Numerical Methods
Natural Sciences Tripos 1B
Lecture Notes
Lent Term 1999

© Stuart Dalziel

Department of Applied Mathematics and Theoretical Physics
University of Cambridge

Phone: (01223) 337911
E-mail: *s.dalziel@damtp.cam.ac.uk*
WWW: `http://www.damtp.cam.ac.uk/user/fdl/people/sd103/`
Lecture Notes: `http://www.damtp.cam.ac.uk/user/fdl/people/sd103/lectures/`

Formatting and visibility in this version:

Subsections

Sub-subsections

Fourth order subsections

All versions

Full text

Common equations

Handout/lecture equations

Handout/lecture figures

Tables

CONTENTS

- Page numbers are correct only for the hand-out with which they are given

1 Introduction

These lecture notes are written for the Numerical Methods course as part of the Natural Sciences Tripos, Part IB. The notes are intended to compliment the material presented in the lectures rather than replace them.

1.1 Objective

- To give an overview of *what* can be done
- To give insight into *how* it can be done
- To give the confidence to tackle numerical solutions

An understanding of how a method works aids in choosing a method. It can also provide an indication of what can and will go wrong, and of the accuracy which may be obtained.

- To gain insight into the underlying physics
- "*The aim of this course is to introduce numerical techniques that can be used on computers, rather than to provide a detailed treatment of accuracy or stability*" – Lecture Schedule.

Unfortunately the course is now examinable and therefore the material must be presented in a manner consistent with this.

1.2 Books

General:

- *Numerical Recipes - The Art of Scientific Computing*, by Press, Flannery, Teukolsky & Vetterling (CUP)
- *Numerical Methods that Work*, by Acton (Harper & Row)
- *Numerical Analysis*, by Burden & Faires (PWS-Kent)
- *Applied Numerical Analysis*, by Gerald & Wheatley (Addison-Wesley)
- *A Simple Introduction to Numerical Analysis*, by Harding & Quinney (Institute of Physics Publishing)
- *Elementary Numerical Analysis*, 3rd Edition, by Conte & de Boor (McGraw-Hill)

More specialised:

- *Numerical Methods for Ordinary Differential Systems*, by Lambert (Wiley)
- *Numerical Solution of Partial Differential Equations: Finite Difference Methods*, by Smith (Oxford University Press)

For many people, Numerical Recipes is the *bible* for simple numerical techniques. It contains not only detailed discussion of the algorithms and their use, but also sample source code for each.

Numerical Recipes is available for three tastes: Fortran, C and Pascal, with the source code examples being taylored for each.

1.3 Programming

While a number of programming examples are given during the course, the course and examination do **not** require any knowledge of programming. Numerical results are given to illustrate a point and the code used to compute them presented in these notes purely for completeness.

1.4 Tools

Unfortunately this course is too short to be able to provide an introduction to the various tools available to assist with the solution of a wide range of mathematical problems. These tools are widely available on nearly all computer platforms and fall into two general classes:

1.4.1 Software libraries

These are intended to be linked into your own computer program and provide routines for solving particular classes of problems.

- NAG
- IMFL
- Numerical Recipes

The first two are commercial packages providing object libraries, while the final of these libraries mirrors the content of the Numerical Recipes book and is available as source code.

1.4.2 Maths systems

These provide a *shrink-wrapped* solution to a broad class of mathematical problems. Typically they have easy-to-use interfaces and provide graphical as well as text or numeric output. Key features include algebraic analytical solution. There is fierce competition between the various products available and, as a result, development continues at a rapid rate.

- Derive
- Maple
- Mathcad
- Mathematica
- Matlab
- Reduce

1.5 Course Credit

Prior to the 1995-1996 academic year, this course was not examinable. Since then, however, there have been two examination questions each year. Some indication of the type of exam questions may be gained from earlier tripos papers and from the later examples sheets. Note that there has, unfortunately, been a tendency to concentrate on the more analysis side of the course in the examination questions.

Some of the topics covered in these notes are not examinable. This situation is indicated by an asterisk at the end of the section heading.

1.6 Versions

These lecture notes are available in three forms: the lecture notes distributed during lectures, and the set available in two formats on the web.

1.6.1 Notes disctributed during lectures

The version distributed during lectures includes *blanks* for you to fill in the missing details. These details will be given during the lectures themselves.

1.6.2 Acrobat

The lecture notes are also available over the web. This year's notes will be provided in Acrobat format (pdf) and may be found at

`http://www.damtp.cam.ac.uk/user/fdl/people/sd103/lectures/`

These notes contain all the information, and any *blanks* have been filled in.

1.6.3 HTML

In previous years these have been provided through an html format, and these notes remain available, although may not contain the latest revisions. The HTML version of the notes also has all the blanks filled in.

The HTML is generated from a source Word document that contains graphics, display equations and inline equations and symbols. All graphics and complex display equations (where the Microsoft Equation Editor has been used) are converted to GIF files for the HTML version. However, many of the simpler equations and most of the inline equations and symbols do not use the Equation Editor as this is very inefficient. As a consequence, they appear as characters rather than GIF files in the HTML document. This has major advantages in terms of document size, but can cause problems with older World Wide Web browsers.

Due to limitations in HTML and many older World Wide Web browsers, Greek and Symbols used within the text and single line equations may not be displayed correctly. Similarly, some browsers do not handle superscript and subscript. To avoid confusion when using older browsers, all Greek and Symbols are formatted in **Green**. Thus if you find a green Roman character, read it as the Greek equivalent. Table 1of the correspondences is given below. Variables and normal symbols

are treated in a similar way but are coloured dark **Blue** to distinguish them from the Greek. The context and colour should distinguish them from HTML hypertext links. Similarly, subscripts are shown in dark **Cyan** and superscripts in dark **Magenta**. Greek subscripts and superscripts are the same **Green** as the normal characters, the context providing the key to whether it is a subscript or superscript. For a similar reason, the use of some mathematical symbols (such as less than or equal to) has been avoided and their Basic computer equivalent used in stead.

Fortunately many newer browsers (Microsoft Internet Explorer 3.0 and Netscape 3.0 on the PC, but on many Unix platforms the Greek and Symbol characters are unavailable) do not have the same character set limitations. The colour is still displayed, but the characters appear as intended.

Greek/Symbol character	Name
α	alpha
β	beta
δ	delta
Δ	Delta
ε	epsilon
φ	phi
Φ	Phi
λ	lambda
μ	mu
π	pi
θ	theta
σ	sigma
ψ	psi
Ψ	Psi
<=	less than or equal to
>=	greater than or equal to
<>	not equal to
=~	approximately equal to
vector	vectors are represented as bold

Table 1: Correspondence between colour and characters.

1.6.4 Copyright

2 Key Idea

The central idea behind the majority of methods discussed in this course is the Taylor Series expansion of a function about a point. For a function of a single variable, we may represent the expansion as

$$f(x + \delta x) = \tag{1}$$

In two dimensions we have

$$f(x + \delta x, y + \delta y) = \tag{2}$$

Similar expansions may be constructed for functions with more independent variables.

3 Root finding in one dimension

3.1 Why?

Solutions $\boldsymbol{x} = \boldsymbol{x}_0$ to equations of the form $f(\boldsymbol{x}) = 0$ are often required where it is impossible or infeasible to find an analytical expression for the vector $\boldsymbol{x}$. If the scalar function f depends on m independent variables $x_1,x_2,\ldots,x_m$, then the solution $\boldsymbol{x}_0$ will describe a surface in $m–1$ dimensional space. Alternatively we may consider the vector function $\boldsymbol{f}(\boldsymbol{x})=\boldsymbol{0}$, the solutions of which typically collapse to particular values of $\boldsymbol{x}$. For this course we restrict our attention to a single independent variable x and seek solutions to $f(x)=0$.

3.2 Bisection

This is the simplest method for finding a root to an equation and is also known as *binary chopping*. As we shall see, it is also the most robust. One of the main drawbacks is that we need two initial guesses x_a and x_b which bracket the root: let $f_a = f(x_a)$ and $f_b = f(x_b)$ such that $f_a\, f_b <= 0$. An example of this is shown graphically in figure 1. Clearly, if $f_a\, f_b = 0$ then one or both of x_a and x_b must be a root of $f(x) = 0$.

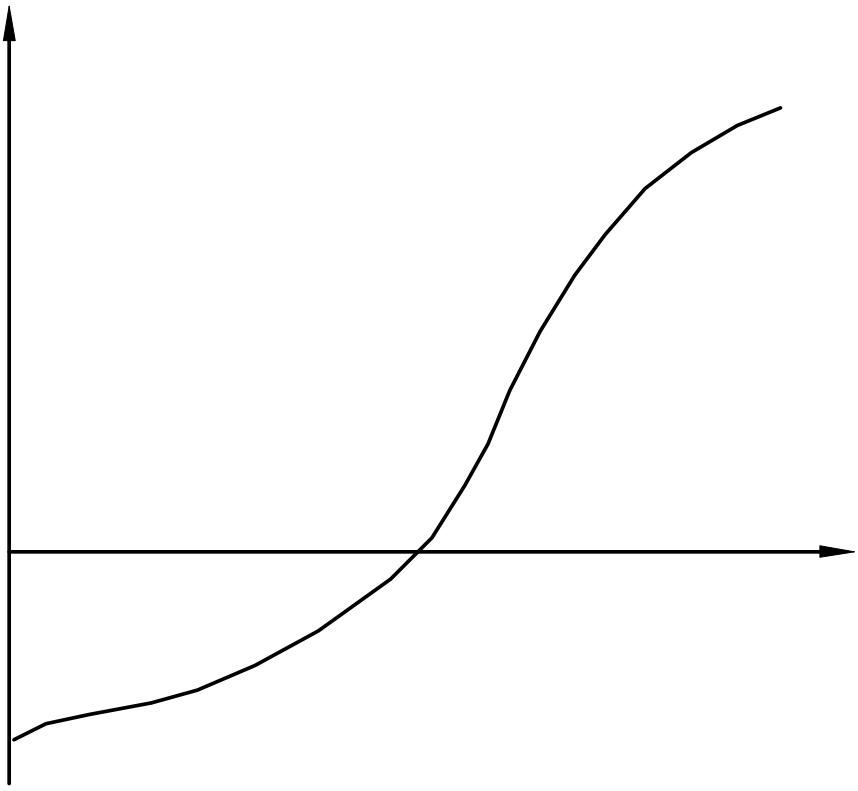


Figure 1: Graphical representation of the bisection method showing two initial guesses (x_a and x_b bracketting the root).

The basic algorithm for the bisection method relies on repeated application of

- Let $x_c = (x_a+x_b)/2$,

- if $f_c = f(c) = 0$ then $x = x_c$ is an exact solution,
- elseif $f_a\, f_c < 0$ then the root lies in the interval (x_a,x_c),
- else the root lies in the interval (x_c,x_b).

By replacing the interval (x_a,x_b) with either (x_a,x_c) or (x_c,x_b) (whichever brackets the root), the error in our estimate of the solution to $f(x) = 0$ is, on average, halved. We repeat this interval halving until either the exact root has been found or the interval is smaller than some specified tolerance.

3.2.1 Convergence

Since the interval (x_a,x_b) always bracets the root, we know that the error in using either x_a or x_b as an estimate for root at the nth iteration, e_n, must be

$$e_n < |x_a - x_b|. \tag{3}$$

Now since the interval (x_a,x_b) is halved for each iteration, then

$$e_{n+1} \sim \tag{4}$$

More generally, if x_n is the estimate for the root x^* at the nth iteration, then the error in this estimate is

$$\varepsilon_n = \tag{5}$$

In many cases we may express the error at the $n+1$th time step in terms of the error at the nth time step as

$$|\varepsilon_{n+1}| \sim \tag{6}$$

Indeed this criteria applies to all techniques discussed in this course, but in many cases it applies only asymptotically as our estimate x_n converges on the exact solution. The exponent p in equation (6) gives the order of the convergence. The larger the value of p, the faster the scheme converges on the solution, at least provided $\varepsilon_{n+1} < \varepsilon_n$. For first order schemes (*i.e.* $p = 1$), $|C| < 1$ for convergence.

For the bisection method we may estimate ε_n as e_n. The form of equation (4) then suggests $p = 1$ and $C = 1/2$, showing the scheme is first order and converges linearly. Indeed convergence is guaranteed - a root to $f(x) = 0$ will *always* be found - provided $f(x)$ is continuous over the initial interval.

3.2.2 Criteria

In general, a numerical root finding procedure will not find the exact root being sought ($\varepsilon = 0$), rather it will find some suitably accurate approximation to it. In order to prevent the algorithm continuing to refine the solution for ever, it is necessary to place some conditions under which the solution process is to be finished or aborted. Typically this will take the form of an error tolerance on $e_n = |a_n – b_n|$, the value of f_c, or both.

For some methods it is also important to ensure the algorithm is converging on a solution (*i.e.* $|\varepsilon_{n+1}| < |\varepsilon_n|$ for suitably large n), and that this convergence is sufficiently rapid to attain the solution in a reasonable span of time. The guaranteed convergence of the bisection method does not require

such safety checks which, combined with its extreme simplicity, is one of the reasons for its widespread use despite being relatively slow to converge.

3.3 Linear interpolation (regula falsi)

This method is similar to the bisection method in that it requires two initial guesses to bracket the root. However, instead of simply dividing the region in two, a linear interpolation is used to obtain a new point which is (hopefully, but not necessarily) closer to the root than the equivalent estimate for the bisection method. A graphical interpretation of this method is shown in figure 2.

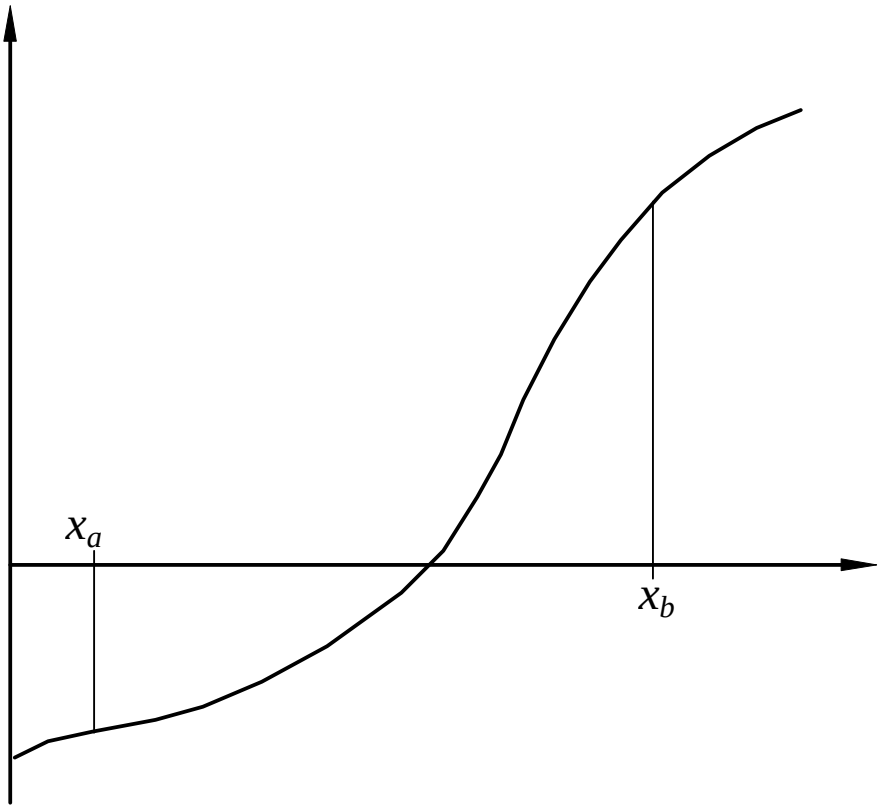


Figure 2: Root finding by the linear interpolation (regula falsi) method. The two initial gueses x_a and x_b must bracket the root.

The basic algorithm for the linear interpolation method is

- Let $x_c = x_a - \frac{x_b - x_a}{f_b - f_a} f_a = x_b - \frac{x_b - x_a}{f_b - f_a} f_b = \frac{x_a f_b - x_b f_a}{f_b - f_a}$, then
- if $f_c = f(x_c) = 0$ then $x = x_c$ is an exact solution,
- elseif $f_a f_c < 0$ then the root lies in the interval (x_a,x_c),
- else the root lies in the interval (x_c,x_b).

Because the solution remains bracketed at each step, convergence is guaranteed as was the case for the bisection method. The method is first order and is exact for linear f.

3.4 Newton-Raphson

Consider the Taylor Series expansion of $f(x)$ about some point $x = x_0$:

$f(x) =$ (7)

Setting the quadratic and higher terms to zero and solving the linear approximation of $f(x) = 0$ for x gives

$x_1 =$ (8)

Subsequent iterations are defined in a similar manner as

$x_{n+1} =$ (9)

Geometrically, x_{n+1} can be interpreted as the value of x at which a line, passing through the point $(x_n, f(x_n))$ and tangent to the curve $f(x)$ at that point, crosses the y axis. Figure 3 provides a graphical interpretation of this.

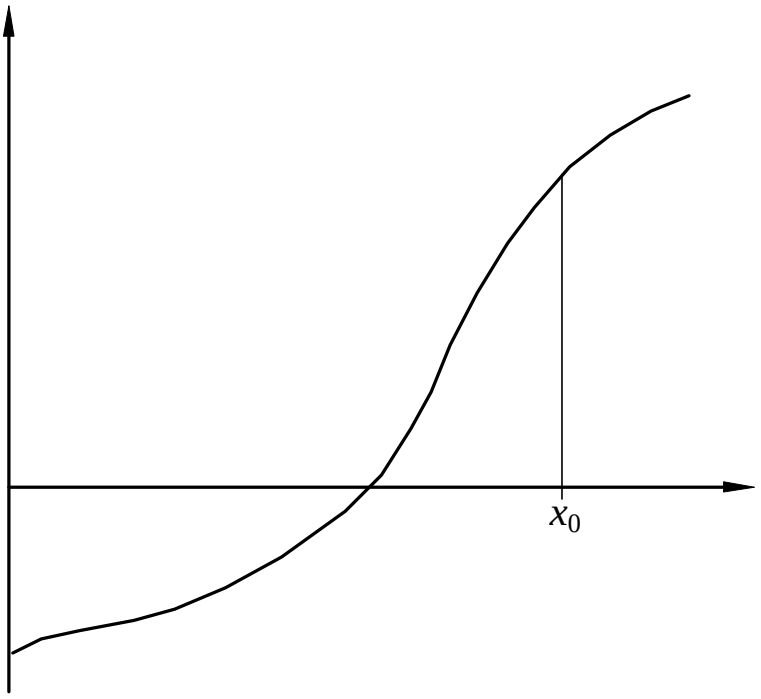


Figure 3: Graphical interpretation of the Newton Raphson algorithm.

When it works, Newton-Raphson converges much more rapidly than the bisection or linear interpolation. However, if f' vanishes at an iteration point, or indeed even between the current estimate and the root, then the method will fail to converge. A graphical interpretation of this is given in figure 4.

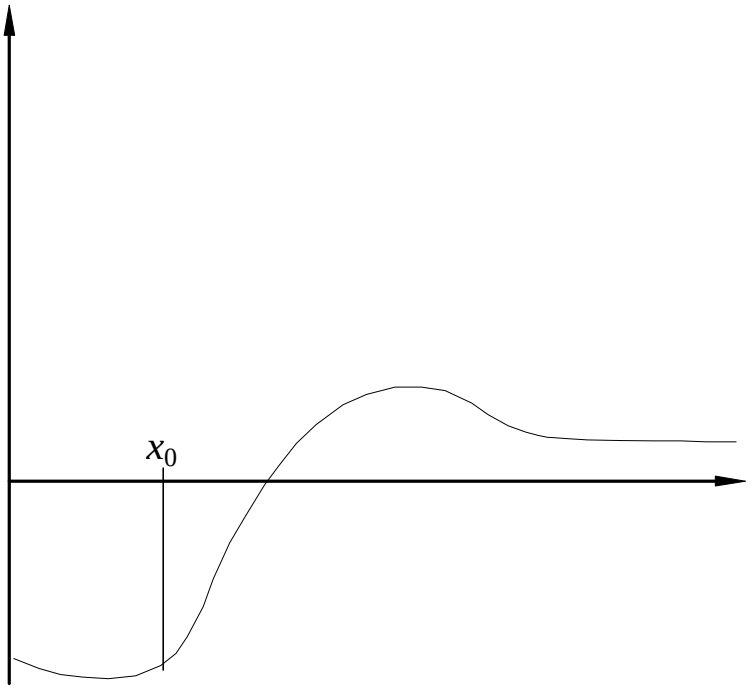


Figure 4: Divergence of the Newton Raphson algorithm due to the presence of a turning point close to the root.

3.4.1 Convergence

To study how the Newton-Raphson scheme converges, expand $f(x)$ around the root $x = x^*$,

$f(x) =$ (10)

and substitute into the iteration formula. This then shows

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

(11)

since $f(x^*)=0$. Thus, by comparison with (5), there is second order (quadratic) convergence. The presence of the f' term in the denominator shows that the scheme will not converge if f' vanishes in the neighbourhood of the root.

3.5 Secant (chord)

This method is essentially the same as Newton-Raphson except that the derivative $f'(x)$ is approximated by a finite difference based on the current and the preceding estimate for the root, *i.e.*

$f'(x_n) \approx$ (12)

and this is substituted into the Newton-Raphson algorithm (9) to give

$x_{n+1} =$ (13)

This formula is identical to that for the Linear Interpolation method discussed in section 3.3. The difference is that rather than replacing one of the two estimates so that the root is always bracketed, the oldest point is always discarded in favour of the new. This means it is not necessary to have two initial guesses bracketing the root, but on the other hand, convergence is not guaranteed. A graphical representation of the method working is shown in figure 5 and failure to converge in figure 6. In some cases, swapping the two initial guesses x_0 and x_1 will change the behaviour of the method from convergent to divergent.

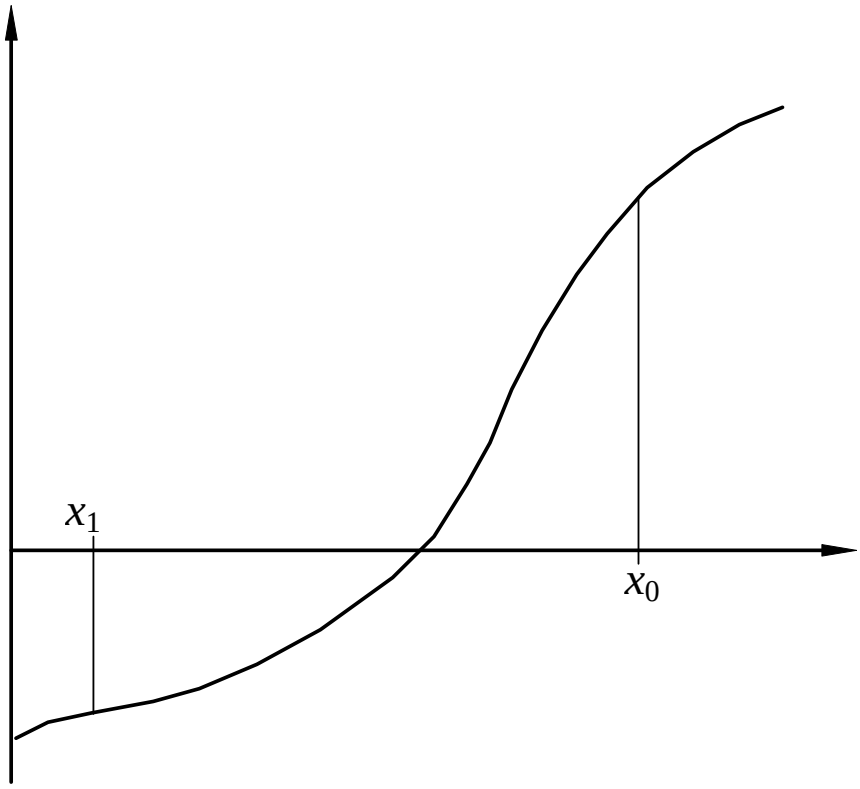


Figure 5: Convergence on the root using the secant method.

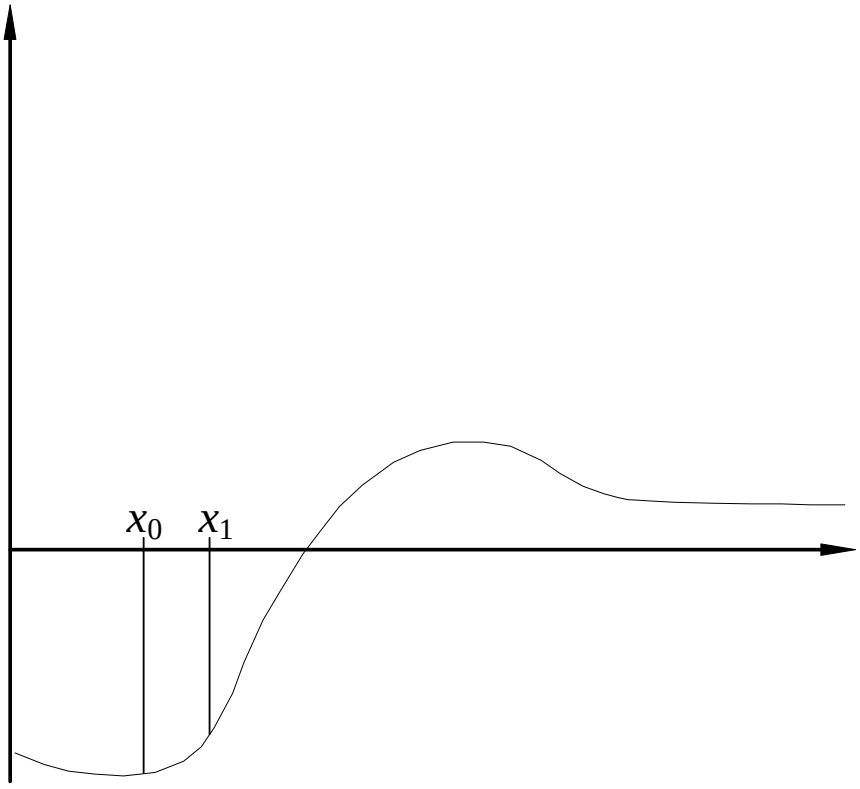


Figure 6: Divergence using the secant method.

3.5.1 Convergence

The order of convergence may be obtained in a similar way to the earlier methods. Expanding around the root $x = x^*$ for x_n and x_{n+1} gives

$f(x_n) =$ (14a)

$f(x_{n-1}) =$ (14b)

and substituting into the iteration formula

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

(15)

Note that this expression for ε_{n+1} includes both ε_n and ε_{n-1}. In general we would like it in terms of ε_n only. The form of this expression suggests a power law relationship. By writing

$\varepsilon_{n+1} =$ (16)

and substituting into the error evolution equation (15) gives

$$\varepsilon_{n+1} = \left(\frac{f''(x^*)}{2f'(x^*)}\right)\varepsilon_n\varepsilon_{n-1}$$
$$=$$

(17)

which we equate with our assumed relationship to show

$$\alpha = \frac{1+\alpha}{\alpha} =$$
$$\beta = \frac{\alpha}{1+\alpha} = \qquad (18)$$

Thus the method is of non-integer order 1.61803… (the golden ratio). As with Newton-Raphson, the method may diverge if *f'* vanishes in the neighbourhood of the root.

3.6 Direct iteration

A simple and often useful method involves rearranging and possibly transforming the function $f(x)$ by $T(f(x),x)$ to obtain $g(x) = T(f(x),x)$. The only restriction on $T(f(x),x)$ is that solutions to $f(x) = 0$ have a one to one relationship with solutions to $g(x) = x$ for the roots being sort. Indeed, one reason for choosing such a transformation for an equation with multiple roots is to eliminate known roots and thus simplify the location of the remaining roots. The efficiency and convergence of this method depends on the final form of $g(x)$.

The iteration formula for this method is then just

$$x_{n+1} = \qquad (19)$$

A graphical interpretion of this formula is given in figure 7.

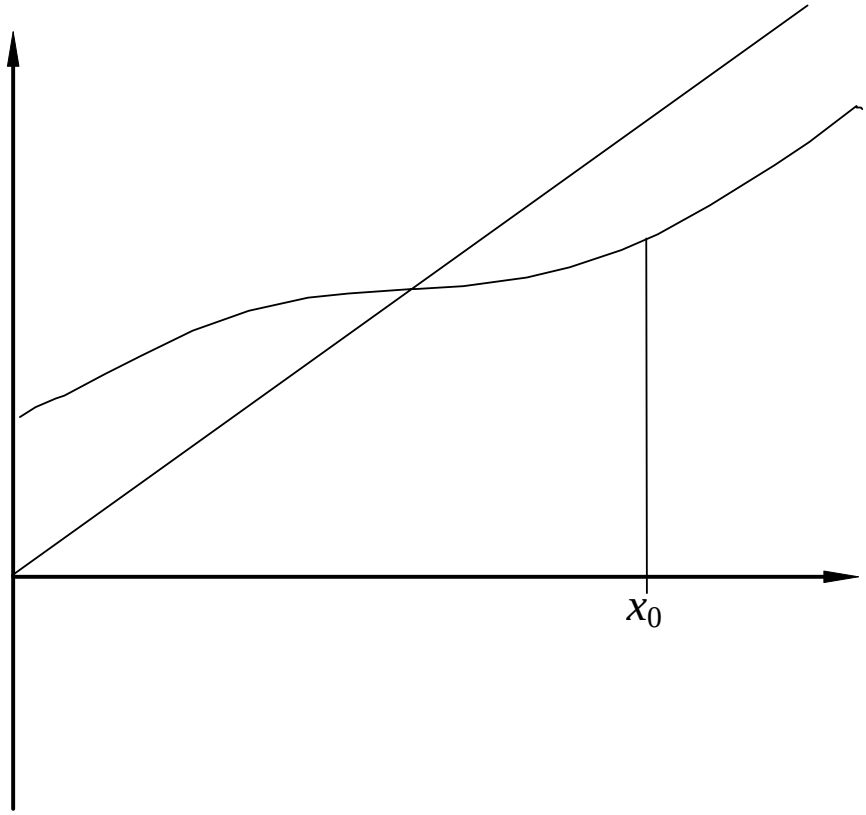


Figure 7: Convergence on a root using the Direct Iteration method.

3.6.1 Convergence

The convergence of this method may be determined in a similar manner to the other methods by expanding about x^*. Here we need to expand $g(x)$ rather than $f(x)$. This gives

$$g(x_n) = \tag{20}$$

so that the evolution of the error follows

$$\varepsilon_{n+1} = x_{n+1} - x$$

(21)

The method is clearly first order and will converge only if $|g'| < 1$. The sign of g' determines whether the convergence (or divergence) is monotonic (positive g') or oscillatory (negative g'). Figure 8 shows how the method will diverge if this restriction on g' is not satisfied. Here $g' < -1$ so the divergence is oscilatory.

Obviously our choice of $T(f(x),x)$ should try to minimise $g'(x)$ in the neighbourhood of the root to maximise the rate of convergence. In addition, we should choose $T(f(x),x)$ so that the curvature $|g''(x)|$ does not become too large.

If $g'(x) < 0$, then we get oscillatory convergence/divergence.

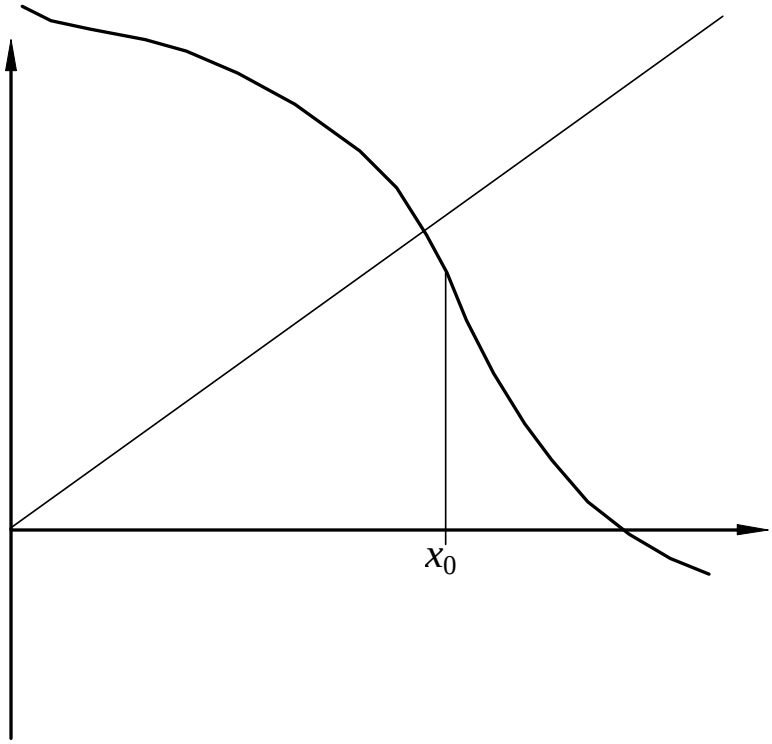


Figure 8: The divergence of a Direct Iteration when $g' < -1$.

3.7 Examples

Consider the equation

$$f(x) = \cos x - 1/2. \tag{22}$$

3.7.1 Bisection method

- Initial guesses $x = 0$ and $x = \pi/2$.
- Expect linear convergence: $|\varepsilon_{n+1}| \sim |\varepsilon_n|/2$.

Iteration	Error	e_{n+1}/e_n
0	-0.261799	-0.500001909862
1	0.130900	-0.4999984721161
2	-0.0654498	-0.5000015278886
3	0.0327250	-0.4999969442322
4	-0.0163624	-0.5000036669437
5	0.00818126	-0.4999951107776
6	-0.00409059	-0.5000110008581
7	0.00204534	-0.4999755541866
8	-0.00102262	-0.5000449824959

9	0.000511356	-0.4999139542706
10	-0.000255634	-0.5001721210794
11	0.000127861	-0.4996574405018
12	-0.0000638867	-0.5006848060707
13	0.0000319871	-0.4986322611303
14	-0.0000159498	-0.5027411002019
15	0.00000801862	

3.7.2 Linear interpolation

- Initial guesses $x = 0$ and $x = \pi/2$.
- Expect linear convergence: $|\varepsilon_{n+1}| \sim c|\varepsilon_n|$.

Iteration	Error	e_{n+1}/e_n
0	-0.261799	0.1213205550823
1	-0.0317616	0.0963178807113
2	-0.00305921	0.09340810209172
3	-0.000285755	0.09312907910623
4	-0.0000266121	0.09310313729469
5	-0.00000247767	0.09310037252741
6	-0.000000230672	0.09310059304987
7	-0.0000000214757	0.09310010849472
8	-0.00000000199939	0.09310039562066
9	-0.000000000186144	0.09310104005501
10	-0.0000000000173302	0.09310567679542
11	-0.00000000000161354	0.09316100003719
12	-0.000000000000150319	0.09374663216227
13	-0.0000000000000140919	0.10000070962752
14	-0.0000000000000014092	0.1620777746239
15	-0.0000000000000002284	

3.7.3 Newton-Raphson

- Initial guess: $x = \pi/2$.
- Note that can not use $x = 0$ as derivative vanishes here.
- Expect quadratic convergence: $\varepsilon_{n+1} \sim c\varepsilon_n^{\,2}$.

Iteration	Error	e_{n+1}/e_n	$e_{n+1}/e_n^{\,2}$
0	0.0235988	0.00653855280777	0.2770714107399
1	0.000154302	0.0000445311143083	0.2885971297087
2	0.00000000687124	0.000000014553	-
3	1.0E-15		
4	*Machine accuracy*		

3.7.4 Secant method

- Initial guesses $x = 0$ and $x = \pi/2$.
- Expect convergence: $\varepsilon_{n+1} \sim c\varepsilon_n^{\,1.618}$.

Iteration	Error	e_{n+1}/e_n	$\lvert e_{n+1}\rvert/\lvert e_n\rvert^{1.618}$
0	-0.261799	0.1213205550823	0.2777
1	-0.0317616	-0.09730712558561	0.8203
2	0.00309063	-0.009399086917554	0.3344
3	-0.0000290491	0.0008898244696049	0.5664
4	-0.0000000258486	-0.000008384051747483	0.4098
5	0.000000000000216716		
6	*Machine accuracy*		

- Convergence substantially faster than linear interpolation.

3.7.5 Direct iteration

There are a variety of ways in which equation (22) may be rearranged into the form required for direct iteration.

3.7.5.1 Addition of x

Use

$$x_{n+1} = g(x) = x_n + \cos x - 1/2 \tag{23}$$

- Initial guess: $x = 0$ (also works with $x = \pi/2$)
- Expect convergence: $\varepsilon_{n+1} \sim g'(x^*)\,\varepsilon_n \sim 0.13\,\varepsilon_n$.

Iteration	Error	e_{n+1}/e_n
0	-0.547198	0.30997006568
1	-0.169615	0.1804233116175
2	-0.0306025	0.1417596601585
3	-0.00433820	0.1350620072841
4	-0.000585926	0.1341210323488
5	-0.0000785850	0.1339937647134
6	-0.0000105299	0.1339775306508
7	-0.00000141077	0.1339750632633
8	-0.000000189008	0.1339747523914
9	-0.0000000253223	0.1339747969181
10	-0.00000000339255	0.1339744440023
11	-0.000000000454515	0.1339748963181
12	-0.0000000000608936	0.1339759843399
13	-0.00000000000815828	0.1339878013503
14	-0.00000000000109311	0.1340617138257
15	-0.000000000001465442	

3.7.5.2 Multiplcation by x

Use

$$x_{n+1} = g(x) = 2x \cos x \tag{24}$$

- Initial guess: $x = \pi/2$ (fails with $x = 0$ as this is a new solution to $g(x)=x$)

- Expect convergence: $\varepsilon_{n+1} \sim g'(x^*)\,\varepsilon_n \sim 0.81\,\varepsilon_n$.

Iteration	Error	e_{n+1}/e_n
0	0.0635232	-0.9577980958138
1	-0.0608424	-0.6773664418235
2	0.0412126	-0.9070721090152
3	-0.0373828	-0.7297714456916
4	0.0272809	-0.8754733164962
5	-0.0238837	-0.7600455540808
6	0.0181527	-0.854809477378
7	-0.0155171	-0.778843985023
8	0.0120854	-0.8410892481838
9	-0.0101649	-0.7908921878228
10	0.00803934	-0.8319464035605
11	-0.00668830	-0.7987216482514
12	0.00534209	-0.8258546748557
13	-0.00441179	-0.8038528579103
14	0.00354643	-0.8218010788314
15	-0.00291446	

3.7.5.3 Approximating $f'(x)$

The Direct Iteration method is closely related to the Newton Raphson method when a particular choice of transformation $T(f(x))$ is made. Consider

$$f(x) = f(x) + (x - x)h(x) = 0. \tag{25}$$

Rearranging equation (25) for one of the x variables and labelling the different variables for different steps in the interation gives

$$x_{n+1} = g(x_n) = x_n - f(x_n)/h(x_n). \tag{26}$$

Now if we choose $h(x)$ such that $g'(x)=0$ everywhere (which requires $h(x) = f'(x)$), then we recover the Newton-Raphson method with its quadratic convergence.

In some situations calculation of $f'(x)$ may not be feasible. In such cases it may be necessary to rely on the first order and secant methods which do not require a knowledge of $f'(x)$. However, the convergence of such methods is very slow. The Direct Iteration method, on the otherhand, provides us with a framework for a faster method. To do this we select $h(x)$ as an approximation to $f'(x)$. For the present $f(x) = \cos x - 1/2$ we may approximate $f'(x)$ as

$$h(x) = 4x(x - \pi)/\pi^2 \tag{27}$$

- Initial guess: $x = 0$ (fails with $x = \pi/2$ as $h(x)$ vanishes).
- Expect convergence: $\varepsilon_{n+1} \sim g'(x^*)\,\varepsilon_n \sim 0.026\,\varepsilon_n$.

Iteration	Error	e_{n+1}/e_n
0	0.0235988	0.02985973863078
1	0.000704654	0.02585084310882
2	0.0000182159	0.02572477890195
3	0.000000468600	0.02572151088348
4	0.0000000120531	0.02572134969427
5	0.000000000310022	0.02572107785899
6	0.00000000000797410	0.02570835580191

7	0.000000000000205001	0.02521207213623
8	0.0000000000000516850	
9	*Machine accuracy*	

The convergence, while still formally linear, is significantly more rapid than with the other first order methods. For a more complex example, the computational cost of having more iterations than Newton Raphson may be significantly less than the cost of evaluating the derivative.

A further potential use of this approach is to avoid the divergence problems associated with $f'(x)$ vanishing in the Newton Raphson scheme. Since $h(x)$ only approximates $f'(x)$, and the accuracy of this approximation is more important close to the root, it may be possible to choose $h(x)$ in such a way as to avoid a divergent scheme.

3.7.6 Comparison

Figure 9 shows graphicall a comparison between the different approaches to finding the roots of equation (22). The clear winner is the Newton-Raphson scheme, with the approximated derivative for the Direct Iteration proving a very good alternative.

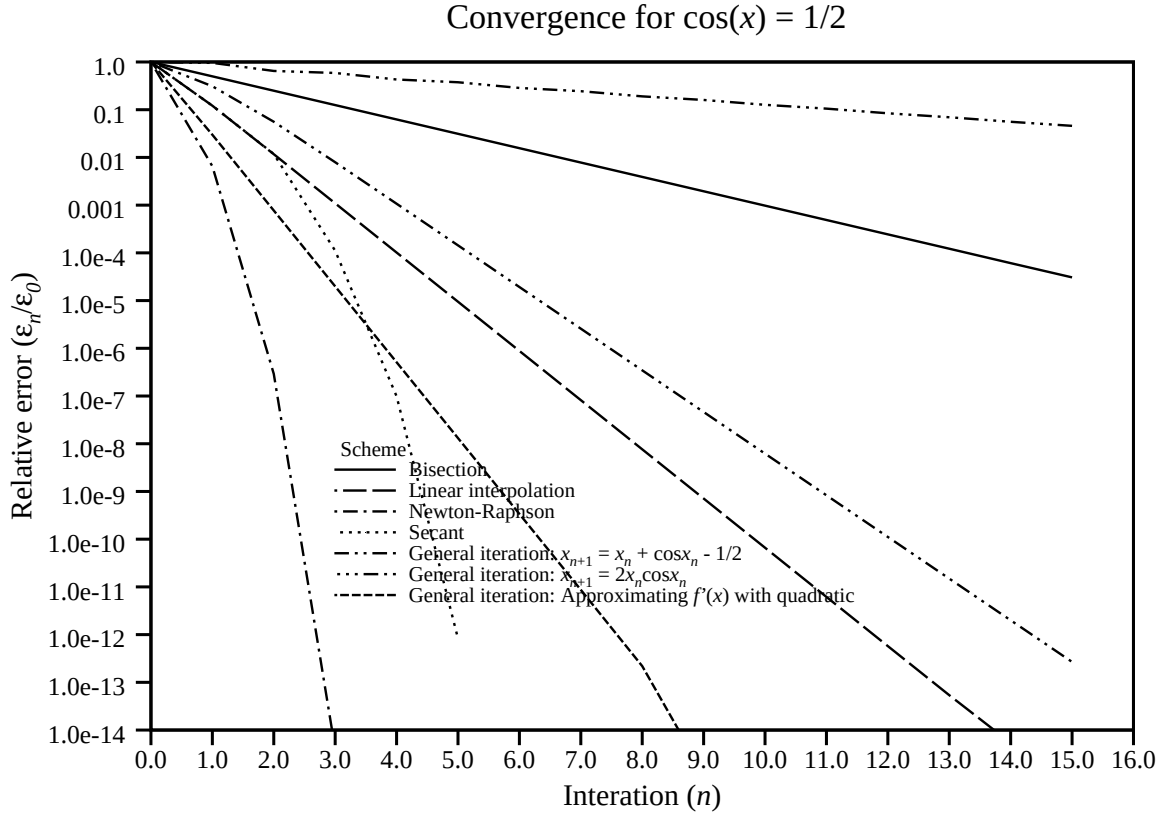


Figure 9: Comparison of the convergence of the error in the estimate of the root to $\cos x = 1/2$ for a range of different root finding algorithms.

3.7.7 Fortran program[*]

The following program was used to generate the data presented for the above examples. Note that this is included as an illustrative example. No knowledge of Fortran or any other programming language is required in this course.

```
      PROGRAM Roots
      INTEGER*4 i,j
      REAL*8    x,xa,xb,xc,fa,fb,fc,pi,xStar,f,df
      REAL*8    Error(0:15,0:15)
      f(x)=cos(x)-0.5
      df(x) = -SIN(x)
      pi = 3.141592653
      xStar = ACOS(0.5)
      WRITE(6,*)'# ',xStar,f(xStar)
C=====Bisection
      xa = 0
      fa = f(xa)
      xb = pi/2.0
      fb = f(xb)
      DO i=0,15
        xc = (xa + xb)/2.0
        fc = f(xc)
        IF (fa*fc .LT. 0.0) THEN
          xb = xc
          fb = fc
        ELSE
          xa = xc
          fa = fc
        ENDIF
        Error(0,i) = xc - xStar
      ENDDO
C=====Linear interpolation
      xa = 0
      fa = f(xa)
      xb = pi/2.0
      fb = f(xb)
      DO i=0,15
        xc = xa - (xb-xa)/(fb-fa)*fa
        fc = f(xc)
        IF (fa*fc .LT. 0.0) THEN
          xb = xc
          fb = fc
        ELSE
          xa = xc
          fa = fc
        ENDIF
        Error(1,i) = xc - xStar
      ENDDO
C=====Newton-Raphson
```

[*] Not examinable

```
      xa = pi/2.0
      DO i=0,15
        xa = xa - f(xa)/df(xa)
        Error(2,i) = xa - xStar
      ENDDO
C=====Secant
      xa = 0
      fa = f(xa)
      xb = pi/2.0
      fb = f(xb)
      DO i=0,15
        IF (fa .NE. fb) THEN
C         If fa = fb then either method has converged (xa=xb)
C         or will diverge from this point
          xc = xa - (xb-xa)/(fb-fa)*fa
          xa = xb
          fa = fb
          xb = xc
          fb = f(xb)
        ENDIF
        Error(3,i) = xc - xStar
      ENDDO
C=====Direct iteration using x + f(x) = x
      xa = 0.0
      DO i=0,15
        xa = xa + f(xa)
        Error(4,i) = xa - xStar
      ENDDO
C=====Direct iteration using  xf(x)=0 rearranged for x
C-----Starting point prevents convergence
      xa = pi/2.0
      DO i=0,15
        xa = 2.0*xa*(f(x)-0.5)
        Error(5,i) = xa - xStar
      ENDDO
C=====Direct iteration using xf(x)=0 rearranged for x
      xa = pi/4.0
      DO i=0,15
        xa = 2.0*xa*COS(xa)
        Error(6,i) = xa - xStar
      ENDDO
C=====Direct iteration using 4x(x-pi)/pi/pi to approximate f'
      xa = pi/2.0
      DO i=0,15
        xa = xa - f(xa)*pi*pi/(4.0*xa*(xa-pi))
        Error(7,i) = xa - xStar
      ENDDO
C=====Output results
      DO i=0,15
        WRITE(6,100)i,(Error(j,i),j=0,7)
      ENDDO
100   FORMAT(1x,i4,8(1x,g12.6))
      END
```

4 Linear equations

Solving equation of the form $\mathbf{A}\boldsymbol{x} = \boldsymbol{r}$ is central to many numerical algorithms. There are a number of methods which may be used, some algebraically correct, while others iterative in nature and providing only approximate solutions. Which is *best* will depend on the structure of $\mathbf{A}$, the context in which it is to be solved and the size compared with the available computer resources.

4.1 Gauss elimination

This is what you would probably do if you were computing the solution of a non-trivial system by hand. For example, if

$$\begin{aligned} x &+ 2y &+ 3z &= 6 \\ 2x &+ 2y &+ 3z &= 7, \\ x &+ 4y &+ 4z &= 9 \end{aligned} \tag{28}$$

we might then subtract 2 times the first equation from the second equation, and subtract the first equation from the third equation to get

(29)

In the second step we might add the second equation to the third to obtain

(30)

The third equation now involves only z giving $z = 1$. Substituting this back into the second equation gives an equation for y and so-on. In particular we have

(31)

We may write this system in terms of a matrix **A**, an *unknown* vector ***x*** and the *known* right-hand side ***b*** as

(32)

and do exactly the same manipulations on the rows of the matrix **A** and right-hand side ***b***. From the system

(33)

we subract 2 times the first row from the second row, and subtract the first row from the third row to obtain

(34)

Before adding the second and third rows, this time we will divide the second row through by –2, the element on the diagonal, getting

(35)

This may seem pointless in this example, but in general it simplifies the next step where we subtract a_{32} times the second row from the third row. Here $a_{32} = 2$ and represents the value in the second column of the third row of the matrix. Thus the next step is

(36)

(37)

We again divide the resulting row (now the third row) by the element on the diagonal ($a_{33} = -2$) to obtain

(38)

and retrieve immediately the value $z = 1$ from the last row of the equation. Substituting back we get

(39)

and finally we recover the answer

(40)

In general, for the system

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= r_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= r_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n &= r_3 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= r_n \end{aligned} , \tag{41}$$

we first divide the first row by a_{11} and then subtract a_{21} times the new first row from the second row, a_{31} times the new first row from the third row … and a_{n1} times the new first row from the nth row. This gives

$$\begin{bmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & \cdots & a_{1n}/a_{11} \\ 0 & a_{22} - (a_{21}/a_{11})a_{12} & a_{23} - (a_{21}/a_{11})a_{13} & & a_{2n} - (a_{21}/a_{11})a_{1n} \\ 0 & a_{32} - (a_{31}/a_{11})a_{12} & a_{33} - (a_{31}/a_{11})a_{13} & & a_{3n} - (a_{31}/a_{11})a_{1n} \\ & & \vdots & & \\ 0 & a_{n2} - (a_{n1}/a_{11})a_{12} & a_{n3} - (a_{n1}/a_{11})a_{13} & & a_{nn} - (a_{n1}/a_{11})a_{1n} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} r_1/a_{11} \\ r_2 - (a_{21}/a_{11})r_1 \\ r_3 - (a_{31}/a_{11})r_1 \\ \vdots \\ r_n - (a_{n1}/a_{11})r_1 \end{pmatrix}. \tag{42}$$

By repeating this process for rows 3 to n, this time using the new contents of element 2,2, we gradually replace the region below the leading diagonal with zeros. Once we have

$$\begin{bmatrix} 1 & \cdots & \\ 0 & & \\ 0 & & \\ & \vdots & \\ 0 & \cdots & \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{r}_1 \\ \hat{r}_2 \\ \hat{r}_3 \\ \vdots \\ \hat{r}_n \end{pmatrix} \tag{43}$$

the final solution may be obtained by back substitution.

$$\begin{aligned} x_n &= \\ x_{n-1} &= \\ x_{n-2} &= \\ &\vdots \\ x_1 &= \end{aligned} \tag{44}$$

If the arithmetic is exact, and the matrix **A** is not singular, then the answer computed in this manner will be exact (provided no zeros appear on the diagonal - see below). However, as computer arithmetic is not exact, there will be some truncation and rounding error in the answer. The cumulative effect of this error may be very significant if the loss of precision is at an early stage in the computation. In particular, if a numerically *small* number appears on the diagonal of the row, then its use in the elimination of subsequent rows may lead to differences being computed between very large and very small values with a consequential loss of precision. For example, if $a_{22}-(a_{21}/a_{11})a_{12}$ were very small, 10^{-6}, say, and both $a_{23}-(a_{21}/a_{11})a_{13}$ and $a_{33}-(a_{31}/a_{11})a_{13}$ were 1, say, then at the next stage of the computation the 3,3 element would involve calculating the difference between $1/10^{-6}=10^{6}$ and 1. If single precision arithmetic (representing real values using approximately six significant digits) were being used, the result would be simply 1.0 and subsequent calculations would be unaware of the contribution of a_{23} to the solution. A more extreme case which may often occur is if, for example, $a_{22}-(a_{21}/a_{11})a_{12}$ is zero – unless something is done it will not be possible to proceed with the computation!

A zero value occuring on the leading diagonal does not mean the matrix is singular. Consider, for example, the system

$$\begin{bmatrix} 0 & 3 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}, \tag{45}$$

the solution of which is obviously $x_1 = x_2 = x_3 = 1$. However, if we were to apply the Gauss Elimination outlined above, we would need to divide through by $a_{11} = 0$. Clearly this leads to difficulties!

4.2 Pivoting

One of the ways around this problem is to ensure that small values (especially zeros) do not appear on the diagonal and, if they do, to remove them by rearranging the matrix and vectors. In the example given in (45) we could simply interchange rows one and two to produce

(46)

or columns one and two to give

(47)

either of which may then be solved using standard Guass Elimination.

More generally, suppose at some stage during a calculation we have

$$\begin{bmatrix} 1 & 4 & 1 & 8 & 3 & 2 & \dots & 5 \\ 0 & 10^{-6} & 1 & 10 & 201 & 13 & & 4 \\ 0 & 9 & 4 & 6 & -8 & 2 & & 18 \\ 0 & 3 & 2 & -3 & 4 & 6003 & & 15 \\ 0 & 15 & 1 & 9 & 33 & -2 & & 1 \\ 0 & -155 & 23 & 4 & 25 & 73 & & 2 \\ & & & \vdots & & & & \\ 0 & 8 & 56 & 4 & -4 & 4 & \dots & 88 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{r}_1 \\ \hat{r}_2 \\ \hat{r}_3 \\ \hat{r}_4 \\ \hat{r}_5 \\ \hat{r}_6 \\ \vdots \\ \hat{r}_n \end{pmatrix} \tag{48}$$

where the element 2,5 (201) is numerically the largest value in the second row and the element 6,2 (155) the numerically largest value in the second column. As discussed above, the very small 10^{-6} value for element 2,2 is likely to cause problems. (In an extreme case we might even have the value 0 appearing on the diagonal – clearly something **must** be done to avoid a *divide by zero* error occurring!) To remove this problem we may again rearrange the rows and/or columns to bring a larger value into the 2,2 element.

4.2.1 Partial pivoting

In partial or column pivoting, we rearrange the rows of the matrix and the right-hand side to bring the numerically largest value in the column onto the diagonal. For our example matrix the largest value is in element 6,2 and so we simply swap rows 2 and 6 to give

$$\begin{bmatrix} 1 & 4 & 1 & 8 & 3 & 2 & \dots & 5 \\ 0 & -155 & 23 & 4 & 25 & 73 & & 2 \\ 0 & 9 & 4 & 6 & -8 & 2 & & 18 \\ 0 & 3 & 2 & -3 & 4 & 6003 & & 15 \\ 0 & 15 & 1 & 9 & 33 & -2 & & 1 \\ 0 & 10^{-6} & 1 & 10 & 201 & 13 & & 4 \\ & & & \vdots & & & & \\ 0 & 8 & 56 & 4 & -4 & 4 & \dots & 88 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{r}_1 \\ \hat{r}_6 \\ \hat{r}_3 \\ \hat{r}_4 \\ \hat{r}_5 \\ \hat{r}_2 \\ \vdots \\ \hat{r}_n \end{pmatrix} . \tag{49}$$

Note that our variables remain in the same order which simplifies the implementation of this procedure. The right-hand side vector, however, has been rearranged. Partial pivoting may be implemented for every step of the solution process, or only when the diagonal values are sufficiently small as to potentially cause a problem. Pivoting for every step will lead to smaller errors being introduced through numerical inaccuracies, but the continual reordering will slow down the calculation.

4.2.2 Full pivoting

The philosophy behind full pivoting is much the same as that behind partial pivoting. The main difference is that the numerically largest value in the column *or* row containing the value to be replaced. In our example above element the magnitude of element 2,5 (201) is the greatest in either row 2 or column 2 so we shall rearrange the columns to bring this element onto the diagonal. This will also entail a rearrangement of the solution vector $\boldsymbol{x}$. The rearranged system becomes

$$\begin{bmatrix} 1 & 3 & 1 & 8 & 3 & 2 & \dots & 5 \\ 0 & 201 & 1 & 10 & 10^{-6} & 13 & & 4 \\ 0 & -8 & 4 & 6 & 9 & 2 & & 18 \\ 0 & 4 & 2 & -3 & 3 & 6003 & & 15 \\ 0 & 33 & 1 & 9 & 15 & -2 & & 1 \\ 0 & 25 & 23 & 4 & -155 & 73 & & 2 \\ & & & \vdots & & & & \\ 0 & -4 & 56 & 4 & 8 & 4 & \dots & 88 \end{bmatrix} \begin{pmatrix} x_1 \\ x_5 \\ x_3 \\ x_4 \\ x_2 \\ x_6 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \\ \hat{b}_4 \\ \hat{b}_5 \\ \hat{b}_6 \\ \vdots \\ \hat{b}_n \end{pmatrix} . \tag{50}$$

The ultimate degree of accuracy can be provided by rearranging both rows and columns so that the numerically largest value in the submatrix not yet processed is brought onto the diagonal. In our example above, the largest value is 6003 occurring at position 4,6 in the matrix. We may bring this onto the diagonal for the next step by interchanging columns one and six **and** rows two and four. The order in which we do this is unimportant. The final result is

$$\begin{bmatrix} 1 & 4 & 1 & 8 & 3 & 2 & \dots & 5 \\ 0 & 6003 & 2 & -3 & 4 & 3 & & 4 \\ 0 & 2 & 4 & 6 & -8 & 9 & & 18 \\ 0 & 13 & 1 & 10 & 201 & 10^{-6} & & 15 \\ 0 & -2 & 1 & 9 & 33 & 15 & & 1 \\ 0 & 73 & 23 & 4 & 25 & -155 & & 2 \\ & & & \vdots & & & & \\ 0 & 4 & 56 & 4 & -4 & 8 & \dots & 88 \end{bmatrix} \begin{pmatrix} x_1 \\ x_6 \\ x_3 \\ x_4 \\ x_5 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{r}_1 \\ \hat{r}_4 \\ \hat{r}_3 \\ \hat{r}_2 \\ \hat{r}_5 \\ \hat{r}_6 \\ \vdots \\ \hat{r}_n \end{pmatrix} . \tag{51}$$

Again this process may be undertaken for every step, or only when the value on the diagonal is considered *too small* relative to the other values in the matrix.

If it is not possible to rearrange the columns or rows to remove a zero from the diagonal, then the matrix $\mathbf{A}$ is singular and no solution exists.

4.3 LU factorisation

A frequently used form of Gauss Elimination is LU Factorisation also known as LU Decomposition or Crout Factorisation. The basic idea is to find two matrices **L** and **U** such that

$$\mathbf{A} = \qquad (52)$$

where **L** is a lower triangular matrix (zero above the leading diagonal) and **U** is an upper triangular matrix (zero below the diagonal). Note that this decomposition is underspecified in that we may choose the relative scale of the two matrices arbitrarily. By convention, the **L** matrix is scaled to have a leading diagonal of unit values. Once we have computed **L** and **U** we need solve only

$$= \boldsymbol{b} \qquad (53)$$

then

$$\boldsymbol{x} = \qquad (54)$$

a procedure requiring $O(n^2)$ operations compared with $O(n^3)$ operations for the full Gauss elimination. While the factorisation process requires $O(n^3)$ operations, this need be done only once whereas we may wish to solve $\mathbf{A}\boldsymbol{x}=\boldsymbol{b}$ for with whole range of $\boldsymbol{b}$.

Since we have decided the diagonal elements l_{ii} in the lower triangular matrix will always be unity, it is not necessary for us to store these elements and so the matrices **L** and **U** can be stored together in an array the same size as that used for **A**. Indeed, in most implementations the factorisation will simply overwrite **A**.

The basic decomposition algorithm for overwriting **A** with **L** and **U** may be expressed as

```
# Factorisation
FOR i=1 TO n
  FOR p=i TO n
    a_pi = a_pi − Σ_{k=1}^{i−1} a_pk a_ki
  NEXT p
  FOR q=i+1 TO n
    a_iq = (a_iq − Σ_{k=1}^{i−1} a_ik a_kq) / a_ii
  NEXT q
NEXT i
# Forward Substitution
FOR i=1 TO n
  FOR q=n+1 TO n+m
    a_iq = (a_iq − Σ_{k=1}^{i−1} a_ik a_kq) / a_ii
  NEXT q
NEXT i
# Back Substitution
FOR i=n−1 TO 1
```

```
  FOR q=n+1 TO n+m
    a_iq = a_iq − Σ_{k=i+1}^{n} a_ik a_kq
  NEXT q
NEXT i
```

This algorithm assumes the right-hand side(s) are initially stored in the same array structure as the matrix and are positioned in the column(s) $n+1$ (to $n+m$ for m right-hand sides). To improve the efficiency of the computation for right-hand sides known in advance, the forward substitution loop may be incorporated into the factorisation loop.

Figure 10 indicates how the LU Factorisation process works. We want to find vectors $\boldsymbol{l}_i^{\mathrm{T}}$ and $\boldsymbol{u}_j$ such that $a_{ij} = \boldsymbol{l}_i^{\mathrm{T}}\boldsymbol{u}_j$. When we are at the stage of calculating the ith element of $\boldsymbol{u}_j$, we will already have the i nonzero elements of $\boldsymbol{l}_i^{\mathrm{T}}$ and the first $i-1$ elements of $\boldsymbol{u}_j$. The ith element of $\boldsymbol{u}_j$ may therefore be chosen simply as $u_{j(i)} = a_{ij} - \boldsymbol{l}_i^{\mathrm{T}}\boldsymbol{u}_j$ where the dot-product is calculated assuming $u_{j(i)}$ is zero.

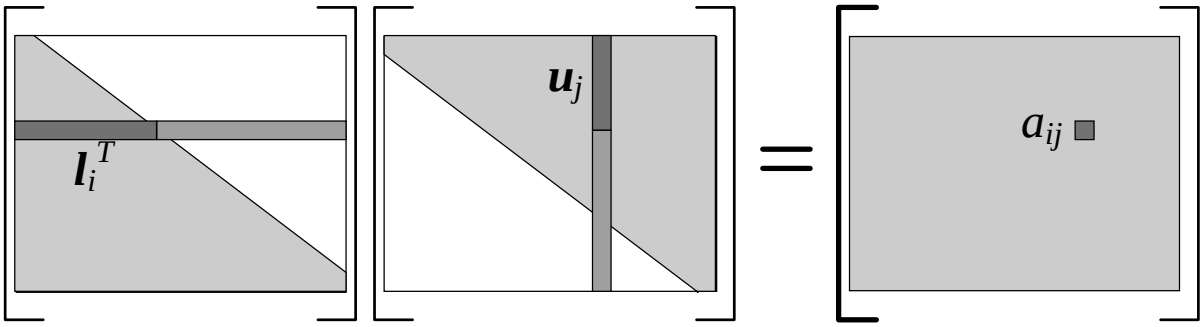


Figure 10: Diagramatic representation of how LU factorisation works for calculating u_{ij} to replace a_{ij} where $i < j$. The white areas represent zeros in the **L** and **U** matrices.

As with normal Gauss Elimination, the potential occurrence of small or zero values on the diagonal can cause computational difficulties. The solution is again pivoting – partial pivoting is normally all that is required. However, if the matrix is to be used in its factorised form, it will be essential to record the pivoting which has taken place. This may be achieved by simply recording the row interchanges for each i in the above algorithm and using the same row interchanges on the right-hand side when using **L** in subsequent forward substitutions.

4.4 Banded matrices

The LU Factorisation may readily be modified to account for banded structure such that the only non-zero elements fall within some distance of the leading diagonal. For example, if elements outside the range $a_{i,i-b}$ to $a_{i,i+b}$ are all zero, then the summations in the LU Factorisation algorithm need be performed only from $k=i$ or $k=i+1$ to $k=i+b$. Moreover, the factorisation loop FOR $q=i+1$ TO n can terminate at $i+b$ instead of n.

One problem with such banded structures can occur if a (near) zero turns up on the diagonal during the factorisation. Care must then be taken in any pivoting to try to maintain the banded structure. This may require, for example, pivoting on both the rows and columns as described in section 4.2.2.

Making use of the banded structure of a matrix can save substantially on the execution time and, if the matrix is stored intelligently, on the storage requirements. Software libraries such as NAG and

IMSL provide a range of routines for solving such banded linear systems in a computationally and storage efficient manner.

4.5 Tridiagonal matrices

A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal ($b=1$). It is sometimes possible to rearrange the rows and columns of a matrix which does not initially have this structure in order to gain this structure and hence greatly simplify the solution process. As we shall see later in sections 6 to 8, tridiagonal matrices frequently occur in numerical solution of differential equations.

A tridiagonal system may be written as

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_5 & b_5 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & b_{n-3} & c_{n-3} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & a_{n-2} & b_{n-2} & c_{n-2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & a_n & b_n \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ \vdots \\ x_{n-3} \\ x_{n-2} \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ \vdots \\ r_{n-3} \\ r_{n-2} \\ r_{n-1} \\ r_n \end{pmatrix} \tag{55}$$

or

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = r_i \tag{56}$$

for $i=1,\ldots,n$. Clearly x_{-1} and x_{n+1} are not required and we set $a_1=c_n=0$ to reflect this.

If solved by standard Gauss elimination, then we start by dividing the first equation by b_0 before subtracting a_2 times this from the second equation to eliminate a_2. We then divide the new second equation by the new value in place of b_2, before subtracting a_3 times this equation from the third, and so on.

Solution, by analogy with the LU Factorisation, may be expressed as

```
# Factorisation
FOR i=1 TO n
  b_i = b_i - a_i c_i-1
  c_i = c_i / b_i
NEXT i
# Forward Substitution
FOR i=1 TO n
  r_i = (r_i - a_i r_i-1) / b_i
NEXT i
# Back Substitution
FOR i=n-1 TO 1
  r_i = r_i - c_i r_i+1
NEXT i
```

4.6 Other approaches to solving linear systems

There are a number of other methods for solving general linear systems of equations including approximate iterative techniques. Many large matrices which need to be solved in practical situations have very special structures which allow solution - either exact or approximate - much faster than the general $O(n^3)$ solvers presented here. We shall return to this topic in section 8.1 where we shall discuss a system with a special structure resulting from the numerical solution of the Laplace equation.

4.7 Over determined systems*

If the matrix $\mathbf{A}$ contains m rows and n columns, with $m > n$, the system is probably over-determined (unless there are $m–n$ redundant rows). Such a system may be the result from fitting a model with unknown coefficients to experimental data or observations. For example, fitting data points x_i,y_i ($i = 0,n–1$) with the model $a + bx + cx^2 + de^x = y$ leads to the linear system

$$\begin{bmatrix} 1 & x_0 & x_0^2 & e^{x_0} \\ 1 & x_1 & x_1^2 & e^{x_1} \\ 1 & x_2 & x_2^2 & e^{x_2} \\ 1 & x_3 & x_3^2 & e^{x_3} \\ 1 & x_4 & x_4^2 & e^{x_4} \\ 1 & x_5 & x_5^2 & e^{x_5} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & e^{x_{n-1}} \end{bmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ \vdots \\ y_{n-1} \end{pmatrix}. \tag{57}$$

which is of the form $\mathbf{Ax} = \mathbf{r}$. While the *solution* to $\mathbf{Ax} = \mathbf{r}$ will not exist in an algebraic sense, it can be valuable to determine the solution in an approximate sense. The *error* in this approximate solution is then

$$\mathbf{e} = \tag{58}$$

The approximate solution is chosen by optimising this error in some manner. Most useful among the classes of *solution* is the Least Squares solution. In this solution we minimise the *residual sum of squares*, which is simply

$$rss = \tag{59}$$

Substituting for $\mathbf{e}$ we obtain

$$rss = \tag{60}$$

and setting $\partial rss/\partial x$ to zero gives

$$\frac{\partial rss}{\partial \boldsymbol{x}} = \tag{61}$$

Thus, if we solve the n by n problem $\mathbf{A}^{\mathrm{T}}\mathbf{A}\boldsymbol{x} = \mathbf{A}^{\mathrm{T}}\boldsymbol{r}$, the solution vector $\boldsymbol{x}$ will give us the solution in a least squares sense.

Warning: The matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ is often poorly conditioned (nearly singular) and can lead to significant errors in the resulting Least Squares solution due to rounding error. While these errors may be reduced using pivoting in combination with Gauss Elimination, it is generally better to solve the Least Squares problem using the Householder transformation, as this produces less rounding error, or better still by Singular Value Decomposition which will highlight any redundant or nearly redundant variables in $\boldsymbol{x}$.

The Householder transformation avoids the poorly conditioned nature of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ by solving the problem directly without evaluating this matrix. Suppose $\mathbf{Q}$ is an orthogonal matrix such that

$$\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = \mathbf{I}, \tag{62}$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{Q}$ is chosen to transform $\mathbf{A}$ into

$$\mathbf{QA} = \begin{bmatrix} & \\ & \end{bmatrix}, \tag{63}$$

where $\mathbf{R}$ is a square matrix of a size n and $\mathbf{0}$ is a zero matrix of size m-n by n. The right-hand side of the system $\mathbf{QA}\boldsymbol{x} = \mathbf{Q}\boldsymbol{r}$ becomes

$$\mathbf{Q}\boldsymbol{r} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{c} \end{pmatrix}, \tag{64}$$

where $\mathbf{b}$ is a vector of size n and $\mathbf{c}$ is a vector of size m-n.

Now the turning point (global minimum) in the residual sum of squares, (61), this occurs when

$$\begin{aligned} \frac{\partial rss}{\partial \boldsymbol{x}} &= 2\left[\mathbf{A}^{\mathrm{T}}\mathbf{A}\boldsymbol{x} - \mathbf{A}^{\mathrm{T}}\boldsymbol{r}\right] \\ &= 2\left[\mathbf{A}^{\mathrm{T}} \quad \mathbf{A}\boldsymbol{x} - \mathbf{A}^{\mathrm{T}} \quad \boldsymbol{r}\right] \end{aligned} \tag{65}$$

* Not examinable

vanishes. For a non-trivial solution, that occurs when

$$\mathbf{R}\boldsymbol{x} = \boldsymbol{b}. \tag{66}$$

This system may be solved to obtain the least squares solution $\boldsymbol{x}$ using any of the normal linear solvers discussed above.

Further discussion of these methods is beyond the scope of this course.

4.8 Under determined systems*

If the matrix $\mathbf{A}$ contains m rows and n columns, with $m < n$, the system is under determined. The *solution* maps out a $n–m$ dimensional subregion in n dimensional space. Solution of such systems typically requires some form of *optimisation* in order to further constrain the solution vector.

Linear programming represents one method for solving such systems. In Linear Programming, the solution is optimised such that the *objective function* $z=\boldsymbol{c}^T\boldsymbol{x}$ is minimised. The "Linear" indicates that the underdetermined system of equations is linear and the objective function is linear in the solution variable $\boldsymbol{x}$. The "Programming" arose to enhance the chances of obtaining funding for research into this area when it was developing in the 1960s.

* Not examinable

5 Numerical integration

There are two main reasons for you to need to do numerical integration: analytical integration may be impossible or infeasible, or you may wish to integrate tabulated data rather than known functions. In this section we outline the main approaches to numerical integration. Which is preferable depends in part on the results required, and in part on the function or data to be integrated.

5.1 Manual method

If you were to perform the integration by hand, one approach is to superimpose a grid on a graph of the function to be integrated, and simply count the squares, counting only those covered by 50% or more of the function. Provided the grid is sufficiently fine, a reasonably accurate estimate may be obtained. Figure 11 demonstrates how this may be achieved.

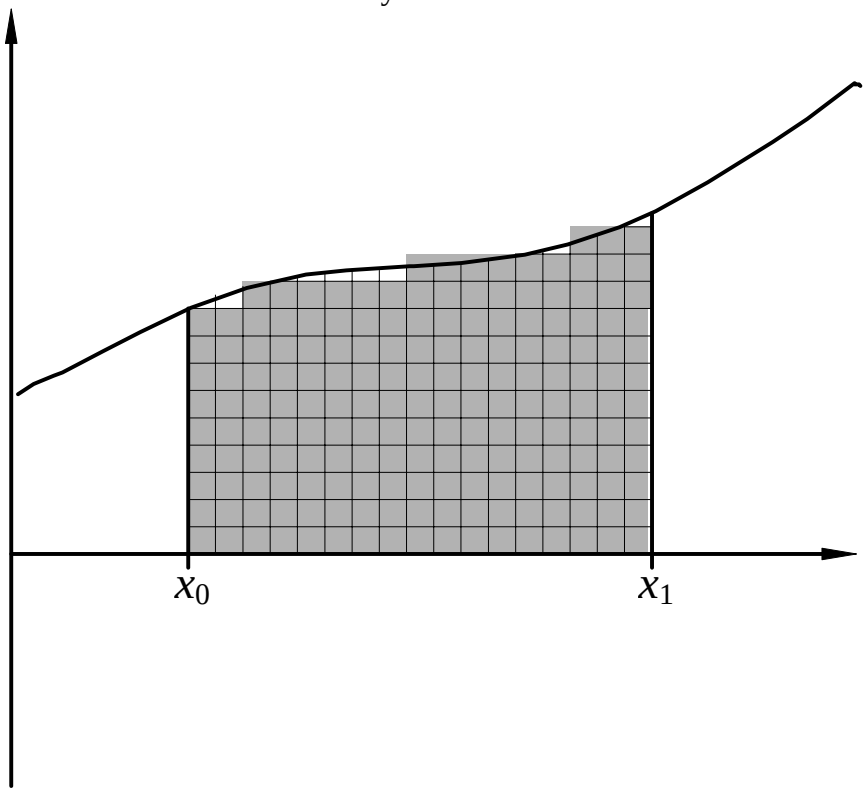


Figure 11: Manual method for determining integral by superimposing a grid on a graph of the integrand. The boxes indicated in grey are counted.

5.2 Constant rule

Perhaps the simplest form of numerical integration is to assume the function $f(x)$ is constant over the interval being integrated. Such a scheme is illustrated in figure 12. Clearly this is not going to be a very accurate method of integrating, and indeed leads to an ambiguous result, depending on whether the constant is selected from the lower or the upper limit of the integral.

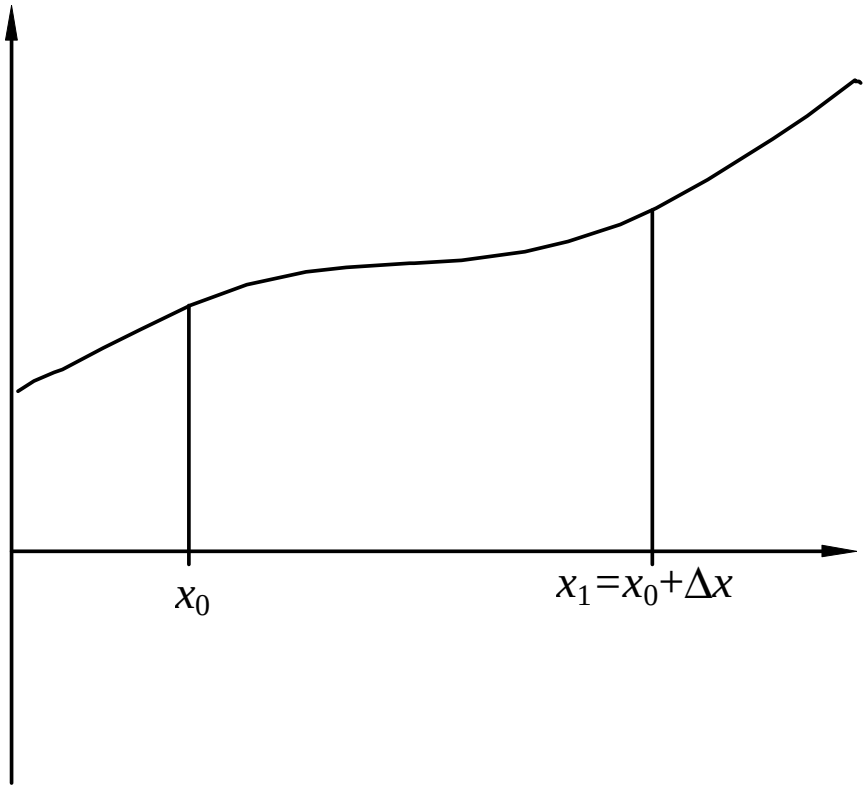


Figure 12: Integration by *constant rule* whereby the value of $f(x)$ is assumed constant over the interval.

Integration of a Taylor Series expansion of $f(x)$ shows the error in this approximation to be

$$\int_{x_0}^{x+\Delta x} f(x)\mathrm{d}x = \int_{x_0}^{x+\Delta x} \quad (67)$$

$$=$$

$$=$$

if the constant is taken from the lower limit, or $f(x_0+\Delta x)\Delta x$ if taken from the upper limit. In both cases the error is $O(\Delta x^2)$, with the coefficient being derived from $f'(x)$.

Clearly we can do much better than this, and as a result this rule is not used in practice, although a knowledge of it helps with understanding the solution of ordinary differential equations (see §6).

5.3 Trapezium rule

Consider the Taylor Series expansion integrated from x_0 to $x_0+\Delta x$:

$$\int_{x_0}^{x+\Delta x} f(x)\mathrm{d}x = \int_{x_0}^{x+\Delta x} f(x_0) + f'(x_0)\Delta x + \tfrac{1}{2} f''(x_0)\Delta x^2 + \ldots \, \mathrm{d}x$$
$$= f(x_0)\Delta x + \tfrac{1}{2} f'(x_0)\Delta x^2 + \tfrac{1}{6} f''(x_0)\Delta x^3 + \ldots \; . \qquad (68)$$
$$=$$

The approximation represented by $\frac{1}{2}[f(x_0) + f(x_0+\Delta x)]\Delta x$ is called the Trapezium Rule based on its geometric interpretation as shown in figure 13.

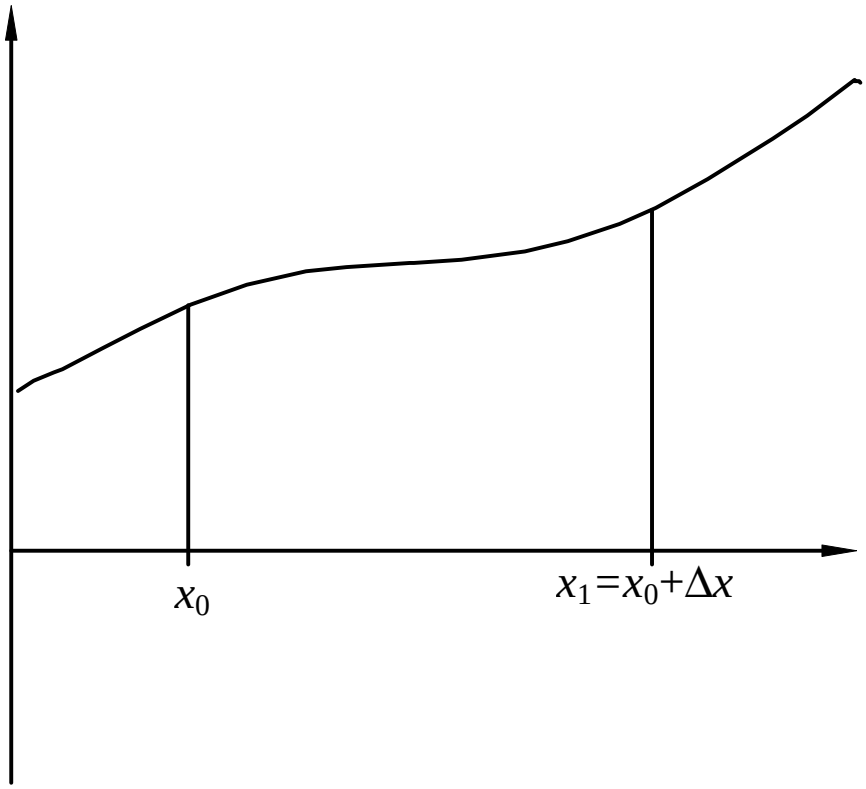


Figure 13: Graphical interpretation of the trapezium rule.

As we can see from equation (68), the error in the Trapezium Rule is proportional to Δx^3. Thus, if we were to halve Δx, the error would be decreased by a factor of eight. However, the size of the domain would be halved, thus requiring the Trapezium Rule to be evaluated twice and the contributions summed. The net result is the error decreasing by a factor of four rather than eight. The Trapezium Rule used in this manner is sometimes termed the Compound Trapezium Rule, but more often simply the Trapezium Rule. In general it consists of the sum of integrations over a smaller distance Δx to obtain a smaller error.

Suppose we need to integrate from x_0 to x_1. We shall subdivide this interval into n steps of size $\Delta x=(x_1-x_0)/n$ as shown in figure 14.

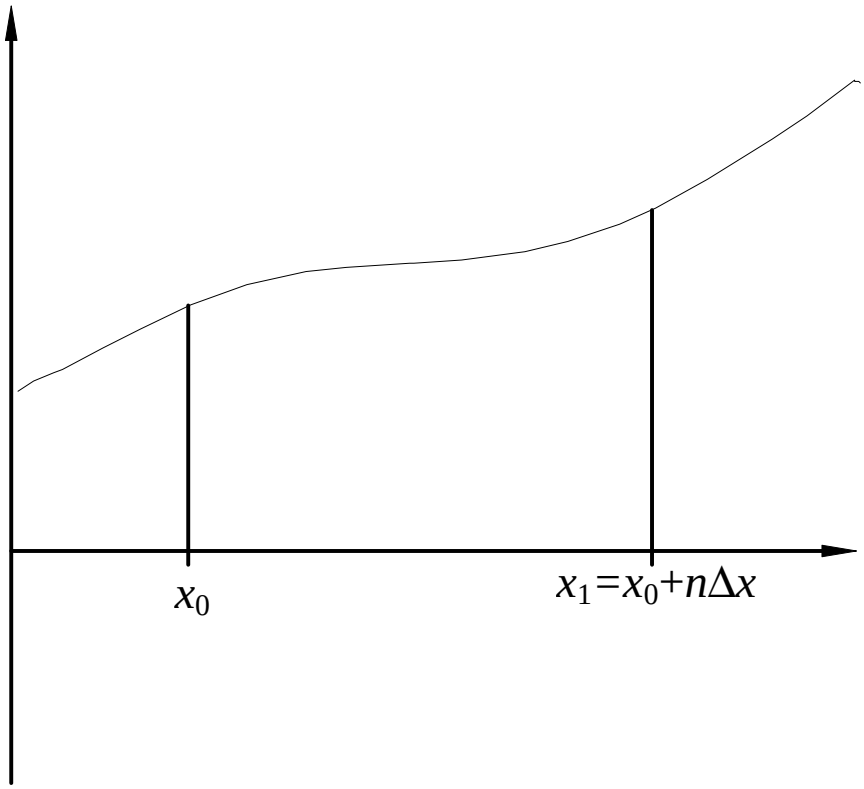


Figure 14: Compound Trapezium Rule.

The Compound Trapezium Rule approximation to the integral is therefore

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = \sum_{i=0}^{n-1} \int_{x_0+i\Delta x}^{x_0+(i+1)\Delta x} f(x)\,\mathrm{d}x$$

$$\approx \sum_{i=0}^{n-1} \qquad . \tag{69}$$

$$=$$

While the error for each step is $O(\Delta x^3)$, the cumulative error is n times this or $O(\Delta x^2) \sim O(n^{-2})$.

The above analysis assumes Δx is constant over the interval being integrated. This is not necessary and an extension to this procedure to utilise a smaller step size Δx_i in regions of high curvature would reduce the total error in the calculation, although it would remain $O(\Delta x^2)$. We would choose to reduce Δx in the regions of high curvature as we can see from equation (68) that the leading order truncation error is scaled by f''.

5.4 Mid-point rule

A variant on the Trapezium Rule is obtained by integrating the Taylor Series from $x_0-\Delta x/2$ to $x_0+\Delta x/2$:

$$\int_{x_0-\frac{1}{2}\Delta x}^{x_0+\frac{1}{2}\Delta x} f(x)\,\mathrm{d}x = \int_{x_0-\frac{1}{2}\Delta x}^{x_0+\frac{1}{2}\Delta x} f(x_0)+f'(x_0)\Delta x+\tfrac{1}{2}f''(x_0)\Delta x^2+\ldots\,\mathrm{d}x \qquad (70)$$

$$=$$

By evaluating the function $f(x)$ at the midpoint of each interval the error may be slightly reduced relative to the Trapezium rule (the coefficient in front of the curvature term is 1/24 for the Mid-point Rule compared with 1/12 for the Trapezium Rule) but the method remains of the same order. Figure 15 provides a graphical interpretation of this approach.

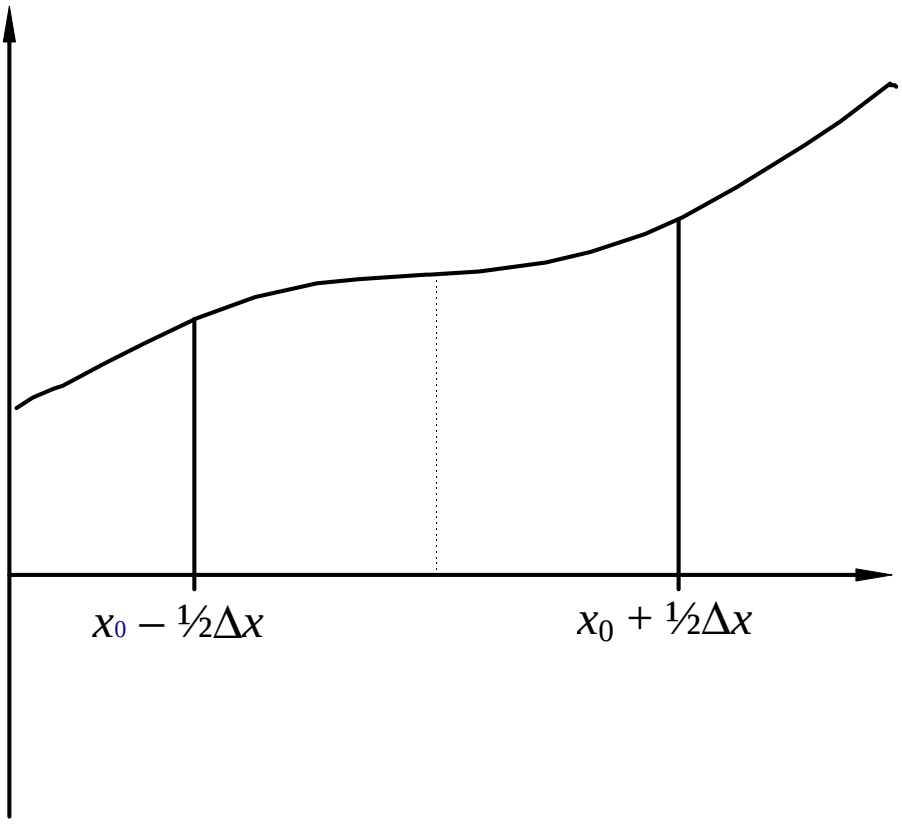


Figure 15: Graphical interpretation of the midpoint rule. The grey region defines the midpoint rule as a rectangular approximation with the dashed lines showing alternative trapeziodal aproximations containing the same area.

Again we may reduce the error when integrating the interval x_0 to x_1 by subdividing it into n smaller steps. This Compound Mid-point Rule is then

$$\int_{x_0}^{x_1} f(x)\mathrm{d}x \approx \sum_{i=0}^{n-1} \tag{71}$$

with the graphical interpretation shown in figure 16. The difference between the Trapezium Rule and Mid-point Rule is greatly diminished in their compound forms. Comparison of equations (69) and (71) show the only difference is in the phase relationship between the points used and the domain, plus how the first and last intervals are calculated.

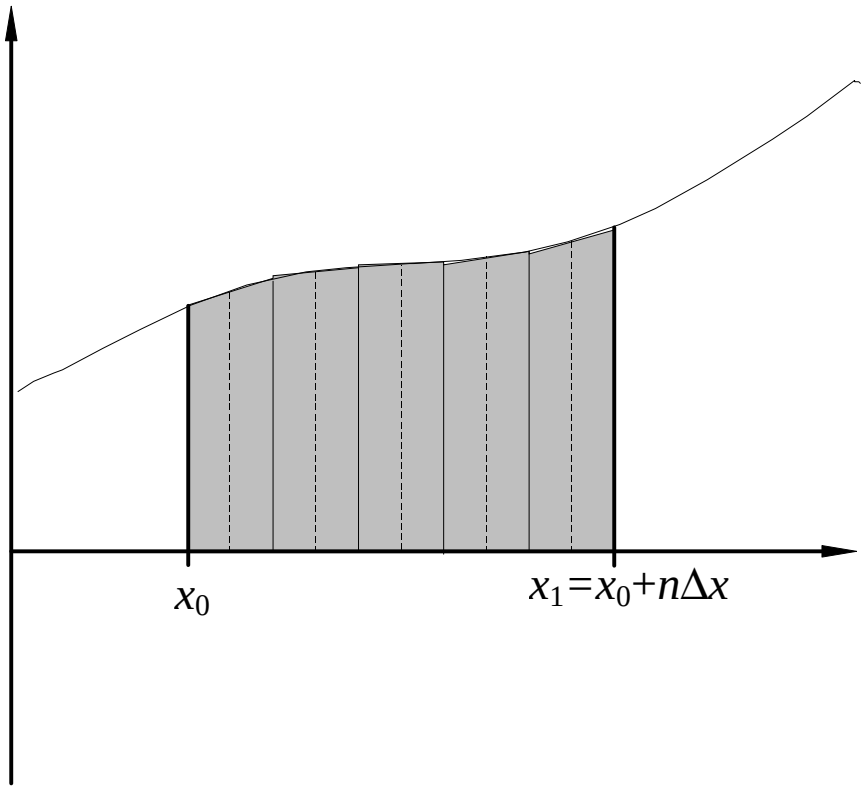


Figure 16: Compound Mid-point Rule.

There are two further advantages of the Mid-point Rule over the Trapezium Rule. The first is that is requires one fewer function evaluations for a given number of subintervals, and the second that it can be used more effectively for determining the integral near an integrable singularity. The reasons for this are clear from figure 17.

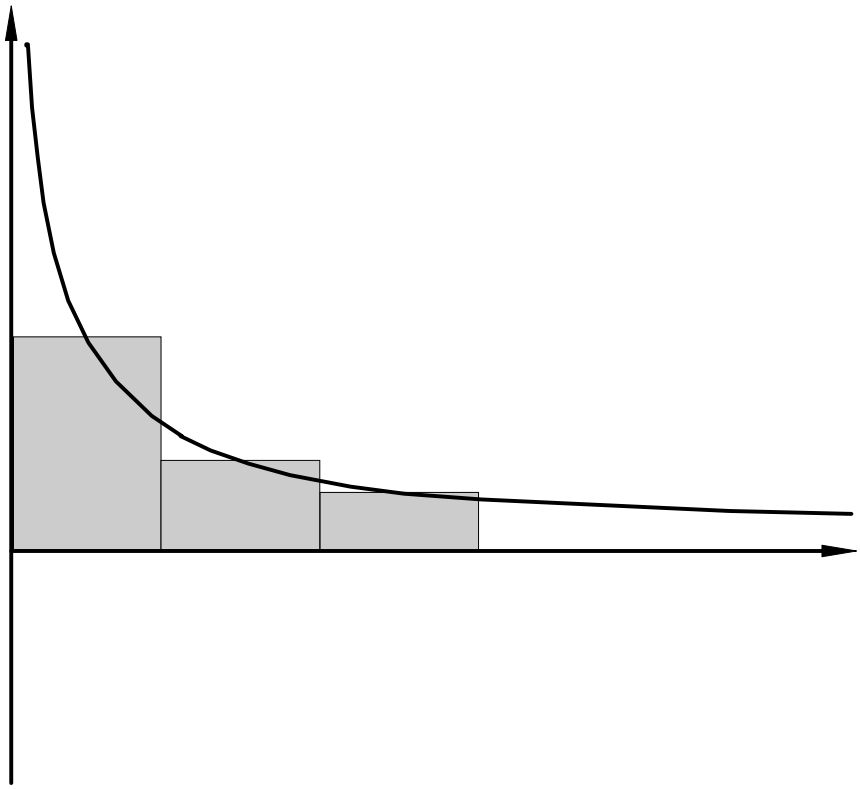


Figure 17: Applying the Midpoint Rule where the singular integrand would cause the Trapezium Rule to fail.

5.5 Simpson's rule

An alternative approach to decreasing the step size Δx for the integration is to increase the accuracy of the functions used to approximate the integrand. Figure 18 sketches one possibility, using a quadratic approximation to $f(x)$.

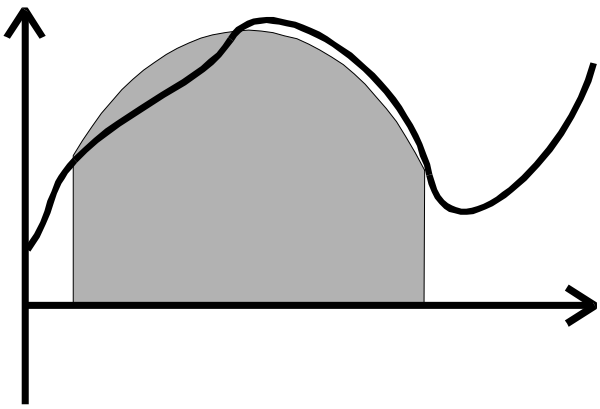


Figure 18: Quadratic approximation to integrand is the basis of Simpson's Rule.

Integrating the Taylor series over an interval $2\Delta x$ shows

$$\int_{x_0}^{x+2\Delta x} f(x)\,\mathrm{d}x = \tag{72}$$

$$= \frac{\Delta x}{3}\Big[$$
$$f(x_0)$$
$$+4\Big(f(x_0)+f'(x_0)\Delta x+\tfrac{1}{2}f''(x_0)\Delta x^2+\tfrac{1}{6}f'''(x_0)\Delta x^3+\tfrac{1}{24}f^{iv}(x_0)\Delta x^4+\ldots\Big)$$
$$+\Big(f(x_0)+2f'(x_0)\Delta x+2f''(x_0)\Delta x^2+\tfrac{4}{3}f'''(x_0)\Delta x^3+\tfrac{2}{3}f^{iv}(x_0)\Delta x^4+\ldots\Big)$$
$$-\tfrac{17}{30}f^{iv}(x_0)\Delta x^4\ldots\Big]$$
$$=$$

Whereas the error in the Trapezium rule was $O(\Delta x^3)$, Simpson's rule is two orders more accurate at $O(\Delta x^5)$, giving exact integration of cubics.

To improve the accuracy when integrating over larger intervals, the interval x_0 to x_1 may again be subdivided into n steps. The three-point evaluation for each subinterval requires that there are an even number of subintervals. Hence we must be able to express the number of intervals as $n=2m$. The Compound Simpson's rule is then

$$\int_{x_0}^{x_1} f(x)\mathrm{d}x \approx \frac{\Delta x}{3}\sum_{i=0}^{m-1} f\left(x_0+2i\Delta x\right)+4f\left(x_0+(2i+1)\Delta x\right)+f\left(x_0+(2i+2)\Delta x\right) \tag{73}$$

$$=$$

and the corresponding error $O(n\Delta x^5)$ or $O(\Delta x^4)$.

5.6 Quadratic triangulation*

Simpson's Rule may be employed in a manual way to determine the integral with nothing more than a ruler. The approach is to cover the domain to be integrated with a triangle or trapezium (whichever is geometrically more appropriate) as is shown in figure 19. The integrand may cross the side of the trapezium (triangle) connecting the end points. For each arc-like region so created (there are two in figure 19) the maximum deviation (indicated by arrows in figure 19) from the line should be measured, as should the length of the chord joining the points of crossing. From Simpson's rule we may approximate the area between each of these arcs and the chord as

* Not examinable

$$area = {}^{2}\!/_{3} \times chord \times maxDeviation, \tag{74}$$

remembering that some increase the area while others decrease it relative to the initial trapezoidal (triangular) estimate. The overall estimate (ignoring linear measurement errors) will be $O(l^5)$, where l is the length of the (longest) chord.

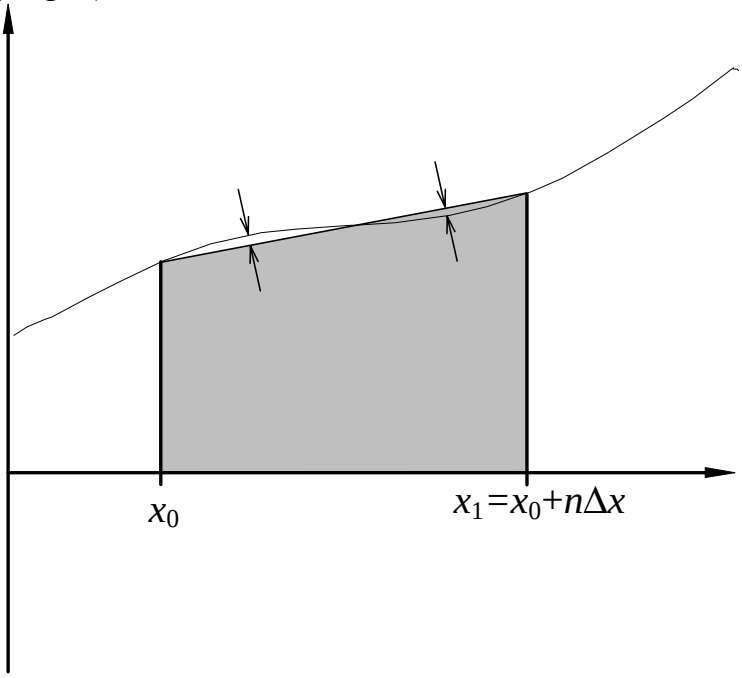


Figure 19: Quadratic triangulation to determine the area using a manual combination of the Trapezium and Simpson's Rules.

5.7 Romberg integration

With the Compound Trapezium Rule we know from section 5.3 the error in some estimate $T(\Delta x)$ of the integral I using a step size Δx goes like $c\Delta x^2$ as $\Delta x \to 0$, for some constant c. Likewise the error in $T(\Delta x/2)$ will be $c\Delta x^2/4$. From this we may construct a revised estimate $T^{(1)}(\Delta x/2)$ for I as a weighted mean of $T(\Delta x)$ and $T(\Delta x/2)$:

$$T^{(1)}(\Delta x/2) = \alpha T(\Delta x/2) + (1-\alpha)T(\Delta x)$$
$$= \alpha[I + c\Delta x^2/4 + O(\Delta x^4)] + (1-\alpha)[I + c\Delta x^2 + O(\Delta x^4)]. \tag{75}$$

By choosing the weighting factor $\alpha = 4/3$ we elimate the leading order ($O(\Delta x^2)$) error terms, relegating the error to $O(\Delta x^4)$. Thus we have

$$T^{(1)}(\Delta x/2) = [4T(\Delta x/2) - T(\Delta x)]/3. \tag{76}$$

Comparison with equation (73) shows that this formula is precisely that for Simpson's Rule.

This same process may be carried out to higher orders using $\Delta x/4$, $\Delta x/8$, … to eliminate the higher order error terms. For the Trapezium Rule the errors are all even powers of Δx and as a result it can be shown that

$$T^{(m)}(\Delta x/2) = [2^{2m}T^{(m-1)}(\Delta x/2) - T^{(m-1)}(\Delta x)]/(2^{2m}-1). \tag{77}$$

A similar process may also be applied to the Compound Simpson's Rule.

5.8 Gauss quadrature

By careful selection of the points at which the function is evaluated it is possible to increase the precision for a given number of function evaluations. The Mid-point rule is an example of this: with just a single function evaluation it obtains the same order of accuracy as the Trapezium Rule (which requires two points).

One widely used example of this is Gauss quadrature which enables exact integration of cubics with only two function evaluations (in contrast Simpson's Rule, which is also exact for cubics, requires three function evaluations). Gauss quadrature has the formula

$$\int_{x_0}^{x_1=x_0+\Delta x} f(x)\mathrm{d}x \approx \frac{\Delta x}{2}\left[f\left(x_0 + \left(\frac{1}{2}-\frac{\sqrt{3}}{6}\right)\Delta x\right) + f\left(x_0 + \left(\frac{1}{2}+\frac{\sqrt{3}}{6}\right)\Delta x\right)\right] + O\left(\Delta x^4\right). \tag{78}$$

In general it is possible to choose M function evaluations per interval to obtain a formula exact for all polynomials of degree $2M-1$ and less.

The Gauss Quadrature accurate to order $2M-1$ may be determined using the same approach required for the two-point scheme. This may be derived by comparing the Taylor Series expansion for the integral with that for the points $x_0+\alpha\Delta x$ and $x_0+\beta\Delta x$:

$$\begin{aligned}\int_{x_0}^{x_1=x_0+\Delta x} f(x)\mathrm{d}x &= \Delta x f\left(x_0\right) + \frac{\Delta x^2}{2} f'\left(x_0\right) + \frac{\Delta x^3}{6} f''\left(x_0\right) + \frac{\Delta x^4}{24} f'''\left(x_0\right) + O\left(\Delta x^5\right) \\ &= \frac{\Delta x}{2}\left[f\left(x_0\right) + \alpha\Delta x f'\left(x_0\right) + \frac{(\alpha\Delta x)^2}{2} f''\left(x_0\right) + \frac{(\alpha\Delta x)^3}{6} f'''\left(x_0\right) + \ldots \right. \\ &\quad \left. + f\left(x_0\right) + \beta\Delta x f'\left(x_0\right) + \frac{(\beta\Delta x)^2}{2} f''\left(x_0\right) + \frac{(\beta\Delta x)^3}{6} f'''\left(x_0\right) + \ldots \right] \\ &= \Delta x f\left(x_0\right) + (\alpha+\beta)\frac{\Delta x^2}{2} f'\left(x_0\right) + \left(\alpha^2+\beta^2\right)\frac{\Delta x^3}{4} + \left(\alpha^3+\beta^3\right)\frac{\Delta x^4}{12} + \ldots\end{aligned} \tag{79}$$

Equating the various terms reveals

(80)

the solution of which gives the positions stated in equation (78).

Using three function evaluations: symmetry suggests for the interval $x_0-\Delta x$ to $x_0+\Delta x$ the points should be evaluated at x_0 and $x_0\pm\alpha$ with weightings $2\Delta xA$ and $2\Delta xB$. The Taylor Series expansion of the integral gives

$$\int_{x_0-\Delta x}^{x_0+\Delta x} f(x)\mathrm{d}x = 2\Delta x\left[f\left(x_0\right) + \frac{\Delta x^2}{6} f''\left(x_0\right) + \frac{\Delta x^4}{120} f^{iv}\left(x_0\right) + \frac{\Delta x^6}{5040} f^{vi}\left(x_0\right) + O\left(\Delta x^8\right)\right], \tag{81}$$

while the expansion of the function for the three points gives

$$\int_{x_0-\Delta x}^{x_0+\Delta x} f(x)\mathrm{d}x = 2\Delta x\left[Af(x_0)+Bf(x_0-\alpha)+Bf(x_0+\alpha)\right]$$

$$= 2\Delta x\Big[Af + B\Big(f-\alpha f'+\tfrac{1}{2}\alpha^2 f''-\tfrac{1}{6}\alpha^3 f'''+\tfrac{1}{24}\alpha^4 f^{iv}-\tfrac{1}{120}\alpha^5 f^{v}+\tfrac{1}{720}\alpha^6 f^{vi}-\tfrac{1}{5040}\alpha^7 f^{vii}+O(\alpha^8)\Big)$$
$$+B\Big(f+\alpha f'+\tfrac{1}{2}\alpha^2 f''+\tfrac{1}{6}\alpha^3 f'''+\tfrac{1}{24}\alpha^4 f^{iv}+\tfrac{1}{120}\alpha^5 f^{v}+\tfrac{1}{720}\alpha^6 f^{vi}+\tfrac{1}{5040}\alpha^7 f^{vii}+O(\alpha^8)\Big)\Big]$$

$$= 2\Delta x\left[(A+2B)f + B\alpha^2 f'' + \tfrac{1}{12}B\alpha^4 f^{iv} + \tfrac{1}{360}\alpha^6 f^{vi} + O(\alpha^8)\right]. \qquad (82)$$

Comparing the terms between (81) and (82) gives

$$A + 2B = 1$$

$$B\alpha^2 = \Delta x^2/6$$

$$B\alpha^4/12 = \Delta x^4/120 \qquad (83)$$

which may be solved to obtain the position of the evaluations and the weightings

$$(B\alpha^4/12)/B\alpha^2 = (\Delta x^4/120) / (\Delta x^2/6)$$

$$\Rightarrow \alpha^2 = (6/10)\Delta x^2$$

$$\Rightarrow \alpha = (3/5)^{1/2}\Delta x$$

$$B = 5/18$$

$$A = 4/9, \qquad (84)$$

thus

$$\int_{x_0-\Delta x}^{x_0+\Delta x} f(x)\mathrm{d}x = \frac{1}{9}\Delta x\left[8f(x_0)+5f\left(x_0-\left(\frac{3}{5}\right)^{1/2}\Delta x\right)+5f\left(x_0+\left(\frac{3}{5}\right)^{1/2}\Delta x\right)\right]. \qquad (85)$$

5.9 Example of numerical integration

Consider the integral

$$\int_0^{\pi} \sin x\, \mathrm{d}x = 2\,, \qquad (86)$$

which may be integrated numerically using any of the methods described in the previous sections. Tables 2 to 6 summarise the error in the numerical estimates for the Trapezium Rule, Midpoint Rule, Simpson's Rule, Gauss Quadrature and a three point Gauss Quadrature (formula derived in lectures). Table 7 compares these errors. The results are presented in terms of the number of function evaluations required. The calculations were performed in double precision.

No. intervals	No. $f(x)$	Trapezium Rule	Error Ratio: e_{2n}/e_n
1	2	-2.00000000	0.2146
2	3	-0.429203673	0.24203
4	5	-0.103881102	0.24806
8	9	-0.0257683980	0.24952
16	17	-0.00642965622	0.24988
32	33	-0.00160663902	0.24997
64	65	-0.000401611359	0.24999
128	129	-0.000100399815	0.25
256	257	-0.0000250997649	0.25
512	513	-0.00000627492942	0.25
1024	1025	-0.00000156873161	0.25
2048	2049	-0.000000392182860	0.25
4096	4097	-0.0000000980457133	0.25
8192	8193	-0.0000000245114248	0.25
16384	16385	-0.00000000612785222	0.25
32768	32769	-0.00000000153194190	0.24999
65536	65537	-0.000000000382977427	0.24994
131072	131073	-0.0000000000957223189	0.25014
262144	262145	-0.0000000000239435138	0.24898
524288	524289	-0.0000000000596145355	

Table 2: Error in Trapezium Rule. Note error ratio $\to 2^{-2}$ (Δx^2)

No. intervals	No. $f(x)$	Midpoint Rule	Error Ratio: e_{2n}/e_n
1	1	1.14189790	0.19392
2	2	0.221441469	0.23638
4	4	0.0523443059	0.24662
8	8	0.0129090855	0.24916
16	16	0.00321637816	0.24979
32	32	0.000803416309	0.24995
64	64	0.000200811728	0.24999
128	128	0.0000502002859	0.25
256	256	0.0000125499060	0.25
512	512	0.00000313746618	0.25
1024	1024	0.000000784365898	0.25
2048	2048	0.000000196091438	0.25
4096	4096	0.0000000490228564	0.25
8192	8192	0.0000000122557182	0.25
16384	16384	0.00000000306393221	0.25
32768	32768	0.000000000765979280	0.25
65536	65536	0.000000000191497040	0.24979
131072	131072	0.0000000000478341810	0.25081
262144	262144	0.0000000000119970700	0.25286
524288	524288	0.0000000000303357339	

Table 3: Error in Mid-point Rule. Note error ratio $\to 2^{-2}$ (Δx^2)

No. intervals	No. $f(x)$	Simpson's Rule	Error Ratio: e_{2n}/e_n
1	3	0.0943951023	0.0483
2	5	0.00455975498	0.05903
4	9	0.000269169948	0.06164
8	17	0.0000165910479	0.06228
16	33	0.00000103336941	0.06245
32	65	0.0000000645300022	0.06249
64	129	0.00000000403225719	0.0625

128	257	0.000000000252001974	0.0625
256	513	0.0000000000157500679	0.0624
512	1025	0.00000000000982769421	

Table 4: Error in Simpson's Rule. Note error ratio → 2^{-4} (Δx^4)

No. intervals	No. $f(x)$	Gauss Quadrature	Error Ratio: e_{2n}/e_n
1	2	-0.0641804253	0.0476
2	4	-0.00305477319	0.05881
4	8	-0.000179666460	0.06158
8	16	-0.0000110640837	0.06227
16	32	-0.000000688965642	0.06244
32	64	-0.0000000430208237	0.06249
64	128	-0.000000000268818500	0.0625
128	256	-0.000000000168002278	0.06249
256	512	-0.000000000104984909	0.06248
512	1024	-0.00000000000655919762	

Table 5: Error in Gauss Quadrature. Note error ratio → 2^{-4} (Δx^4)

No. intervals	No. $f(x)$	Gauss Quadrature - three point	Error Ratio: e_{2n}/e_n
1	3	0.001388913607743625	0.01169
2	6	0.00001624311099668319	0.01464
4	12	0.000000237821995874298 9	0.01538
8	24	0.000000003657474767493341	0.01556
16	48	0.000000000056922910829371 18	0.0156
32	96	0.000000000000888178419700125 2	0.016
64	192	0.000000000000014210854715202 00	-0.01563
128	384	-0.000000000000000022204460492503 13	1
256	768	-0.000000000000000022204460492503 13	

Table 6: Error in Three point Gauss Quadrature. Note error ratio → 2^{-6} (Δx^6)

No. Intervals	Trapezium Rule	Midpoint Rule	Simpson's Rule	Gauss Quadrature	3 Pnt Gauss Quadrature
1	-2.0000E+00	1.1418E+00	9.4395E-02	-6.4180E-02	1.3889E-03
2	-4.2920E-01	2.2144E-01	4.5597E-03	-3.0547E-03	1.6243E-05
4	-1.0388E-01	5.2344E-02	2.6916E-04	-1.7966E-04	2.3782E-07
8	-2.5768E-02	1.2909E-02	1.6591E-05	-1.1064E-05	3.6574E-09
16	-6.4296E-03	3.2163E-03	1.0333E-06	-6.8896E-07	5.6922E-11
32	-1.6066E-03	8.0341E-04	6.4530E-08	-4.3020E-08	8.8817E-13
64	-4.0161E-04	2.0081E-04	4.0322E-09	-2.6881E-09	1.4210E-14
128	-1.0039E-04	5.0200E-05	2.5200E-10	-1.6800E-10	-2.2204E-16
256	-2.5099E-05	1.2549E-05	1.5750E-11	-1.0498E-11	-2.2204E-16
512	-6.2749E-06	3.1374E-06	9.8276E-13	-6.5591E-13	
1024	-1.5687E-06	7.8436E-07			
2048	-3.9218E-07	1.9609E-07			
4096	-9.8045E-08	4.9022E-08			
8192	-2.4511E-08	1.2255E-08			
16384	-6.1278E-09	3.0639E-09			
32768	-1.5319E-09	7.6597E-10			
65536	-3.8297E-10	1.9149E-10			
131072	-9.5722E-11	4.7834E-11			
262144	-2.3943E-11	1.1997E-11			
524288	-5.9614E-12	3.0335E-12			
1048576					

Table 7: Error in numerical integration of (86) as a function of the number of subintervals.

5.9.1 Program for numerical integration[*]

Note that this program is written for clarity rather than speed. The number of function evaluations actually computed may be approximately halved for the Trapezium rule and reduced by one third for Simpson's rule if the compound formulations are used. Note also that this example is included for illustrative purposes only. No knowledge of Fortran or any other programming language is required in this course.

```
      PROGRAM Integrat
      REAL*8    x0,x1,Value,Exact,pi
      INTEGER*4 i,j,nx
C=====Functions
      REAL*8    TrapeziumRule
      REAL*8    MidpointRule
      REAL*8    SimpsonsRule
      REAL*8    GaussQuad
C=====Constants
      pi = 2.0*ASIN(1.0D0)
      Exact = 2.0
C=====Limits
      x0 = 0.0
      x1 = pi
C=======================================================================
C=    Trapezium rule                                                   =
C=======================================================================
      WRITE(6,*)
      WRITE(6,*)'Trapezium rule'
      nx = 1
      DO i=1,20
        Value = TrapeziumRule(x0,x1,nx)
        WRITE(6,*)nx,Value,Value - Exact
        nx = 2*nx
      ENDDO
C=======================================================================
C=    Midpoint rule                                                    =
C=======================================================================
      WRITE(6,*)
      WRITE(6,*)'Midpoint rule'
      nx = 1
      DO i=1,20
        Value = MidpointRule(x0,x1,nx)
        WRITE(6,*)nx,Value,Value - Exact
        nx = 2*nx
      ENDDO
C=======================================================================
C=    Simpson's rule                                                   =
C=======================================================================
      WRITE(6,*)
      WRITE(6,*)'Simpson''s rule'
      WRITE(6,*)
      nx = 2
      DO i=1,10
        Value = SimpsonsRule(x0,x1,nx)
        WRITE(6,*)nx,Value,Value - Exact
        nx = 2*nx
      ENDDO
C=======================================================================
C=    Gauss Quadrature                                                 =
```

[*] Not examinable

```
C=====================================================================
      WRITE(6,*)
      WRITE(6,*)'Gauss quadrature'
      nx = 1
      DO i=1,10
        Value = GaussQuad(x0,x1,nx)
        WRITE(6,*)nx,Value,Value - Exact
        nx = 2*nx
      ENDDO
      END

      FUNCTION f(x)
C=====parameters
      REAL*8    x,f
      f = SIN(x)
      RETURN
      END

      REAL*8 FUNCTION TrapeziumRule(x0,x1,nx)
C=====parameters
      INTEGER*4 nx
      REAL*8    x0,x1
C=====functions
      REAL*8    f
C=====local variables
      INTEGER*4 i
      REAL*8    dx,xa,xb,fa,fb,Sum
      dx = (x1 - x0)/DFLOAT(nx)
      Sum = 0.0
      DO i=0,nx-1
        xa = x0 + DFLOAT(i)*dx
        xb = x0 + DFLOAT(i+1)*dx
        fa = f(xa)
        fb = f(xb)
        Sum = Sum + fa + fb
      ENDDO
      Sum = Sum * dx / 2.0
      TrapeziumRule = Sum
      RETURN
      END

      REAL*8 FUNCTION MidpointRule(x0,x1,nx)
C=====parameters
      INTEGER*4 nx
      REAL*8    x0,x1
C=====functions
      REAL*8    f
C=====local variables
      INTEGER*4 i
      REAL*8    dx,xa,fa,Sum
      dx = (x1 - x0)/Dfloat(nx)
      Sum = 0.0
      DO i=0,nx-1
        xa = x0 + (DFLOAT(i)+0.5)*dx
        fa = f(xa)
        Sum = Sum + fa
      ENDDO
      Sum = Sum * dx
      MidpointRule = Sum
      RETURN
      END

      REAL*8 FUNCTION SimpsonsRule(x0,x1,nx)
C=====parameters
      INTEGER*4 nx
```

```
      REAL*8    x0,x1
C=====functions
      REAL*8    f
C=====local variables
      INTEGER*4 i
      REAL*8    dx,xa,xb,xc,fa,fb,fc,Sum
      dx = (x1 - x0)/DFLOAT(nx)
      Sum = 0.0
      DO i=0,nx-1,2
        xa = x0 + DFLOAT(i)*dx
        xb = x0 + DFLOAT(i+1)*dx
        xc = x0 + DFLOAT(i+2)*dx
        fa = f(xa)
        fb = f(xb)
        fc = f(xc)
        Sum = Sum + fa + 4.0*fb + fc
      ENDDO
      Sum = Sum * dx / 3.0
      SimpsonsRule = Sum
      RETURN
      END

      REAL*8 FUNCTION GaussQuad(x0,x1,nx)
C=====parameters
      INTEGER*4 nx
      REAL*8    x0,x1
C=====functions
      REAL*8    f
C=====local variables
      INTEGER*4 i
      REAL*8    dx,xa,xb,fa,fb,Sum,dxl,dxr
      dx = (x1 - x0)/DFLOAT(nx)
      dxl = dx*(0.5D0 - SQRT(3.0D0)/6.0D0)
      dxr = dx*(0.5D0 + SQRT(3.0D0)/6.0D0)
      Sum = 0.0
      DO i=0,nx-1
        xa = x0 + DFLOAT(i)*dx + dxl
        xb = x0 + DFLOAT(i)*dx + dxr
        fa = f(xa)
        fb = f(xb)
        Sum = Sum + fa + fb
      ENDDO
      Sum = Sum * dx / 2.0
      GaussQuad = Sum
      RETURN
      END
```

6 First order ordinary differential equations

Ths section of the course introduces some commonly used methods for determining the numerical solutions of ordinary differential equations. These methods will then be used in §8 as the basis for solving some types of partial differential equations.

6.1 Taylor series

The key idea behind numerical solution of odes is the combination of function values at different points or times to approximate the derivatives in the required equation. The manner in which the function values are combined is determined by the Taylor Series expansion for the point at which the derivative is required. This gives us a *finite difference* approximation to the derivative.

6.2 Finite difference

Consider a first order ode of the form

$$\frac{dy}{dt} = \qquad (87)$$

subject to some boundary/initial condition $y(t=t_0) = c$. The finite difference solution of this equation proceeds by discretising the independent variable t to t_0, $t_0+\Delta t$, $t_0+2\Delta t$, $t_0+3\Delta t$, … We shall denote the exact solution at some $t = t_n = t_0+n\Delta t$ by $y_n = y(t=t_n)$ and our approximate solution by Y_n. We then look to solve

(88)

at each of the points in the domain, where Y'_n is an approximation to dy/dt based on the discrete set of Y_n.

If we take the Taylor Series expansion for the grid points in the neighbourhood of some point $t = t_n$,

...

$Y_{n-2} =$

$Y_{n-1} =$

$Y_n =$

$Y_{n+1} =$

$Y_{n+2} =$

... (89)

we may then take linear combinations of these expansions to obtain an approximation for the derivative Y'_n at $t = t_n$, *viz.*

$$Y'_n \approx \sum_{i=a}^{b} \alpha_i Y_{n+i} \,. \tag{90}$$

The linear combination is chosen so as to eliminate the term in Y_n, requiring

(91)

and, depending on the method, possibly some of the terms of higher order. We shall look at various strategies for choosing α_i in the following sections. As an example, we may select the Y_n and Y_{n-1} values as the basis of an approximation to approximate Y'_n. Adding the appropriate equations, noting that the weightings sum to zero so may be expressed as $\alpha_1 = \alpha$ and $\alpha_2 = -\alpha$, and setting equal to Y'_n, gives

$Y'_n \approx \alpha Y_n - \alpha Y_{n-1}$

$=$ (92)

Equating left- and right-hand sides reveals $\alpha = 1/\Delta t$, so

$$\frac{Y_n - Y_{n-1}}{\Delta t} = Y'_n + \frac{1}{2}\Delta t Y''_n + O\left(\Delta t^2\right). \tag{93}$$

Before looking at this in any further detail, we need to consider the error associated with approximating y_n by Y_n.

6.3 Truncation error

The *global truncation error* at the nth step is the cumulative total of the truncation error at the previous steps and is

$E_n =$ (94)

In contrast, the *local truncation error* for the nth step is

$e_n =$ (95)

where y_n* the exact solution of our differential equation but with the initial condition y_{n-1}*=Y_{n-1}. Note that E_n is not simply

$$\sum_{i=1}^{n} e_n \,, \tag{96}$$

which would give $E_n = O(ne_n)$. It also depends on the *stability* of the method (see section 6.7 for details) and we aim for $E_n = O(e_n)$.

6.4 Euler method

The Euler method is the simplest finite difference scheme to understand and implement. Following on from (93) we approximate the derivative in (88) as

(97)

in our differential equation for Y_n to obtain

(98)

Given the initial/boundary condition $Y_0 = c$, we may obtain Y_1 from $Y_0 + \Delta t f(t_0, Y_0)$, Y_2 from $Y_1 + \Delta t f(t_1, Y_1)$ and so on, marching forwards through time. This process is shown graphically in figure 20.

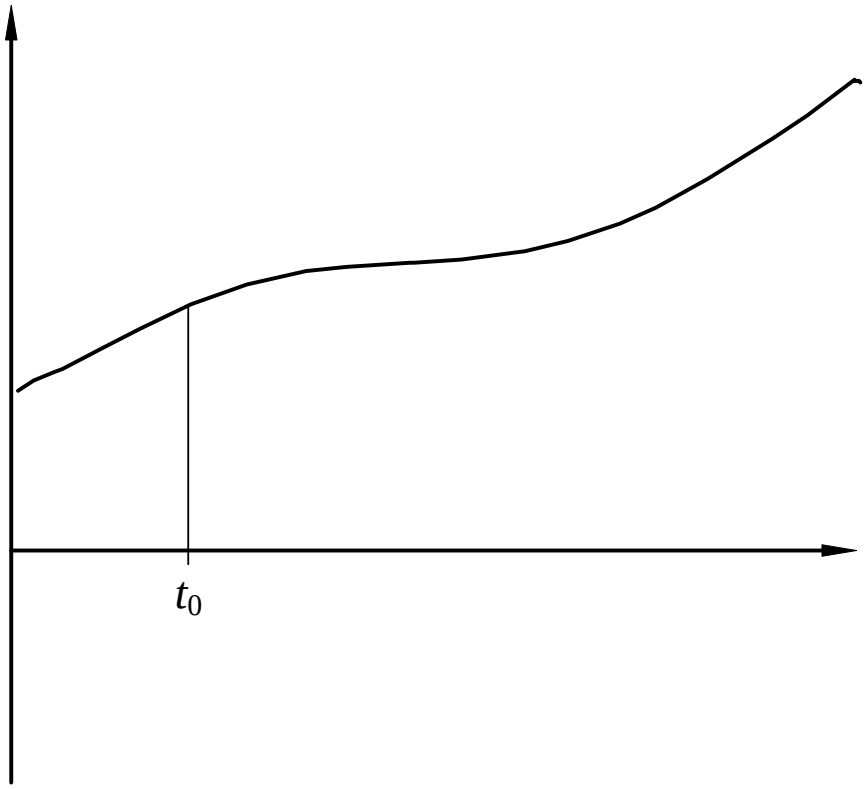


Figure 20: Sketch of the function $y(t)$ (dark line) and the Euler method solution (arrows). Each arrow is tangental to to the solution of (87) passing through the point located at the start of the arrow. Note that this point need not be on the desired $y(t)$ curve.

The Euler method is termed an *explicit* method because we are able to write down an explicit solution for Y_{n+1} in terms of "known" values at t_n.

Comparing the Euler method equation for Y_n in (98) with the the Taylor Series expansion for Y_{n-1} in (89), shows that the $O(\Delta t)$ error in the finite difference approximation (93) becomes an $O(\Delta t^2)$ error in (98). Thus the Euler method is first order accurate and is often referred to as a *first order*

method. Moreover, it can be shown that if $Y_n=y_n+O(\Delta t^2)$, then $Y_{n+1}=y_{n+1}+O(\Delta t^2)$ *provided* the scheme is stable (see section 6.7).

6.5 Implicit methods

The Euler method outlined in the previous section may be summarised by the update formula $Y_{n+1} = g(Y_n,t_n,\Delta t)$. In contrast implicit methods have have Y_{n+1} on both sides: $Y_{n+1} = h(Y_n,Y_{n+1},t_n,\Delta t)$, for example. Such implicit methods are computationally more expensive for a single step than explicit methods, but offer advantages in terms of stability and/or accuracy in many circumstances. Often the computational expense *per step* is more than compensated for by it being possible to take larger steps (see section 6.7).

6.5.1 Backward Euler

The backward Euler method is almost identical to its explicit relative, the only difference being that the derivative Y'_n is approximated by

$$Y'_n \approx \qquad (99)$$

to give the evolution equation

$$Y_{n+1} = \qquad (100)$$

This is shown graphically in figure 21.

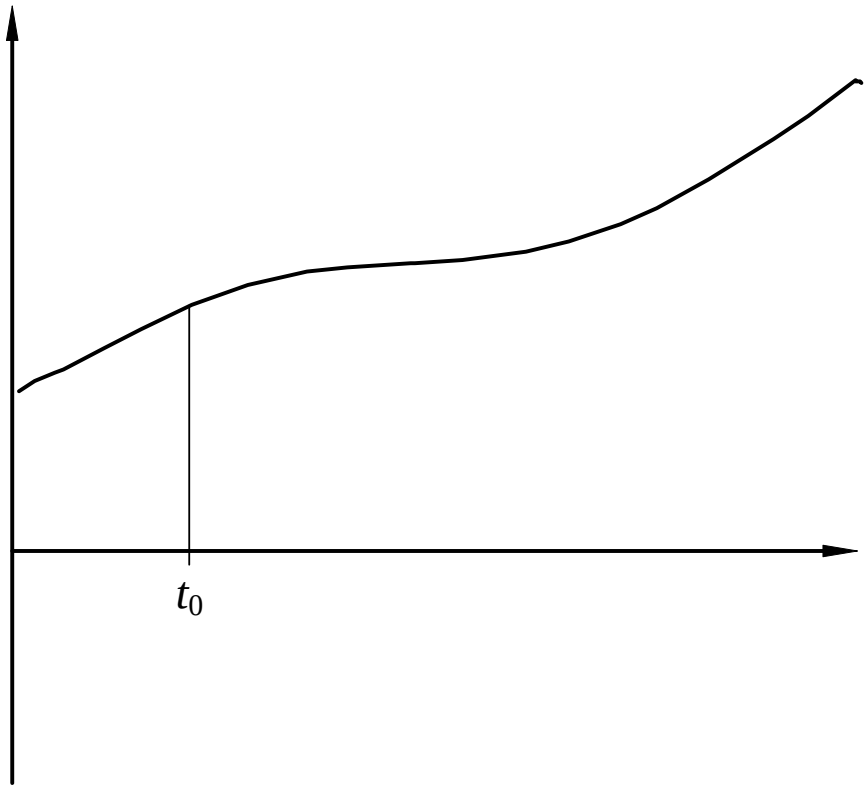


Figure 21: Sketch of the function $y(t)$ (dark line) and the Euler method solution (arrows). Each arrow is tangental to to the solution of (87) passing through the point located at the end of the arrow. Note that this point need not be on the desired $y(t)$ curve.

The dependence of the right-hand side on the variables at t_{n+1} rather than t_n means that it is not, in general possible to give an explicit formula for Y_{n+1} only in terms of Y_n and t_{n+1}. (It may, however, be possible to recover an explicit formula for some functions f.)

As the derivative Y'_n is approximated only to the first order, the Backward Euler method has errors of $O(\Delta t^2)$, exactly as for the Euler method. The solution process will, however, tend to be more stable and hence accurate, especially for *stiff* problems (problems where f' is large). An example of this is shown in figure 22.

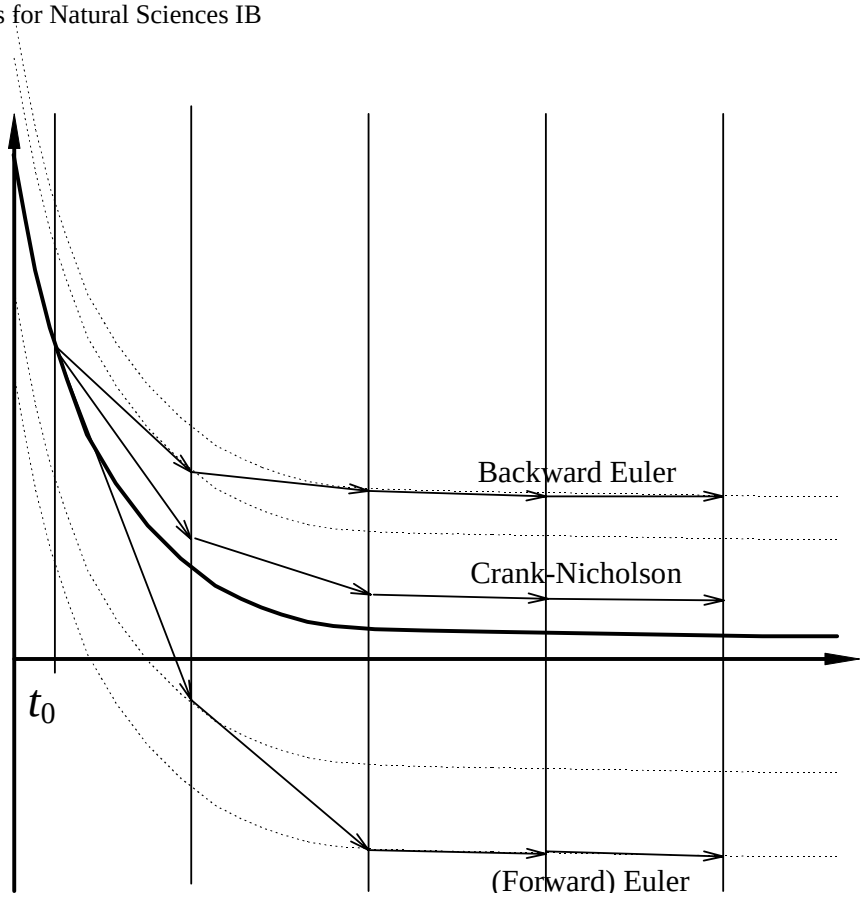


Figure 22: Comparison of ordinary differential equation solvers for a stiff problem.

6.5.2 Richardson extrapolation

The Romberg integration approach (presented in section 5.7) of using two approximations of different step size to construct a more accurate estimate may also be used for numerical solution of ordinary differential equations. Again, if we have the estimate for some time t calculated using a time step Δt, then for both the Euler and Backward Euler methods the approximate solution is related to the true solution by $Y(t,\Delta t) = y(t) + c\Delta t^2$. Similarly an estimate using a step size $\Delta t/2$ will follow $Y(t,\Delta t/2) = y(t) + {}^1/_4 c\Delta t^2$ as $\Delta t \to 0$. Combining these two estimates to try and cancel the $O(\Delta t^2)$ errors gives the improved estimate as

$$Y^{(1)}(t,\Delta t/2) = [4Y(t,\Delta t/2) - Y(t,\Delta t)\,]/3. \qquad (101)$$

The same approach may be applied to higher order methods such as those presented in the following sections. It is generally preferable, however, to utilise a higher order method to start with, the exception being that calculating both $Y(t,\Delta t)$ and $Y(t,\Delta t/2)$ allows the two solutions to be compared and thus the trunctation error estimated.

6.5.3 Crank-Nicholson

If we use *central differences* rather than the *forward difference* of the Euler method or the *backward difference* of the backward Euler, we may obtain a second order method due to cancellation of the terms of $O(\Delta t^2)$. Using the same discretisation of t we obtain

(102)

Substitution into our differential equation for Y_n gives

(103)

The requirement for $f(t_{n+1/2},Y_{n+1/2})$ is then satisfied by a linear interpolation for f between $t_{n-1/2}$ and $t_{n+1/2}$ to obtain

$Y_{n+1} - Y_n =$ (104)

As with the Backward Euler, the method is implicit and it is not, in general, possible to write an explicit expression for Y_{n+1} in terms of Y_n.

Formal proof that the Crank-Nicholson method is second order accurate is slightly more complicated than for the Euler and Backward Euler methods due to the linear interpolation to approximate $f(t_{n+1/2},Y_{n+1/2})$. The overall approach is much the same, however, with a requirement for Taylor Series expansions about t_n::

$$y_{n+1} = y_n + \frac{dy}{dt}\Delta t + \frac{1}{2}\frac{d^2y}{dt^2}\Delta t^2 + O\left(\Delta t^3\right) \tag{105a}$$

$=$

$$f\left(t_{n+1}, y_{n+1}\right) = \tag{105b}$$

Substitution of these into the left- and right-hand sides of equation (104) reveals

$$y_{n+1} - y_n = f_n\Delta t + \frac{1}{2}\left(\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial y}\right)\Delta t^2 + O\left(\Delta t^3\right) \tag{106a}$$

and

$$\begin{aligned}\frac{1}{2}\left(f\left(t_n, y_n\right) + f\left(t_{n+1}, y_{n+1}\right)\right)\Delta t &= \frac{1}{2}\left(f_n + f_n + \left(\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial y}\right)\Delta t + O\left(\Delta t^2\right)\right)\Delta t \\ &= f_n\Delta t + \frac{1}{2}\left(\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial y}\right)\Delta t^2 + O\left(\Delta t^3\right)\end{aligned} \tag{106b}$$

which are equal up to $O(\Delta t^3)$.

6.6 Multistep methods

As an alternative, the accuracy of the approximation to the derivative may be improved by using a linear combination of additional points. By utilising only $Y_{n-s+1}, Y_{n-s+2}, \ldots, Y_n$ we may construct an approximation to the derivatives of orders 1 to s at t_n. For example, if $s = 2$ then

$$Y'_n =$$

$$Y''_n \approx \qquad (107)$$

and so we may construct a second order method from the Taylor series expansion as

$$Y_{n+1} = Y_n + \Delta t Y'_n + \tfrac{1}{2}\Delta t^2 Y''_n$$

$$= \qquad (108)$$

For $s=3$ we also have Y'''_n and so can make use of a second-order one-sided finite difference to approximate $Y''_n = f'_n = (3f_n - 4f_{n-1} + f_{n-2})/2\Delta t$ and include the third order $Y'''_n = f''_n = (f_n - 2f_{n-1} + f_{n-2})/\Delta t^2$ to obtain

$$Y_{n+1} = Y_n + \Delta t Y'_n + {}^1/_2\Delta t^2 Y''_n + {}^1/_6\Delta t^3 Y''_n$$

$$= \qquad (109)$$

These methods are called *Adams-Bashforth* methods. Note that $s = 1$ recovers the Euler method.

Implicit Adams-Bashforth methods are also possible if we use information about f_{n+1} in addition to earlier time steps. The corresponding $s = 2$ method then uses

$$Y'_n = f_n,$$

$$Y''_n \approx$$

$$Y'''_n \approx \qquad (110)$$

to give

$$Y_{n+1} = Y_n + \Delta t Y'_n + {}^1/_2\Delta t^2 Y''_n + {}^1/_6\Delta t^3 Y'''_n$$

$$= Y_n + (1/12)\Delta t(5f_{n+1} + 8f_n - f_{n-1}). \qquad (111)$$

This family of implicit methods is known as *Adams-Moulton* methods.

6.7 Stability

The stability of a method can be even more important than its accuracy as measured by the order of the truncation error. Suppose we are solving

(112)

for some complex λ. The exact solution is bounded (*i.e.* does not increase without limit) provided $\mathrm{Re}\lambda \leq 0$. Substituting this into the Euler method shows

$$Y_{n+1} = Y_n + \Delta t\, f(t_n, Y_n)$$

$$=$$

(113)

If Y_n is to remain bounded for increasing n and given $\mathrm{Re}\lambda < 0$ we require

(114)

If we choose a time step Δt which does not satisfy (114) then Y_n will increase without limit. This condition (114) on Δt is very restrictive if $\lambda << 0$ as it demonstrates the Euler method must use very small time steps $\Delta t < 2|\lambda|^{-1}$ if the solution is to converge on $y = 0$.

The reason why we consider the behaviour of equation (112) is that it is a model for the behaviour of small errors. Suppose that at some stage during the solution process our approximate solution is

$$y_\$ = \qquad (115)$$

where ε is the (small) error. Substituting this into our differential equation of the form $y' = f(t,y)$ and using a Taylor Series expansion gives

$$\frac{dy_\$}{dt} = \frac{dy}{dt} + \frac{d\varepsilon}{dt}$$

$$= f\left(t, y_\$\right)$$

$$=$$

(116)

Thus, to the leading order, the error obeys an equation of the form given by (112), with $\lambda = \partial f/\partial y$. As it is desirable for errors to decrease (and thus the solution remain stable) rather than increase (and the solution be unstable), the limit on the time step suggested by (114) applies for the application of the Euler method to any ordinary differential equation. A consequence of the decay of errors present at one time step as the solution process proceeds is that memory of a particular time step's contribution to the global truncation error decays as the solution advances through time. Thus the global truncation error is dominated by the local trunction error(s) of the most recent step(s) and $O(E_n) = O(e_n)$.

In comparison, solution of (112) by the Backward Euler method

$$Y_{n+1} = Y_n + \Delta t\, f(t_{n+1}, Y_{n+1})$$

$$= \qquad (117)$$

can be rearranged for Y_{n+1} and

$$Y_{n+1} = \qquad (118)$$

which will be stable provided

$$\qquad (119)$$

For $\mathrm{Re}\lambda \leq 0$ this is always satisfied and so the Backward Euler method is unconditionally stable.

The Crank-Nicholson method may be analysed in a similar fashion with

$$(1-\lambda\Delta t/2)Y_{n+1} = (1+\lambda\Delta t/2)Y_n, \qquad (120)$$

to arrive at

$$Y_{n+1} = [(1+\lambda\Delta t/2)/(1-\lambda\Delta t/2)]^{n+1}\, Y_0, \qquad (121)$$

with the magnitude of the term in square brackets always less than unity for $\mathrm{Re}\lambda < 0$. Thus, like Backward Euler, Crank-Nicholson is unconditionally stable.

In general, explicit methods require less computation per step, but are only conditionally stable and so may require far smaller step sizes than an implicit method of nominally the same order.

6.8 Predictor-corrector methods

One approach to using an implicit method is to use a direct iteration to solve the implicit equation. For example, the Backward Euler method has the form

$$Y_{n+1} = \qquad (122)$$

Applying the theory for direct iteration outlined in §3.6, we may write

$$Y_{n+1,i} = \qquad (123)$$

setting $Y_{n+1,0} = Y_n$, and iterate over i until the solution has converged. From (21) we know it will converge if

$$\qquad (124)$$

and the solution will gradually approach the Backward Euler solution. Note that this convergence criterion is more stringent than the stability criterion established in §6.7!

What we are effectively doing here is making a *prediction* with the Euler method (our equation for $Y_{n+1,1}$ is just the Euler method), then multiple *corrections* to this to get the solution to converge on the Backward Euler method. A similar strategy may be applied to the Crank-Nicholson method to obtain

$$Y_{n+1,i} = \qquad (125)$$

requiring $\frac{1}{2}\,\Delta t\,\partial f/\partial Y < 1$ for convergence. Once convergence is achieved, we are left with a second-order implicit scheme. However, it may not be necessary or desirable to carry on iterating until convergence is achieved. Moreover, the condition for convergence using (125) imposes the same restrictions upon Δt as stability of the Euler method. There are other reasons, however, for taking this approach.

The above examples, if iterated only a finite number of times rather than until convergence is achieved, belong to a class of methods known as predictor-corrector methods. Predictor-corrector methods try to combine the advantages of the simplicity of explicit methods with the improved accuracy of implicit methods. They achieve this by using an explicit method to *predict* the solution $Y_{n+1}^{(p)}$ at t_{n+1} and then utilise $f(t_{n+1},Y_{n+1}^{(p)})$ as an approximation to $f(t_{n+1},Y_{n+1})$ to *correct* this prediction using something similar to an implicit step.

6.8.1 Improved Euler method

The simplest of these methods combines the Euler method as the predictor

$$Y_{n+1}^{(1)} = \qquad (126)$$

and then the Backward Euler to give the corrector

$$Y_{n+1}^{(2)} = \qquad (127)$$

The final solution is the mean of these:

$$Y_{n+1} = \qquad (128)$$

To understand the stability of this method we again use the $y' = \lambda y$ so that the three steps described by equations (126) to (128) become

$$Y_{n+1}^{(1)} = Y_n + \lambda\Delta t Y_n, \qquad (129a)$$

$$Y_{n+1}^{(2)} = Y_n + \lambda\Delta t Y_{n+1}^{(1)}$$

$$=$$

$$=, \qquad (129b)$$

$$Y_{n+1} = (Y_{n+1}^{(1)} + Y_{n+1}^{(2)})/2$$

$$=$$

$$=$$

$$= \qquad (129c)$$

Stability requires $|1 + \lambda\Delta t + {}^1/_2\lambda^2\Delta t^2| < 1$ (for $\mathrm{Re}\lambda < 0$) which in turn restricts $\Delta t < 2|\lambda|^{-1}$. Thus the stability of this method, commonly known as the Improved Euler method, is identical to the Euler method. It is also the same as the criterion for the iterative Crank-Nicholson method given in (125) to converge. This is not surprising as it is limited by the stability of the initial predictive step. The accuracy of the method is, however, second order as may be seen by comparison of (129c) with the Taylor Series expansion.

6.8.2 Runge-Kutta methods

The Improved Euler method is the simplest of a family of similar predictor corrector methods following the form of a single *predictor* step and one or more *corrector* steps. The corrector step may be repeated a fixed number of times, or until the estimate for Y_{n+1} converges to some tolerance.

One subgroup of this family are the Runge-Kutta methods which use a fixed number of corrector steps. The Improved Euler method is the simplest of this subgroup. Perhaps the most widely used of these is the fourth order method:

$$Y_{n+1} - Y_n = k = \Delta t\, f(t,Y)$$

$$k^{(1)} = \Delta t f(t_n,Y_n)\,, \qquad (130a)$$

$$k^{(2)} = \Delta t f(t_n + {}^1/_2\Delta t\,, Y_n + \tfrac{1}{2}k^{(1)})\,, \qquad (130b)$$

$$k^{(3)} = \Delta t f(t_n + {}^1/_2\Delta t\,, Y_n + \tfrac{1}{2}k^{(2)})\,, \qquad (130c)$$

$$k^{(4)} = \Delta t f(t_n + \Delta t\,, Y_n + k^{(3)})\,, \qquad (130d)$$

$$Y_{n+1} = Y_n + (k^{(1)} + 2k^{(2)} + 2k^{(3)} + k^{(4)})/6. \qquad (130e)$$

In order to analyse this we need to construct Taylor-Series expansions for $k^{(2)} = \Delta t f(t_n + {}^1/_2\Delta t, Y_n + {}^1/_2 k^{(1)}) = \Delta t[f(t_n,Y_n) + (\Delta t/2)(\partial f/\partial t + f\partial f/\partial y)]$, and similarly for $k^{(3)}$ and $k^{(4)}$. This is then compared with a full Taylor-Series expansion for Y_{n+1} up to fourth order. To achieve this we require

$$Y'' = df/dt$$

$$= \partial f/\partial t + \partial y/\partial t\, \partial f/\partial y$$

$$= \qquad (131)$$

$$Y'' = d^2f/dt^2$$

$$= \partial^2 f/\partial t^2 + 2f\,\partial^2 f/\partial t\partial y + \partial f/\partial t\,\partial f/\partial y + f^2\,\partial^2 f/\partial y^2 + f\,(\partial f/\partial y)^2, \quad (132)$$

$$Y'''' =$$

and similarly for Y''''. All terms up to order Δt^4 can be shown to match, with the error coming in at Δt^5.

7 Higher order ordinary differential equations

In this section we look at how to solve higher order ordinary differential equations. Some of the techniques discussed here are based on the techniques introduced in §6, and some rely on a combination of this material and the linear algebra of §4.

7.1 Initial value problems

The discussion so far has been for first order ordinary differential equations. All the methods given may be applied to higher ordinary differential equations, provided it is possible to write an explicit expression for the highest order derivative and the system has a complete set of initial conditions. Consider some equation

(133)

where at $t = t_0$ we know the values of y, dy/dt, d^2y/dt^2, …, $d^{n-1}y/dt^{n-1}$. By writing $x_0=y$, $x_1=\mathrm{d}y/\mathrm{d}t$, $x_2=\mathrm{d}^2y/\mathrm{d}t^2$, …, $x_{n-1}=\mathrm{d}^{n-1}y/\mathrm{d}t^{n-1}$, we may express this as the system of equations

$x_0' =$

$x_1' =$

$x_2' =$

. . . .

$x_{n-2}' =$

$x_{n-1}' =$ (134)

and use the standard methods for updating each x_i for some t_{n+1} before proceeding to the next time step. A decision needs to be made as to whether the values of x_i for t_n or t_{n+1} are to be used on the right hand side of the equation for x_{n-1}'. This decision may affect the order and convergence of the method. Detailed analysis may be undertaken in a manner similar to that for the first order ordinary differential equations.

7.2 Boundary value problems

For second (and higher) order odes, two (or more) initial/boundary conditions are required. If these two conditions do not correspond to the same point in time/space, then the simple extension of the first order methods outlined in section 7.1 can not be applied without modification. There are two relatively simple approaches to solve such equations.

7.2.1 Shooting method

Suppose we are solving a second order equation of the form $y'' = f(t,y,y')$ subject to $y(0) = c_0$ and $y(1) = c_1$. With the shooting method we apply the $y(0)=c_0$ boundary condition and make some *guess* that $y'(0) = \alpha_0$. This gives us two *initial* conditions so that we may apply the simple time-stepping methods already discussed in section 7.1. The calculation proceeds until we have a value for $y(1)$. If this does not satisfy $y(1) = c_1$ to some acceptable tolerance, we revise our guess for $y'(0)$ to some value α_1, say, and repeat the time integration to obtain an new value for $y(1)$. This process continues until we *hit* $y(1)=c_1$ to the acceptable tolerance. The number of iterations which will need to be made in order to achieve an acceptable tolerance will depend on how good the refinement algorithm for α is. We may use the root finding methods discussed in section 3 to undertake this refinement.

The same approach can be applied to higher order ordinary differential equations. For a system of order n with m boundary conditions at $t = t_0$ and $n–m$ boundary conditions at $t = t_1$, we will require guesses for $n–m$ initial conditions. The computational cost of refining these $n–m$ guesses will rapidly become large as the dimensions of the space increase.

7.2.2 Linear equations

The alternative is to rewrite the equations using a finite difference approximation with step size $\Delta t = (t_1–t_0)/N$ to produce a system of $N+1$ simultaneous equations. Consider the second order linear system

(135)

with boundary conditions $y(t_0) = \alpha$ and $y(t_1) + y'(t_1) = \beta$. If we use the central difference approximations

$y'_i =$ (136a)

$y''_i =$ (136b)

Substitution into (135) gives

(137)

and we may write the boundary condition at $t = t_0$ as

(138)

For $t = t_1$ we must express the derivative in the boundary condition in finite difference form using points that exist in our mesh. The obvious thing to do is to take the backward difference so that the boundary condition becomes

(139)

but as we shall see, this choice is not ideal.

The above strategy leads to the system of equations

…

(140)

This tridiagonal system may be readily solved using the method discussed in section 4.5.

There is one problem with the scheme outlined above: while the Y'' and Y' derivatives within the domain are second order accurate, the boundary condition at $t = t_1$ is imposed only as a first-order approximation to the derivative. We have two ways in which we may improve the accuracy of this boundary condition.

The first way is the most obvious: make use of the Y_{n-2} value to construct a second-order approximation to Y'' so that we have

(141)

so that the last equation in our system becomes

(142)

The problem with this is that we then loose our simple tridiagonal system.

The second approach is to artificially increase the size of the domain by one mesh point and impose the boundary condition *inside* the domain. The additional mesh point is often referred to as a *dummy mesh point* or *dummy cell*. This leads to the last three equations being

$$(1+{}^1\!/_2 a\Delta t)Y_{n-2} + (b\Delta t^2-2)Y_{n-1} + (1-{}^1\!/_2 a\Delta t)Y_n = c\Delta t^2,$$

(143)

The attraction of this approach is that we maintain our simple tri-diagonal system.

Clearly the same tricks can be applied if we have a boundary condition on the derivative at t_0.

Higher order linear equations may be catered for in a similar manner and the matrix representing the system of equations will remain banded, but not as sparse as tridiagonal. The solution may be undertaken using the modified LU decomposition introduced in section 4.4.

Nonlinear equations may also be solved using this approach, but will require an iterative solution of the resulting matrix system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ as the matrix $\mathbf{A}$ will be a function of $\boldsymbol{x}$. In most circumstances this is most efficiently achieved through a Newton-Raphson algorithm, similar in principle to that introduced in section 3.4 but where a system of linear equations requires solution for each iteration.

7.3 Other considerations*

Not examinable

*7.3.1 Truncation error**

Not examinable

*7.3.2 Error and step control**

Not examinable

* Not examinable

* Not examinable

* Not examinable

8 Partial differential equations

In this section we shall concentrate on the numerical solution of second order linear partial differential equations.

8.1 Laplace equation

Consider the Laplace equation in two dimensions

(144)

in some rectangular domain described by x in $[x_0,x_1]$, y in $[y_0,y_1]$. Suppose we discretise the solution φ onto a $m+1$ by $n+1$ rectangular grid (or mesh) given by

$x_i =$ (145a)

$y_j =$ (145b)

where $i=0,m$, $j=0,n$. The mesh spacing is $\Delta x = (x_1-x_0)/m$ and $\Delta y = (y_1-y_0)/n$. Let

$\varphi_{ij} =$ (146)

be the exact solution at the mesh point i,j, and $\Phi_{ij} \approx \varphi_{ij}$ be the approximate solution at that mesh point.

By considering the Taylor Series expansion for φ about some mesh point i,j,

$\varphi_{i+1,j} =$ (147a)

$\varphi_{i-1,j} =$ (147b)

$\varphi_{i,j+1} =$ (147c)

$\varphi_{i,j-1} =$ (147d)

it is clear that we may approximate $\partial^2\varphi/\partial x^2$ and $\partial^2\varphi/\partial y^2$ to the second order using the four adjacent mesh points to obtain the finite difference approximation

(148)

for the internal points $0<i<m$, $0<j<n$. In addition to this we will have either Dirichlet, von Neumann or mixed boundary conditions to specify the boundary values of φ_{ij}. The system of linear equations described by (148) in combination with the boundary conditions may be solved in a variety of ways.

8.1.1 Direct solution

Provided the boundary conditions are linear in φ, our finite difference approximation is itself linear and the resulting system of equations may be written as

(149)

with $(m+1)(n+1)$ equations. This system may be solved directly using Gauss Elimination as discussed in section 4.1. However, this approach may be feasible if the total number of mesh points $(m+1)(n+1)$ required is relatively small, but as the matrix **A** used to represent the complete system will have $[(m+1)(n+1)]^2$ elements, the storage and computational cost of such a solution will become prohibitive even for relatively modest m and n.

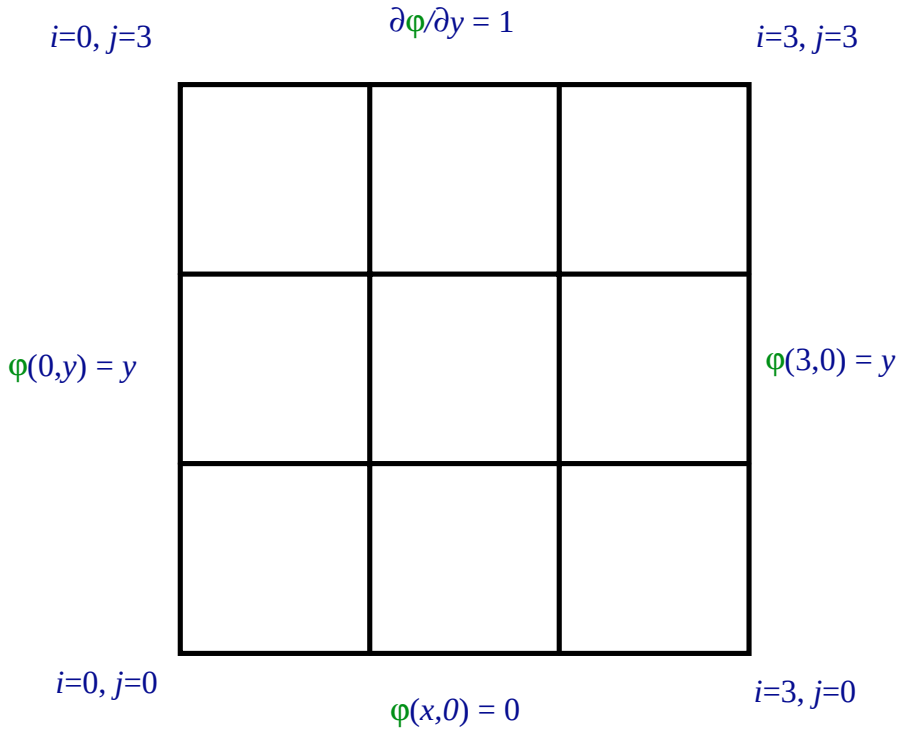


Figure 23: Sketch of domain for Laplace equation. The linear system for this domain is given in (150).

The structure of the system ensures **A** is relatively sparse, consisting of a tridiagonal core with one nonzero diagonal above and another below this. These nonzero diagonals are offset by either m or n from the leading diagonal. For example, in the domain shown in figure 23, the linear system is

$$\begin{bmatrix}
1 & & & & & & & & & & & & & & & \\
 & 1 & & & & & & & & & & & & & & \\
 & & 1 & & & & & & & & & & & & & \\
 & & & 1 & & & & & & & & & & & & \\
 & & & & 1 & & & & & & & & & & & \\
 & 1 & & & 1 & -4 & 1 & & & 1 & & & & & & \\
 & & 1 & & & 1 & -4 & 1 & & & 1 & & & & & \\
 & & & & & & & 1 & & & & & & & & \\
 & & & & & & & & 1 & & & & & & & \\
 & & & & & 1 & & & 1 & -4 & 1 & & & 1 & & \\
 & & & & & & 1 & & & 1 & -4 & 1 & & & 1 & \\
 & & & & & & & & & & & 1 & & & & \\
 & & & & & & & & -1 & & & & 1 & & & \\
 & & & & & & & & & -1 & & & & 1 & & \\
 & & & & & & & & & & -1 & & & & 1 & \\
 & & & & & & & & & & & -1 & & & & 1
\end{bmatrix}
\begin{pmatrix} \varphi_{00} \\ \varphi_{10} \\ \varphi_{20} \\ \varphi_{30} \\ \varphi_{01} \\ \varphi_{11} \\ \varphi_{21} \\ \varphi_{31} \\ \varphi_{02} \\ \varphi_{12} \\ \varphi_{22} \\ \varphi_{32} \\ \varphi_{03} \\ \varphi_{13} \\ \varphi_{23} \\ \varphi_{33} \end{pmatrix}
=
\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 2 \\ 0 \\ 0 \\ 2 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{150}$$

Provided pivoting (if required) is conducted in such a way that it does not place any nonzero elements outside this band then solution by Gauss Elimination or LU Decomposition will only produce nonzero elements inside this band, substantially reducing the storage and computational requirements (see section 4.4). Careful choice of the order of the matrix elements (*i.e.* by x or by y) may help reduce the size of this matrix so that it need contain only $O(m^3)$ elements for a square domain.

Because of the wide spread need to solve Laplace's and related equations, specialised solvers have been developed for this problem. One of the best of these is Hockney's method for solving $\mathbf{A}\varphi = \boldsymbol{b}$ which may be used to reduce a block tridiagonal matrix (and the corresponding right-hand side) of the form

$$\mathbf{A} = \begin{bmatrix}
\hat{\mathbf{A}} & \mathbf{I} & & & & & & \\
\mathbf{I} & \hat{\mathbf{A}} & \mathbf{I} & & & & & \\
 & \mathbf{I} & \hat{\mathbf{A}} & \mathbf{I} & & & & \\
 & & \mathbf{I} & \hat{\mathbf{A}} & \mathbf{I} & & & \\
 & & & \mathbf{I} & \hat{\mathbf{A}} & \mathbf{I} & & \\
 & & & & \mathbf{I} & \hat{\mathbf{A}} & \ddots & \\
 & & & & & \ddots & \ddots & \mathbf{I} \\
 & & & & & & \mathbf{I} & \hat{\mathbf{A}}
\end{bmatrix}, \tag{151}$$

into a block diagonal matrix of the form

$$\begin{bmatrix} \hat{\mathbf{B}} & & & & & & \\ & \hat{\mathbf{B}} & & & & & \\ & & \hat{\mathbf{B}} & & & & \\ & & & \hat{\mathbf{B}} & & & \\ & & & & \hat{\mathbf{B}} & & \\ & & & & & \hat{\mathbf{B}} & \ddots \\ & & & & & \ddots & \ddots \\ & & & & & & \hat{\mathbf{B}} \end{bmatrix}, \tag{152}$$

where $\hat{\mathbf{A}}$ and $\hat{\mathbf{B}}$ are themselves block tridiagonal matrices and $\mathbf{I}$ is an identiy matrix.. This process may be performed iteratively to reduce an n dimensional finite difference approximation to Laplace's equation to a tridiagonal system of equations with n–1 applications. The computational cost is $O(p \log p)$, where p is the total number of mesh points. The main drawback of this method is that the boundary conditions must be able to be cast into the block tridiagonal format.

8.1.2 Relaxation

An alternative to direct solution of the finite difference equations is an iterative numerical solution. These iterative methods are often referred to as relaxation methods as an initial *guess* at the solution is allowed to slowly relax towards the true solution, reducing the errors as it does so. There are a variety of approaches with differing complexity and speed. We shall introduce these methods before looking at the basic mathematics behind them.

8.1.2.1 Jacobi

The Jacobi Iteration is the simplest approach. For clarity we consider the special case when $\Delta x = \Delta y$. To find the solution for a two-dimensional Laplace equation simply:

1. Initialise Φ_{ij} to some initial *guess*.
2. Apply the boundary conditions.
3. For each internal mesh point set

$$\Phi^*_{ij} = \tag{153}$$

4. Replace old solution Φ with new estimate Φ^*.
5. If solution does not satisfy tolerance, repeat from step 2.

$\Phi^*_{ij} =$ (154)

An example of this algorithm is given in the table below where $\varphi = 0$ on the boundaries and the initial guess is $\varphi = 1$ in the interior (clearly the answer is $\varphi = 0$ everywhere). The first four iterations are labelled *a*, *b*, *c* and *d*.

The coefficients in (154) (here all 1/4) used to calculate the refined estimate is often referred to as the *stencil* or *template*. Higher order approximations may be obtained by simply employing a stencil which utilises more points. Other equations (*e.g.* the bi-harmonic equation, $\nabla^4\Psi = 0$) may be solved by introducing a stencil appropriate to that equation.

While very simple and cheap per iteration, the Jacobi Iteration is very slow to converge, especially for larger grids. Corrections to errors in the estimate Φ_{ij} diffuse only slowly from the boundaries taking $O(\max(m,n))$ iterations to diffuse across the entire mesh.

8.1.2.2 Gauss-Seidel

The Gauss-Seidel Iteration is very similar to the Jacobi Iteration, the only difference being that the new estimate Φ^*_{ij} is returned to the solution Φ_{ij} as soon as it is completed, allowing it to be used immediately rather than deferring its use to the next iteration. The advantages of this are:

- Less memory required (there is no need to store Φ^*).
- Faster convergence (although still relatively slow).

On the other hand, the method is less amenable to vectorisation as, for a given iteration, the new estimate of one mesh point is dependent on the new estimates for those already scanned.

An example of the first iteration of the Gauss-Seidel approach using the same initial guess as our Jacobi iteration in §8.1.2.1 is given below, with the iteration starting at the bottom left and moving across then up.

Clearly in this example the solution is converging on zero more rapidly, but the symmetry that existed in the initial guess has been lost.

8.1.2.3 Red-Black ordering

A variant on the Gauss-Seidel Iteration is obtained by updating the solution Φ_{ij} in two passes rather than one. If we consider the mesh points as a chess board, then the white squares would be updated on the first pass and the black squares on the second pass. The advantages

- No interdependence of the solution updates within a single pass aids vectorisation.
- Faster convergence at low wave numbers.

An example, based on those in the previous two sections, is given below. Note that the cells labelled “Red” are updated on the first pass and those labelled “Black” on the second pass. Unlike the Gauss-Seidel, the order in which the cells are updated within one pass does not matter.

In this example the solution is converging faster than the Jacobi iteration and at a comparable rate to Gauss-Seidel, yet is maintaining the symmetries which existed in the initial guess. For many problems, particularly those looking at instabilities, the maintenance of symmetries is important.

8.1.2.4 Successive Over Relaxation (SOR)

It has been found that the errors in the solution obtained by any of the three preceding methods decrease only slowly and often decrease in a monotonic manner. Hence, rather than setting

$$\Phi^*_{ij} = (\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1})/4,$$

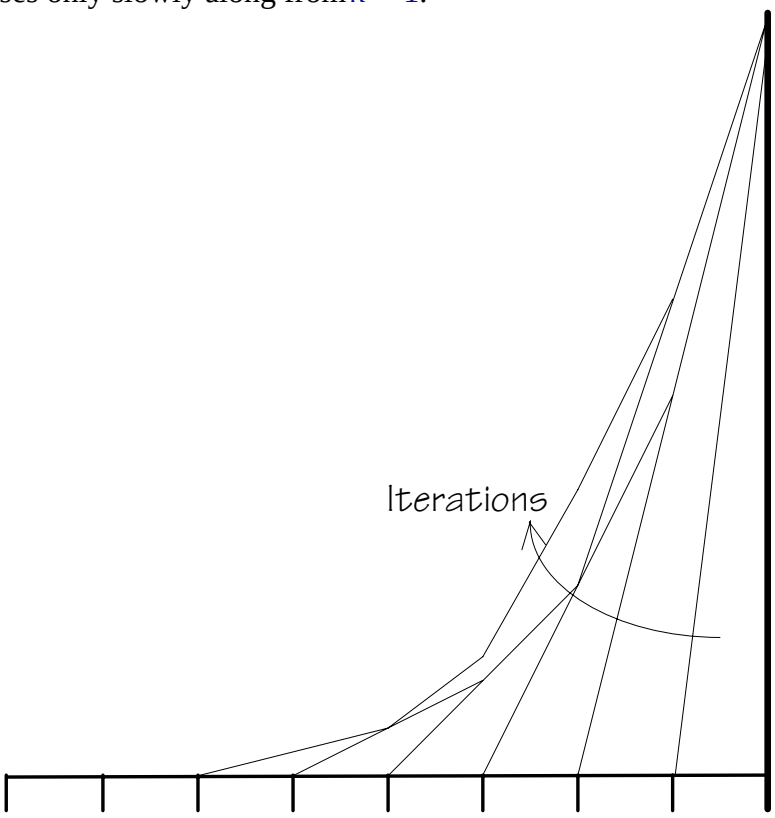
for each internal mesh point, we use

$\Phi^*_{ij} =$ (155)

for some value σ. The *optimal* value of σ will depend on the problem being solved and may vary as the iteration process converges. Typically, however, a value of around 1.2 to 1.4 produces good results. In some special cases it is possible to determine an optimal value analytically.

8.1.3 Multigrid[*]

The big problem with relaxation methods is their slow convergence. If $\sigma = 1$ then application of the stencil removes all the error in the solution at the wave length of the mesh for that point, but has little impact on larger wave lengths. This may be seen if we consider the one-dimensional equation $d^2\varphi/dx^2 = 0$ subject to $\varphi(x=0) = 0$ and $\varphi(x=1) = 1$. Suppose our initial guess for the iterative solution is that $\Phi_i = 0$ for all internal mesh points. With the Jacobi Iteration the correction to the internal points diffuses only slowly along from $x = 1$.



Multigrid methods try to improve the rate of convergence by considering the problem of a hierarchy of grids. The larger wave length errors in the solution are dissipated on a coarser grid while the shorter wave length errors are dissipated on a finer grid. for the example considered above, the solution would converge in one complete Jacobi multigrid iteration, compared with the slow asymptotic convergence above.

For linear problems, the basic multigrid algorithm for one complete iteration may be described as

1. Select the initial finest grid resolution $p=P_0$ and set $\boldsymbol{b}^{(p)} = 0$ and make some initial guess at the solution $\Phi^{(p)}$
2. If at coarsest resolution ($p=0$) then solve $\mathbf{A}^{(p)}\Phi^{(p)}=\boldsymbol{b}^{(p)}$ *exactly* and jump to step 7
3. Relax the solution at the current grid resolution, applying boundary conditions

[*] Not examinable

4. Calculate the error $\boldsymbol{r} = \mathbf{A}\Phi^{(p)} - \boldsymbol{b}^{(p)}$
5. Coarsen the error $\boldsymbol{b}^{(p-1)} \leftarrow \boldsymbol{r}$ to the next coarser grid and decrement p
6. Repeat from step 2
7. Refine the correction to the next finest grid $\Phi^{(p+1)} = \Phi^{(p+1)} + \alpha\Phi^{(p)}$ and increment p
8. Relax the solution at the current grid resolution, applying boundary conditions
9. If not at current finest grid (P_0), repeat from step 7
10. If not at final desired grid, increment P_0 and repeat from step 7
11. If not converged, repeat from step 2.

Typically the relaxation steps will be performed using Successive Over Relaxtion with Red-Black ordering and some relaxation coefficient σ. The hierarchy of grids is normally chosen to differ in dimensions by a factor of 2 in each direction. The factor α is typically less than unity and effectively damps possible instabilities in the convergence. The refining of the correction to a finer grid will be achieved by (bi-)linear or higher order interpolation, and the coarsening may simply be by sub-sampling or averaging the error vector $\boldsymbol{r}$.

It has been found that the number of iterations required to reach a given level of convergence is more or less independent of the number of mesh points. As the number of operations per complete iteration for n mesh points is $O(n)+O(n/2^d)+ +O(n/2^{2d})+\ldots$, where d is the number of dimensions in the problem, then it can be seen that the Multigrid method may often be faster than a direction solution (which will require $O(n^3)$, $O(n^2)$ or $O(n \log n)$ operations, depending on the method used). This is particularly true if n is large or there are a large number of dimensions in the problem. For small problems, the coefficient in front of the n for the Multigrid solution may be relatively large so that direct solution may be faster.

A further advantage of Multigrid and other iterative methods when compared with direct solution, is that irregular shaped domains or complex boundary conditions are implemented more easily. The difficulty with this for the Multigrid method is that care must be taken in order to ensure consistent boundary conditions in the embedded problems.

8.1.4 The mathematics of relaxation[*]

In principle, relaxation methods which are the basis of the Jacobi, Gauss-Seidel, Successive Over Relaxation and Multigrid methods may be applied to any system of linear equations to interatively improve an approximation to the exact solution. The basis for this is identical to the Direct Iteration method described in section 3.6. We start by writing the vector function

$$\boldsymbol{f}(\boldsymbol{x}) = \mathbf{A}\boldsymbol{x} - \boldsymbol{b}, \tag{156}$$

and search for the vector of roots to $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{0}$ by writing

$$\boldsymbol{x}_{n+1} = \boldsymbol{g}(\boldsymbol{x}_n), \tag{157}$$

where

$$\boldsymbol{g}(\boldsymbol{x}) = \mathbf{D}^{-1}\{[\mathbf{A}+\mathbf{D}]\boldsymbol{x} - \boldsymbol{b}\}, \tag{158}$$

[*] Not examinable

with **D** a diagonal matrix (zero for all off-diagonal elements) which may be chosen arbitrarily. We may analyse this system by following our earlier analysis for the Direct Iteration method (section 3.6). Let us assume the exact solution is $\boldsymbol{x}^* = \boldsymbol{g}(\boldsymbol{x}^*)$, then

$$\begin{aligned}\varepsilon_{n+1} &= \boldsymbol{x}_{n+1} - \boldsymbol{x}^* \\ &= \mathbf{D}^{-1}\{[\mathbf{A}+\mathbf{D}]\boldsymbol{x}_n - \boldsymbol{b}\} - \mathbf{D}^{-1}\{[\mathbf{A}+\mathbf{D}]\boldsymbol{x}^* - \boldsymbol{b}\} \\ &= \mathbf{D}^{-1}[\mathbf{A}+\mathbf{D}](\boldsymbol{x}_n - \boldsymbol{x}^*) \\ &= \mathbf{D}^{-1}[\mathbf{A}+\mathbf{D}]\varepsilon_n \\ &= \{\mathbf{D}^{-1}[\mathbf{A}+\mathbf{D}]\}^{n+1}\,\varepsilon_0. \end{aligned} \tag{159}$$

From this it is clear that convergence will be linear and requires

$$\|\varepsilon_{n+1}\| = \|\mathbf{B}\varepsilon_n\| < \|\varepsilon_n\|, \tag{160}$$

where $\mathbf{B} = \mathbf{D}^{-1}[\mathbf{A}+\mathbf{D}]$ for some suitable norm. As any error vector ε_n may be written as a linear combination of the eigen vectors of our matrix **B**, it is sufficient for us to consider the eigen value problem

$$\mathbf{B}\varepsilon_n = \lambda\varepsilon_n, \tag{161}$$

and require $\max(|\lambda|)$ to be less than unity. In the asymptotic limit, the smaller the magnitude of this maximum eigen value the more rapid the convergence. The convergence remains, however, linear.

Since we have the ability to choose the diagonal matrix **D**, and since it is the eigen values of $\mathbf{B} = \mathbf{D}^{-1}[\mathbf{A}+\mathbf{D}]$ rather than **A** itself which are important, careful choice of **D** can aid the speed at which the method converges. Typically this means selecting **D** so that the diagonal of **B** is small.

8.1.4.1 Jacobi and Gauss-Seidel for Laplace equation*

The structure of the finite difference approximation to Laplace's equation lends itself to these relaxation methods. In one dimension,

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & 1 & -2 & 1 & & \\ & & & 1 & -2 & 1 & \\ & & & & \ddots & \ddots & \ddots \\ & & & & & 1 & -2 \end{bmatrix} \tag{162}$$

and both Jacobi and Gauss-Seidel iterations take **D** as $2\mathbf{I}$ (**I** is the identity matrix) on the diagonal to give $\mathbf{B} = \mathbf{D}^{-1}[\mathbf{A}+\mathbf{D}]$ as

* Not examinable

$$\mathbf{B} = \begin{bmatrix} 0 & 1/2 & & & & & \\ 1/2 & 0 & 1/2 & & & & \\ & 1/2 & 0 & 1/2 & & & \\ & & 1/2 & 0 & 1/2 & & \\ & & & 1/2 & 0 & 1/2 & \\ & & & & \ddots & \ddots & \ddots \\ & & & & & 1/2 & 0 \end{bmatrix} \tag{163}$$

The eigen values λ of this matrix are given by the roots of

$$\det(\mathbf{B}-\lambda\mathbf{I}) = 0. \tag{164}$$

In this case the determinant may be obtained using the recurrence relation

$$\det(\mathbf{B}-\lambda)_{(n)} = -\lambda \det(\mathbf{B}-\lambda)_{(n-1)} - \tfrac{1}{4} \det(\mathbf{B}-\lambda)_{(n-2)} , \tag{165}$$

where the subscript gives the size of the matrix **B**. From this we may see

$$\begin{gathered} \det(\mathbf{B}-\lambda)_{(1)} = -\lambda , \\ \det(\mathbf{B}-\lambda)_{(2)} = \lambda^2 - \tfrac{1}{4} , \\ \det(\mathbf{B}-\lambda)_{(3)} = -\lambda^3 + \tfrac{1}{2}\lambda , \\ \det(\mathbf{B}-\lambda)_{(4)} = \lambda^4 - \tfrac{3}{4}\lambda^2 + (1/16) , \\ \det(\mathbf{B}-\lambda)_{(5)} = -\lambda^5 + \lambda^3 - (3/16)\lambda , \\ \det(\mathbf{B}-\lambda)_{(6)} = \lambda^6 - (5/4)\lambda^4 + (3/8)\lambda^2 - (1/64) , \\ \dots \end{gathered} \tag{166}$$

$$\begin{gathered} \lambda_{(1)} = 0 , \\ \lambda^2_{(2)} = 1/4 , \\ \lambda^2_{(3)} = 0, 1/2 , \\ \lambda^2_{(4)} = (3 \pm \sqrt{5})/8 , \\ \lambda^2_{(5)} = 0, 1/4, 3/4 , \\ \dots \end{gathered} \tag{167}$$

It can be shown that for a system of any size following this general form, all the eigen values satisfy $|\lambda| < 1$, thus proving the relaxation method will always converge. As we increase the number of mesh points, the number of eigen values increases and gradually fills up the range $|\lambda| < 1$, with the numerically largest eigen values becoming closer to unity. As a result of $\lambda \to 1$, the convergence of the relaxation method slows considerably for large problems. A similar analysis may be applied to

Laplace's equation in two or more dimensions, although the expressions for the determinant and eigen values is correspondingly more complex.

The large eigen values are responsible for decreasing the error over large distances (many mesh points). The multigrid approach enables the solution to converge using a much smaller system of equations and hence smaller eigen values for the larger distances, bypassing the slow convergence of the basic relaxation method.

8.1.4.2 Successive Over Relaxation for Laplace equation*

The analysis of the Jacobi and Gauss-Seidel iterations may be applied equally well to Successive Over Relaxation. The main difference is that $\mathbf{D} = (2/\sigma)\mathbf{I}$ so that

$$\mathbf{B}_{SOR} = \begin{bmatrix} 1-\sigma & \sigma/2 & & & & & \\ \sigma/2 & 1-\sigma & \sigma/2 & & & & \\ & \sigma/2 & 1-\sigma & \sigma/2 & & & \\ & & \sigma/2 & 1-\sigma & \sigma/2 & & \\ & & & \sigma/2 & 1-\sigma & \sigma/2 & \\ & & & & \ddots & \ddots & \ddots \\ & & & & & \sigma/2 & 1-\sigma \end{bmatrix} \tag{168}$$

and thus

$$\mathbf{B}_{SOR} - \lambda\mathbf{I} = \sigma\left[\mathbf{B}_J - \frac{\lambda+\sigma-1}{\sigma}\mathbf{I}\right] \tag{169}$$

and the corresponding eigen values λ_{SOR} are related to the eigen values λ_J for the Jacobi and Gauss-Seidel methods by

$$\lambda_{SOR} = 1 + \sigma(\lambda_J - 1) \tag{170}$$

Thus if σ is chosen inappropriately, the eigen values of **B** will exceed unity and the relaxation method will diverge. On the otherhand, careful choice of σ will allow the eigen values of **B** to be less than those for Jacobi and Gauss-Seidel, thus increasing the rate of convergence.

8.1.4.3 Other equations*

Relaxation methods may be applied to other differential equations or more general systems of linear equations in a similar manner. As a rule of thumb, the solution will converge if the **A** matrix is diagonally dominant, *i.e.* the numerically largest values occur on the diagonal. If this is not the case, SOR can still be used, but it may be necessary to choose $\sigma < 1$ whereas for Laplace's equation $\sigma \geq 1$ produces a better rate of convergence.

* Not examinable

* Not examinable

8.1.5 FFT*

One of the most common ways of solving Laplace's equation is to take the Fourier transform of the equation to convert it into wave number space and there solve the resulting algebraic equations. This conversion process can be very efficient if the Fast Fourier Transform algorithm is used, allowing a solution to be evaluated with $O(n \log n)$ operations.

In its simplest form the FFT algorithm requires there to be $n = 2^p$ mesh points in the direction(s) to be transformed. The efficiency of the algorithm is achieved by first calculating the transform of pairs of points, then of pairs of transforms, then of pairs of pairs and so on up to the full resolution. The idea is to *divide and conquer!* Details of the FFT algorithm may be found in any standard text.

8.1.6 Boundary elements*

8.1.7 Finite elements*

8.2 Poisson equation

The Poisson equation $\nabla^2\varphi = f(\boldsymbol{x})$ may be treated using the same techniques as Laplace's equation. It is simply necessary to set the right-hand side to f, scaled suitably to reflect any scaling in $\mathbf{A}$.

8.3 Diffusion equation

Consider the two-dimensional diffusion equation,

(171)

subject to $u(x,y,t) = 0$ on the boundaries x=0,1 and y=0,1. Suppose the initial conditions are $u(x,y,t=0) = u_0(x,y)$ and we wish to evaluate the solution for $t > 0$. We shall explore some of the options for achieving this in the following sections.

8.3.1 Semi-discretisation

One of the simplest and most useful approaches is to discretise the equation in space and then solve a system of (coupled) ordinary differential equations in time in order to calculate the solution. Using a square mesh of step size $\Delta x = \Delta y = 1/m$, and taking the diffusivity $D = 1$, we may utilise our earlier approximation for the Laplacian operator (equation (148)) to obtain

* Not examinable

* Not examinable

* Not examinable

$$\frac{\partial u_{i,j}}{\partial t} \approx \tag{172}$$

for the internal points i=1,m–1 and j=1,m–1. On the boundaries (i=0,j), (i=m,j), (i,j=0) and (i,j=m) we simply have u_{ij}=0. If U_{ij} represents our approximation of u at the mesh points x_{ij}, then we must simply solve the $(m-1)^2$ coupled ordinary differential equations

(173)

In principle we may utilise any of the time stepping algorithms discussed in earlier lectures to solve this system. As we shall see, however, care needs to be taken to ensure the method chosen produces a stable solution.

8.3.2 Euler method

Applying the Euler method $Y_{n+1} = Y_n+\Delta t f(Y_n,t_n)$ to our spatially discretised diffusion equation gives

(174)

where the Courant number

$$\mu = \tag{175}$$

describes the size of the time step relative to the spatial discretisation. As we shall see, stability of the solution depends on μ in contrast to an ordinary differential equation where it is a function of the time step Δt only.

8.3.3 Stability

Stability of the Euler method solving the diffusion equation may be analysed in a similar way to that for ordinary differential equations. We start by asking the question “does the Euler method converge as t–>$infinity$?” The exact solution will have u –> 0 and the numerical solution must also do this if it is to be stable.

We choose

$$U^{(0)}_{i,j} = \tag{176}$$

for some α and β chosen as multiples of π/m to satisfy $u = 0$ on the boundaries. Substituting this into (174) gives

(177)

Applying this at consecutive times shows the solution at time t_n is

$U^{(n)}_{i,j} =$ (178)

which then requires $|1 - 4\mu[\sin^2(\alpha/2) + \sin^2(\beta/2)]| < 1$ for this to converge as *n->infinity*. For this to be satisfied for arbitrary α and β we require $\mu < 1/4$. Thus we must ensure

$\Delta t <$ (179)

A doubling of the spatial resolution therefore requires a factor of four more time steps so overall the expense of the computation increases sixteen-fold.

The analysis for the diffusion equation in one or three dimensions may be computed in a similar manner.

8.3.4 Model for general initial conditions

Our analysis of the Euler method for solving the diffusion equation in section 8.3.3 assumed initial conditions of the form $\sin(k\pi x/L_x)\sin(l\pi y/L_y)$ where k,l are integers and L_x, L_y are the dimensions of the domain. In addition to satisfying the boundary conditions, these initial conditions represent a set of orthogonal functions which may be used to construct any arbitrary initial conditions as a Fourier series. Now, since the diffusion equation is linear, and as our stability analysis of the previous section shows the conditions under which the solution for each Fourier mode is stable, we can see that the equation (179) applies equally for arbitrary initial conditions.

8.3.5 Crank-Nicholson

The implicit Crank-Nicholson method is significantly better in terms of stability than the Euler method for ordinary differential equations. For partial differential equations such as the diffusion equation we may analyse this in the same manner as the Euler method of section 8.3.3.

For simplicity, consider the one dimensional diffusion equation

(180)

with $u(x=0,t) = u(x=1,t) = 0$ and apply the standard spatial discretisation for the curvature term to obtain

(181)

for the $i=1,m-1$ internal points. Solution of this expression will involve the solution of a tridiagonal system for this one-dimensional problem *at each time step*:

$$\begin{bmatrix} 1 & & & & & & & & & \\ -\frac{1}{2}\mu & 1+\mu & -\frac{1}{2}\mu & & & & & & & \\ & -\frac{1}{2}\mu & 1+\mu & -\frac{1}{2}\mu & & & & & & \\ & & -\frac{1}{2}\mu & 1+\mu & -\frac{1}{2}\mu & & & & & \\ & & & -\frac{1}{2}\mu & 1+\mu & -\frac{1}{2}\mu & & & & \\ & & & & -\frac{1}{2}\mu & 1+\mu & -\frac{1}{2}\mu & & & \\ & & & & & -\frac{1}{2}\mu & 1+\mu & -\frac{1}{2}\mu & & \\ & & & & & & -\frac{1}{2}\mu & 1+\mu & -\frac{1}{2}\mu & \\ & & & & & & & -\frac{1}{2}\mu & 1+\mu & -\frac{1}{2}\mu \\ & & & & & & & & -\frac{1}{2}\mu & 1 \end{bmatrix} \begin{pmatrix} U_0^{(n+1)} \\ U_1^{(n+1)} \\ U_2^{(n+1)} \\ U_3^{(n+1)} \\ U_4^{(n+1)} \\ U_5^{(n+1)} \\ U_6^{(n+1)} \\ \\ U_{m-1}^{(n+1)} \\ U_m^{(n+1)} \end{pmatrix} = \begin{pmatrix} 0 \\ U_1^n + \frac{1}{2}\mu\left(U_2^{(n)} - 2U_1^{(n)} + U_0^{(n)}\right) \\ U_2^n + \frac{1}{2}\mu\left(U_3^{(n)} - 2U_2^{(n)} + U_1^{(n)}\right) \\ \\ \\ \\ \\ \\ \\ U_{m-1}^n + \frac{1}{2}\mu\left(U_m^{(n)} - 2U_{m-1}^{(n)} + U_{m-2}^{(n)}\right) \\ 0 \end{pmatrix} \quad (182)$$

To test the stability we again choose a Fourier mode. Here we have only one spatial dimension so we use $U^{(0)}_i = \sin(\theta i)$ which satisfies the boundary condition if θ is a multiple of π. Substituting this into (183) we find

$$U_i^n = U_i^{n-1}\left(\frac{1-2\mu\sin^2\frac{\theta}{2}}{1+2\mu\sin^2\frac{\theta}{2}}\right) = U_i^0\left(\frac{1-2\mu\sin^2\frac{\theta}{2}}{1+2\mu\sin^2\frac{\theta}{2}}\right)^n. \quad (183)$$

Since the term $[1–2\mu\sin^2(\theta/2)]/[1+2\mu\sin^2(\theta/2)] < 1$ for all $\mu > 0$, the Crank-Nicholson method is unconditionally stable. The step size Δt may be chosen on the grounds of truncation error independently of Δx.

8.3.6 ADI[*]

Not examinable

8.4 Advection[*]

Not examinable

8.4.1 Upwind differencing[*]

Not examinable

8.4.2 Courant number[*]

Not examinable

8.4.3 Numerical dispersion[*]

Not examinable

8.4.4 Shocks[*]

Not examinable

8.4.5 Lax-Wendroff[*]

Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable

8.4.6 Conservative schemes[*]

Not examinable

[*] Not examinable

9. Number representation*

This section outlines some of the fundamentals as to how numbers are represented in computers. None of the material in this section is examinable.

9.1. Integers*

Normally represented as 2's compliment. Positive integers are presented as their binary equivalents:

Decimal	Binary	Two's compliment (16 bit)
1	1	0000000000000001
5	101	0000000000000101
183	10110111	0000000010110111
255	11111111	0000000011111111
32767	111111111111111	0111111111111111

Negative numbers are obtained by taking the one's compliment (*i.e.* inverting all bits) and adding one: $-A = (A \texttt{ xor } 1111111111111_2) + 1$. Any *carry bit* generated is discarded. This operation is its self-inverse

	Decimal	Two's compliment (16 bit)
A	5	0000000000000101
xor 1111111111111		1111111111111010
+ 1	–5	1111111111111011

	Decimal	Two's compliment (16 bit)
–A	**–5**	**1111111111111011**
xor 1111111111111		**0000000000000100**
+ 1	**5**	**0000000000000101**

Examples

Decimal	Binary	Two's compliment (16 bit)
–1	–1	1111111111111111
–5	–101	1111111111111011
–183	–10110111	1111111101001001
–255	–11111111	1111111100000001
–32767	–111111111111111	1000000000000001
–32768	–1111111111111110	1000000000000000

- The number must be represented by a known number of bits, n (say)
- The highest order bit indicates the sign of the number
- If n bits are used to represent the integer, values in the range -2^{n-1} to $2^{n-1}-1$ may be represented. For $n=16$, values from –32768 to 32767 are represented

* Not examinable

* Not examinable

- Normal binary addition of two two's compliment numbers produces a correct result if any carry bit is discarded

	Decimal	**Two's compliment (16 bit)**
	−5	1111111111111011
+	+ 183	0000000010110111
binary sum	???	10000000010110010
discard *carry bit*	178	0000000010110010

9.2. Floating point*

Represented as a binary mantissa and a binary exponent. There are a number of strategies for encoding the two parts. The most common of these is the IEEE floating point format which we describe here.

Figure 24 shows the IEE format for four byte floating point values, and figure 25 the same for eight byte values. In both cases the number is stored as a sign bit followed by the expondent and the mantissa. The number of bits used to represent the exponent and mantissa varies depending on the total number of bits.

Sign bit. The sign bit gives the overall sign for the number. A value of 0 indicates positive values, while 1 indicates negative values.

Exponent. This occupies eight bits for four byte values and eleven bits for eight byte values. The value stored in the exponent field is offset such that for four byte reals exponents of n = -1 ,0 and +1 are stored as 126 ($=01111110_2$), 127 ($=01111111_2$) and 128 ($=10000000_2$) respectively. The corresponding stored values for eight byte numbers are 1022 ($=01111111110_2$), 1023 ($=01111111111_2$) and 1024 ($=10000000000_2$). Each of these exponents represents a power of two scaling on the mantissa.

Mantissa. The mantissa is stored as unsigned binary in the remaining 23 (four byte values) or 52 (eight byte values) bits. The use of a mantissa plus an exponent means that the binary representation of all floating point values apart from zero will start with a one. It is thus unnecessary to store the first binary digit, and so improve the accuracy of the number representation. There is an assumed binary point (equivalent of a decimal point) following the unstored part of the number.

Zero. The value zero (0) does not follow the pattern of a unit value for the first binary digit in the mantissa. Given that, plus it being, in general, a *special case*, zero does not follow the above pattern. Instead, it is stored with all three components set to zero.

In the example below we show the four byte IEE representations of a selection of values.Binary values are indicated by a subscript 2.

Decimal	**Sign**	**Exponent**	**Mantissa**	**Value Stored**
				seeeeeee emmmmmmm mmmmmmmm mmmmmmmm
0.0	0_2	0 00000000_2	0.0 0.00000_2	0.0 00000000 00000000 00000000 00000000_2
1.0	+ 0_2	0 01111111_2	1.0 1.00000_2	1.0 00111111 10000000 00000000 00000000_2
8.0	+ 0_2	3 10000100_2	1.0 1.00000_2	8.0 01000001 00000000 00000000 00000000_2

* Not examinable

3.0	+ 0_2	1 10000000_2	1.5 1.10000_2	3.0 01000000 01000000 00000000 00000000_2
-3.0	- 1_2	1 10000000_2	1.5 1.10000_2	-3.0 11000000 01000000 00000000 00000000_2
0.25	+ 0_2	-2 01111101_2	1.0 1.00000_2	0.25 00111110 10000000 00000000 00000000_2
0.2	+ 0_2	-3 01111100_2	1.6 1.10011_2	0.2 + 0.0000000149.. 00111110 01001100 11001100 11001101

The first six values in this table yield exact representations under the IEE format. The final value (0.2), however, while an exact decimal, is represented as a recurring binary and thus suffers from truncation error in the IEEE format. Here the final bit has been rounded up, whereas the recurring sequence would carry on the 11001100 pattern. The error for a single value is small, but when carried through the a number of stages of computation, may lead to substantial errors being generated.

9.3. Rounding and truncation error*

The finite precision of floating point arithmetic in computers leads to rounding and truncation error. Truncation error is the result of the computer not being able to represent most floating point values exactly.

If two floating point values are combined, by addition, for example, then there may be a further loss of precision through *rounding error* even if both starting values may be represented exactly. Consider the sum of $0.00390625=0.00000001_2$ and $16=10000_2$, both of which have exact representations in our binary floating point format. The sum of these, 10000.00000001_2 would require a 13 bit mantissa. If we were to have a format where we have only 12 bits stored for the mantissa, it would be necessary to reduce the number by either *rounding* the binary number up to 10000.0000001_2, or truncating it to give 10000.0000000_2. In either case there is a loss of precision. The effect of this will accumulate of successive computations an may lead to a meaningless final result if care is not taken to ensure adequate precision and a suitable order for calculations.

9.4. Endians*

While almost all modern computers use formats discussed in the previous sections to store integer and floating point values, the same value may not be stored in the same way on two different machines. The reason for this is in the architecture of the central processing unit (CPU). Some machines store the least significant byte of a word at the lower memory address and the most significant byte at the upper memory address occupied by the value. This is the case for machines based on Intel and Digital processors (*e.g.* PCs and DEC alphas), amongst others. However, other machines store the values the opposite way around in memory, storing the least significant byte at the higher memory address. Many (but not all) Unix work stations use the latter strategy.

The result of this is that files containing binary data (*e.g.* in the raw IEEE format rather than as ASCII text) will not be easily portable from a machine that uses one convention to one that uses the

* Not examinable

* Not examinable

other. A two byte integer value of 1 = $\texttt{00000000 00000001}_2$ written by one machine would be read as 256 = $\texttt{00000001 00000000}_2$ when read by the other machine.

10. Computer languages*

A complete discussion of computer languages would require a whole course in itself. These notes are simply intended to give the reader a feeling for the strengths and uses of different languages. The notes are written from the perspective of someone who was brought up on a wide range of languages with programs suitable for both procedural and object oriented approaches. The discussion is necessarily brief and may be unfairly critical on some languages.

Ideally you would use the language which is most appropriate for a given application. For most projects, the language will be selected from the intersection of the subset of languages you know and the set of languages supported on the specific platform you require. Only for large projects is it worth investing the time and money required to utilise the *optimal* language.

None of the material in this section is examinable.

10.1. Procedural verses Object Oriented*

For many applications, the current trend is towards *object oriented* languages such as C++. *Procedural* languages are considered *behind the times* by many computer scientists, yet they remain the language of choice for many scientific programs. The main procedural contender is, and has always been Fortran. Both C++ and Fortran have their advantages, depending on the task at hand. This section is intended to act as an aid for choosing the appropriate language for a given task.

10.2. Fortran 90*

Fortran is one of the oldest languages still in wide spread use. It has long had a *standard*, enabling programs written for one type of machine to be relatively easily *ported* to run on a different type of machine, provided a Fortran compiler was available. The standard has evolved to reflect changes in coding practices and requirements. Unfortunately for many programmers, the standard lags many years behind current thinking. For example, Fortran 77 is the standard to which the majority of current compilers adhere. As the name suggests, the standard was released in 1977 and the demands of computing have changed significantly since then.

The latest standard, Fortran 90, was not finally released until about 1994. Like earlier improvements to the standard, it maintains a high level of backward compatibility. A Fortran 90 compiler should be able to compile all but a few of the oldest of Fortran programs (assuming they adhere to the relevant standard). While Fortran 90 answers the majority of criticisms about the earlier version, work has

* Not examinable

* Not examinable

* Not examinable

already started on the next generation, helping to ensure Fortran remains in wide spread use.

10.2.1. Procedural oriented[*]

The origins of Fortran as a scientific computer language are reflected in the name which stands for **For**mula **Tran**slation. The language is *procedural oriented* with the algorithm (or "formula") playing the central rôle controling the way the program is written and structured. In contrast, the *object oriented* C++ (see below) focuses on the data. For the majority of scientific programs using the types of algorithms introduced in this course, the procedural oriented approach is more appropriate. The program generates data from input parameters. The underlying numerical algorithms process one type of data in a consistent manner.

As the most computationally intensive programs have traditionally followed the procedural model, and have been written in Fortran, there has been a tremendous effort put into developing optimisation strategies for Fortran compilers. As a result the code produced by Fortran compilers is normally more efficient *for a numerical program* than that produced by other language compilers. In contrast, if you were to develop a word processor, for example, in Fortran, the end result would be less efficient (in terms of speed and size) and more cumbersome than something with the same functionality developed in C++.

10.2.2. Fortran enhancements[*]

For those familiar with Fortran 77, the Fortran 90 standard has introduced many of the features sadly missing from the earlier version. Control structures such as DO loops have done away with the need for a numeric label. WHILE, CASE and BREAK statements have been added to simplify program logic, and the ability to call subroutines recursively allows the program to operate more efficiently. From the data side structures (records), unions an maps greatly simplify the passing of parameters and the organisation of variables without resort to a multitude of COMMON blocks.

Perhaps the most significant changes accompany a move from all data structures being static (*i.e.* the size of and space required by variables is determined at compile time) to allowing dynamic allocation of memory. This greatly improves the flexibility of a program and frequently allows you to achieve more with less memory as the same memory can be reused for different purposes.

Many of these enhancements have been borrowed from languages such as C++, and adapted to the Fortran environment. One of the most powerful (and, if abused, most dangerous) is the ability to *overload* a function or operator. By overloading a function or operator, the precise result will depend on the type (and number) of parameters or operands used in executing the function/operator. For example, the operator "+" could be defined such that, when it "adds" two strings, then it concatenates them together, perhaps stripping any trailing spaces in the process.

[*] Not examinable

[*] Not examinable

Similarly it would be possible to define “number=string” such that the string “string” was passed to an interpreter function and the contents of it evaluated. Clearly doing something like defining “WRITE” as “READ” would lead to a great deal of confusion!

10.3. C++*

Like Fortran, C++ has evolved from earlier languages. The original, BCPL, was adopted and enhanced by Bell Laboratories to produce B and then C as part of their project to develop Unix.

*10.3.1. C**

The main features of C are its access to low-level features of the system and its extremely compact (and often confusing) notation. C has often been described as a *write only language* as it is relatively easy to write code which almost no one will be able to decipher. The popularity of C stems from its close association with Unix rather than from any inherent qualities. While there are probably still more programs written in Fortran and COBOL than there are in C, it is almost certain that more people *use* programs written in C (or C++) than in any other language. The vast majority of *shrink wrapped* applications are now written in C or C++.

While C compilers are still widely available and in common use, there is little or no justification for using C for any new project. C++ (which is more than just an improved version of C) provides all the functionality of C and allows the code to be written in a more flexible and readable manner.

Many programs were written in C during the late 1980s because it was the “in” language rather than the best for a given application. As a result of this investment and the relative simplicity of gradually moving over to C++, most mainstream applications in C

*10.3.2. Object Oriented**

As systems became bigger, it became desirable – often essential – to reuse components as much as possible. Historically this was achieved through the use of software libraries, but such libraries tend to be useful only for low-level routines. The concept of Object Oriented Programming was introduced to C to provide a straight forward way of handling many similar – but not identical – *objects* without having to write specialised code for each object. An object is simply some data which describes a coherent unit.

* Not examinable

* Not examinable

* Not examinable

Suppose we have an object *my_cup* which belongs to some class *Ccup_of_drink* (say). The object *my_cup* will take the form of a description of the cup which might include information about its colour, its texture, its use and its contents. Suppose we also have an object *my_plate* and that this belongs to a class *Cplate_of_food*. Clearly there is much in common between a cup and a plate. They are both crockery, although *my_cup* will contain a drink while *my_plate* will contain food. To save us having to define the crockery aspects separately for *my_cup* and *my_plate*, we shall allow them to inherit these attributes from a class *Ccrockery*. Tableware will, in turn, inherit attributes from other classes such as *Ctableware*, *Ccolour* and *Ctexture*. This hierarchy may be expressed as

Ccup_of_drink: *Ccrockery, Cdrink*

Cplate_of_food: *Ccrockery, Cfood*

Ccrockery: *Ctableware, Ccolour, Ctexture*

Similarly we might have a teaspoon *my_teaspoon* belonging to *Ccutlery* which follows

Ccutlery: *Ctableware, Ccolour*

The object-oriented approach comes in if we have an object *my_dishwasher*. The dishwasher will accept all objects belonging to *Ctableware*. The classes *Ccrockery* and *Ccutlery* are both derived from *Ctableware* and so any *Ccrockery* or *Ccutlery* object is also a *Ctableware* object, and can thus be washed by the *my_dishwasher*. Similarly as *Ccup_of_drink* and *Cplate_of_food* are derived (indirectly) from *Ctableware* and so can be washed by *my_dishwasher*.

The object *my_dishwasher* would simply say “wash thyself” to any object it contains. For a *Ccrockery* or *Ccutlery* object the process of washing would be identical, and could thus be embodied at the *Ctableware* level. It is not quite so simple for the *Ccup_of_drink* and *Cplate_of_food* objects as they contain food which would be destroyed by the dishwasher. Thus when *my_dishwasher* tries to wash the *Ctableware* derived object *my_plate*, it is necessary for the *Cplate_of_food* class to override the *Ctableware* procedure for washing. In this example the *Cplate_of_food* response to the dishwasher may be to declare the *my_plate* empty of food and then pass the request for washing the plate down to its embedded *Ccrockery* class which will in turn pass it down to *Ctableware*.

10.3.3. Weaknesses[*]

On the surface, the class structure would appear to allow C++ to be adapted to any type of application. In practice there are some things that are difficult to achieve in a seamless way and others which lead to relatively inefficient programming. This inefficiency may not matter for a wordprocessor (although users of Microsoft Word may not agree), but can be critical for a large scale numerical simulation.

For someone brought up on most other languages, the most obvious omission is support for multidimensional arrays. While C++ is able to have an array of arrays, and

[*] Not examinable

so have some of the functionality of a multidimensional array, they can not be used in the same way and the usual notation can not be employed. Further, it is easy to get confused as to what you have an array of, especially with the cryptic notation shared by C and C++.

C++ itself is a very small language with only a few built in functions and operators. It gains its functionality through standard and specialised libraries. While this adds to the flexibility, it has implications on the time taken to compile a program and, more importantly, on the execution speed. Even with the most carefully constructed libraries and the functions being inserted *in-line* (instead of being called), the fact the compiler does not *understand* the functions within the library limits the degree of optimisation which can be achieved.

10.4. Others*

*10.4.1. Ada**

Touted as the language to replace all other languages, Ada was introduced in the 1980s with a substantial backing from the US military. Its acceptance was slow due to its cumbersome nature and its unsuitability for many of the computers of the period.

*10.4.2. Algol**

In the late 1960s and early 1970s Algol (**Algo**rithmic **L**anguage) looked as though it would become *the* scientific computer language. Algol forced the programmer to adopt a much more structured approach than other languages at the time and offered a vast array of built-in functionality such as matrix manipulation. Ultimately it was the shear power that lead to the demise of Algol as it was unable to fit on the smaller mini and microcomputers as they were developed.

*10.4.3. Basic**

Basic has developed from a design exercise in Southampton to the *language* which is known by more people than any other. The name Basic (**B**eginners **S**ymbolic **I**nstruction **C**ode) does not really represent a language, but rather a family of languages which appear similar in concept but will differ in almost every important detail. Most implementations of Basic are proprietary and some are much better than others. Basic was designed as an interpreted language and most implementations follow this heritage. The name Basic is often used for the interpreted macro language in many modern applications, especially those from Microsoft.

---

* Not examinable

* Not examinable

* Not examinable

* Not examinable

10.4.4. Cobol[*]

Cobol (**Co**mmon **B**usiness **O**riented **L**anguage) and Fortran are the only two early high-level languages remaining in widespread use. However, whereas Fortran has evolved with time, Cobol is essentially the same as it was 30 years ago. It is an extremely verbose, English-like language aimed at business applications. It remains in widespread use not because of any inherent advantages, but because of the huge investment which would be required to upgrade the software to a more modern language.

10.4.5. Delphi[*]

Developed by Borland International as a competitor for Microsoft's Visual Basic, Delphi tries to combine the graphical development tools of Visual Basic with the structure and consistency of Pascal and the Object Oriented approach of C++ and Smalltalk. It is being predicted that Delphi will become increasingly popular on PCs.

10.4.6. Forth[*]

A threaded-interpreted language. Developed to control telescopes. Uses reverse-polish notation. Extremely efficient for an interpreted language, but relatively hard to read and program.

10.4.7. Lisp[*]

Good for Artificial Intellegence, provided you like recursion and counting brackets.

10.4.8. Modula-2[*]

An *improved* form of Pascal. Very good, but never widely accepted.

10.4.9. Pascal[*]

Developed as a teaching language. The *standard* lacks many of the features essential for effective computing. Most implementations provide significant enhancements to this standard, but are often not portable. Can be used for scientific programming, but the compilers are often inferior to those of Fortran and C++.

---

[*] Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable

10.4.10. PL/1[*]

Introduced by IBM in the 1970s as a *replacement* for both Cobol and Fortran, PL/1 has some of the advantages and some of the disadvantages of both languages. However, it never gained much of a following due to a combination of its proprietary nature and a requirement for very substantial mainframe resources.

10.4.11. PostScript[*]

Often thought of as a printer protocol, PostScript is in fact a full-fledged language with all the normal capabilities *plus* built in graphics! Like Forth, PostScript is a threaded-interpreted language which results in efficient but hard to read code. While PostScript can be used for general-purpose computing, it is inadvisable as it would tie up the printer for a long time! Unlike most languages in wide-spread use, PostScript is a proprietary language belonging to Adobe. There are, however, many emulations of PostScript with suitably disguised names: GhostScript, StarScript, …

10.4.12. Prolog[*]

10.4.13. Smalltalk[*]

One of the first object-oriented programming languages. It provides a more consistent view than C++, but is less widely accepted.

10.4.14. Visual Basic[*]

This flavour of Basic was introduced by Microsoft as an easy method of writing programs for the Windows GUI. It has been an extremely successful product, but remains relatively inefficient for computational work when compared with C++ or Fortran.

[*] Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable

[*] Not examinable