h_i} + a_i\frac{(x-x_i)^3}{6h_i} + b_i(x_{i+1}-x) + c_i(x-x_i)\,. \qquad (3.64)$$

This leads to $n$-2 equations for the $n$ unknowns $a_0$, …., $a_{n-1}$:

$$\frac{h_i}{6}a_{i-1} + \frac{h_i+h_{i+1}}{3}a_i + \frac{h_{i+1}}{6}a_{i+1} = \frac{y_{i+2}-y_{i+1}}{h_{i+1}} - \frac{y_{i+1}-y_i}{h_i} \quad (i=1,\ldots,n-2)\,. \qquad (3.65)$$

(For a proof of this, see one of the references at the end). It is easy to verify that this goes through each point and has continuous derivatives up to order 2 at each point.

As we know from our work on linear equations above, we require $n$ equations in $n$ unknowns to solve this system. The undetermined part of the above equations is what we do at the end points of the interpolating spline. Some choices can be made as per the following table.

| Choice | Meaning | Name (in Matlab) |
|---|---|---|
| $S''(x)=0 \quad x=x_1,\, x=x_n$ | Set the second derivative to be zero at the end points | "Natural" |
| $S(x_1)=S(x_n)$<br>$S'(x_1)=S'(x_n)$<br>$S''(x_1)=S''(x_n)$ | Make the function and its first and second derivatives equal at the end points | "Periodic" |
| $S'(x_1)=\alpha$<br>$S'(x_n)=\beta$ | Set fixed values for the first derivative of $\alpha$ and $\beta$ | "Clamped" or "Complete" |
| $S''(x_1)=\alpha$<br>$S''(x_n)=\beta$ | Set fixed values for the second derivative of $\alpha$ and $\beta$ | "Second" |

Illustrations of these endpoints using Matlab "splinetool" (Note this is an R2010b or higher function)

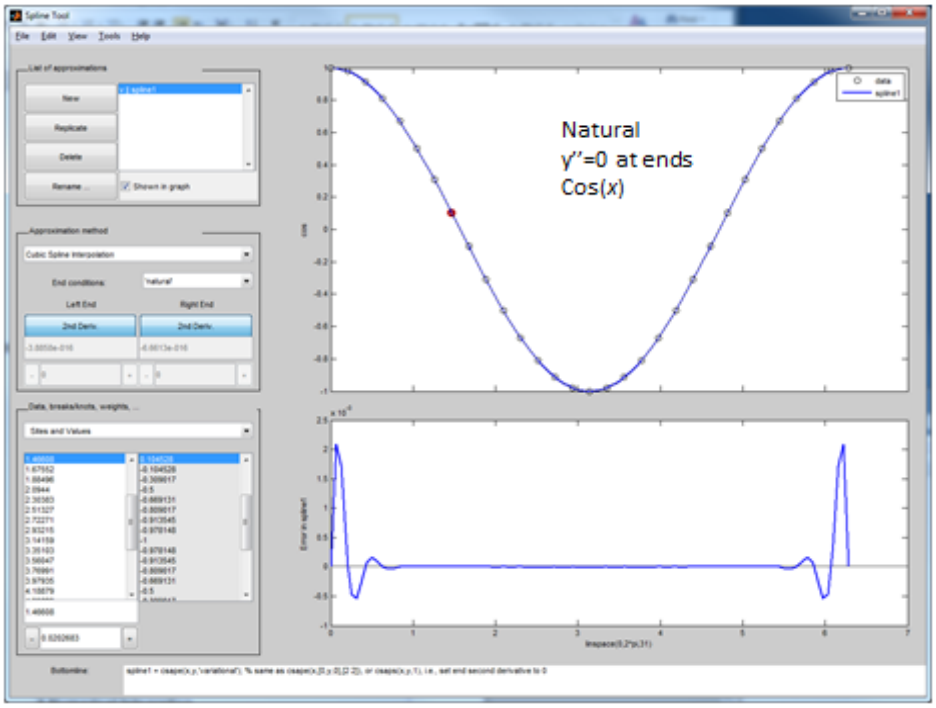



Figure 5 Natural End conditions

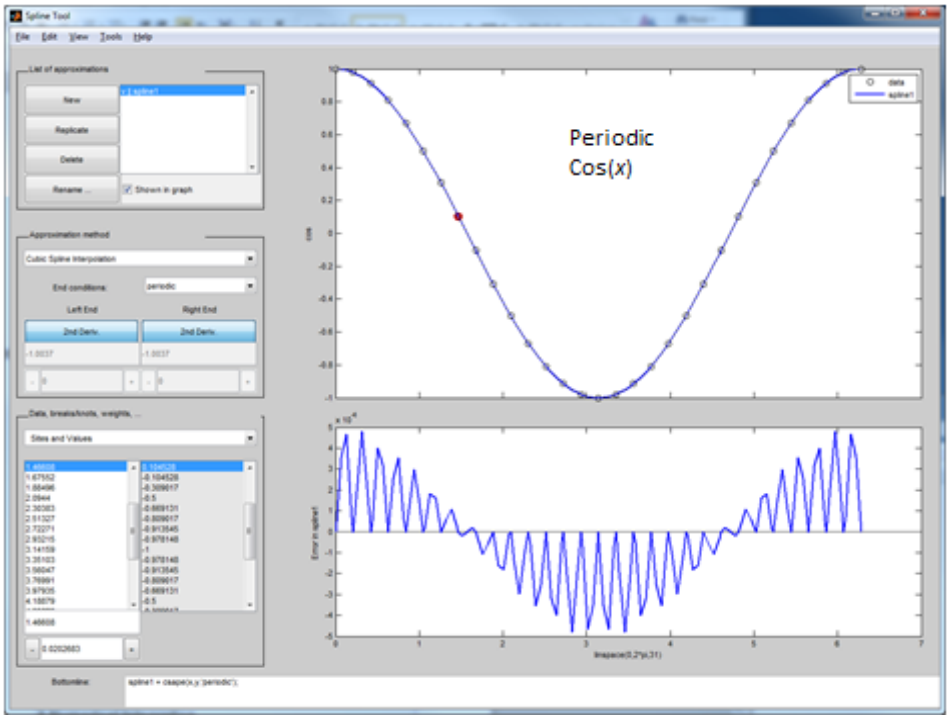



Figure 6 Periodic End conditions

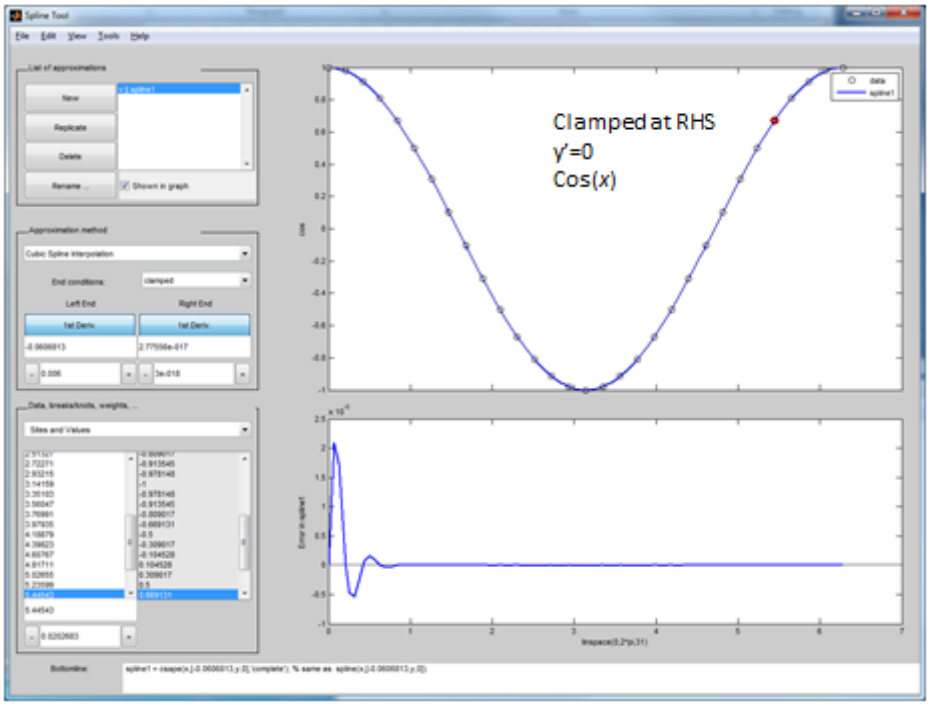



Figure 7 Clamped End conditions

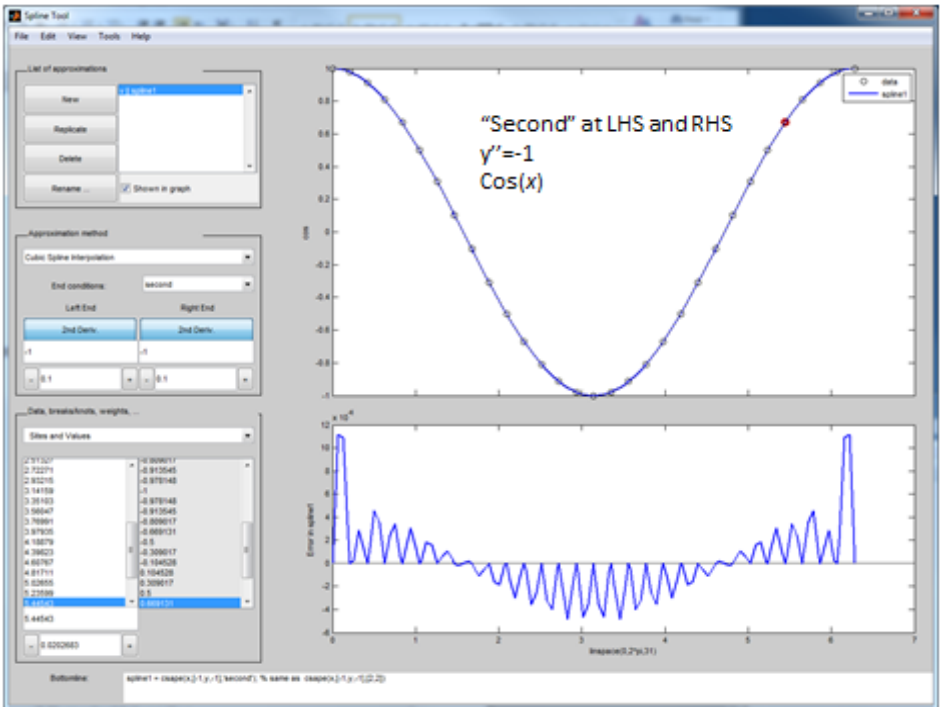


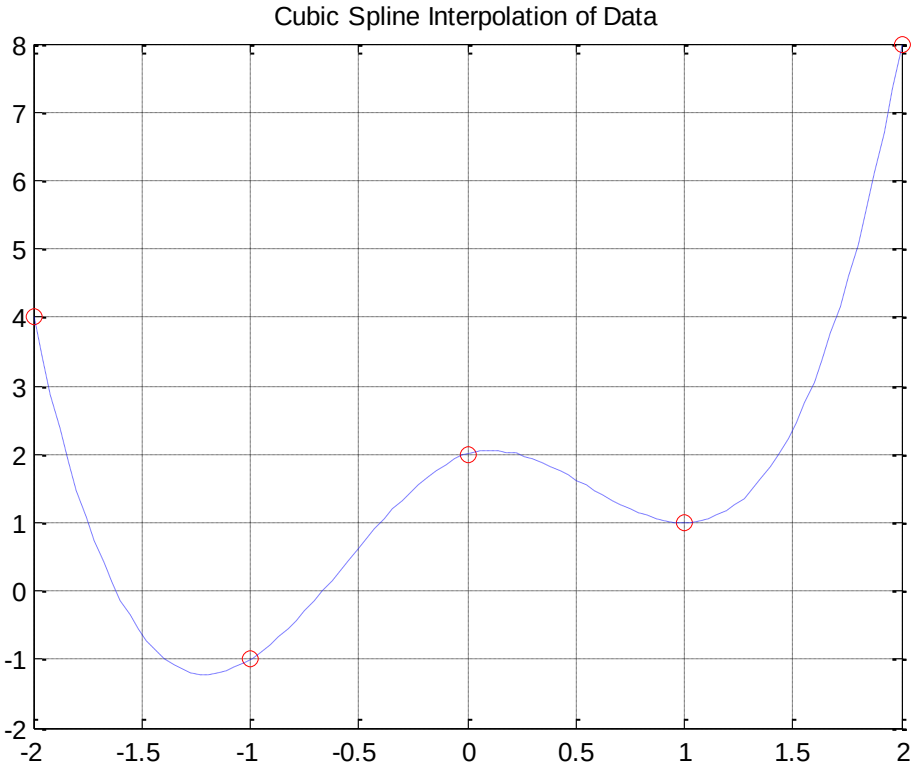

Figure 8 "Second" End conditions

Let us now consider the Natural Spline formulation in which the 2nd derivatives at the endpoints are set to zero so the values of $b_i$ and $c_i$ ($i$=1,… $n$-1) are expressed in terms of the $a_i$ coefficients as follows:

$$b_i = \frac{y_i}{h_i} - \frac{a_{i-1}h_i}{6}, \quad c_i = \frac{y_{i+1}}{h_i} - \frac{a_i h_i}{6}. \tag{3.66}$$

This leads to a triadiagonal set of equations for the coefficients, which can then be used to give each of the piecewise cubic splines. We can also show the interpolation in Matlab.

```
x=[-2,-1,0,1,2];
y=[4,-1,2,1,8];
spline_x = linspace(-2,2,100);
cubic_spline = interp1(x,y,spline_x,'spline');
plot(x, y, 'ro', spline_x, cubic_spline, 'b:')
title('Cubic Spline Interpolation of Data')
grid on
```




With the curve fitting toolbox installed, we can also use `splinetool([-2,-1,0,1,2],[4,-1,2,1,8])` to explore this interactively and examine different endpoint conditions.

# 4 Numerical Integration

[For Python information on integration, see Chapter 16 of Prof Fangohr's notes at:

http://www.soton.ac.uk/~sesg2006/textbook/Python-for-Computational-Science-and-Engineering.pdf]

The purpose of numerical integration (or 'quadrature') is to find the area under a graph.

- Make any possible analytic progress . Even a complicated analytic result is likely to be quicker and more accurate to evaluate than using a numerical method
- Certain integrals have been well studied (for example it may be possible to represent the integrand as a series and then integrate term by term). In this case it may be easier to control the accuracy using this alternative method.

$$\int_{x=u}^{\infty} \frac{x\exp(-\mu x)}{\sqrt{x^2-u^2}}\,dx = uK_1(u\mu) \quad [u>0, \mathrm{Re}\,\mu>0] \tag{4.67}$$

The integral above can be written in terms of a Bessel function ($K_1$), which can be evaluated using standard techniques.

- For the remainder of integrals it is necessary to use numerical integration. The strategy will be to evaluate the function to be integrated at a number of sample points and find the area enclosed between these points and the axis.

## *4.1 Trapezium Rule*

This is the simplest method and consists of joining up each of our sample points with a straight line (like linear interpolation). The area under the curve becomes the sum of the areas under the resulting trapezia.

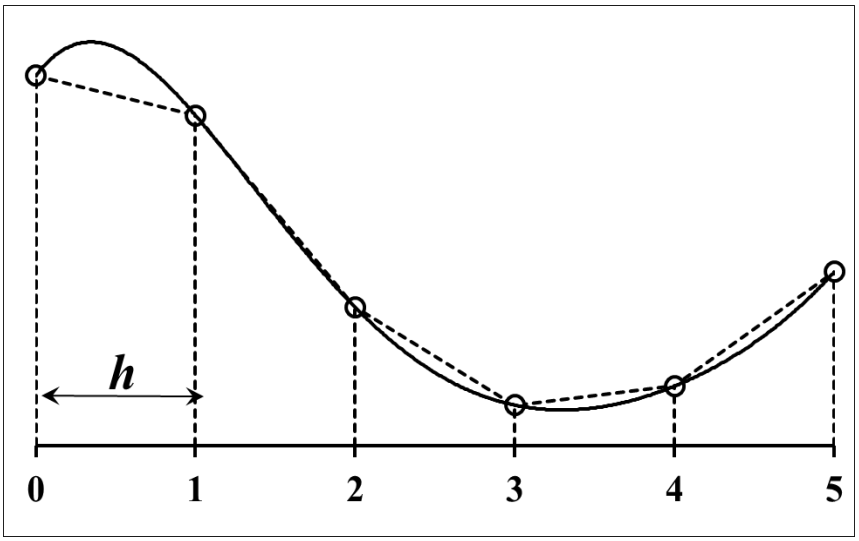



Figure 9 Trapezium Rule

Let $y_0 \ldots y_n$ be the function values at each of the sample points.

$$\int_a^b f(x)\,\mathrm{d}x \approx h\left\{\frac{1}{2}(y_0+y_n)+(y_1+y_2+y_3+\ldots y_{n-1})\right\} \tag{4.68}$$

Numerical Example. Consider the following integral

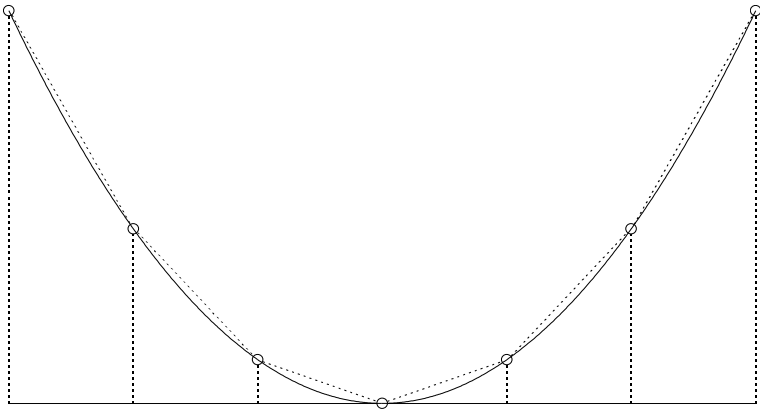
$$\int_{x=0}^{x=4} \exp(x)\,\mathrm{d}x \tag{4.69}$$

| | $y_i = \exp(x_i)$ | Weight | $y_i \times$ weight |
|---|---|---|---|
| $x_0 = 0$ | $y_0 = 1.00000$ | 0.5 | 0.50000 |
| $x_1 = 0.5$ | $y_1 = 1.64872$ | 1 | 1.64872 |
| $x_2 = 1$ | $y_2 = 2.71828$ | 1 | 2.71828 |
| $x_3 = 1.5$ | $y_3 = 4.48169$ | 1 | 4.48169 |
| $x_4 = 2$ | $y_4 = 7.38906$ | 1 | 7.38906 |
| $x_5 = 2.5$ | $y_5 = 12.18249$ | 1 | 12.18249 |
| $x_6 = 3$ | $y_6 = 20.08554$ | 1 | 20.08554 |
| $x_7 = 3.5$ | $y_7 = 33.11545$ | 1 | 33.11545 |
| $x_8 = 4$ | $y_8 = 54.59815$ | 0.5 | 27.29908 |
| | | Sum | 109.42031 |
| | | $h$ | 0.50000 |
| | | Result = $h \times$ sum | 54.71015 |

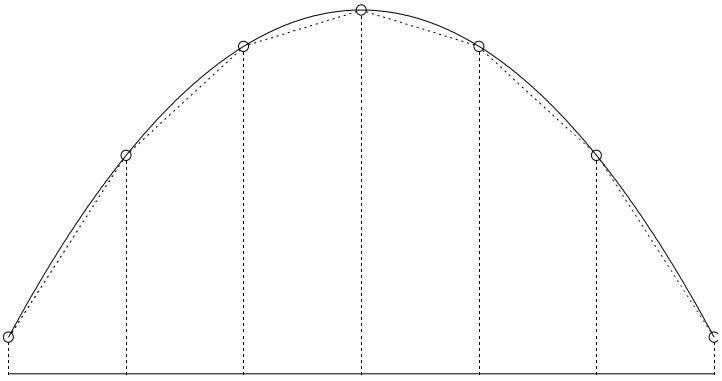
The final result is 54.71015 (5dp). This compares well to the exact answer of exp(4) – exp(0) = 53.59815.

### 4.1.1 Notes on the trapezium Rule

1) In the general case the error is of the form (constant)$\times h^2$. Thus if the number of points is doubled then the error decreases by a factor of 4.
2) The trapezium rule overestimates when the function is convex upwards and underestimates when it is concave downwards. When the function has an equal number of maxima and minima then the trapezium rule will give a better result.



(a)



(b)

Figure 10 Trapezium Rule (a) overestimate when function is convex upwards (b) underestimate when function is convex downwards

3) It can be shown that :

$$\int_a^b f(x)\,\mathrm{d}x \approx \text{trapezium rule} + \frac{h^2}{12}\left(f'(a) - f'(b)\right) + O(h^3) \qquad (4.70)$$

This 'endpoint' correction, which is calculated using the derivatives of the function evaluated at the endpoints can be used to provide a more accurate estimate of the integral when these derivatives are available. In the above case we have

$$f(x) = \exp(x) \Rightarrow f'(x) = \exp(x) \qquad (4.71)$$

So the endpoint correction is

$$\eta = \frac{h^2}{12}\left(f'(a) - f'(b)\right) = \frac{0.5^2}{12}(1.00000 - 54.59815) = -1.11663 \qquad (4.72)$$

Applying this correction gives a result of 54.71015 - 1.11663 = 53.59352 for the integral (which is even closer to the exact answer of 53.59815).

There are circumstances when the endpoint correction vanishes (a) when $f'(a) = f'(b) = 0$ or (b) simply $f'(a) = f'(b)$ - such as for a periodic function. However, if $f'(a)$ or $f'(b)$ is large then the result from the trapezium rule will be poor.

## *4.2 Simpson's Rule*

In the trapezium rule we perform linear interpolation between each of the sample points. By fitting a curve through sets of points (as in the spline interpolation) we can increase the accuracy of the method. Simpson's rule fits a parabola through sets of three consecutive sample points.

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\{(y_0 + y_n) + 4(y_1 + y_3 + y_5 + \ldots y_{n-1}) + 2(y_2 + y_4 + \ldots y_{n-2})\} \qquad (4.73)$$

This can also be remembered as "$\frac{h}{3}$ ×(first + last + 4× sum of odds + 2× sum of evens)".

For Simpson's rule doubling the number of sample points decreases the error by a factor of 16.

We now repeat the integral computed previously using the trapezium rule. In this case the weighting of the points in the calculation changes.

$$\int_{x=0}^{x=4} \exp(x)\,\mathrm{d}x \qquad (4.74)$$

| | $y_i = \exp(x_i)$ | Weight | $y_i \times$ weight |
|---|---|---|---|
| $x_0 = 0$ | $y_0 = 1.00000$ | 1 | 1.00000 |
| $x_1 = 0.5$ | $y_1 = 1.64872$ | 4 | 6.59489 |
| $x_2 = 1$ | $y_2 = 2.71828$ | 2 | 5.43656 |
| $x_3 = 1.5$ | $y_3 = 4.48169$ | 4 | 17.92676 |
| $x_4 = 2$ | $y_4 = 7.38906$ | 2 | 14.77811 |
| $x_5 = 2.5$ | $y_5 = 12.18249$ | 4 | 48.72998 |
| $x_6 = 3$ | $y_6 = 20.08554$ | 2 | 40.17107 |
| $x_7 = 3.5$ | $y_7 = 33.11545$ | 4 | 132.46181 |
| $x_8 = 4$ | $y_8 = 54.59815$ | 1 | 54.59815 |
| | | Sum | 321.69732 |
| | | *h/3* | 0.16667 |
| | | Result = *h/3* × sum | 53.61622 |

The final answer is thus 53.61622, which is more accurate than the result given by the trapezium rule using the same sample points.

The end correction for Simpson's Rule is:

$$\eta = \frac{h^4}{180}\{y'''(a) - y'''(b)\} \tag{4.75}$$

Substituting in the values

$$\eta = (0.5^4) / 180 \times (1 - 54.5982) = -0.0186105, \tag{4.76}$$

which gives a final result for the integral of 53.59761. This agrees to 4 figures of accuracy with the exact answer (=53.59815)!

## *4.3 Recursive Quadrature*

How many points should be used in the integration? Clearly it is hard to know this in advance! However, it is possible to add points to the integration without wasting the effort of the previous function evaluations.

The explicit formulae for the integral using 1, 2, 4, and 8 strips are

$$\begin{aligned}
I(1) &= \tfrac{1}{2} f(0) + \tfrac{1}{2} f(1) \\
I(2) &= \tfrac{1}{4} f(0) + \tfrac{1}{2} f(\tfrac{1}{2}) + \tfrac{1}{4} f(1) \\
I(4) &= \tfrac{1}{8} f(0) + \tfrac{1}{4} f(\tfrac{1}{4}) + \tfrac{1}{4} f(\tfrac{1}{2}) + \tfrac{1}{4} f(\tfrac{3}{4}) + \tfrac{1}{8} f(1) \\
I(8) &= \tfrac{1}{16} f(0) + \tfrac{1}{8} f(\tfrac{1}{8}) + \tfrac{1}{8} f(\tfrac{1}{4}) + \tfrac{1}{8} f(\tfrac{3}{8}) + \tfrac{1}{8} f(\tfrac{1}{2}) + \tfrac{1}{8} f(\tfrac{5}{8}) \\
&\quad + \tfrac{1}{8} f(\tfrac{3}{4}) + \tfrac{1}{8} f(\tfrac{7}{8}) + \tfrac{1}{16} f(1)
\end{aligned} \tag{4.77}$$

By judiciously regrouping the terms it is possible to determine the value of the integral $I(2n)$ using only function evaluations not already computed in finding $I(n)$:

$$\begin{aligned}
I(1) &= \tfrac{1}{2} f(0) + \tfrac{1}{2} f(1) \\
I(2) &= \tfrac{1}{4} f(0) + \tfrac{1}{4} f(1) + \tfrac{1}{2} f(\tfrac{1}{2}) \\
I(4) &= \tfrac{1}{8} f(0) + \tfrac{1}{8} f(1) + \tfrac{1}{4} f(\tfrac{1}{2}) + \tfrac{1}{4} f(\tfrac{1}{4}) + \tfrac{1}{4} f(\tfrac{3}{4}) \\
I(8) &= \tfrac{1}{16} f(0) + \tfrac{1}{16} f(1) + \tfrac{1}{8} f(\tfrac{1}{2}) + \tfrac{1}{8} f(\tfrac{1}{4}) + \tfrac{1}{8} f(\tfrac{3}{4}) + \tfrac{1}{8} f(\tfrac{1}{8}) + \\
&\quad + \tfrac{1}{8} f(\tfrac{3}{8}) + + \tfrac{1}{8} f(\tfrac{5}{8}) + \tfrac{1}{8} f(\tfrac{7}{8})
\end{aligned} \tag{4.78}$$

This leads to

$$\begin{aligned}
I(1) &= \tfrac{1}{2} f(0) + \tfrac{1}{2} f(1) \\
I(2) &= \tfrac{1}{2} I(1) + \tfrac{1}{2}\left[f(\tfrac{1}{2})\right] \\
I(4) &= \tfrac{1}{2} I(2) + \tfrac{1}{4}\left[f(\tfrac{1}{4}) + f(\tfrac{3}{4})\right] \\
I(8) &= \tfrac{1}{2} I(4) + \tfrac{1}{8}\left[f(\tfrac{1}{8}) + f(\tfrac{3}{8}) + f(\tfrac{5}{8}) + f(\tfrac{7}{8})\right]
\end{aligned} \tag{4.79}$$

Over a more general interval $[a, b]$ this yields the formula:

$$I(2n) = \tfrac{1}{2} I(n) + h\sum_{i=1}^{n} f(a + (2i-1)h), \tag{4.80}$$

where $h = (b - a) / (2n)$.

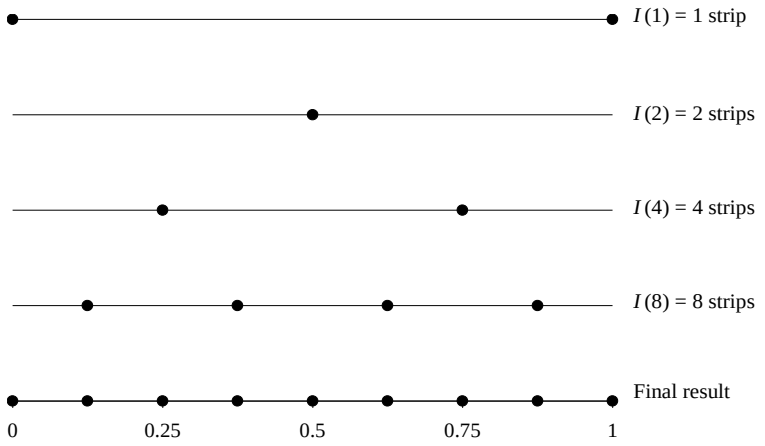



Figure 11 Recursive Trapezium Rule. At each level we add the points marked, doubling the number of strips used in the integration at each step and re-using previous work

## *4.4 Adaptive Quadrature*

Adaptive quadrature methods take into account the behaviour of the integrand by subdividing the interval of integration into sections so that numerical integration on the sub-intervals will provide results of sufficient precision. If, for example, the function you wish to integrate is smooth in one region of the interval of interest, but oscillatory in another, then it is necessary to use more strips in the oscillatory part. This is known as 'adaptive quadrature.'

## *4.5 Monte Carlo Methods*

It is possible to estimate an integral using the following formula:

$$I = \int_{x=a}^{b} f(x)\,\mathrm{d}x \approx (b-a)\left\langle f(x_i)\right\rangle \pm \frac{(b-a)}{\sqrt{N}}\sqrt{\left\langle f(x_i)^2\right\rangle - \left\langle f(x_i)\right\rangle^2} \qquad (4.81)$$

,

where the $x_i$ are chosen randomly between $a$ and $b$, $N$ is the number of sample points, and:

$\left\langle f(x_i)\right\rangle$ = `mean(f)` and $\sqrt{\left\langle f(x_i)^2\right\rangle - \left\langle f(x_i)\right\rangle^2}$ = `std(f)`.

To scale random numbers in the range 0 to 1 to be in the range $a$ to $b$, you can use `(b - a)*rand(N,1) + a` in Matlab, where $b > a$ and $N$ is the number of points to generate.

In Python, you should use `from numpy import random` and then use `(b - a)*random.random((N,1)) + a`.

Why might you ever want to use this method? For multi-dimensional integrals in $d$ dimensions, the error for the trapezium rule falls off as $N^{(-2/d)}$, where $N$ is the number of points used/ function evaluations made. However, the $N^{(-1/2)}$ dependence of the error for the Monte Carlo integration is *independent* of the number of dimensions; hence for large $d$, the Monte Carlo method is actually more accurate than the trapezium rule for a given number of function evaluations. For Simpson's rule the error goes as $N^{(-4/d)}$ in $d$ dimensions. By using quasi-random sequences (such as Sobol sequences) it is possible to arrange for the error to drop off as $N^{(-1)}$ independent of $d$.

# 5 Non-Linear Equations

[For Python information on non-linear equations, see Chapter 16 of Prof Fangohr's notes at:

http://www.soton.ac.uk/~sesg2006/textbook/Python-for-Computational-Science-and-Engineering.pdf]

## *5.1 Introduction*

Consider comparing two algorithms which have time complexities O($N^2$) and O($N$ log $N$) respectively. Although for large enough $N$ it will be better to use the O($N$ log $N$) method, the O($N^2$) method may be faster for small $N$. Which is the fastest algorithm for a given $N$?

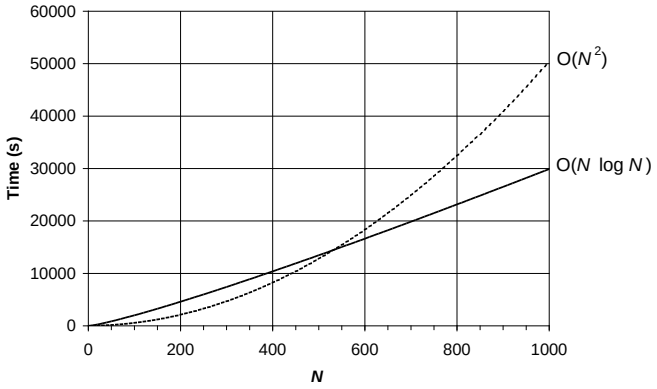



Figure 12 Comparison between two algorithms with different time complexities.

The graph in Figure 12 shows that in this particular case it is better to use the O($N^2$) method for $N$ less than about 540. The approach in the figure is to use a graphical solution method. Can we do any better? Unfortunately, there is no closed form solution to $f(N \log N) = g(N^2)$ which does not involve a numerical calculation: in this section we will consider various numerical approaches to allow us to solve equations of the form $F = 0$, where $F$ is some function. We can re-arrange the above problem into this form by writing $F = f(N \log N) - g(N^2) = 0$.

## *5.2 Bracketing a root*

Suppose that a function $f(x)$ is continuous and we have found two values $a$ and $b$ such that $f(a).f(b) < 0$. Then by the intermediate value theorem (from calculus), we know that there is at least one root in the interval $[a, b]$. The condition simply means that the function changes sign on the interval.

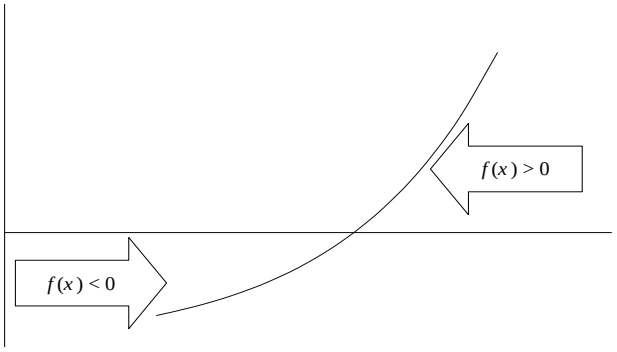



Figure 13 Sign change indicates at least one root in the region

## *5.3 Incremental Search*

This is the most primitive way to identify a root.

(i) Identify a region containing a root by proceeding from left to right over the search region checking the sign of the function. This has performed the bracketing in the previous section.

(ii) Divide the interval into smaller regions and repeat until the value of the root is found to sufficient accuracy.

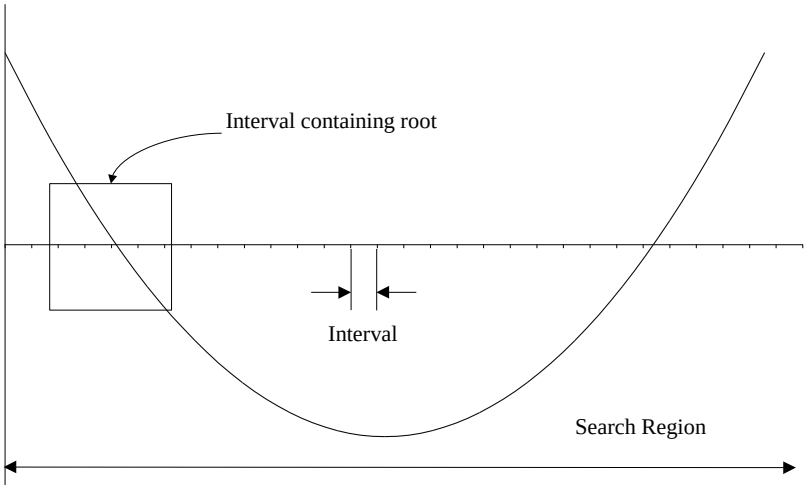



Figure 14 Incremental Search

This is very inefficient! However in the next sections we will see that the 'respectable' methods for finding zeros are just variations on this theme. The only difference is that as the sophistication of the method increases, we use more information about the function to select the next guess at the root and we try to increase the speed at which we converge to the root.

## *5.4 Bisection*

In this case we proceed thus:

(i) Bracket the root between $x_{lower}$ and $x_{upper}$.

(ii) Construct a new point midway between these two:

$$x_{new} = \frac{x_{lower} + x_{upper}}{2} \tag{5.82}$$

(iii) Replace whichever of $x_{lower}$ and $x_{upper}$ is such that $f(x_{lower})$ or $f(x_{upper})$ has the same sign as $f(x_{new})$.

(iv) Repeat until bored.

This method is bound to succeed. If the interval contains more than one root, then bisection will find one of them. It will also converge on a singularity- so beware! How fast does it converge on the root? If after $n$ iterations the interval has width $\varepsilon_n$ then

$$\varepsilon_{n+1} = \frac{\varepsilon_n}{2}. \tag{5.83}$$

So to converge to a tolerance of $\varepsilon$ we require

$$n = \log_2 \frac{\varepsilon_0}{\varepsilon} \tag{5.84}$$

steps, where $\varepsilon_0$ is the width of the initial interval. When a method converges as a factor less than 1 times the previous uncertainty to the first power then the method is said to converge linearly: this is the case for the bisection method. This means that extra significant digits are won linearly with computational effort. Superlinear convergence occurs when we can write

$$\varepsilon_{n+1} = (\text{constant}) \times (\varepsilon_n)^m, \tag{5.85}$$

with $m$>1.

## *5.5 Secant Method and Method of False Position*

The secant method and the method of false position ('regula falsi') take account of the function value to determine where to choose the next guess at the root. In both methods, we assume that the function is approximately linear in the region of interest and the next improvement is taken as the point where the approximating line crosses the axis. After each iteration one of the previous boundary points is discarded in favour of the latest estimate of the root.

### 5.5.1 Method of false position

In this case we require that the root always remains bracketed, so that we retain the prior estimate for which the function value has opposite sign from the function value at the current best estimate of the root.

Mathematically, if $a$ and $b$ were the previous brackets then we interpolate between ($a$, $f(a)$) and ($b$, $f(b)$) and construct a new point, $x_{new}$, where this line crosses the $x$-axis:

$$x_{new} = \frac{a \times f(b) - b \times f(a)}{f(b) - f(a)}. \tag{5.86}$$

We then replace $a$ or $b$ with $x_{new}$ such that $f(x_{new}) \times f(a \text{ or } b) < 0$.

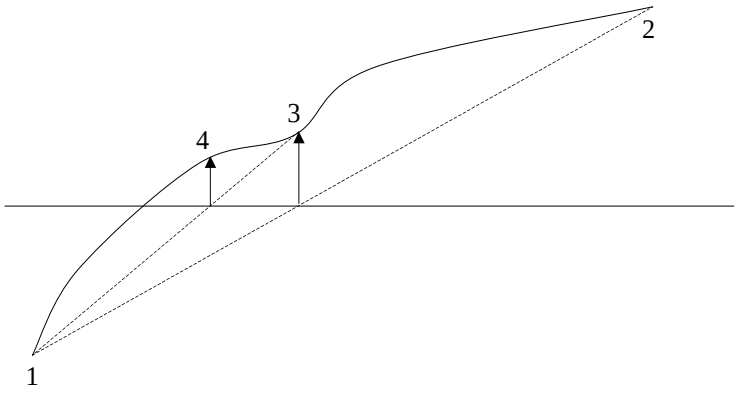



Figure 15 Method of False Position

The construction of the new point follows as follows. The equation of the line joining ($a$, $f(a)$) and ($b$, $f(b)$) is

$$y = f(a) + [f(b) - f(a)] \times \frac{(x-a)}{(b-a)}. \tag{5.87}$$

This line crosses the $x$-axis at $y = 0$:

$$\begin{aligned} 0 &= f(a) + [f(b) - f(a)] \times \frac{(x-a)}{(b-a)} \\ \Rightarrow x &= -f(a) \times \frac{(b-a)}{[f(b) - f(a)]} + a \\ \Rightarrow x &= \frac{a \times (f(b) - f(a)) - f(a) \times (b-a)}{[f(b) - f(a)]} \\ \Rightarrow x_{new} &= \frac{a \times f(b) - b \times f(a)}{[f(b) - f(a)]} \quad \text{QED.} \end{aligned} \tag{5.88}$$

The method of false position converges linearly in general, but sometimes converges superlinearly. Exact determination of its order is hard.

### 5.5.2 Secant method

The price for demanding that we always bracket the root is slower convergence. In the secant method we just interpolate between the latest two points found. However, since the root may not remain bracketed this method may not be stable and local behaviour of the function may send it towards infinity.

Given two guesses for the root, $x_n$ and $x_{n-1}$, then a better guess at the root is:

$$x_{n+1} = \frac{x_{n-1} f(x_n) - x_n f(x_{n-1})}{f(x_n) - f(x_{n-1})} \tag{5.89}$$

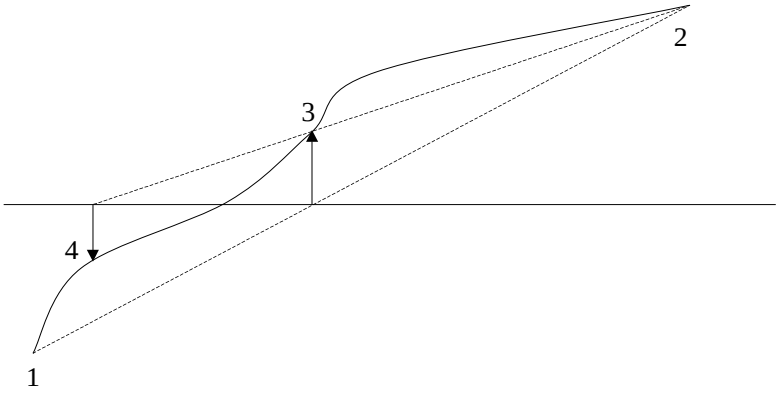



Figure 16 Secant Method

The secant method converges superlinear (for sufficiently smooth functions).

$$\lim_{n\to\infty}\left|\varepsilon_{n+1}\right| \approx \text{constant} \times \left|\varepsilon_n\right|^{1.618} \tag{5.90}$$

The number appearing in this fraction is the 'golden ratio', which was known to the ancient Pythagoreans.

### 5.5.3 Problems

The following function proves particularly troublesome to the false position (and secant) methods.

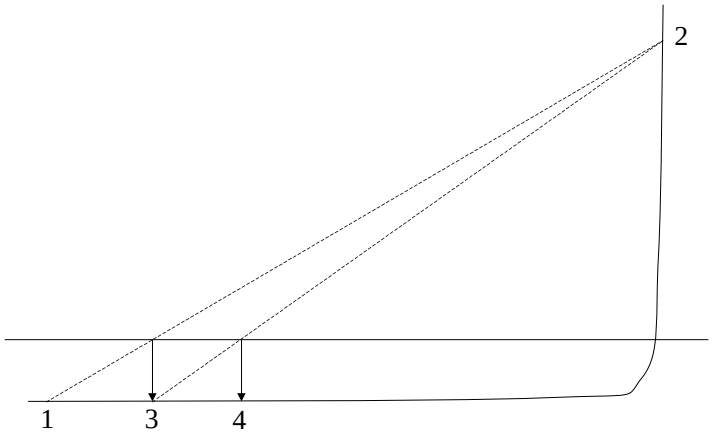



Figure 17 The method of false position (and the secant method) take many iterations to converge to the true root. The numbering shows the iterations of the method of false position.

## *5.6 Newton-Raphson Method*

By using the derivative of the function, it is possible to obtain quadratic convergence to the zero, once we are sufficiently close to it. The Newton-Raphson formula consists geometrically of extending the tangent line at a current point $x_i$ until it equals zero, and then setting the next guess, $x_{i+1}$ to the abscissa of that zero-crossing.

The iteration scheme is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{5.91}$$

It is easy to remember that the function appears on the top line, since at the root $f(x_n) = 0$, so the correction to the approximation to the root disappears. This is also a warning that if $f'(x_n)$, the value of the first derivative, is close (or equal) to zero then there will be numerical problems- in fact the convergence of Newton's method is only linear near to multiple roots.

We stop the iterations when

$$\begin{aligned} &\text{(i)} \quad |f(x_n)| < \varepsilon, \text{ or} \\ &\text{(ii)} \quad |x_{n+1} - x_n| < \varepsilon \end{aligned} \tag{5.92}$$

We may also specify a maximum number of iterations allowed and return a 'method not converged error' if this number is exceeded. It is possible to derive the following formula for the evolution of the error as the iterations progress:

$$\varepsilon_{n+1} = -\varepsilon_n^2 \frac{f''(x)}{2 \times f'(x)} \tag{5.93}$$

Near to a root the number of significant figures approximately doubles with each step.

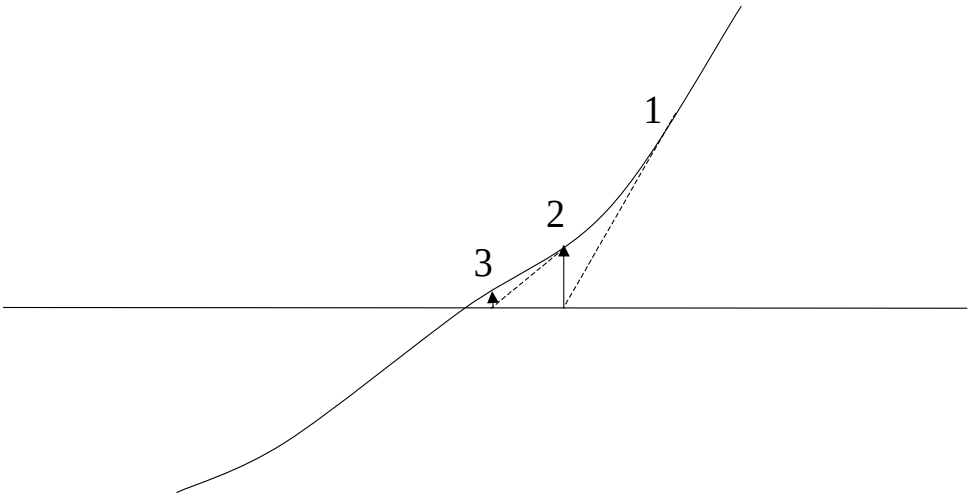



Figure 18 Newton-Raphson Iterations

## *5.7 Hybrid Methods*

Is there a way to combine the superlinear convergence of e.g. the secant method with the sureness of bisection? Yes. We keep track of whether the superlinear method is converging the way it is supposed to and, if it is not, then we can interperse some bisection steps to guarantee at least linear convergence. Such a method is the Van Wijngaarden-Dekker-Brent method, developed in the 1960s. See Press *et al* Chapter 9 for more details. There are also ways to combine the superlinear convergence of Newton's method (if the derivative can be evaluated) with the bisection method.

## *5.8 Multidimensional Methods*

Newton's method for systems of non-linear equations follows from the method for a single equation. Consider the following pair of equations:

$$\left.\begin{aligned} f_1(x_1, x_2) = 0 \\ f_2(x_1, x_2) = 0 \end{aligned}\right\} \tag{5.94}$$

Suppose that ($x_1$, $x_2$) is an approximate solution. We will now compute corrections $h_1$ and $h_2$ such that ($x_1 + h_1$, $x_2 + h_2$) is a better solution. We can expand the above equations using Taylor's theorem to give:

$$\left.\begin{aligned} f_1(x_1 + h_1, x_2 + h_2) = f_1(x_1, x_2) + h_1 \frac{\partial f_1}{\partial x_1} + h_2 \frac{\partial f_1}{\partial x_2} \\ f_2(x_1 + h_1, x_2 + h_2) = f_2(x_1, x_2) + h_1 \frac{\partial f_2}{\partial x_1} + h_2 \frac{\partial f_2}{\partial x_2} \end{aligned}\right\} \tag{5.95}$$

The partial derivatives in (5.95) are evaluated at ($x_1$, $x_2$). Equation (5.95) represents a pair of linear equations for determining $h_1$ and $h_2$:

$$\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} \bullet \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} -f_1(x_1, x_2) \\ -f_2(x_1, x_2) \end{bmatrix}, \tag{5.96}$$

where the matrix is known as the Jacobian, $J$:

$$J \bullet h = -F \tag{5.97}$$

The values of $h_1$ and $h_2$ are found by solving (5.97) as a system of equations. Hence Newton's method for two non-linear equations in 2 variables is:

$$\begin{bmatrix} x_1^{(n+1)} \\ x_2^{(n+1)} \end{bmatrix} = \begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \end{bmatrix} + \begin{bmatrix} h_1^{(n)} \\ h_2^{(n)} \end{bmatrix} \tag{5.98}$$

The algorithm for performing Newton's method for systems is as follows

1) Set $n = 0$. Pick an initial guess $(x_1^{(0)}, x_2^{(0)})$. Set $\varepsilon = 1\times10^{-6}$.
2) Calculate $F$ and $J$ at the point $(x_1^{(n)}, x_2^{(n)})$.
3) Solve the system of equations $J.h = -F$ to find the value of $h_1$ and $h_2$.
4) Set $\begin{bmatrix} x_1^{(n+1)} \\ x_2^{(n+1)} \end{bmatrix} = \begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \end{bmatrix} + \begin{bmatrix} h_1^{(n)} \\ h_2^{(n)} \end{bmatrix}$.
5) If $\| h \|_2 < \varepsilon$ then stop, otherwise set $n = n + 1$ and repeat from (2)

It is usual to specify some maximum number of iterations after which the method outputs a 'not converged' error to avoid an infinite loop in the algorithm.

Example. This example follows the steps in the algorithm above:

$$\left.\begin{aligned} f_1(x_1, x_2) &= x_1^{\,3} + 2x_2^{\,2} - 9 = 0 \\ f_2(x_1, x_2) &= x_1^{\,2} - 3x_2^{\,2} + 11 = 0 \end{aligned}\right\} \tag{5.99}$$

1) Initial Guess: $(x_1^{(0)}, x_2^{(0)}) = (2, 3)$

2) $F = \begin{bmatrix} 2^3 + 2\times 3^2 - 9 \\ 2^2 - 3\times 3^2 + 11 \end{bmatrix} = \begin{bmatrix} 17 \\ -12 \end{bmatrix}$ and $J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 3x_1^{\,2} & 4x_2 \\ 2x_1 & -6x_2 \end{bmatrix} = \begin{bmatrix} 12 & 12 \\ 4 & -18 \end{bmatrix}$

3) Solve the system of equations $J.h = -F$:

$$\begin{bmatrix} 12 & 12 \\ 4 & -18 \end{bmatrix} \bullet \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = -\begin{bmatrix} 17 \\ -12 \end{bmatrix} \text{ (note that the right hand side is } -F\text{)}$$

$$\Rightarrow \begin{bmatrix} 12 & 12 \\ 4 & -18 \end{bmatrix} \bullet \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} -17 \\ 12 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 12 & 12 \\ 4-\frac{12}{3} & -18-\frac{12}{3} \end{bmatrix} \bullet \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} -17 \\ 12+\frac{17}{3} \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 12 & 12 \\ 0 & -22 \end{bmatrix} \bullet \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} -17 \\ 17\frac{2}{3} \end{bmatrix}$$

$$\Rightarrow -22h_2 = 17\tfrac{2}{3}$$

$$\Rightarrow h_2 = -0.80\overline{30}$$

$$\Rightarrow 12h_1 + 12h_2 = -17$$

$$\Rightarrow h_1 = \frac{-17 - 12h_2}{12} = -0.61\overline{36}$$

4) Set $\begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \end{bmatrix} + \begin{bmatrix} h_1^{(0)} \\ h_2^{(0)} \end{bmatrix}$

$$\begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} + \begin{bmatrix} -0.61\overline{36} \\ -0.80\overline{30} \end{bmatrix} = \begin{bmatrix} 1.38\overline{63} \\ 2.19\overline{69} \end{bmatrix}$$

So an improved guess is (1.386, 2.197) (4 significant figures of accuracy).

5) We could then repeat again.

Here are the results for the next iteration of the loop for you to check:

$$F = \begin{bmatrix} 3.31795 \\ -1.55802 \end{bmatrix},\ J = \begin{bmatrix} 5.76601 & 8.78788 \\ 2.77273 & -13.1818 \end{bmatrix},\ \begin{bmatrix} h_1^{(2)} \\ h_2^{(2)} \end{bmatrix} = \begin{bmatrix} -0.299332 \\ -0.181158 \end{bmatrix}$$

$$\text{so } \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} = \begin{bmatrix} 1.08703 \\ 2.01581 \end{bmatrix}$$

```
x = [2;3]
F = [x(1)^3 + 2* x(2)^2 - 9; x(1)^2 - 3*x(2)^2 + 11]
J = [3*x(1)^2, 4*x(2); 2*x(1), -6*x(2)]
h = J\(-F)
for i=1:10
x =x+h
F = [x(1)^3 + 2* x(2)^2 - 9; x(1)^2 - 3*x(2)^2 + 11]
J = [3*x(1)^2, 4*x(2); 2*x(1), -6*x(2)]
h = J\(-F)
end
```

Figure 19 A simple Matlab code for 10 iterations of multidimensional Newton's method example

```
x=array([ [2],[3]])
F=array([[x[0,0]**3+2*x[1,0]**2-9],[x[0,0]**2-3*x[1,0]**2+11]])
J=array( [[3*x[0,0]**2, 4*x[1,0]],[2*x[0,0], -6*x[1,0]]])
h=solve(J,-F)

for i in range(10):
   x=x+h
   F=array([[x[0,0]**3+2*x[1,0]**2-9],[x[0,0]**2-3*x[1,0]**2+11]])
   J=array( [[3*x[0,0]**2, 4*x[1,0]],[2*x[0,0], -6*x[1,0]]])
   h=solve(J,-F)
```

Figure 20 A simple Python code for 10 iterations of multidimensional Newton's method example

The solution to the system is (1, 2). Starting sufficiently close to the solution yields rapid convergence to the solution follows from starting sufficiently close to the solution yields.

# 6 References and useful links

Whilst there is no single book which covers all of the material for the course, you should find much of it in either of the first two books.

Burden, RL and Faires, JD (2005) "Numerical Analysis." Brooks/Cole ISBN 0534404995.

Fausett L (2007) "Applied Numerical Analysis: Using Matlab" Pearson Education. ISBN 0132397285.

Press, WH, Teukolsky, SA, Vetterling, WT, and Flannery BP (1992, 1996, 2007, and later) "Numerical Recipes in C" , "Numerical Recipes in Fortran", "Numerical Recipes 3rd Edition". These books contain both the code and algorithms. See also www.nr.com for more details and you can also read these books online there too.

For more detailed mathematics and proofs of equations see

Stoer J and Bulirsch R (2010) "Introduction to Numerical Analysis" Springer. ISBN 144193006X

# Python Information

This is a good Python book on Numerical Methods with a focus on Engineering:

Jaan Kiusalaas "Numerical Methods in Engineering with Python" Hardcover: 432 pages Publisher: Cambridge University Press; 2 edition (29 Jan 2010). ISBN-10: 0521191327 and ISBN-13: 978-0521191326

For more Python examples on numerical methods, see Chapter 16 of Prof Fangohr's notes at:

http://www.southampton.ac.uk/~fangohr/training/python/pdfs/Python-for-Computational-Science-and-Engineering.pdf [last checked October 2012]

This online tutorial is useful: http://www.learnpythonthehardway.org/

If you have a Windows machine, then you can use

- Python Tools for Visual Studio at http://pytools.codeplex.com/
- and get Visual Studio through Dreamspark: http://www.dreamspark.com/

You might also consider getting yourself a Raspberry Pi. Python is one of the languages used on this: http://www.raspberrypi.org/

# Past Exam paper Questions for SESG6025 ONLY

These are the past questions from some past papers in the Uni repository which are relevant/ representative of the type of exam questions for this year for **SESG 6025**

| | |
|---|---|
| 2012: SESG6025 | 1,2,3,4 |
| 2011: SESG6025 | 1,2,3,4 |
| 2001: cm204 | A1, B2, B3 |
| 2000: cm204 | 2, 3, 4, 5(a), 5(b) |
| 1999: cm204 | A1, B1(a i), |
| 1999: cm310 | 2,3,4(a),5 |
| 1998: cm310 | 1,2,5 |

**For FEEG1001, please see FEEG1001 web pages (http://www.soton.ac.uk/~feeg1001)**
**For SESG 2006, the questions on the exam will be a programming assignment – please see past papers for SESA2006.**

# 7 Matlab Appendix

See the Matlab hand-out for notes on the use of Matlab. Other packages offering a high-level 'problem solving environment (PSE)' for computational modelling include Python (open source), Mathematica (numerical and symbolic calculations), and Maple (symbolic calculations). Spreadsheets such as Excel can also be useful for simple calculations and visualization of results.

## *7.1 MATLAB and Python features*

MATLAB and Python are versatile and interactive tools for performing numerical calculations.

Can be used for

- testing algorithms
- running small programs
- interactive visualisation of data

Other features:

- Specialist toolboxes can be used to solve particular problems
- Numerical Computation
- Interaction visualization and presentation graphics
- High Level Programming Language based on vectors and matrices
- Specialist toolboxes written by experts
- Tools for interface building
- Integrated debugger, editor and performance profiler
- On-line electronic documentation

## *7.2 Availability of MATLAB/ Python*

The Matlab User's guide, software, and thousands of pages of online documentation is available in a student version. It is on iSolutions PC Clusters and the major University High Performance Computing (HPC) facilities. Python is available as Open Source.

### 7.2.1 Matlab Features

| Key Features | Additional Features | Extras |
|---|---|---|
| Basic Mathematics (1.2) | Handling arrays (1.6) | Complex Numbers (1.3) |
| "Help" (1.4)<br>Array operations (1.5) | Relational Operators (1.8) | Linear Algebra (1.9.2 – 1.9.5) |
| 2D plotting (1.7) | Linear Algebra (1.9.1) | Special Matrices (1.10) |
| 3D plotting (1.18) | Text Handling (1.11) | Polynomials (1.14) |
| Programming (1.12 and 1.13) | Curve fitting & interpolation (1.15.1-1.15.6) | Numerical Analysis (1.15.7 – 1.15.10) |
| | Sparse Matrices & Optimisation tips (1.17) | Data Analysis example (1.16) |

## *7.3 Basic Features*

- Mathematical calculations can be typed in as you would write them
- Variables are defined using '`a = 3`'
- Once a variable is defined, it can be used in mathematical expressions
- The commands `who` and `whos` display information about variables

## *7.4 Help*

- Typing `help <command>` gives a summary of the command
- `lookfor` allows you to search for a keyword
- Under Windows, `help` can be accessed interactively and online

## *7.5 Array Operations*

- Commas separate columns of a matrix : "`,`"
- Use a semicolon: "`;`" to start a new row
- To index individual array elements use

  e.g. `x(1), x(3)`

## *7.6 Colon Notation*

The colon can be used in several ways

- To index an array:

```
x(start element: step : last)
```

e.g. `x(1:2:5)` returns elements `x(1), x(3), x(5)`

- To construct an array:

```
x=(first number : step : last)
```

e.g. `x=(2:2:6)` is the same as `x=[2,4,6]`

## *7.7 Array Mathematics*

Array Manipulation

- Elementwise:

`a.*b` gives $a_{1,1}\times b_{1,1}$ and $a_{1,2}\times b_{1,2}$ etc.

- Conventional linear algebra (the dimensions of the matrices must be compatible)

`a*b` means use $a_{1,1}\times b_{1,1} + a_{1,2}\times b_{2,1} + a_{1,3}\times b_{3,1}$ etc.

## *7.8 2D Plotting*

- `plot(x1,y1,s1, x2,y2,s2, x3,y3,s3,...)` places plots of the vectors `(x1,y1)` with style `s1` and `(x2,y2)` with style `s2` etc. on the same axes.
- `xlabel('text')` adds a label to the x axis.
- `ylabel('text')` add a label to the y axis.
- `grid on` turns on a grid over the plot.

## *7.9 Other plots*

- `semilogy(x,y)` and `semilogx(x,y)` gives axes marked in powers of 10.
- `loglog(x,y)` plots both axes with a logarithmic scale.
- MATLAB supports many common types of plot (see the booklet).
- `fill(x,y,s)` draws a fills a polygon with the colour `r`.

## *7.10 3D Plotting*

- `plot3(x1,y1,z1,s1, x2,y2,s2,z2, .. )` plots the points defined by the triples `(x1,y1,z1)` with style `s1` and `(x2,y2,z2)` with style `s2` etc. on the same axes.
- `zlabel('text')` adds a label to the z axis.
- `mesh(x,y,z)` draws a wire frame grid for the surface defined by `(x,y,z)`.
- `surf(x,y,z)` gives a shaded surface plot for the surface defined by `(x,y,z)`.

## *7.11 3D plotting*

- Change the shading using `colormap(map)`
- To examine a coloured map of a matrix, A, use `imagesc(A)`
- `colorbar` displays the colour coding for the matrix shading

## *7.12 Programming MATLAB*

- MATLAB provides loops using

```
for k = 1:n
[instructions]
end
```

- MATLAB commands can be put together in a script (or text) file to group together a set of instructions.
- MATLAB also provides tools to build user-friendly interfaces for programs.