NUMERICAL METHODS

Contents Summary

Contents

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 written in Microsoft Word 7.0 for Windows 95. The same Word document is used as the source for both printed and HTML versions. Conversion from Word to HTML is achieved through a combination of custom macros to adjust the formatting and Microsoft Internet Assistant for Word.

1.6.1 Word version

The Word version of the notes is available for those who may wish to alter or print it out. The Word 7.0 file format is interchangeable with Word 6.0.

1.6.2 Notation in HTML formatted notes

The source Word document 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.

1.6.3 Copyright

These notes may be duplicated freely for the purposes of education or research. Any such reproductions, in whole or in part, should contain details of the author and this copyright notice.

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

$$f(x+\delta x)=f(x)+\delta x f'(x)+\frac{\delta x^2}{2}f''(x)+\frac{\delta x^3}{6}f'''(x)+\ldots \quad (1)$$

In two dimensions we have

$$f(x+\delta x,y+\delta y)=f(x,y)+\delta x\frac{\partial f}{\partial x}+\delta y\frac{\partial f}{\partial y}+\frac{\delta x^2}{2}\frac{\partial^2 f}{\partial x^2}+\frac{\delta y^2}{2}\frac{\partial^2 f}{\partial y^2}+\delta x\delta y\frac{\partial^2 f}{\partial x\partial y}+\ldots \quad (2)$$

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. 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 $f(x) = 0$.

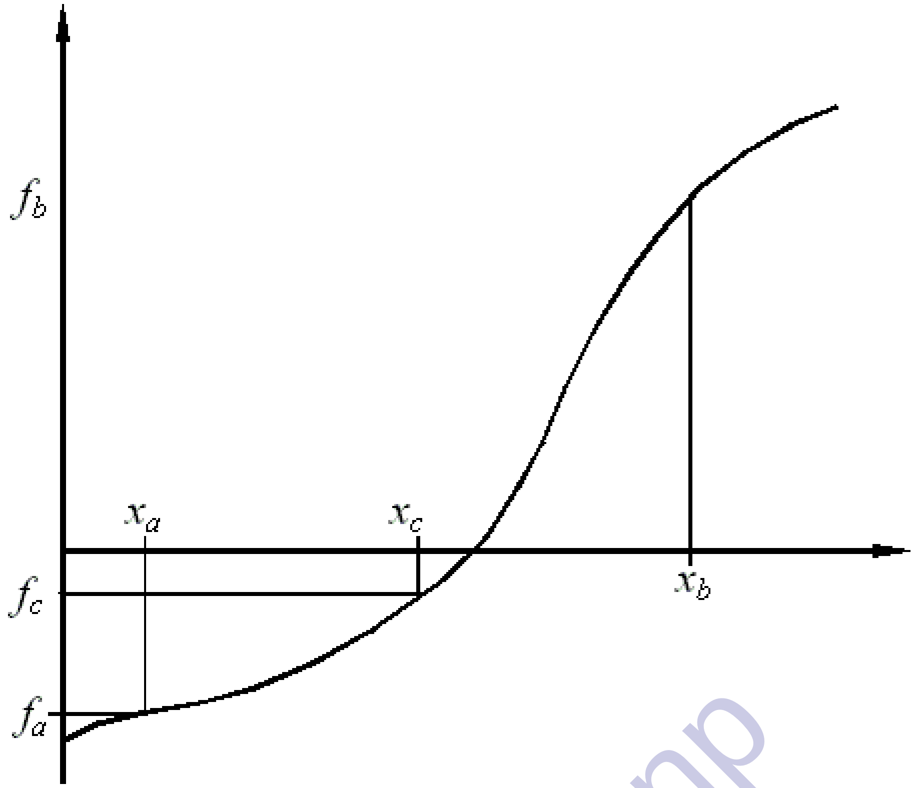


Figure1: Graphical representation of the bisection method showing two initial guesses (x_a and x_b bracketting the root).

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 must be $e_n < |x_a\ \ x_b|$. Now since the interval (x_a,x_b) is halved for each iteration, then

$$e_{n+1} \sim e_n/2. \qquad (3)$$

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

$$\varepsilon_n = x_n\ \ x^*. \qquad (4)$$

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

$$|\varepsilon_{n+1}| \sim C|\varepsilon_n|^p. \tag{5}$$

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 (5) 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 (3) 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 $e_n = |a_n\text{-}b_n|$, 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 .

Figure2

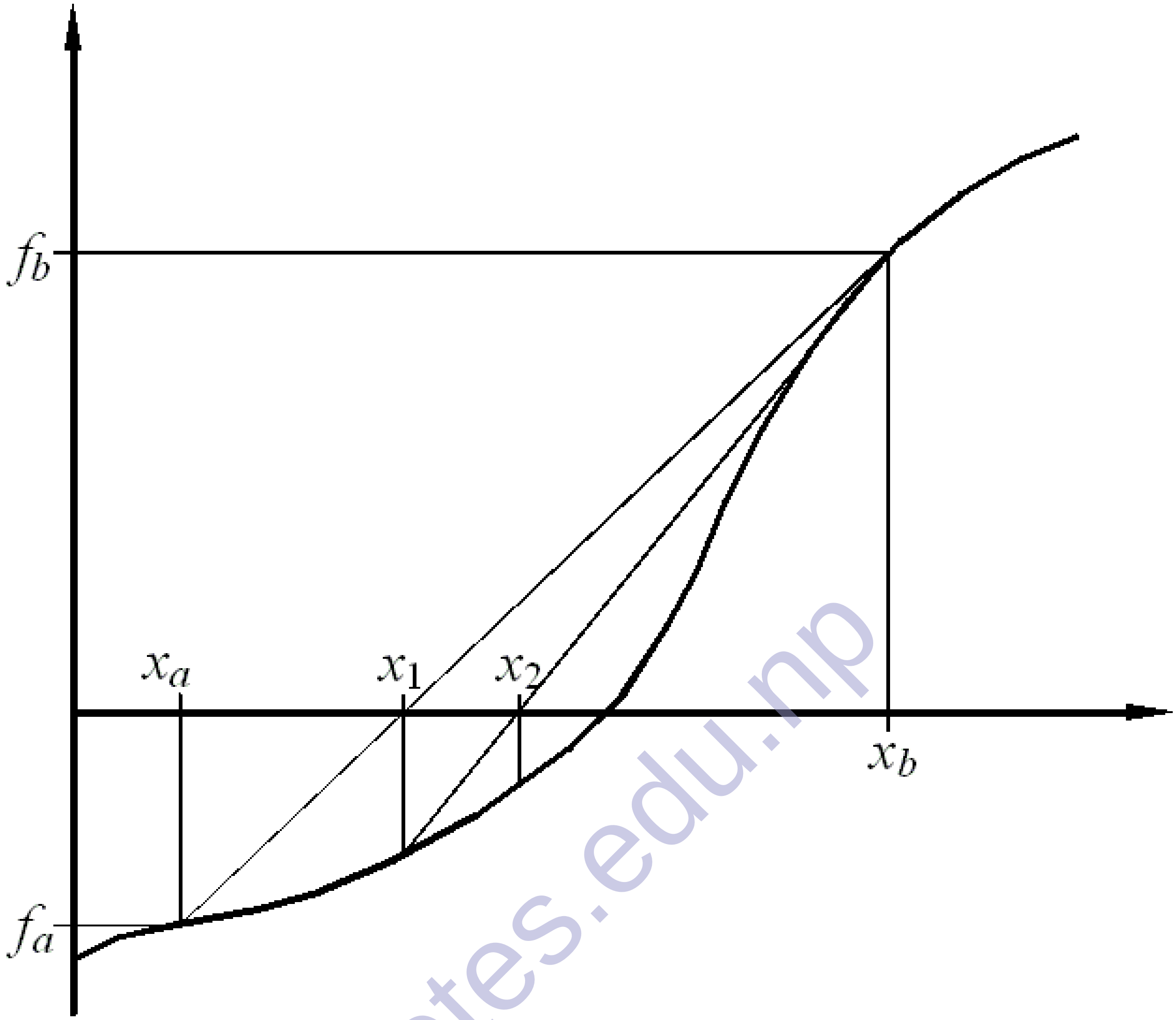


Figure2: Root finding by the linear interpolation (regula falsi) method. The two initial gueses x_a and x_b must bracket the root.

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 $x = x_0$:

$$f(x) = f(x_0) + (x-x_0)f'(x_0) + \tfrac{1}{2}(x-x_0)^2 f''(x_0) + O(|x-x_0|^3). \tag{6}$$

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

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{7}$$

Subsequent iterations are defined in a similar manner as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{8}$$

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.

Figure3

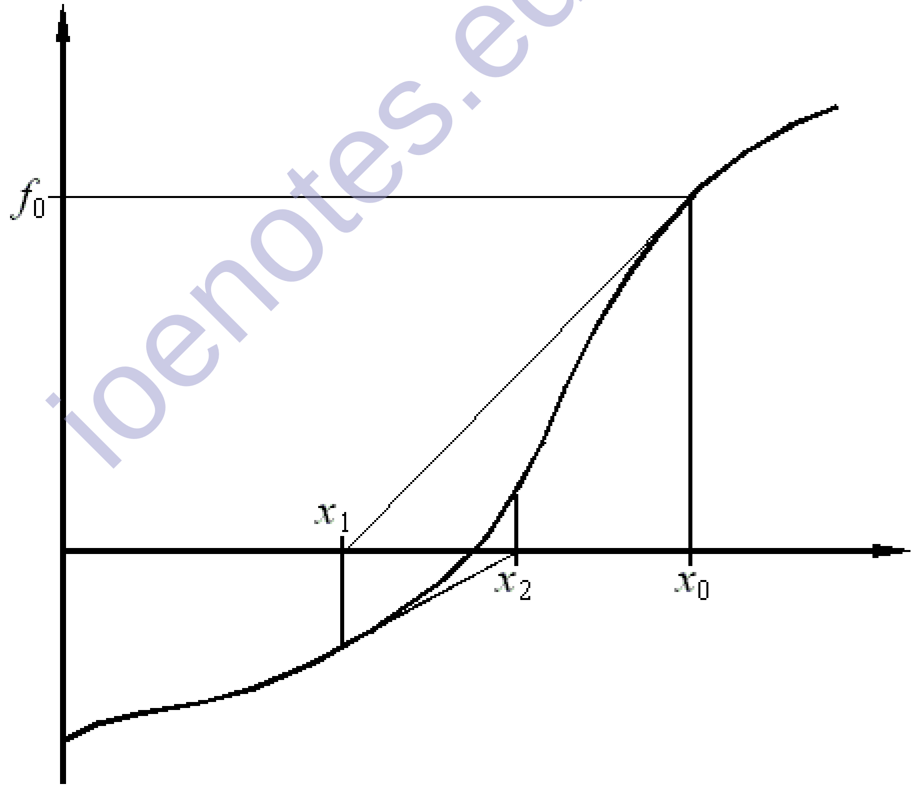


Figure3: Graphical interpretation of the Newton Raphson algorithm.

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 .

Figure4

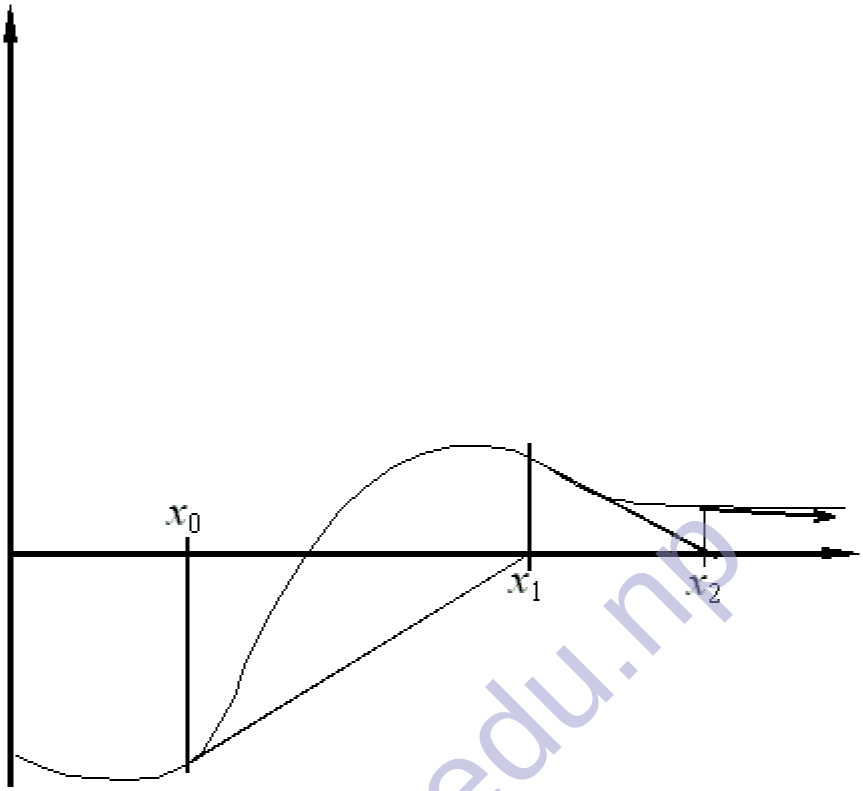


Figure4: Divergence of the Newton Raphson algorithm due to the presence of a turning point close to the root.

3.4.1 Convergence

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

$$f(x) = f(x^*) + (x - x^*)f'(x^*) + {}^1/_2(x - x^*)^2 f''(x^*) + O(|x - x^*|^3), \qquad (9)$$

and substitute into the iteration formula. This then shows

$$\begin{aligned}
\varepsilon_{n+1} &= x_{n+1} - x^* \\
&= x_n - x^* - \frac{f(x_n)}{f'(x_n)} \\
&= \varepsilon_n - \frac{\varepsilon_n f'(x^*) + \tfrac{1}{2}\varepsilon_n^2 f''(x^*) + \ldots}{f'(x^*) + \varepsilon_n f''(x^*) + \ldots} \\
&= \varepsilon_n - \left[\varepsilon_n f'(x^*) + \tfrac{1}{2}\varepsilon_n^2 f''(x^*) + \ldots\right]\frac{1}{f'(x^*)}\left[1 - \varepsilon_n \frac{f''(x^*)}{f'(x^*)} + \ldots\right] \\
&= \varepsilon_n - \varepsilon_n + \varepsilon_n^2 \frac{f''(x^*)}{f'(x^*)} - \tfrac{1}{2}\varepsilon_n^2 \frac{f''(x^*)}{f'(x^*)} + O\left(\varepsilon_n^3\right) \\
&= \tfrac{1}{2}\varepsilon_n^2 \frac{f''(x^*)}{f'(x^*)} + O\left(\varepsilon_n^3\right)
\end{aligned} \tag{10}$$

since $f(x^*)=0$. Thus, by comparison with (4), 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.*

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}, \tag{11}$$

and this is substituted into the Newton-Raphson algorithm (8) to give

$$x_{n+1} = x_n - \frac{(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})} f(x_n). \tag{12}$$

This formula is identical to that for the Linear Interpolation method discussed in section3.3 . 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.

Figure5

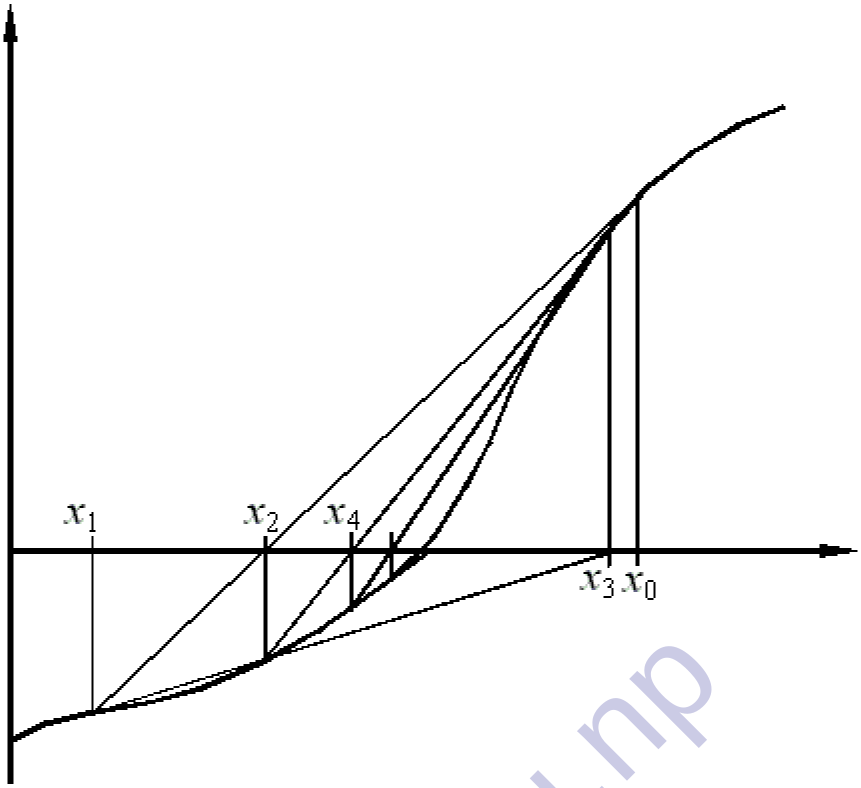


Figure5: Convergence on the root using the secant method.

Figure6

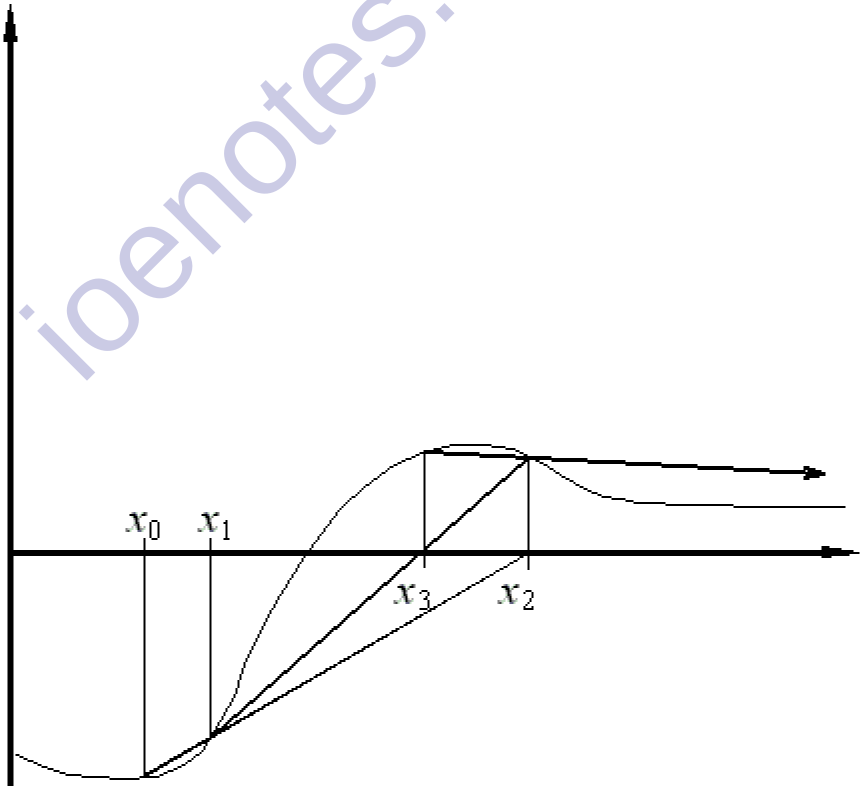


Figure6: Divergence using the secant method.

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

$$f(x_n) = f(x^*) + \varepsilon_n f'(x^*) + {}^1/_2\varepsilon_n^2 f''(x^*) + O(|\varepsilon_n|^3), \qquad (13a)$$

$$f(x_{n1}) = f(x^*) + \varepsilon_{n-1} f'(x^*) + {}^1/_2\varepsilon_{n-1}^2 f''(x^*) + O(|\varepsilon_{n-1}|^3), \qquad (13b)$$

and substituting into the iteration formula

$$\begin{aligned}
\varepsilon_{n+1} &= x_{n+1} - x^* \\
&= x_n - x^* - \frac{f(x_n)}{f(x_n) - f(x_{n-1})}(x_n - x_{n-1}) \\
&= \varepsilon_n - \frac{\varepsilon_n f'(x^*) + \frac{1}{2}\varepsilon_n^2 f''(x^*) + \ldots}{\varepsilon_n f'(x^*) + \frac{1}{2}\varepsilon_n^2 f''(x^*) + \ldots - \left[\varepsilon_{n-1} f'(x^*) + \frac{1}{2}\varepsilon_{n-1}^2 f''(x^*) + \ldots\right]}(\varepsilon_n - \varepsilon_{n-1}) \\
&= \varepsilon_n - \frac{\varepsilon_n f'(x^*) + \frac{1}{2}\varepsilon_n^2 f''(x^*) + \ldots}{(\varepsilon_n - \varepsilon_{n-1}) f'(x^*)\left[1 + \frac{1}{2}(\varepsilon_n + \varepsilon_{n-1}) f''(x^*) + \ldots\right]}(\varepsilon_n - \varepsilon_{n-1}) \\
&= \varepsilon_n - \left[\varepsilon_n + \tfrac{1}{2}\varepsilon_n^2 \frac{f''(x^*)}{f'(x^*)} + \ldots\right]\left[1 - \tfrac{1}{2}(\varepsilon_n + \varepsilon_{n-1})\frac{f''(x^*)}{f'(x^*)} + \ldots\right] \\
&= \varepsilon_n - \varepsilon_n + \tfrac{1}{2}\varepsilon_n(\varepsilon_n + \varepsilon_{n-1})\frac{f''(x^*)}{f'(x^*)} - \tfrac{1}{2}\varepsilon_n^2\frac{f''(x^*)}{f'(x^*)} + O(\varepsilon_n^3) \\
&= \tfrac{1}{2}\frac{f''(x^*)}{f'(x^*)}\varepsilon_n\varepsilon_{n-1} + O(\varepsilon_{n-1}^3)
\end{aligned} \qquad (14)$$

.

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

$$\varepsilon_{n+1} = \left(\frac{f''(x^*)}{2f'(x^*)}\right)^{\beta}\varepsilon_n^{\alpha}, \qquad (15)$$

and substituting into the error evolution equation (14) gives

$$\begin{aligned}
\varepsilon_{n+1} &= \left(\frac{f''(x^*)}{2f'(x^*)}\right)\varepsilon_n\varepsilon_{n-1} \\
&= \left(\frac{f''(x^*)}{2f'(x^*)}\right)\varepsilon_n\left(\frac{f''(x^*)}{2f'(x^*)}\right)^{-\beta/\alpha}\varepsilon_n^{1/\alpha} \\
&= \left(\frac{f''(x^*)}{2f'(x^*)}\right)^{1-\beta/\alpha}\varepsilon_n^{\frac{\alpha+1}{\alpha}}
\end{aligned} \qquad (16)$$

which we equate with our assumed relationship to show

$$\alpha = \frac{1+\alpha}{\alpha} = \frac{1+\sqrt{5}}{2},$$
$$\beta = \frac{\alpha}{1+\alpha} = \frac{1}{\alpha} = \frac{2}{1+\sqrt{5}}. \tag{17}$$

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

$$x_{n+1} = g(x_n). \tag{18}$$

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

Figure7

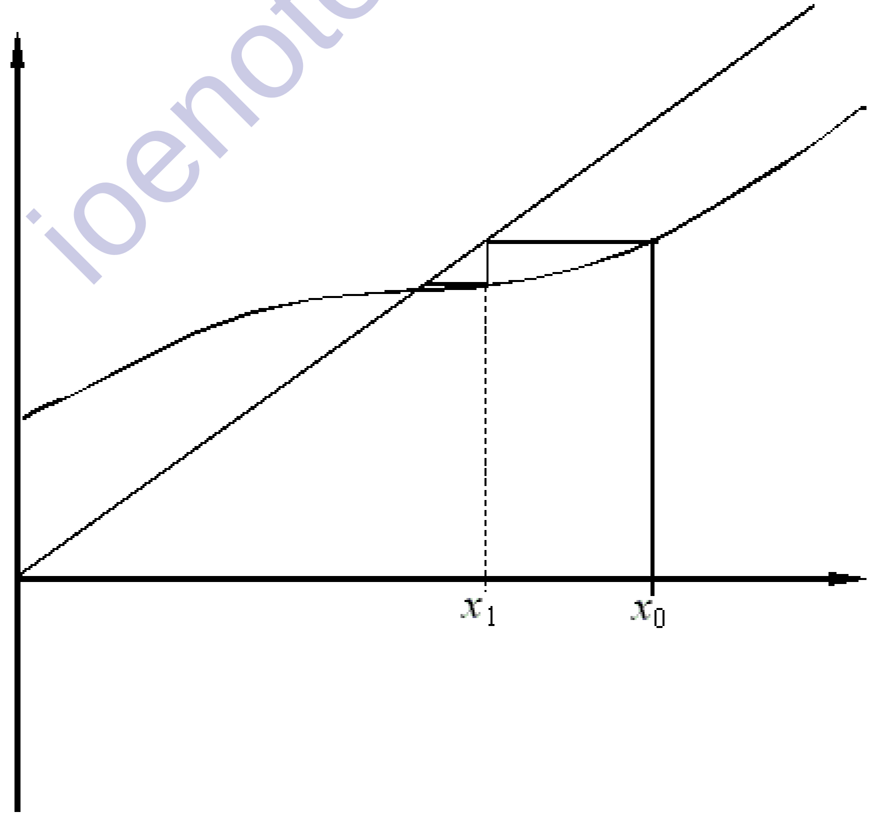


Figure7: Convergence on a root using the Direct Iteration method.

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

$$g(x_n) = g(x^*) + \varepsilon_n g'(x^*) + {}^1\!/_2 \varepsilon_n^2 g''(x^*) + O(|\varepsilon_n|^3), \tag{19}$$

so that the evolution of the error follows

$$\begin{aligned} \varepsilon_{n+1} &= x_{n+1} - x^* \\ &= g(x_n) - x^* \\ &= g(x^*) + \varepsilon_n g'(x^*) + {}^1\!/_2 \varepsilon_n^2 g''(x^*) + O(\varepsilon_n^3) - x^* \\ &= \varepsilon_n g'(x^*) + {}^1\!/_2 \varepsilon_n^2 g''(x^*) + O(\varepsilon_n^2) \end{aligned} \tag{20}$$

.

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.

Figure8

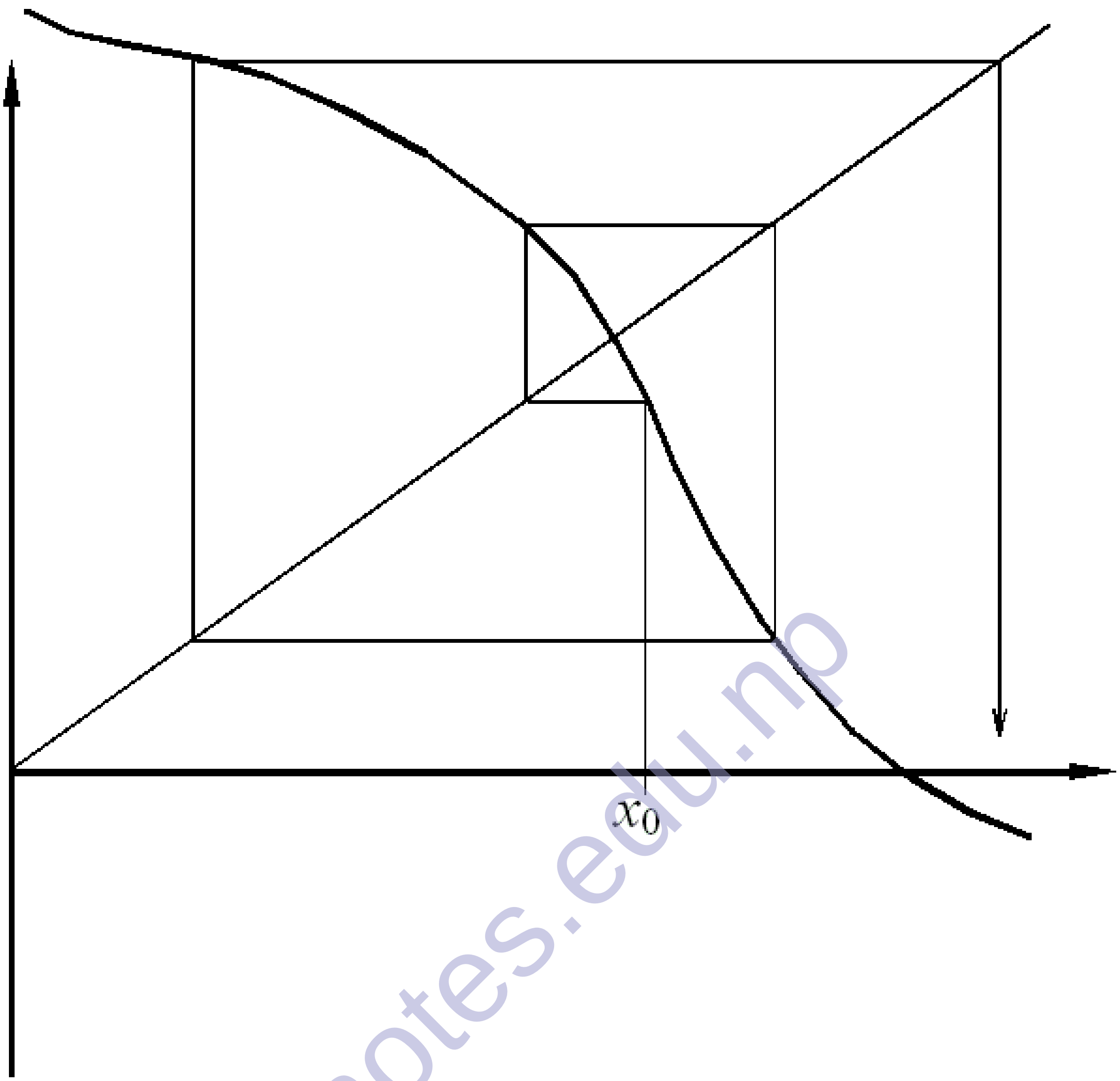


Figure8: The divergence of a Direct Iteration when $g' < 1$.

3.7 Examples

Consider the equation

$$f(x) = \cos x \;\; 1/2. \qquad (21)$$

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	-50274110020.188

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	-0.00000000000000002284	

- Convergence linear, but fast.

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	0.0000000687124	0.000000014553	-
3	1.0E-15		
4	*Machine accuracy*		

- Solution found to roundoff error ($O(10^{-15})$) in three iterations.

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 (21) may be rearranged into the form required for direct iteration.

3.7.5.1 Addition of *x*

Use

$$x_{n+1} = g(x) = x_n + \cos x - 1/2 \qquad (22)$$

- 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	-0.0000000000001465442	

3.7.5.2 Multiplcation by *x*

Use

$$x_{n+1} = g(x) = 2x \cos x \qquad (23)$$

- 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

$$f(x) = f(x) + (x-x)h(x) = 0. \qquad ()$$

Rearranging equation (24) for one of the x variables and labelling the different variables for different steps in the interation gives

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

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

$$h(x) = 4x(x - \pi)/\pi^2 \qquad (26)$$

- 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**	$\boldsymbol{e_{n+1}/e_n}$
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	0.00000000000000516850	
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 (21). The clear winner is the Newton-Raphson scheme, with the approximated derivative for the Direct Iteration proving a very good alternative.

Figure9

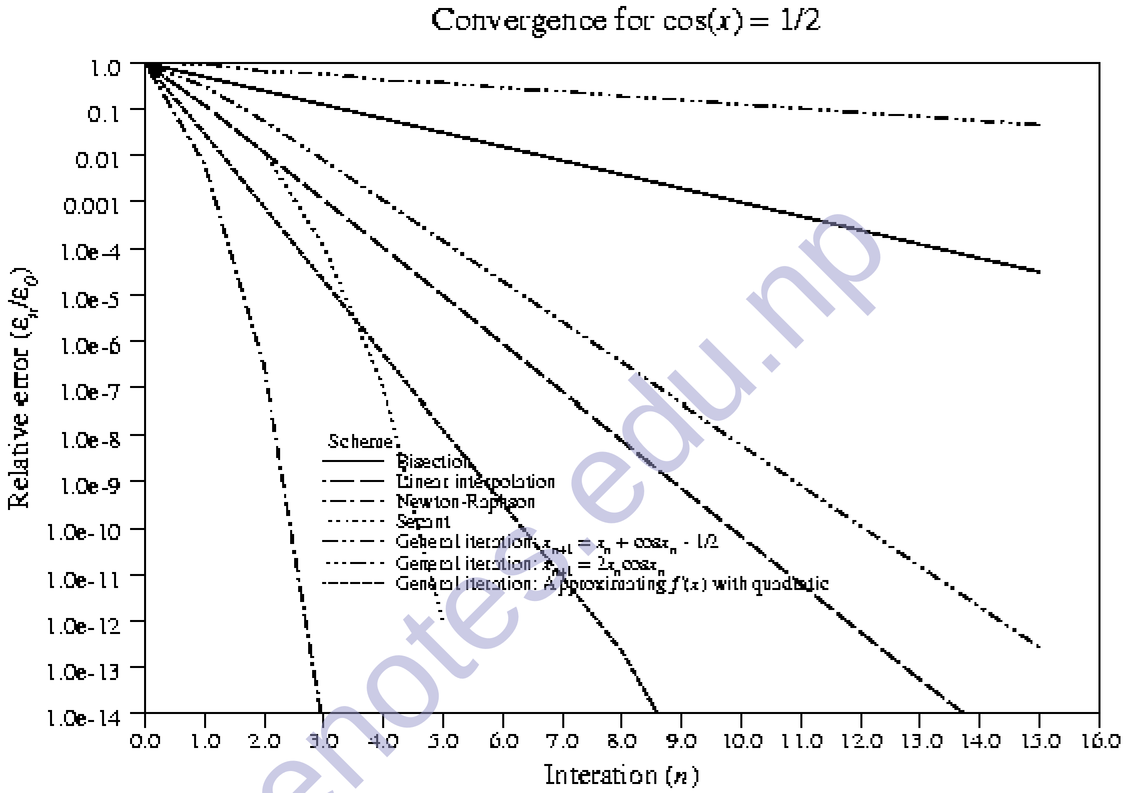


Figure9: Comparison of the convergence of the error in the estimate of the root to $\cos x = 1/2$ for a range of different root finding algorithms.

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
      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 **A*x*** = ***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 **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 the system

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= r_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= r_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n &= r_3 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= r_n \end{aligned} , \tag{27}$$

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

$$\begin{bmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & \cdots & a_{1n}/a_{11} \\ 0 & a_{22}-(a_{21}/a_{11})a_{12} & a_{23}-(a_{21}/a_{11})a_{13} & & a_{2n}-(a_{21}/a_{11})a_{1n} \\ 0 & a_{32}-(a_{31}/a_{11})a_{12} & a_{33}-(a_{31}/a_{11})a_{13} & & a_{3n}-(a_{31}/a_{11})a_{1n} \\ & & \vdots & & \\ 0 & a_{n2}-(a_{n1}/a_{11})a_{12} & a_{n3}-(a_{n1}/a_{11})a_{13} & & a_{nn}-(a_{n1}/a_{11})a_{1n} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} r_1/a_{11} \\ r_2-(a_{21}/a_{11})r_1 \\ r_3-(a_{31}/a_{11})r_1 \\ \vdots \\ r_n-(a_{n1}/a_{11})r_1 \end{pmatrix} . \tag{28}$$

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

$$\begin{bmatrix} 1 & \hat{a}_{12} & \hat{a}_{13} & \cdots & \hat{a}_{1n} \\ 0 & 1 & \hat{a}_{23} & & \hat{a}_{2n} \\ 0 & & 1 & & \hat{a}_{3n} \\ & & & \vdots & \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{r}_1 \\ \hat{r}_2 \\ \hat{r}_3 \\ \vdots \\ \hat{r}_n \end{pmatrix} \qquad ()$$

the final solution may be obtained by back substitution.

$$\begin{aligned} x_n &= \hat{r}_n, \\ x_{n-1} &= \hat{r}_{n-1} - \hat{a}_{n-1,n} x_n, \\ x_{n-2} &= \hat{r}_{n-2} - \hat{a}_{n-2,n-1} x_{n-1} - \hat{a}_{n-2,n} x_n, \\ &\vdots \\ x_1 &= \hat{r}_1 - \hat{a}_{1,2} x_2 - \hat{a}_{1,3} x_3 - \cdots - \hat{a}_{1,n} x_n. \end{aligned} \tag{30}$$

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

$$\begin{bmatrix} 0 & 3 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}, \tag{31}$$

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 (31) we could simply interchange rows one and two to produce

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}, \tag{32}$$

or columns one and two to give

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_2 \\ x_1 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}, \tag{33}$$

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

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

$$\begin{bmatrix} 1 & 4 & 1 & 8 & 3 & 2 & \dots & 5 \\ 0 & 10^{-6} & 1 & 10 & 201 & 13 & & 4 \\ 0 & 9 & 4 & 6 & -8 & 2 & & 18 \\ 0 & 3 & 2 & -3 & 4 & 6003 & & 15 \\ 0 & 15 & 1 & 9 & 33 & -2 & & 1 \\ 0 & -155 & 23 & 4 & 25 & 73 & & 2 \\ & & & \vdots & & & & \\ 0 & 8 & 56 & 4 & -4 & 4 & \dots & 88 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{r}_1 \\ \hat{r}_2 \\ \hat{r}_3 \\ \hat{r}_4 \\ \hat{r}_5 \\ \hat{r}_6 \\ \vdots \\ \hat{r}_n \end{pmatrix} \tag{34}$$

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

$$\begin{bmatrix} 1 & 4 & 1 & 8 & 3 & 2 & \dots & 5 \\ 0 & -155 & 23 & 4 & 25 & 73 & & 2 \\ 0 & 9 & 4 & 6 & -8 & 2 & & 18 \\ 0 & 3 & 2 & -3 & 4 & 6003 & & 15 \\ 0 & 15 & 1 & 9 & 33 & -2 & & 1 \\ 0 & 10^{-6} & 1 & 10 & 201 & 13 & & 4 \\ & & & \vdots & & & & \\ 0 & 8 & 56 & 4 & -4 & 4 & \dots & 88 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{r}_1 \\ \hat{r}_6 \\ \hat{r}_3 \\ \hat{r}_4 \\ \hat{r}_5 \\ \hat{r}_2 \\ \vdots \\ \hat{r}_n \end{pmatrix} . \qquad (35)$$

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 ***x***. The rearranged system becomes

$$\begin{bmatrix} 1 & 3 & 1 & 8 & 3 & 2 & \dots & 5 \\ 0 & 201 & 1 & 10 & 10^{-6} & 13 & & 4 \\ 0 & -8 & 4 & 6 & 9 & 2 & & 18 \\ 0 & 4 & 2 & -3 & 3 & 6003 & & 15 \\ 0 & 33 & 1 & 9 & 15 & -2 & & 1 \\ 0 & 25 & 23 & 4 & -155 & 73 & & 2 \\ & & & \vdots & & & & \\ 0 & -4 & 56 & 4 & 8 & 4 & \dots & 88 \end{bmatrix} \begin{pmatrix} x_1 \\ x_5 \\ x_3 \\ x_4 \\ x_2 \\ x_6 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \\ \hat{b}_4 \\ \hat{b}_5 \\ \hat{b}_6 \\ \vdots \\ \hat{b}_n \end{pmatrix} . \qquad (36)$$

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

$$\begin{bmatrix} 1 & 4 & 1 & 8 & 3 & 2 & \dots & 5 \\ 0 & 6003 & 2 & -3 & 4 & 3 & & 4 \\ 0 & 2 & 4 & 6 & -8 & 9 & & 18 \\ 0 & 13 & 1 & 10 & 201 & 10^{-6} & & 15 \\ 0 & -2 & 1 & 9 & 33 & 15 & & 1 \\ 0 & 73 & 23 & 4 & 25 & -155 & & 2 \\ & & & \vdots & & & & \\ 0 & 4 & 56 & 4 & -4 & 8 & \dots & 88 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{r}_1 \\ \hat{r}_2 \\ \hat{r}_3 \\ \hat{r}_4 \\ \hat{r}_5 \\ \hat{r}_6 \\ \vdots \\ \hat{r}_n \end{pmatrix}. \qquad (37)$$

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 **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 **LU** = **A**, 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 **L*y*=*b*** then **U*x*=*y***, 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 **A*x*=*b*** for with whole range of ***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 $n$
    $a_{pi} = a_{pi} - \sum_{k=1}^{i-1} a_{pk} a_{ki}$
  NEXT $p$
  FOR $q$=$i$+1 TO $n$
```

```
    $a_{iq} = \frac{a_{iq} - \sum_{k=1}^{i-1} a_{ik} a_{kq}}{a_{ii}}$
  NEXT q
NEXT i
# Forward Substitution
FOR i=1 TO n
  FOR q=n+1 TO n+m
    $a_{iq} = \frac{a_{iq} - \sum_{k=1}^{i-1} a_{ik} a_{kq}}{a_{ii}}$
  NEXT q
NEXT i
# Back Substitution
FOR i=n-1 TO 1
  FOR q=n+1 TO n+m
    $a_{iq} = a_{iq} - \sum_{k=i+1}^{n} a_{ik} a_{kq}$
  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 $\boldsymbol{l}_i^T$ and $\boldsymbol{u}_j$ such that $a_{ij} = \boldsymbol{l}_i^T\boldsymbol{u}_j$. When we are at the stage of calculating the ith element of $\boldsymbol{u}_j$, we will already have the i nonzero elements of $\boldsymbol{l}_i^T$ and the first $i1$ elements of $\boldsymbol{u}_j$. The ith element of $\boldsymbol{u}_j$ may therefore be chosen simply as $u_{j(i)} = a_{ij}$ $\boldsymbol{l}_i^T\boldsymbol{u}_j$where the dot-product is calculated assuming $u_{j(i)}$ is zero.

- Figure10: 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 section4.2.2 .

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 to8 , tridiagonal matrices frequently occur in numerical solution of differential equations.

A tridiagonal system may be written as

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = r_i \qquad (38)$$

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. 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 **A** contains m rows and n columns, with $m > n$, the system is probably over-determined (unless there are m-n redundant rows). While the *solution* to $\mathbf{A}\boldsymbol{x} = \boldsymbol{r}$ 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 $\boldsymbol{e} = \mathbf{A}\boldsymbol{x}$-$\boldsymbol{r}$. 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 $rss=\boldsymbol{e}^T\boldsymbol{e}$. Substituting for $\boldsymbol{e}$ we obtain

$$\begin{aligned} rss &= \boldsymbol{e}^T\boldsymbol{e} \\ &= [\boldsymbol{x}^T\mathbf{A}^T\ \boldsymbol{r}^T][\mathbf{A}\boldsymbol{x}\ \boldsymbol{r}] \\ &= \boldsymbol{x}^T\mathbf{A}^T\mathbf{A}\boldsymbol{x}\ 2\boldsymbol{x}^T\mathbf{A}^T\boldsymbol{r} + \boldsymbol{r}^T\boldsymbol{r}, \end{aligned} \tag{39}$$

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

$$\frac{\partial rss}{\partial x} = 2\mathbf{A}^T\mathbf{A}\boldsymbol{x} - 2\mathbf{A}^T\boldsymbol{r} = 0 \tag{40}$$

.

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

Warning: The matrix $\mathbf{A}^T\mathbf{A}$ 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 ***x***.

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

$$\mathbf{Q}^{T}\mathbf{Q} = \mathbf{I}, \qquad (41)$$

where **I** is the identity matrix and **Q** is chosen to transform **A** into

$$QA = \begin{bmatrix} R \\ 0 \end{bmatrix}, \qquad (42)$$

where **R** is a square matrix of a size *n* and **0** is a zero matrix of size *mn* by *n*. The right-hand side of the system **QA*x*** = **Q*r*** becomes

$$Q\boldsymbol{r} = \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{c} \end{bmatrix}, \qquad (43)$$

where **b** is a vector of size *n* and **c** is a vector of size *mn*.

Now the turning point (global minimum) in the residual sum of squares (40) occurs when

$$\begin{aligned} \frac{\partial rss}{\partial x} &= 2\left[A^{T}Ax - A^{T}\boldsymbol{r}\right] \\ &= 2\left[A^{T}Q^{T}QAx - A^{T}Q^{T}Q\boldsymbol{r}\right] \\ &= 2\left[[QA]^{T}QAx - [QA]^{T}Q\boldsymbol{r}\right] \\ &= 2[QA]^{T}[QAx - Q\boldsymbol{r}] \\ &= 2R^{T}[Rx - \boldsymbol{b}] \end{aligned} \qquad (44)$$

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

$$\mathbf{R}\boldsymbol{x} = \boldsymbol{b}. \qquad (45)$$

This system may be solved to obtain the least squares solution ***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 **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* $z=\mathbf{c}^T\mathbf{x}$ is minimised. The "Linear" indicates that the underdetermined system of equations is linear and the objective function is linear in the solution variable $\mathbf{x}$. The "Programming" arose to enhance the chances of obtaining funding for research into this area when it was developing in the 1960s.

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.

Figure11

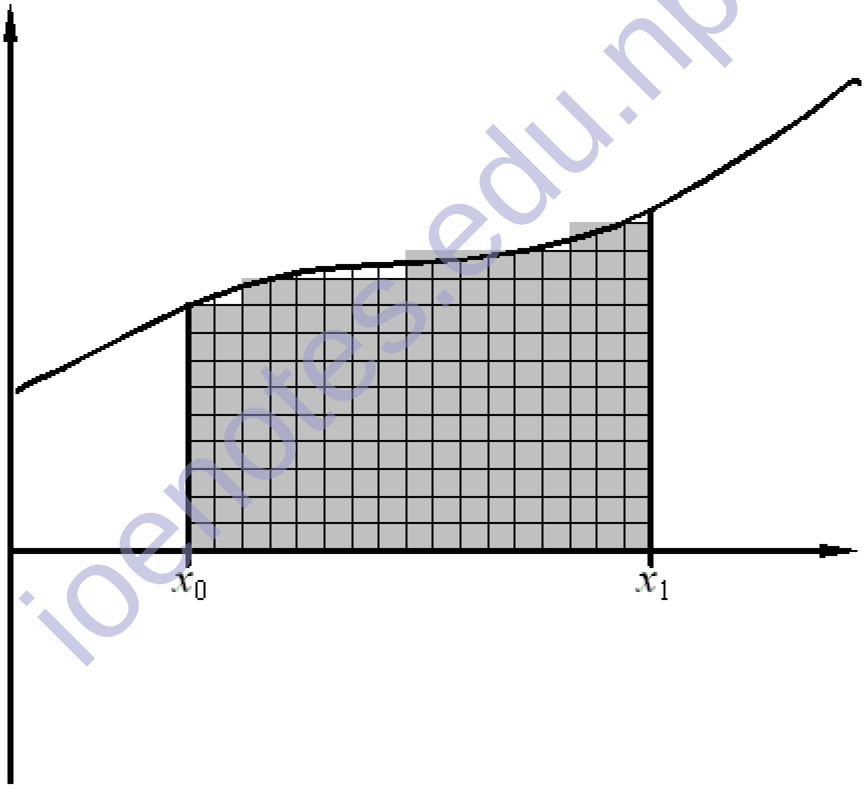


Figure11: Manual method for determining integral by superimposing a grid on a graph of the integrand. The boxes indicated in grey are counted.

5.2 Trapezium rule

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

$$\begin{aligned}\int_{x_0}^{x+\Delta x} f(x)\mathrm{d}x &= f(x_0)\Delta x + \tfrac{1}{2}f'(x_0)\Delta x^2 + \tfrac{1}{6}f''(x_0)\Delta x^3 + \ldots \\ &= \left[\tfrac{1}{2}f(x_0) + \tfrac{1}{2}\left(f(x_0) + f'(x_0)\Delta x + \tfrac{1}{2}f''(x_0)\Delta x + \ldots\right) + \ldots\right]\Delta x \\ &= \tfrac{1}{2}\left(f(x_0) + f(x_0+\Delta x)\right)\Delta x + O\left(\Delta x^3\right)\end{aligned} \tag{46}$$

The approximation represented by $^1/_2[f(x_0) + f(x_0+\Delta x)]\Delta x$ is called the Trapezium Rule based on its geometric interpretation as shown in figure 12 .

Figure12

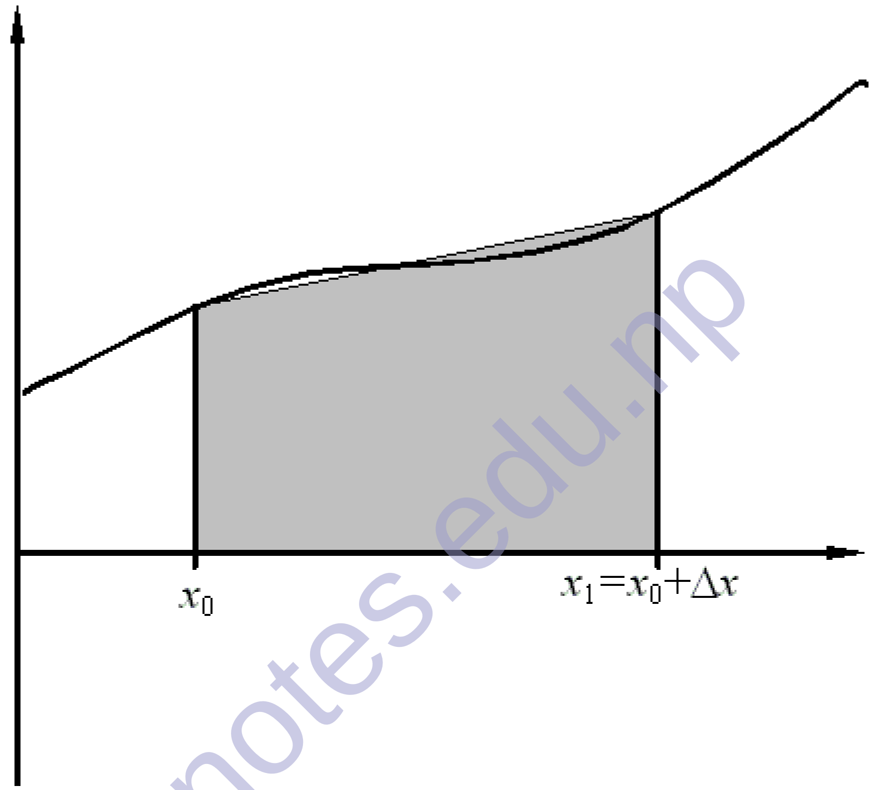


Figure12: Graphical interpretation of the trapezium rule.

As we can see from equation (46), 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 13 .

Figure13

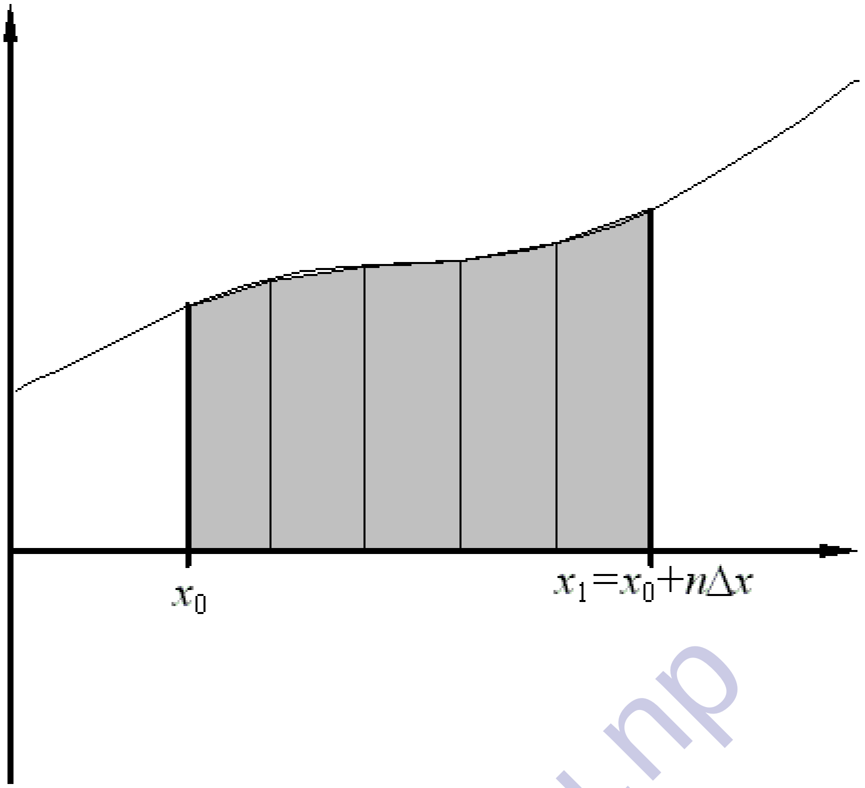


Figure13: Compound Trapezium Rule.

The Compound Trapezium Rule approximation to the integral is therefore

$$\begin{aligned}\int_{x_0}^{x_1} f(x)\mathrm{d}x &\approx \frac{\Delta x}{2}\sum_{i=0}^{n-1} f(x_0+i\Delta x)+f(x_0+(i+1)\Delta x) \\ &= \frac{\Delta x}{2}\left[f(x_0)+2f(x_0+\Delta x)+2f(x_0+2\Delta x)+\ldots+2f(x_0+(n-1)\Delta x)+f(x_1)\right].\end{aligned} \quad (47)$$

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 (46) that the leading order truncation error is scaled by f''.

5.3 Mid-point rule

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

$$\int_{x_0-\frac{1}{2}\Delta x}^{x+\frac{1}{2}\Delta x} f(x)\mathrm{d}x = f(x_0)\Delta x + \frac{1}{24} f''(x_0)\Delta x^3 + \ldots \quad (48)$$

.

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 14 provides a graphical interpretation of this approach.

Figure14

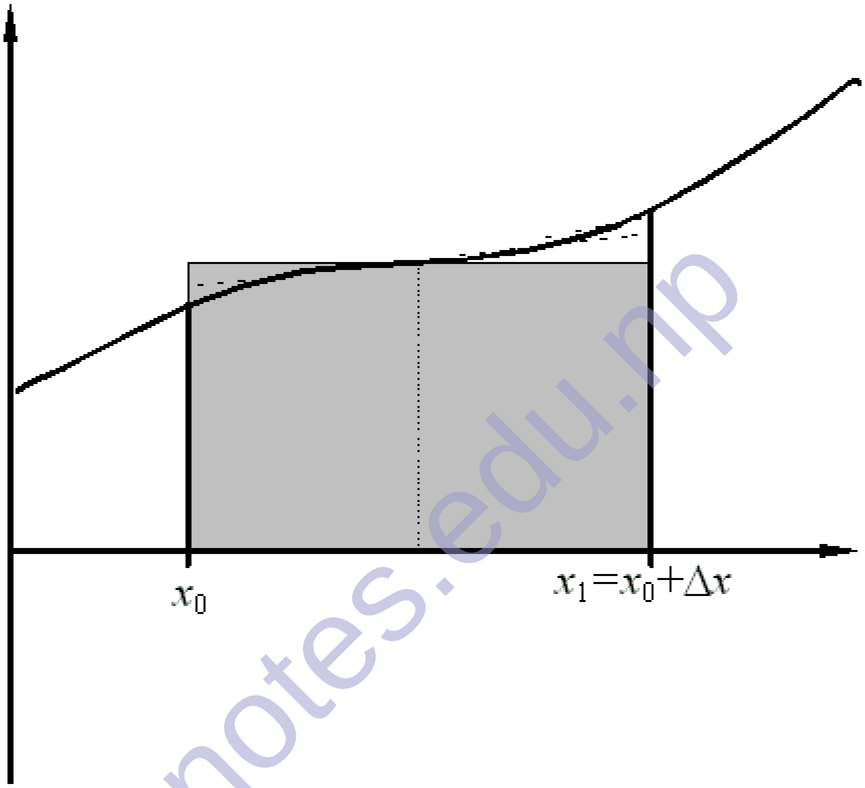


Figure14: Graphical interpretation of the midpoint rule. The grey region defines the midpoint rule as a rectangular approximation with the dashed lines showing alternative trapeziodal aproximations containing the same area.

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

$$\int_{x_0}^{x_1} f(x)\mathrm{d}x \approx \Delta x \sum_{i=0}^{n-1} f\left(x_0 + \left(i + \frac{1}{2}\right)\Delta x\right) \quad (49)$$

,

with the graphical interpretation shown in figure 15 . The difference between the Trapezium Rule and Mid-point Rule is greatly diminished in their compound forms. Comparison of equations (47) and (49) 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.

Figure15

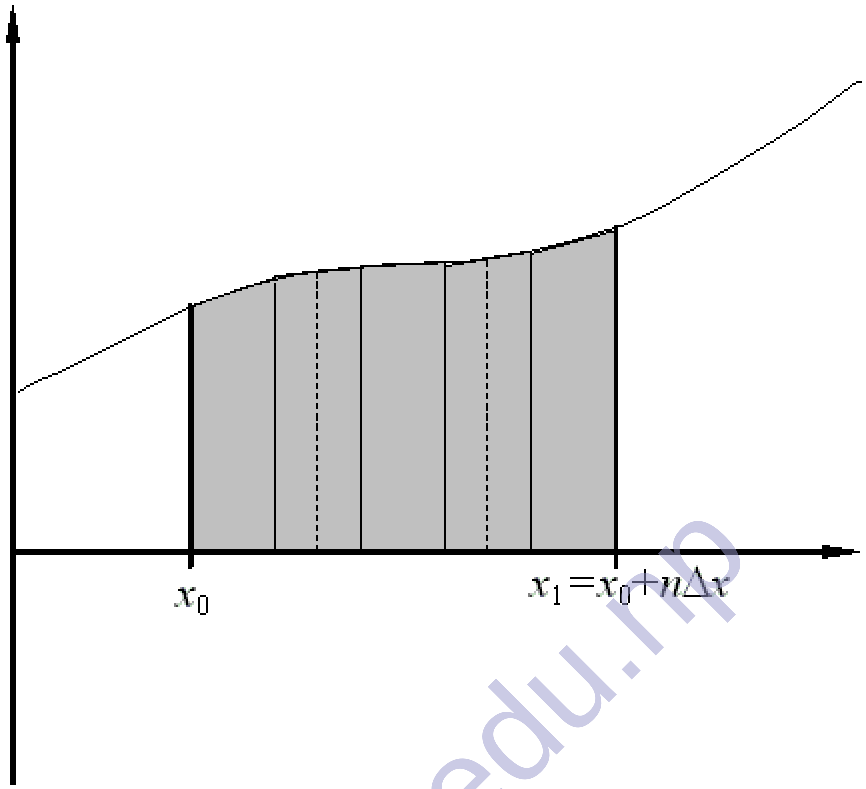


Figure15: Compound Mid-point Rule.

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 16 .

Figure16

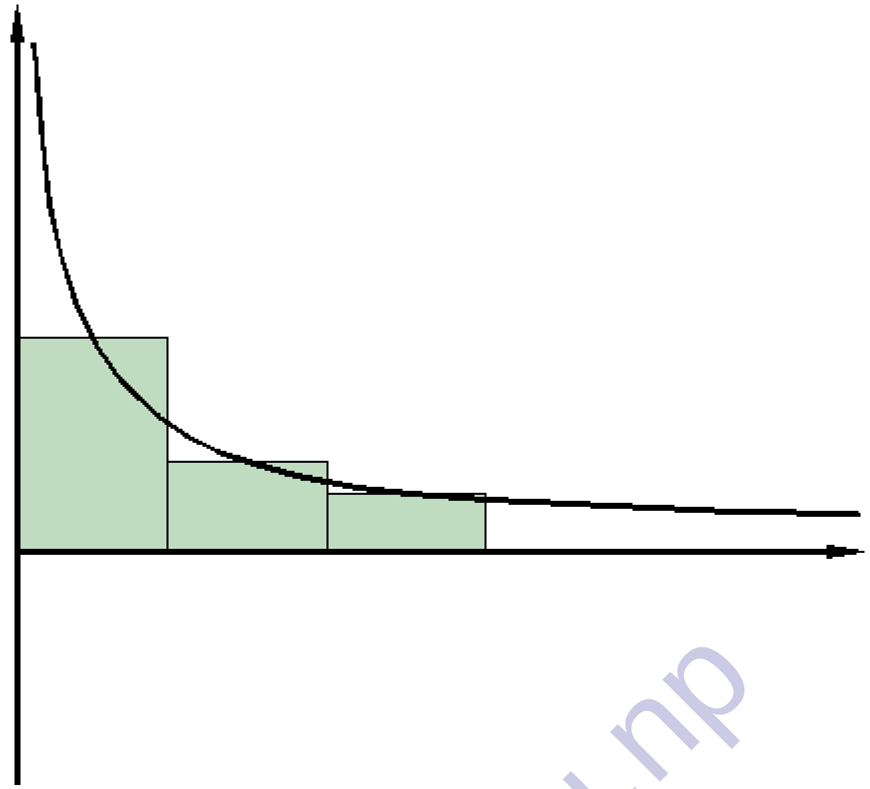


Figure16: Applying the Midpoint Rule where the singular integrand would cause the Trapezium Rule to fail.

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

$$\int_{x_0}^{x+2\Delta x} f(x)\mathrm{d}x = 2f(x_0)\Delta x + 2f'(x_0)\Delta x^2 + \tfrac{4}{3}f''(x_0)\Delta x^3 + \tfrac{2}{3}f'''(x_0)\Delta x^4 + \tfrac{4}{15}f^{iv}(x_0)\Delta x^5 \ldots \quad (50)$$

$$\begin{aligned}
&= \frac{\Delta x}{3}\Big[f(x_0) \\
&+4\left(f(x_0) + f'(x_0)\Delta x + \tfrac{1}{2}f''(x_0)\Delta x^2 + \tfrac{1}{6}f'''(x_0)\Delta x^3 + \tfrac{1}{24}f^{iv}(x_0)\Delta x^4 + \ldots\right) \\
&+\left(f(x_0) + 2f'(x_0)\Delta x + 2f''(x_0)\Delta x^2 + \tfrac{4}{3}f'''(x_0)\Delta x^3 + \tfrac{2}{3}f^{iv}(x_0)\Delta x^4 + \ldots\right) \\
&-\tfrac{17}{30}f^{iv}(x_0)\Delta x^4 \ldots\Big] \\
&= \frac{\Delta x}{3}\left(f(x_0) + 4f(x_0+\Delta x) + f(x_0+2\Delta x)\right) + O\left(\Delta x^5\right)
\end{aligned}$$

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

$$\begin{aligned}\int_{x_0}^{x_1} f(x)\mathrm{d}x &\approx \frac{\Delta x}{3}\sum_{i=0}^{m-1} f(x_0+2i\Delta x)+4f(x_0+(2i+1)\Delta x)+f(x_0+(2i+2)\Delta x) \\ &= \frac{\Delta x}{3}\left[f(x_0)+4f(x_0+\Delta x)+2f(x_0+2\Delta x)+\ldots+4f(x_0+(n-1)\Delta x)+f(x_1)\right],\end{aligned} \tag{51}$$

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

5.5 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 17 . 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 17) the maximum deviation (indicated by arrows in figure 17) 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

$$\mathit{area} = {}^{2}/_{3}\ \mathit{chord}\ \mathit{maxDeviation}, \tag{52}$$

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.

Figure17

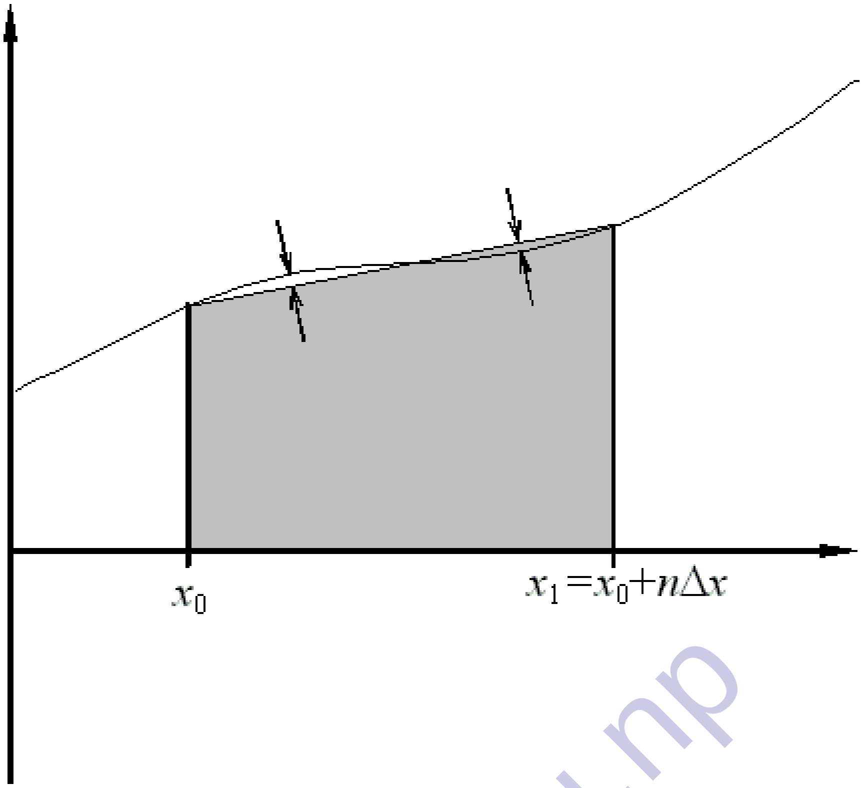


Figure17: Quadratic triangulation to determine the area using a manual combination of the Trapezium and Simpson's Rules.

5.6 Romberg integration

With the Compound Trapezium Rule we know from section 5.2 the error in some estimate $T(\Delta x)$ of the integral I using a step size Δx goes like $c\Delta x^2$ as $\Delta x0$, 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 $T(\Delta x/2)$:

$$T^{(1)}(\Delta x/2) = \alpha T(\Delta x/2) + (1-\alpha)T(\Delta x)$$
$$= \alpha[I + c\Delta x^2/4 + O(\Delta x^4)] + (1-\alpha)[I + c\Delta x^2 + O(\Delta x^4)]. \qquad (53)$$

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

$$T^{(1)}(\Delta x/2) = [4T(\Delta x/2) - T(\Delta x)]/3. \qquad (54)$$

Comparison with equation (51) 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

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

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

5.7 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

$$\int_{x_0}^{x_1=x_0+\Delta x} f(x)\,\mathrm{d}x \approx \frac{\Delta x}{2}\left[f\left(x_0+\left(\frac{1}{2}-\frac{\sqrt{3}}{6}\right)\Delta x\right)+f\left(x_0+\left(\frac{1}{2}+\frac{\sqrt{3}}{6}\right)\Delta x\right)\right]+O\left(\Delta x^4\right). \tag{56}$$

In general it is possible to choose *M* function evaluations per interval to obtain a formula exact for all polynomials of degree *2M1* and less.

The Gauss Quadrature accurate to order *2M1* 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 $x_0+\beta\Delta x$:

$$\begin{aligned}\int_{x_0}^{x_1=x_0+\Delta x} f(x)\mathrm{d}x &= \Delta x f(x_0)+\frac{\Delta x^2}{2}f'(x_0)+\frac{\Delta x^3}{6}f''(x_0)+\frac{\Delta x^4}{24}f'''(x_0)+O\left(\Delta x^5\right)\\ &= \frac{\Delta x}{2}\left[f(x_0)+\alpha\Delta x f'(x_0)+\frac{(\alpha\Delta x)^2}{2}f''(x_0)+\frac{(\alpha\Delta x)^3}{6}f'''(x_0)+\ldots\right.\\ &\quad\left.+f(x_0)+\beta\Delta x f'(x_0)+\frac{(\beta\Delta x)^2}{2}f''(x_0)+\frac{(\beta\Delta x)^3}{6}f'''(x_0)+\ldots\right]\\ &= \Delta x f(x_0)+(\alpha+\beta)\frac{\Delta x^2}{2}f'(x_0)+\left(\alpha^2+\beta^2\right)\frac{\Delta x^3}{4}+\left(\alpha^3+\beta^3\right)\frac{\Delta x^4}{12}+\ldots\end{aligned} \tag{57}$$

Equating the various terms reveals

$$\begin{gathered}\alpha+\beta=1\\ (\alpha^2+\beta^2)/4 = 1/6,\end{gathered} \tag{58}$$

the solution of which gives the positions stated in equation (56).

5.8 Example of numerical integration

Consider the integral

$$\int_0^\pi \sin x \, dx = 2 \qquad (59)$$

,

which may be integrated numerically using any of the methods described in the previous sections. Table 2 gives the error in the numerical estimates for the Trapezium Rule, Midpoint Rule, Simpson's Rule and Gauss Quadrature. The results are presented in terms of the number of function evaluations required. The calculations were performed in double precision.

No. intervals	**Trapezium Rule**	**Midpoint Rule**	**Simpson's Rule**	**Gauss Quadrature**
1		1.14189790E+00		
2	-2.00000000E+00	2.21441469E-01		-6.41804253E-02
4	-4.29203673E-01	5.23443059E-02	9.43951023E-02	-3.05477319E-03
8	-1.03881102E-01	1.29090855E-02	4.55975498E-03	-1.79666460E-04
16	-2.57683980E-02	3.21637816E-03	2.69169948E-04	-1.10640837E-05
32	-6.42965622E-03	8.03416309E-04	1.65910479E-05	-6.88965642E-07
64	-1.60663902E-03	2.00811728E-04	1.03336941E-06	-4.30208237E-08

128	-4.01611359E-04	5.02002859E-05	6.45300022E-08	-2.68818500E-09
256	-1.00399815E-04	1.25499060E-05	4.03225719E-09	-1.68002278E-10
512	-2.50997649E-05	3.13746618E-06	2.52001974E-10	-1.04984909E-11
1024	-6.27492942E-06	7.84365898E-07	1.57500679E-11	-6.55919762E-13
2048	-1.56873161E-06	1.96091438E-07	9.82769421E-13	
4096	-3.92182860E-07	4.90228564E-08		
8192	-9.80457133E-08	1.22557182E-08		
16384	-2.45114248E-08	3.06393221E-09		
32768	-6.12785222E-09	7.65979280E-10		
65536	-1.53194190E-09	1.91497040E-10		
131072	-3.82977427E-10	4.78341810E-11		

262144	-9.57223189E-11	1.19970700E-11		
524288	-2.39435138E-11	3.03357339E-12		
1048576	-5.96145355E-12			

Table 2 : Error in numerical integration of (59) as a function of the number of subintervals.

5.8.1 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                                         =
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

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

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t,y) \quad , \qquad (60)$$

subject to some boundary/initial condition $f(t=t_0) = c$. 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

$$Y'_n = f(t_n,Y_n) \qquad (61)$$

at each of the points in the domain.

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

$$\begin{aligned}
&\ldots \\
Y_{n-2} &= Y_n - 2\Delta t Y'_n + 2\Delta t^2 Y''_n - \frac{4}{3}\Delta t^3 Y'''_n + O(\Delta t^4) \\
Y_{n-1} &= Y_n - \Delta t Y'_n + \frac{1}{2}\Delta t^2 Y''_n - \frac{1}{6}\Delta t^3 Y'''_n + O(\Delta t^4) \\
Y_n &= Y_n \\
Y_{n+1} &= Y_n + \Delta t Y'_n + \frac{1}{2}\Delta t^2 Y''_n + \frac{1}{6}\Delta t^3 Y'''_n + O(\Delta t^4) \\
Y_{n+2} &= Y_n + 2\Delta t Y'_n + 2\Delta t^2 Y''_n + \frac{4}{3}\Delta t^3 Y'''_n + O(\Delta t^4) \\
&\ldots \qquad ,
\end{aligned} \qquad ()$$

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

$$Y'_n \approx \sum_{i=a}^{b} \alpha_i Y_{n+i} \quad . \qquad ()$$

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

$$\sum_{i=a}^{b} \alpha_i = 0 \qquad (64)$$

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. Before doing so, 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

$$E_n = Y_n - y_n. \qquad (65)$$

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

$$e_n = Y_n - y_n^*, \qquad (66)$$

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 the sum of e_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. By approximating the derivative in (61) as

$$Y'_n =\sim (Y_{n+1} - Y_n)/\Delta t, \qquad (67)$$

in our differential equation for Y_n we obtain

$$Y_{n+1} = Y_n + \Delta t f(t_n, Y_n). \qquad (68)$$

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 18 .

Figure18

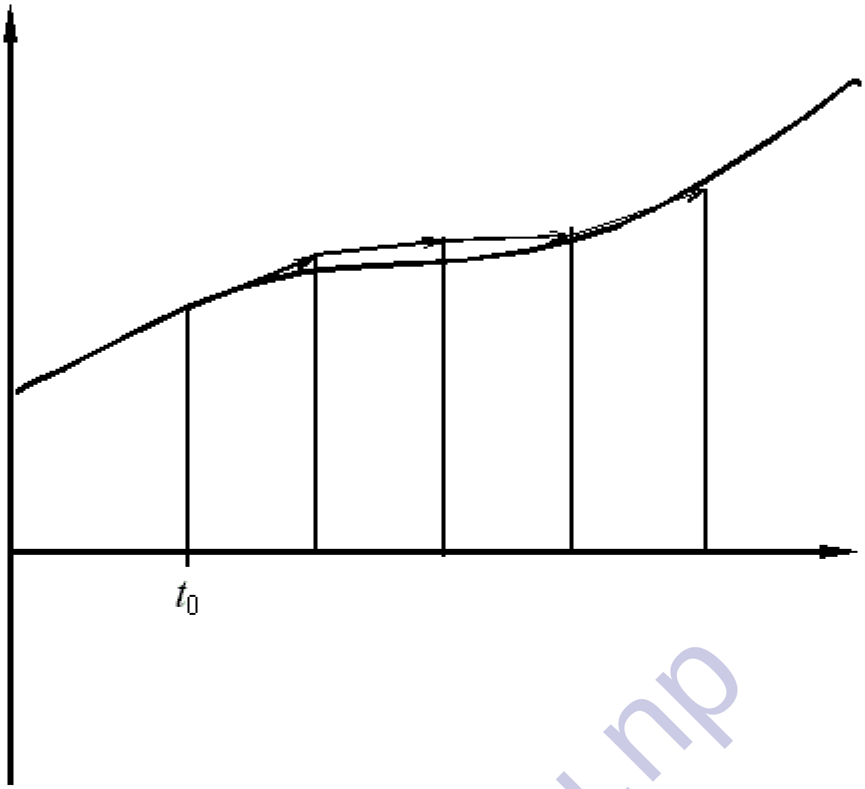


Figure18: Sketch of the function $y(t)$ (dark line) and the Euler method solution (arrows). Each arrow is tangental to to the solution of (60) passing through the point located at the start of the arrow. Note that this point need not be on the desired $y(t)$ curve.

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. Inspection of our approximation for Y'_n shows the error term is of order Δt^2 in our time step formula. This shows that the Euler method is 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 section6.7).

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 section6.7).

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

$$Y'_n =\sim (Y_n - Y_{n-1})/\Delta t, \tag{69}$$

to give the evolution equation

$$Y_{n+1} = Y_n + \Delta t f(t_{n+1}, Y_{n+1}). \tag{70}$$

This is shown graphically in figure 19 .

Figure19

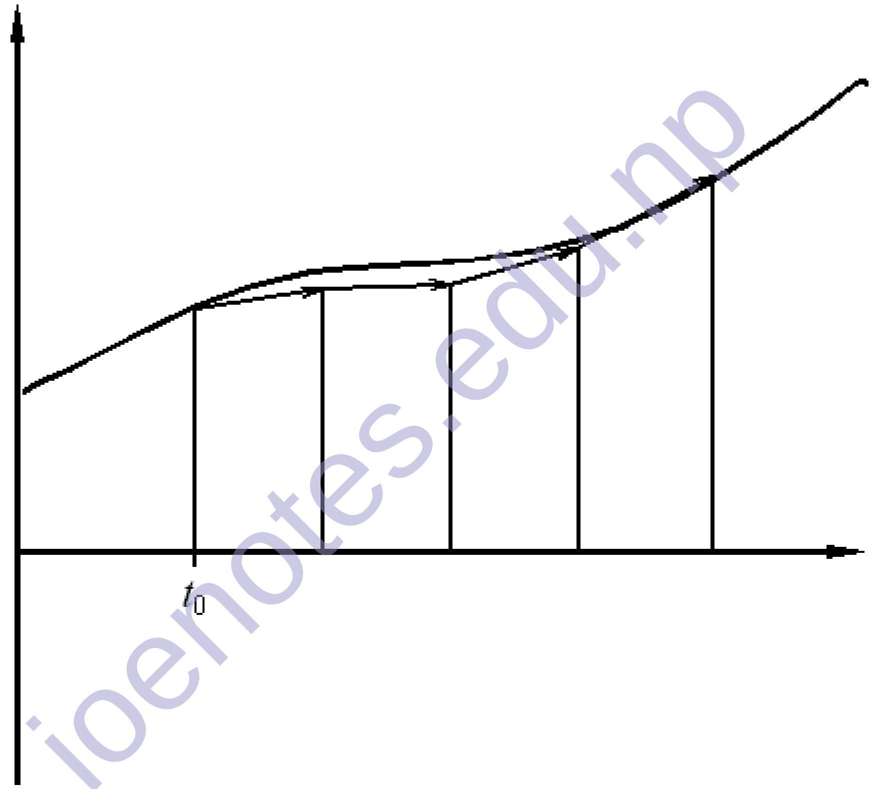


Figure19: Sketch of the function $y(t)$ (dark line) and the Euler method solution (arrows). Each arrow is tangental to to the solution of (60) passing through the point located at the end of the arrow. Note that this point need not be on the desired $y(t)$ curve.

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 20 .

Figure20

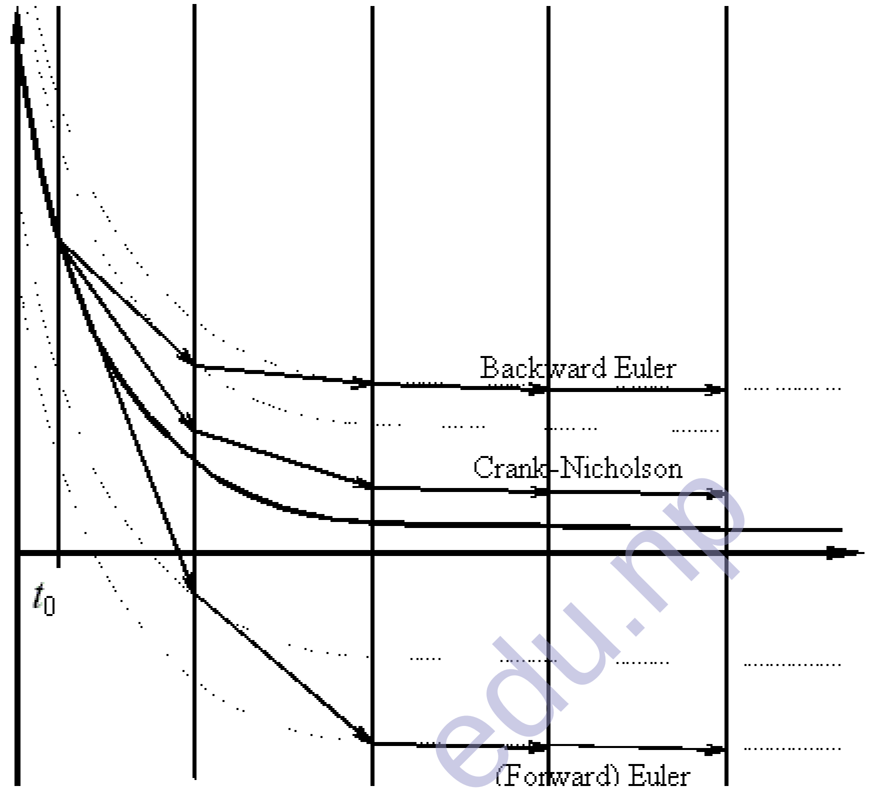


Figure20: Comparison of ordinary differential equation solvers for a stiff problem.

6.5.2 Richardson extrapolation

The Romberg integration approach (presented in section5.6) 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 $\Delta t0$. Combining these two estimates to try and cancel the $O(\Delta t^2)$ errors gives the improved estimate as

$$Y^{(1)}(t,\Delta t/2) = [4Y(t,\Delta t/2) - Y(t,\Delta t)]/3. \qquad (71)$$

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

$$Y'_{n+1/2} =\sim (Y_{n+1} - Y_n)/\Delta t. \tag{72}$$

Substitution into our differential equation for Y_n gives

$$(Y_{n+1} - Y_n)/\Delta t =\sim f(t_{n+1/2}, Y_{n+1/2}). \tag{73}$$

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

$$Y_{n+1} - Y_n =\sim {}^1/_2[f(t_{n+1}, Y_{n+1}) + f(t_n, Y_n)]\Delta t. \tag{74}$$

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::

$$\begin{aligned} y_{n+1} &= y_n + \frac{dy}{dt}\Delta t + \frac{1}{2}\frac{d^2y}{dt^2}\Delta t^2 + O(\Delta t^3) \\ &= y_n + f_n\Delta t + \frac{1}{2}\left(\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial y}\right)\Delta t^2 + O(\Delta t^3) \end{aligned} \tag{75a}$$

$$\begin{aligned} f(t_{n+1}, y_{n+1}) &= f(t_n, y_n) + \frac{\partial f}{\partial t}\Delta t + \frac{\partial f}{\partial y}\Delta y + O(\Delta t^2) + O(\Delta y^2) \\ &= f_n + \frac{\partial f}{\partial t}\Delta t + \frac{\partial f}{\partial y}\frac{dy}{dt}\Delta t + O(\Delta t^2) \\ &= f_n + \left(\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial y}\right)\Delta t + O(\Delta t^2) \end{aligned} \tag{75 b}$$

Substitution of these into the left- and right-hand sides of equation (74) reveals

$$y_{n+1} - y_n = f_n\Delta t + \frac{1}{2}\left(\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial y}\right)\Delta t^2 + O(\Delta t^3) \tag{76a}$$

and

$$\frac{1}{2}\left(f\left(t_n,y_n\right)+f\left(t_{n+1},y_{n+1}\right)\right)\Delta t=\frac{1}{2}\left(f_n+f_n+\left(\frac{\partial f}{\partial t}+f\frac{\partial f}{\partial y}\right)\Delta t+O\left(\Delta t^2\right)\right)\Delta t$$
$$=f_n\Delta t+\frac{1}{2}\left(\frac{\partial f}{\partial t}+f\frac{\partial f}{\partial y}\right)\Delta t^2+O\left(\Delta t^3\right) \qquad (76\ b)$$

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

$$Y'_n = f_n,$$
$$Y''_n =\sim (f_n - f_{n-1})/\Delta t \qquad (77)$$

and so we may construct a second order method as

$$Y_{n+1} = Y_n + \Delta t Y'_n + ½\Delta t^2 Y''_n$$
$$= Y_n + {}^1/_2\Delta t(3f_n - f_{n-1}). \qquad (78)$$

For s=3 we also have Y'''_n and so can make use of a second-order one-sided finite difference to approximate $Y''_n = f'_n = (3f_n\ 4\ f_{n1} + f_{n2})/2\Delta t$ and include the third order $Y'''_n = f''_n = (f_n\ 2f_{n1}+f_{n2})/\Delta t^2$ to obtain

$$Y_{n+1} = Y_n + \Delta t Y'_n + {}^1/_2\Delta t^2 Y''_n + {}^1/_6\Delta t^3 Y''_n$$
$$= Y_n + {}^1/_{12}\Delta t(23f_n - 16f_{n-1} + 5f_{n-2}). \qquad (79a)$$

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

$$Y'_n = f_n,$$
$$Y''_n =\sim (f_{n+1} - f_{n-1})/2\Delta t$$
$$Y'''_n =\sim (f_{n+1} - 2f_n + f_{n-1})/\Delta t^2, \qquad (80)$$

to give

$$Y_{n+1} = Y_n + \Delta t Y'_n + {}^1/_2\Delta t^2 Y''_n + {}^1/_6\Delta t^3 Y'''_n$$
$$= Y_n + (1/12)\Delta t(5f_{n+1} + 8f_n - f_{n-1}). \qquad (81)$$

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

$$y' = \lambda y, \tag{82}$$

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

$$Y_{n+1} = (1 + \lambda\Delta t)Y_n = (1 + \lambda\Delta t)^2 Y_{n-1} = \dots = (1 + \lambda\Delta t)^{n+1} Y_0. \tag{83}$$

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

$$|1 + \lambda\Delta t| <= 1. \tag{84}$$

If we choose a time step Δt which does not satisfy (84) then Y_n will increase without limit. This condition (84) 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 (82) 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 $y_\$ = y + \varepsilon$ 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

$$\begin{aligned} y_\$' &= y + \varepsilon \\ &= f(t, y_\$) \\ &= f(t, y + \varepsilon) \\ &= f(t,y) + \varepsilon\frac{\partial f}{\partial y} + O(\varepsilon^2) \end{aligned} \quad . \tag{85}$$

Thus, to the leading order, the error obeys an equation of the form given by (82), with $\lambda = f/y$. 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 (84) 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 (82) by the Backward Euler method

$$Y_{n+1} = Y_n + \Delta t\lambda Y_{n+1}, \tag{86}$$

can be rearranged for Y_{n+1} and

$$Y_{n+1} = Y_n/(1 - \lambda\Delta t) = Y_{n-1}/(1 - \lambda\Delta t)^2 = \ldots = Y_0/(1 - \lambda\Delta t)^{n+1}, \tag{87}$$

which will be stable provided

$$|1 - \lambda\Delta t| > 1. \tag{88}$$

For $\mathrm{Re}\lambda <= 0$ 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

$$(1\lambda\Delta t/2)Y_{n+1} = (1+\lambda\Delta t/2)Y_n, \tag{89}$$

to arrive at

$$Y_{n+1} = [(1+\lambda\Delta t/2)/(1 - \lambda\Delta t/2)]^{n+1}\ Y_0, \tag{90}$$

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

Predictor-corrector methods try to combine the advantages of the simplicity of explicit methods with the improved stability and 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

$$Y_{n+1}^{(1)} = Y_n + \Delta t f(t_n, Y_n), \tag{91}$$

and then the Backward Euler to give the corrector

$$Y^{(2)}_{n+1} = Y_n + \Delta t f(t_n, Y_{n+1}^{(1)}). \tag{92}$$

The final solution is the mean of these:

$$Y_{n+1} = (Y_{n+1}^{(1)} + Y_{n+1}^{(2)})/2. \qquad (93)$$

To understand the stability of this method we again use the $y' = \lambda y$ so that the three steps described by equations (91) to (93) become

$$Y_{n+1}^{(1)} = Y_n + \lambda\Delta t Y_n, \qquad (94a)$$

$$\begin{aligned} Y_{n+1}^{(2)} &= Y_n + \lambda\Delta t Y_{n+1}^{(1)} \\ &= Y_n + \lambda\Delta t\,(Y_n + \lambda\Delta t Y_n) \\ &= (1 + \lambda\Delta t + \lambda^2\Delta t^2)Y_n, \end{aligned} \qquad (\ 94\ b)$$

$$\begin{aligned} Y_{n+1} &= (Y_{n+1}^{(1)} + Y_{n+1}^{(2)})/2 \\ &= [(1 + \lambda\Delta t)Y_n + (1 + \lambda\Delta t + \lambda^2\Delta t^2)\ Y_n]/2 \\ &= (1 + \lambda\Delta t + {}^1/_2\lambda^2\Delta t^2)\ Y_n \\ &= (1 + \lambda\Delta t + {}^1/_2\lambda^2\Delta t^2)^n\ Y_0. \end{aligned} \qquad (\ 94\ c)$$

Convergence 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. 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 (94 c) 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:

$$k^{(1)} = \Delta t f(t_n, Y_n)\ , \qquad (95a)$$

$$k^{(2)} = \Delta t f(t_n + {}^1/_2\Delta t\ , Y_n + ½k^{(1)})\ , \qquad (\ 95\ b)$$

$$k^{(3)} = \Delta t f(t_n + {}^1/_2\Delta t\ , Y_n + ½k^{(2)})\ , \qquad (\ 95\ c)$$

$$k^{(4)} = \Delta t f(t_n + \Delta t\ , Y_n + k^{(3)})\ , \qquad (\ 95\ d)$$

$$Y_{n+1} = Y_n + (k^{(1)} + 2k^{(2)} + 2k^{(3)} + k^{(4)})/6. \qquad (\ 95\ e)$$

In order to analyse this we need to construct Taylor-Series expansions for $k^{(2)} = \Delta t f(t_n + {}^1/_2\Delta t, Y_n + {}^1/_2 k^{(1)}) = \Delta t[f(t_n, Y_n) + (\Delta t/2)(f/t + ff/y)]$, 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 requiring $Y'' = df/dt = f/t + f\,f/y$, $Y''' = d^2f/dt^2 = {}^2f/t^2 + 2f\,{}^2f/ty + f/t\,f/y + f^2\,{}^2f/y^2 + f\,(f/y)^2$, and similarly for Y''''. All terms up to order Δt^4 can be shown to match, with the error coming in at Δt^5.

Goto next document (HighODE)

7. Higher order ordinary differential equations

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

$$\frac{d^n y}{dt^n} = f\left(t, y, \frac{dy}{dt}, \frac{d^2 y}{d^2 t}, \ldots, \frac{d^{n-1} y}{dt^{n-1}}\right) \qquad (96)$$

,

where at $t = t_0$ we know the values of y, dy/dt, d^2y/dt^2, …, $d^{n1}y/dt^{n1}$. By writing $x_0=y$, $x_1=dy/dt$, $x_2=d^2y/dt^2$, …, $x_{n-1}=d^{n-1}y/dt^{n-1}$, we may express this as the system of equations

$$\begin{aligned} x_0' &= x_1 \\ x_1' &= x_2 \\ x_2' &= x_3 \\ &\;.\;.\;.\;. \\ x_{n-2}' &= x_{n-1} \\ x_{n-1}' &= f(t,x_0,x_1,\ldots,x_{n-2}), \end{aligned} \qquad (97)$$

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 section7.1 . 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 $n\text{-}m$ boundary conditions at $t = t_1$, we will require guesses for $n\text{-}m$ initial conditions. The computational cost of refining these $n\text{-}m$ 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 $\Delta t = (t_1\text{-}t_0)/N$ to produce a system of N+1 simultaneous equations. Consider the second order linear system

$$y'' + ay' + by = c, \qquad (98)$$

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

$$y'_i \;{=}{\sim}\; (Y_{i+1} - Y_{i-1})/2\Delta t, \qquad (99a)$$

$$y''_i \;{=}{\sim}\; (Y_{i+1} - 2Y_i + Y_{i-1})/\Delta t^2, \qquad (99 b)$$

we can write the system as

$$\begin{gathered}
Y_0 = \alpha,\\
(1+{}^1/_2 a\Delta t)Y_0 + (b\Delta t^2\text{-}2)Y_1 + (1\text{-}{}^1/_2 a\Delta t)Y_2 = c\Delta t^2,\\
(1+{}^1/_2 a\Delta t)Y_1 + (b\Delta t^2\text{-}2)Y_2 + (1\text{-}{}^1/_2 a\Delta t)Y_3 = c\Delta t^2,\\
\ldots\\
(1+{}^1/_2 a\Delta t)Y_{n-1} + (b\Delta t^2\text{-}2)Y_{n-1} + (1\text{-}{}^1/_2 a\Delta t)Y_n = c\Delta t^2,\\
Y_n - Y_{n-1} = \beta\Delta t \qquad (100)
\end{gathered}$$

This tridiagonal system may be readily solved using the method discussed in section4.5 .

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 section4.4 .

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[*]

7.3.1 Truncation error[*]

7.3.2 Error and step control[*]

8. Partial differential equations

8.1 Laplace equation

Consider the Laplace equation in two dimensions

$$\nabla^2\varphi = \frac{\partial^2\varphi}{\partial x^2} + \frac{\partial^2\varphi}{\partial y^2} = 0 , \qquad (101)$$

in some rectangular domain described by x in $[x_0,x_1]$, y in $[y_0,y_1]$. Suppose we discretise the solution onto a $m+1$ by $n+1$ rectangular grid (or mesh) given by $x_i = x_0 + i\Delta x$, $y_j = y_0 + j\Delta y$ 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 $_{ij} = (x_i,y_j)$ be the exact solution at the mesh point i,j, and $\Phi_{ij} =\sim {}_{ij}$ be the approximate solution at that mesh point.

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

$$\varphi_{i+1,j} = \varphi_{i,j} + \Delta x\frac{\partial\varphi_{i,j}}{\partial x} + \frac{\Delta x^2}{2}\frac{\partial^2\varphi_{i,j}}{\partial x^2} + O(\Delta x^3) , \qquad (102a)$$

$$\varphi_{i-1,j} = \varphi_{i,j} - \Delta x\frac{\partial\varphi_{i,j}}{\partial x} + \frac{\Delta x^2}{2}\frac{\partial^2\varphi_{i,j}}{\partial x^2} + O(\Delta x^3) , \qquad (102b)$$

$$\varphi_{i,j+1} = \varphi_{i,j} + \Delta y\frac{\partial\varphi_{i,j}}{\partial y} + \frac{\Delta y^2}{2}\frac{\partial^2\varphi_{i,j}}{\partial y^2} + O(\Delta y^3) , \qquad (102b)$$

$$\varphi_{i,j-1} = \varphi_{i,j} - \Delta y\frac{\partial\varphi_{i,j}}{\partial y} + \frac{\Delta y^2}{2}\frac{\partial^2\varphi_{i,j}}{\partial y^2} + O(\Delta y^3) , \qquad (102b)$$

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

$$\frac{\Phi_{i+1,j} - 2\Phi_{i,j} + \Phi_{i-1,j}}{\Delta x^2} + \frac{\Phi_{i,j+1} - 2\Phi_{i,j} + \Phi_{i,j-1}}{\Delta y^2} = 0 \qquad (103)$$

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 $_{ij}$. The system of linear equations described by (103) 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 , our finite difference approximation is itself linear and the resulting system of equations may be solved directly using Gauss Elimination as discussed in section4.1 . 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.

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. 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 section4.4). 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 $\mathbf{Ax} = \boldsymbol{b}$ which may be used to reduce a block tridiagonal matrix (and the corresponding right-hand side) of the form

$$A = \begin{bmatrix} \hat{A} & I & & & & & & \\ I & \hat{A} & I & & & & & \\ & I & \hat{A} & I & & & & \\ & & I & \hat{A} & I & & & \\ & & & I & \hat{A} & I & & \\ & & & & I & \hat{A} & \ddots & \\ & & & & & \ddots & \ddots & I \\ & & & & & & I & \hat{A} \end{bmatrix}, \qquad (104)$$

into a block diagonal matrix of the form

$$\begin{bmatrix} \hat{B} & & & & & & & \\ & \hat{B} & & & & & & \\ & & \hat{B} & & & & & \\ & & & \hat{B} & & & & \\ & & & & \hat{B} & & & \\ & & & & & \hat{B} & \ddots & \\ & & & & & \ddots & \ddots & \\ & & & & & & & \hat{B} \end{bmatrix}, \qquad (105)$$

where $\hat{A}$ and $\hat{B}$ are themselves block tridiagonal matrices and **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

$$\Phi^*_{ij} = (\Phi_{i+1,\mathrm{j}} + \Phi_{i-1,\mathrm{j}} + \Phi_{i,j+1} + \Phi_{i,j-1})/4. \tag{106}$$

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

The coefficients in the expression (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, $^4\Psi = 0$) 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.

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.

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

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

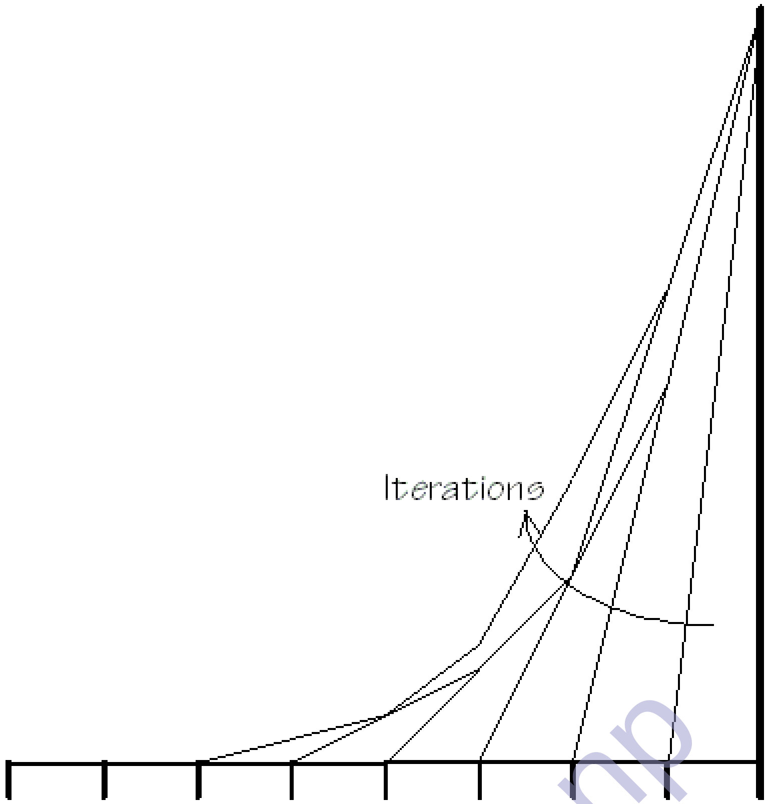
for each internal mesh point, we use

$$\Phi^*_{ij} = (1-\sigma)\Phi_{ij} + \sigma(\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1})/4, \tag{107}$$

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 $d^2/dx^2 = 0$ subject to $\phi(x=0) = 0$ and $\phi(x=1) = 1$. 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 $\boldsymbol{\Phi}^{(p)}$
2. If at coarsest resolution ($p=0$) then solve $\mathbf{A}^{(p)}\boldsymbol{\Phi}^{(p)}=\boldsymbol{b}^{(p)}$ *exactly* and jump to step 7
3. Relax the solution at the current grid resolution, applying boundary conditions
4. Calculate the error $\boldsymbol{r} = \mathbf{A}\boldsymbol{\Phi}^{(p)}-\boldsymbol{b}^{(p)}$
5. Coarsen the error $\boldsymbol{b}^{(p-1)}\boldsymbol{r}$ to the next coarser grid and decrement p
6. Repeat from step 2
7. Refine the correction to the next finest grid $\boldsymbol{\Phi}^{(p+1)} = \boldsymbol{\Phi}^{(p+1)}+\alpha\boldsymbol{\Phi}^{(p)}$ 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 section3.6 . We start by writing the vector function

$$\boldsymbol{f}(\boldsymbol{x}) = \mathbf{A}\boldsymbol{x}\ \boldsymbol{b}, \qquad (108)$$

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

$$\boldsymbol{x}_{n+1} = \boldsymbol{g}(\boldsymbol{x}_n), \qquad (109)$$

where

$$\boldsymbol{g}(\boldsymbol{x}) = \mathbf{D}^{1}\{[\mathbf{A}+\mathbf{D}]\boldsymbol{x}\ \boldsymbol{b}\}, \qquad (110)$$

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 (section3.6). Let us assume the exact solution is $\boldsymbol{x}^* = \boldsymbol{g}(\boldsymbol{x}^*)$, then

$$\begin{aligned} \boldsymbol{\varepsilon}_{n+1} &= \boldsymbol{x}_{n+1}\ \boldsymbol{x}^* \\ &= \mathbf{D}^{1}\{[\mathbf{A}+\mathbf{D}]\boldsymbol{x}_n\ \boldsymbol{b}\}\ \mathbf{D}^{1}\{[\mathbf{A}+\mathbf{D}]\boldsymbol{x}^*\ \boldsymbol{b}\} \\ &= \mathbf{D}^{1}[\mathbf{A}+\mathbf{D}](\boldsymbol{x}_n\ \boldsymbol{x}^*) \\ &= \mathbf{D}^{1}[\mathbf{A}+\mathbf{D}]\boldsymbol{\varepsilon}_n \\ &= \{\mathbf{D}^{1}[\mathbf{A}+\mathbf{D}]\}^{n+1}\ \boldsymbol{\varepsilon}_0. \end{aligned}$$

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

$$||\boldsymbol{\varepsilon}_{n+1}|| = ||\mathbf{B}\boldsymbol{\varepsilon}_n|| < ||\boldsymbol{\varepsilon}_n||, \tag{111}$$

where $\mathbf{B} = \mathbf{D}^{1}[\mathbf{A}+\mathbf{D}]$ for some suitable norm. As any error vector $\boldsymbol{\varepsilon}_n$ 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

$$\mathbf{B}\boldsymbol{\varepsilon}_n = \lambda\boldsymbol{\varepsilon}_n, \tag{112}$$

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,

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & 1 & -2 & 1 & & \\ & & & 1 & -2 & 1 & \\ & & & & \ddots & \ddots & \ddots \\ & & & & & 1 & -2 \end{bmatrix} \tag{113}$$

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

$$\mathbf{B} = \begin{bmatrix} 0 & 1/2 & & & & & \\ 1/2 & 0 & 1/2 & & & & \\ & 1/2 & 0 & 1/2 & & & \\ & & 1/2 & 0 & 1/2 & & \\ & & & 1/2 & 0 & 1/2 & \\ & & & & \ddots & \ddots & \ddots \\ & & & & & 1/2 & 0 \end{bmatrix} \qquad (114)$$

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

$$\det(\mathbf{B}\lambda\mathbf{I}) = 0. \qquad (115)$$

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

$$\det(\mathbf{B}\lambda)_{(n)} = \lambda \det(\mathbf{B}\lambda)_{(n1)} \; {}^{1}/_{4} \det(\mathbf{B}\lambda)_{(n2)} \text{ ,} \qquad (116)$$

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

$$\begin{gathered} \det(\mathbf{B}\lambda)_{(1)} = \lambda \text{ ,} \\ \det(\mathbf{B}\lambda)_{(2)} = \lambda^2 \; ¼ \text{ ,} \\ \det(\mathbf{B}\lambda)_{(3)} = \lambda^3 + ½\lambda \text{ ,} \\ \det(\mathbf{B}\lambda)_{(4)} = \lambda^4 \; ¾ \lambda^2 + (1/16) \text{ ,} \\ \det(\mathbf{B}\lambda)_{(5)} = \lambda^5 + \lambda^3 \; (3/16)\lambda \text{ ,} \\ \det(\mathbf{B}\lambda)_{(6)} = \lambda^6 \; (5/4)\lambda^4 + (3/8)\lambda^2 \; (1/64) \text{ ,} \\ \ldots \end{gathered} \qquad (117)$$

which may be solved to give the eigen values

$$\begin{gathered} \lambda_{(1)} = 0 \text{ ,} \\ \lambda^2_{(2)} = 1/4 \text{ ,} \\ \lambda^2_{(3)} = 0,\ 1/2 \text{ ,} \\ \lambda^2_{(4)} = (3 \; 5)/8 \text{ ,} \\ \lambda^2_{(5)} = 0,\ 1/4,\ 3/4 \text{ ,} \\ \ldots \end{gathered} \qquad (118)$$

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 $\lambda 1$, 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

$$\mathbf{B} = \begin{bmatrix} 1-\sigma & \sigma/2 & & & & & \\ \sigma/2 & 1-\sigma & \sigma/2 & & & & \\ & \sigma/2 & 1-\sigma & \sigma/2 & & & \\ & & \sigma/2 & 1-\sigma & \sigma/2 & & \\ & & & \sigma/2 & 1-\sigma & \sigma/2 & \\ & & & & \ddots & \ddots & \ddots \\ & & & & & \sigma/2 & 1-\sigma \end{bmatrix} \tag{119}$$

and the corresponding eigen values are related by $(1\sigma\lambda)^2$ equal to the values tabulated above. Thus if σ is chosen inappropriately, the eigen values of **B** will exceed unity and the relaxation method will diverge. On the otherhand, careful choise 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 $\sigma >= 1$ produces a better rate of convergence.

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 $^2\varphi = f(\mathbf{x})$ 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 **A**.

8.3 Diffusion equation

Consider the two-dimensional diffusion equation,

$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \tag{120}$$

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 (103)) to obtain

$$\frac{\partial u_{i,j}}{\partial t} \approx \left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}\right) / \Delta x^2 \tag{121}$$

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

$$U'_{i,j}(t) = \left(U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} - 4U_{i,j}\right) / \Delta x^2 . \tag{122}$$

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

$$U_{i,j}^{(n+1)} = U_{i,j}^{(n)} + \mu\left(U_{i+1,j}^{(n)} + U_{i-1,j}^{(n)} + U_{i,j+1}^{(n)} + U_{i,j-1}^{(n)} + -4U_{i,j}^{(n)}\right), \tag{123}$$

where the Courant number

$$\mu = \Delta t/\Delta x^2, \tag{124}$$

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

$$U^{(0)}_{i,j}=\sin(\alpha i)\,\sin(\beta j), \tag{125}$$

for some α and β chosen as multiples of π/m to satisfy $u = 0$ on the boundaries. Substituting this into (123) gives

$$\begin{aligned}U^{(1)}_{i,\mathrm{i}} &= \sin(\alpha i)\sin(\beta j) + \mu\{\sin[\alpha(i+1)]\sin(\beta j) + \sin[\alpha(i1)]\sin(\beta j)\\ &\quad + \sin(\alpha i)\sin[\beta(j+1)] + \sin(\alpha i)\sin[\beta(j1)]\ 4\sin(\alpha i)\sin(\beta j)\}\\ &= \sin(\alpha i)\sin(\beta j) + \mu\{[\sin(\alpha i)\cos(\alpha) + \cos(\alpha i)\sin(\alpha)]\sin(\beta j) + [\sin(\alpha i)\cos(\alpha)\\ &\quad \cos(\alpha i)\sin(\alpha)]\sin(\beta j)\\ &\quad + \sin(\alpha i)[\sin(\beta j)\cos(\beta) + \cos(\beta j)\sin(\beta)] + \sin(\alpha i)[\sin(\beta j)\cos(\beta)\\ &\quad \cos(\beta j)\sin(\beta)]\ 4\sin(\alpha i)\sin(\beta j)\}\\ &= \sin(\alpha i)\sin(\beta j) + 2\mu\{\sin(\alpha i)\cos(\alpha)\ \sin(\beta j) + \sin(\alpha i)\sin(\beta j)\cos(\beta)\ 2\\ &\quad \sin(\alpha i)\sin(\beta j)\}\\ &= \sin(\alpha i)\sin(\beta j)\{1 + 2\mu[\cos(\alpha) + \cos(\beta)\ 2]\}\\ &= \sin(\alpha i)\sin(\beta j)\{1\ 4\mu[\sin^2(\alpha/2) + \sin^2(\beta/2)]\}.\end{aligned} \tag{126}$$

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

$$U^{(\mathrm{n})}_{i,j} = \sin(\alpha i)\sin(\beta j)\ \{1\ 4\mu[\sin^2(\alpha/2) + \sin^2(\beta/2)]\}^n, \tag{127}$$

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

$$\Delta t < \Delta x^2/4. \tag{128}$$

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