Computational Physics
Physics Year 3

Robert Kingham

SECTION 1
Course Description

Welcome to Computational Physics! This course is about the application of computational methods to solve mathematical problems in physics. In your core courses you have seen how physical systems can be described mathematically, often using differential equations, or statistically. Many of the examples you have encountered so far, in mechanics, electromagnetism and quantum physics, have simple analytic solutions. In real life the number of such problems is limited and **numerical methods** are used to solve most problems in mathematical physics, even for apparently simple systems.

In this course you will learn how to select and apply various techniques to solve mathematical physics problems, as well as how to test the suitability of the chosen numerical methods.

The skills acquired will be relevant for future work in both theoretical and experimental physics as well as mathematical modelling.

1.1. Overview

The course will cover the following topics.

- Solution of initial-value ordinary differential equations using finite difference methods.
- Analysis of the accuracy and stability of numerical methods for solving differential equations.
- Solution of initial-value parabolic and hyperbolic partial differential equations using finite difference methods.
- Linear matrix algebra. Iterative methods for solution of linear equations and eigenvalue problems.
- Use of matrix methods to solve elliptic (boundary-value) differential equations.
- Optimisation problems. Methods for finding the minimum of general multi-dimensional functions.
- How random number generators work and how to generate non-uniform random distributions.
- Monte Carlo methods for integration, minimisation and simulating equilibria of assemblies of particles.
- Fourier transform methods and their use in solving differential equations.
- An introduction to some of the library routines available to solve numerical problems.

By the end of the course you should be able to:

- Select and implement finite difference methods to solve differential equations in physics.
- Evaluate the stability, accuracy and efficiency of a given finite difference method to solve a given differential equation.
- Formulate boundary value and eigenvalue problems as matrix equations.

- Select suitable random number generators and use them for simulations and integration.
- Design and implement Monte Carlo methods to simulate dynamic and statistical physics problems.
- Show how Fourier transform methods can be used for differential equations and data processing.
- Design and write computer programs to solve physics problems using any of the above techniques.
- Use numerical library routines as *glass boxes* rather than *black boxes*.

Example problems will be drawn from all areas of physics like mechanics, fluid dynamics, electrostatics, electromagnetism, quantum physics, statistical physics, solid-state physics and high energy physics.

1.2. Course components

The course comprises the following components:

- A series of $15 + 1$ lectures, covering the theory of numerical methods and their applications in physics.
- Eight weekly practical sessions where you will apply the methods, by undertaking ...
- ... two projects, where you will study two particular topics in depth.

Practical sessions will be held in the UG computing suite (level 3 Blackett) on **Wednesday mornings** from **9–12pm** during weeks 3–11 of term.

The course will be assessed in three units:

- Project **A**, to be submitted by **12 noon** on **Monday, 11th November**. This will be worth 20% of the marks for the course. There will be no choice for this project.
- Project **B**, to be submitted by the first Monday after the last day of term (**Monday, 16th December** by **12 noon**). There will be a choice of four topics for this and the project will be worth 40% of the marks.
- A test, to be held in January 2014, worth 40% of the marks. The preliminary time for this test is **Monday, 13th January 2013, 2–4pm**. Make sure you are back at Imperial for the first day of the 2nd term and **allow for travel disruptions over the Christmas and New Year break as no exceptions will be allowed**.

In addition to the assessed work, I will provide 4 problem sheets, which will not be assessed. They contain some theoretical and some programming problems to accompany and extend the lecture material, and examples of applications to physics. These problems will be relevant to the test and to the projects. Test papers from the last few years are available on the course website on Blackboard Learn.

Table 1. Schedule of lectures (held in LT3).

	Date (in 2013)	Topic
1	Thursday, October 3rd, 9am	Introduction
2	Tuesday, October 8th, 11am	Numerical differentiation
3	Wednesday, October 9th, 9am	Evaluating finite difference methods
4	Friday, October 11th, 3pm	Methods for solving ODEs
5	Tuesday, October 15th, 11am	Linear Matrix algebra
6	Wednesday, October 16th, 9am	Boundary value & Eigenvalue problems
7	Friday, October 18th, 4pm	Optimisation
8	Monday, October 28th, 5pm	Random numbers
9	Tuesday, October 29th, 11am	Monte Carlo simulation methods
10	Monday, November 4th, 5pm	PDE introduction and library routines
11	Tuesday, November 5th, 11am	Solving elliptic PDEs
12	Friday, November 8th, 5pm	Solving parabolic and hyperbolic PDEs
13	Tuesday, November 12th, 11am	Overview of project B
14	Friday, November 15th, 3pm	Fourier transform methods
15	Thursday, November 28th, 10am	Poisson's equation by Fourier methods
16	Thursday, November 12th, 10am	Revision

Table 2. Practical Sessions and Project deadlines.

Lab	Date (in 2013)	Topic
1	Wednesday, October 23rd, 9am–12pm	Project A
2	Wednesday, October 30th, 9am–12pm	Project A
3	Wednesday, November 6th, 9am–12pm	Project A
*	**Monday, November 11th, 12 noon**	**Deadline project A**
4	Wednesday, November 13th, 9am–12pm	External libraries, project selection
5	Wednesday, November 20th, 9am–12pm	Project B
6	Wednesday, November 27th, 9am–12pm	Project B
7	Wednesday, December 4th, 9am–12pm	Project B
8	Wednesday, December 11th, 9am–12pm	Project B
*	**Monday, December 16th, 12 noon**	**Deadline project B**

1.3. Course website

The course website is **Computational Physics (2013-14)** on **Blackboard Learn** https://bb.imperial.ac.uk . Lecture notes, course handouts, problems sheets & solutions and reference material for coding (i.e. example codes, numerical libraries) will be made available there. The projects (consisting of the report and the associated programs) should be submitted electronically though the course website on Blackboard Learn. Coursework will be checked by TurnItIn anti-plagiarism software. More details on coursework submission will be given later on.

1.4. Project B

Four projects will be offered. The options will be reviewed briefly in lecture 12 and relatively detailed scripts will be provided for each. You may also propose your own project provided that it relates to the course material. If you have a clear idea of a project to study please discuss it with me. The offered projects are:

1. Optimisation using log likelihood fit to find the lifetime of the D^0 meson.
2. Solution of Laplace's equation to study tolerances in misaligning capacitor plates for the electrostatic field.
3. Solution of the wave equation to study the dynamics of solitons.
4. Metropolis Monte Carlo simulation of a statistical physics problem.

Further details on each project will be available in November.

1.5. Practical sessions

Unlike first and second year computing, you will be working much more on your own and in a less structured way. The practical sessions on Wednesday mornings are organised to give you access to help with programming problems and other questions. However, in practice you will need to be prepared to sort out most compilation errors on your own. You are not required to attend the practical sessions; we are there to help when you have problems, not to check your attendance. Remember that you will get most out of the help available at the practical sessions if you prepare your questions in advance. You are encouraged to help each other with practical aspects of programming, such as debugging code and so on, but the work you hand in for assessment must be your own. Any help from your colleagues or others must be clearly acknowledged in your submitted work. Issues of plagiarism naturally apply to both the written reports and the programs.

The standard platform for practical work is Microsoft Visual Studio 2012 on the PCs in the computing suite. Use of your own laptop and programming environment is allowed. However you do so at your own risk; there is no guarantee that any demonstrator will be familiar enough with your set up to help you out of technical problems. It is important that you (re)acquaint yourselves with Visual Studio (or the development-environment and compiler on you laptop) before the first session.

1.6. Programming language

I imagine that most of you will prefer to use C++ as the programming language. If you wish to work in another language, such as FORTRAN, Python or Matlab, this will normally be acceptable but you must check yourself that the software is available. Please contact me for approval if you plan to use a language other than plain C or C++. The course is not a programming course as such but I will, in different places, provide small bits of advice to improve overall programming style. The quality of programming will influence the assessment of the projects at a minor level.

For graphs and tables you can use whatever tool you are familiar with, such as Excel, Matlab or GnuPlot, to make effective plots for your report. (Of course, make sure axes are labelled and curves are distinguishable and labelled too.)

1.7. Contact

I will be available for discussions and questions during the lab hours in the Computing Suite. During term 1, I will also run **office hours** at the following times; **Mondays 12–1pm** and **Wednesdays 1–2pm**. I can be found in Room 724 Blackett, (ext. 47637). If you cannot make the regular office hour please contact me (rj.kingham@imperial.ac.uk)to arrange an alternative.

For general questions please use the discussion group on area for this course on Blackboard Learn. That way everybody will benefit from the answer. Feel free to take part in discussions there as well.

You can also contact me by email on rj.contaldi@imperial.ac.uk. If you identify errors or other problems with the course material, or have general comments or suggestions, please let me know by email. The lab demonstrators are listed in table 3, please make use of them. If you need help outside the lab hours you can contact them as well but be prepared that you might be asked to come back at another specified time.

Table 3. Computational Physics Demonstrators and Project Assessors.

Name	Position	Group	Room	email
Kingham, Robert	Staff	PLAS	Blackett 724	rj.kingham@imperial.ac.uk
Heavens, Alan	Staff	ASTRO	Blackett 1018E	a.heavens@imperial.ac.uk
Guhl, Hannes	RA	CMTH	Bessemer B321	h.guhl@imperial.ac.uk
Hamm, Joachim	RA	CMTH	Blackett 901A	j.hamm@imperial.ac.uk
Kinsler, Paul	RA	PHOT	Blackett 635	p.kinsler@imperial.ac.uk
Luo, Yu	RA	CMTH	Blackett 816	y.luo09@imperial.ac.uk
Pusch, Andreas	RA	CMTH	Blackett 901	a.pusch11@imperial.ac.uk
Davies, Peter	PG	CMTH	Blackett 817	p.davies10@imperial.ac.uk
Gimeno-Segovia, Mercedes	PG	QUOLS	Elec. Eng.	m.gimeno-segovia@imperial.ac.uk
Kaube, Benjamin	PG	CMTH	Bessemer B321	benjamin.kaube08@imperial.ac.uk
Pecover, James	PG	PLAS	Blackett 738	j.pecover11@imperial.ac.uk
Santos, Edward	PG	HEP	Blackett 514	e.santos10@imperial.ac.uk

1.8. Literature

The following textbook is strongly recommended:

- C. Gerald and P. Wheatley, *Applied Numerical Analysis*, International Edition, 7th edition, (Pearson, 2004), ISBN 0-321-19019-X.
- W. H. Press, B. P. Flannery, S. A. Teukolsky, and W.T Vetterling *Numerical Recipes in C++: The art of scientific computing* (Cambridge: CUP 2007, 3rd ed.). The full text of the second edition is available on the internet at http://www.nr.com/ but the access is rather convoluted.
- J. D. Hoffman, *Numerical Methods for Engineers and Scientists*, 2nd ed., (Marcel Dekker, Inc., 2001), ISBN 0-8247-0443-6.

The following resources are also useful:

- M. Galassi et. al., *GSL - GNU Scientific Library*. This is the numerical library we will use in the course. Source code and full documentation available at http://www.gnu.org/software/gsl/.
- N. J. Giordano and H. Nakanishi. *Computational Physics*, Second Edition, (Pearson 2006), ISBN 0-13-146990-8. Good as background for some of the projects.
- E. Süli and D. Mayers. *An introduction to Numerical Analysis*, Cambridge University Press, ISBN-13 978-0-521-00794-8. A very formal book on Numerical Analysis.
- J. R. Hubbard, *Programming with C++, Schaum's outlines*, Second edition, (McGraw-Hill 2000), ISBN 0-07-135346-1. This is a good (and very cheap) manual to C++ programming.
- E. W. Weisstein, *MathWorld–A Wolfram Web Resource*. http://mathworld.wolfram.com. An authoritative site on large areas of mathematics. Also available as a book.
- G. B. Arfken and H-J. Weber, *Mathematical Methods for Physicists*, (Academic Press Inc 20051992), ISBN 0-12-088584-0. A reference book for most mathematics a physicists will ever need.

You will find many internet web pages related to computational methods and their applications to physics. These can be helpful for understanding, but please remember that web material is in general not completely reliable. Also remember that you must reference any sources (books, reports, websites, etc.) which you use in writing your project reports.

Acknowledgments

These lecture notes are adapted from previous course notes developed by C. Contaldi and U. Egede.

SECTION 2
Introduction to Numerical Calculations

Outline of Section

- Nature of errors
- Numerical differentiation
- Ordinary Differential Equations (ODEs)
- Natural units
- Solving non-linear algebraic equations
- Basic interpolation

2.1. Numerical Accuracy and Errors

Review of Taylor series

A computer approximates continuous functions as a truncated series expansion. This is because a computer can only add, subtract and multiply numbers. We will also be approximating derivatives as truncated series. To do this we review some basic concepts of Taylor series.

The function $f(x)$ can be expanded around the point a to n^{th} order as

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + ... + \frac{1}{n!}f^n(a)(x-a)^n\,,$$

where $f^n(a)$ is the n^{th} derivative of the function evaluated at the point $x = a$. In fact the function can be written as an n^{th} order expansion plus a remainder term which sums up all the remaining terms to infinite order

$$\boxed{f(x) \equiv \sum_{i=0}^{i=n} \frac{1}{i!} f^i(a)(x-a)^i + R_n(x)} \quad \text{where} \quad R_n(x) \equiv \sum_{i=n+1}^{i=\infty} \frac{1}{i!} f^i(a)(x-a)^i\,. \tag{2.1}$$

Using the **Mean Value Theorem** it can be shown that there is a value ξ which lies somewhere in the interval between x and a for which

$$\boxed{R_n(x) = \frac{1}{(n+1)!} f^{n+1}(\xi)(x-a)^{n+1} \qquad \text{where} \qquad (a \le \xi \le x)} \tag{2.2}$$

that is that the remainder can be written as the $(n+1)^{\text{th}}$ term of the expansion evaluated at th point ξ. This is a useful way of expressing the error in the truncated series expansion for what follows. Remember that in order to possess an n^{th} order Taylor expansion, the function has to be n times differentiable (and $n+1$ times, if we want the remainder term).

For us it will be more useful to expand functions as $f(x+h)$ around the point x. This is because we will be interested in the value of the function at the point $x+h$ where h is a *small* step away from the point x where the value of the function $f(x)$ is already known. In this case, substituting in $x = a + h$, the Taylor series can be written as

$$f(x+h) \approx f(x) + f'(x)\,h + \frac{1}{2}f''(x)\,h^2 + ... + \frac{1}{n!}f^n(x)\,h^n\,. \tag{2.3}$$

For functions of two independent variables, say x and y, the Taylor series is

$$f(x+h, y+k) \approx \sum_{i=0}^{i=n} \frac{1}{i!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^i f(x,y) \tag{2.4}$$

where term $(\ldots)^i$ is expanded by the binomial expansion, e.g. $\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^2 = h^2\frac{\partial^2}{\partial x^2} + 2hk\frac{\partial^2}{\partial x \partial y} + k^2\frac{\partial^2}{\partial y^2}$, and operates on the function evaluated at (x, y). The remainder term is similar to (2.2), in that it involves $(n+1)^{\text{th}}$ order partial derivatives of f evaluated at the point (ξ, η), where $x \leq \xi \leq x+h$ and $y \leq \eta \leq y+k$. Equation (2.4) can be generalised to more independent variables by adding the relevant partial derivatives (e.g. $\partial/\partial x$) into the $(\ldots)^i$ term and using a multinomial expansion.

There are a number of errors that can affect the accuracy and stability of a numerical code.

Truncation errors

Even the most precise computer evaluates functions approximately. This approximation can be compared to the truncation of a Taylor series (most programming languages use some form of power expansion to approximate standard mathematical functions). For example the Taylor expansion of $\sin(x)$ is

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ ... \ + (-1)^n\frac{x^{2n+1}}{(2n+1)!} + \ ... \,.$$

If we truncate the series at 3rd order then the truncation error will be of 5th order

$$\sin(x) \approx p_3(x) \equiv x - \frac{x^3}{3!}\,,$$

then

$$\epsilon(x) \equiv \sin(x) - p_3(x) \sim \mathcal{O}(x^5)\,. \tag{2.5}$$

In general, if a Taylor series for a function $f(x)$ around $x = a$ is truncated at n^{th} order the error is

$$\epsilon = \frac{f^{n+1}(\xi)}{(n+1)!}(x-a)^{n+1}\,, \tag{2.6}$$

where ξ lies somewhere between x and a.

Round-off errors

Finally even in the absence of any truncation error or sensitivity to initial conditions a round-off error is inherent to all digital computers. This is due to the fact that a computer stores any number with a finite precision.

The numbers are rounded off to the closest value at the computer's precision and in some cases this error can be propagated and lead to instabilities.

Computers commonly have two levels of precision, single and double. **With `C++` it is always advisable to use double precision.** In `FORTRAN` it is possible to pad any numerical number with trailing zeroes but in `C++` this is not possible and the round-off error means that the value could be very slightly larger or smaller than intended. **For this reason never use single or double precision variables as loop counters in `C++`. Always use integer variables.**

A computer stores numbers in binary representation

binary representation		decimal representation
0	=	0
1	=	1
10	=	2
11	=	3
100	=	4
101	=	5
110	=	6
111	=	7
1000	=	8
etc...		

(2.7)

32 and 64 bits are used to store number in single and double precision respectively with the following scheme e.g. for the number 1.2345×10^{10}

	sign	mantissa	exponent
	$\pm$	.12345	offset + 10
single	1-bit	23-bits	8-bits
double	1-bit	52-bits	11-bits

(2.8)

The exponent is stored as a binary value between 0 and 255 (8-bit) for single precision and 0 and 2047 (11-bits) for double precision. The range includes both negative and positive values, hence the offset to make them all positive. Thus the range of numbers

allowed are roughly $10^{\pm 38}$ and $10^{\pm 308}$ for single and double precision respectively as numbers are stored in base 2 (i.e. $2^{256/2} \sim 10^{38}$).

Numbers smaller or greater than these will cause an **underflow** or **overflow** respectively. Overflows are replaced by the symbol `Inf` in some languages. The IEEE standard for handling e.g. $0/0$, $\sqrt{-1}$, etc. is to replace with `NaN` i.e. Not a Number.

Initial condition errors

Even in the absence of any truncation error or round-off error, an error in defining the starting point of the calculation can put the calculation onto a different solution of the equation being solved. This new solution (e.g. $x(t)$ curve) may diverge from the intended solution, perhaps more & more quickly with time (or whatever the independent variable is).

Some systems can be very sensitive with respect to initial conditions, e.g., uncertainty in modelling of meteorological model dominated by initial uncertainty in atmospheric data, and other chaotic systems such as fractals.

Propagating errors

A propagation error is akin to an initial condition error. It is the error that would be seen in successive steps of a calculation given an error present at the current step, *if the rest of the calculation were to be done exactly (i.e. without truncation or round-off errors).* The **inherited error** at the current step is the accumulation of errors from all previous steps. Figure 2.1 illustrates propagating errors and truncation errors during each step of the numerical solution of an ODE $dy/dx = f(y, x)$. (Concrete examples of numerical schemes & ODEs will be given later.) The numerical scheme attempts to solve the ODE at discrete values of x; x_1, x_2, etc. You an see that the numerical solution (open circles) moves off the exact solution (solid line).

If the propagated error is increasing with each step then the calculation is **unstable**. If it remains constant or decreases then the calculation is **stable**.

The stability of a calculation can depend both on the type of equations involved and the method used to approximate the system numerically.

2.2. Numerical Differentiation

We are used to defining a derivative as

$$\frac{dy(x)}{dx} \equiv y'(x) = \lim_{h\to 0} \frac{y(x+h) - y(x)}{h} \equiv \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}\,. \tag{2.9}$$

We approach this limit on the computer by defining a **forward difference scheme (FDS)**

$$\boxed{\tilde{y}'_f(x) \equiv \frac{y(x+h) - y(x)}{h}\,.} \tag{2.10}$$

The FDS is an example of a **finite difference approximation**. Notation: in this course a tilde ($\sim$) over a quantity signifies that it is an approximated version. By considering

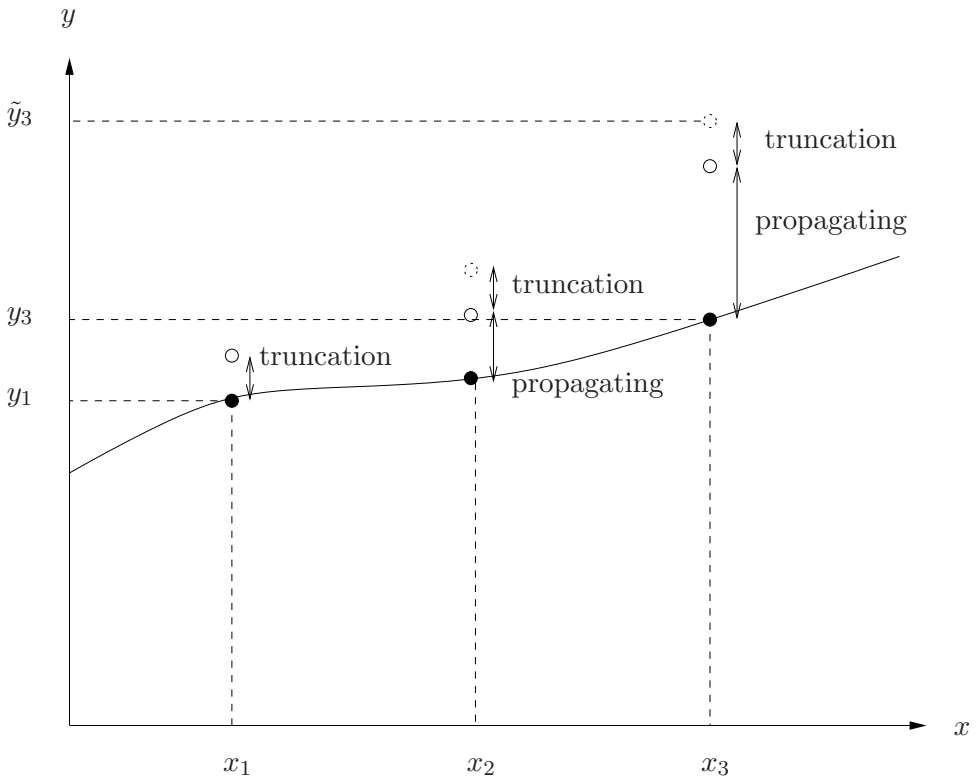

Figure 2.1. Illustration of truncation and propagating errors incurred in solving an ODE $dy/dx = f(y, x)$. The solid line depicts the exact solution. Dashed open circles depict the numerical solution. The calculation starts at (x_0, y_0) (off the plot) where there is no error. $\tilde{y}_3$ is the numerical result after 3 steps.

the Taylor expansion of $y(x + h)$ around the point x we can understand how the error in the above scheme is second order

$$y(x + h) = y(x) + y'(x)\, h + \frac{y''(\xi)}{2}\, h^2\,,$$

where we have defined the truncation error as

$$\epsilon = \frac{y''(\xi)}{2}\, h^2,$$

where ξ is an unknown value which lies in the interval $x < \xi < x + h$. Thus we can write the first derivative as

$$y'(x) = \frac{y(x + h) - y(x)}{h} - \frac{y''(\xi)}{2}\, h = \tilde{y}'_f(x) - \mathcal{O}(h)\,. \tag{2.11}$$

So the simple forward difference scheme has error $\mathcal{O}(h)$. Notice that even if we reduce this truncation error by making h really small, in practice, the accuracy of the derivative will have a limit imposed by the round-off error of the computer.

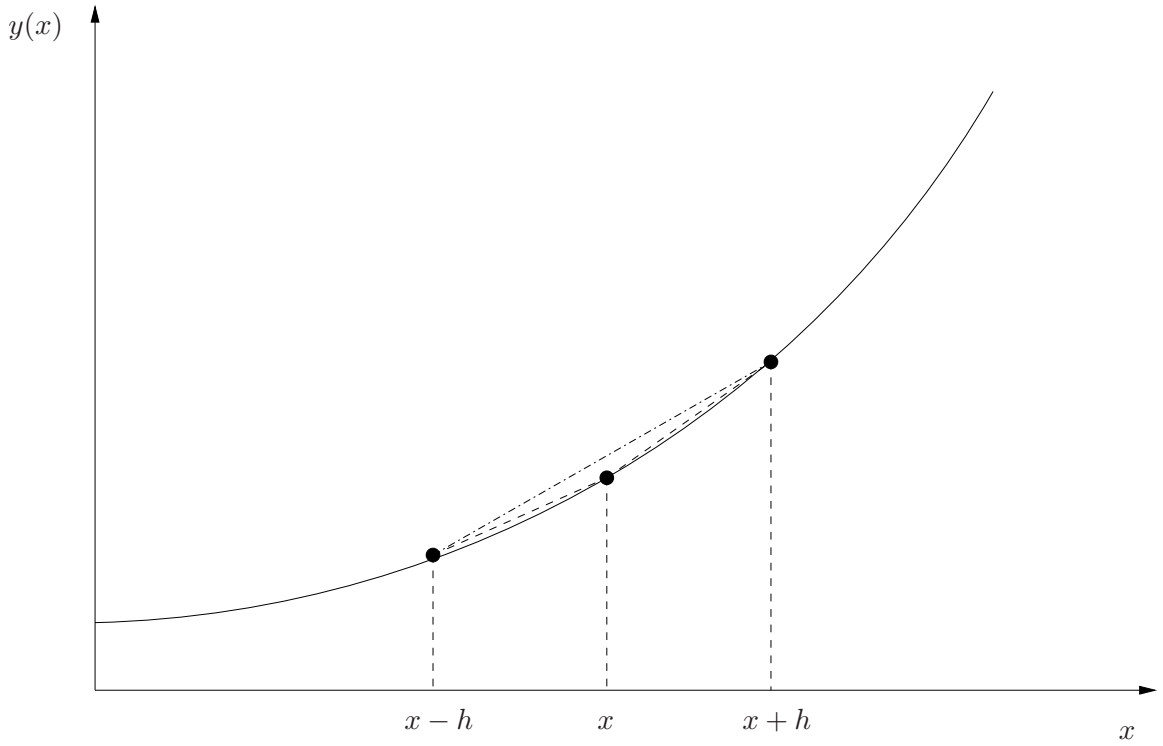

Figure 2.2. Forward, backward, and central difference schemes for approximating the derivative of the function $y(x)$ at the point x

We can also use a **backward difference scheme (BDS)**

$$\boxed{\tilde{y}'_b(x) \equiv \frac{y(x) - y(x-h)}{h}\,,} \tag{2.12}$$

but as you can easily show (expand $y(x-h)$ around x) this will have a similar error to the forward difference estimate. However since they are estimates of the same quantity there must be a way to combine the two to get a more accurate estimate. This is achieved by the **central difference scheme (CDS)** which is the average of the two estimates

$$\boxed{\tilde{y}'_c(x) \equiv \frac{\tilde{y}'_f(x) + \tilde{y}'_b(x)}{2} = \frac{y(x+h) - y(x-h)}{2h}\,.} \tag{2.13}$$

Consider the Taylor series (to 3rd order) for $y(x+h)$ and $y(x-h)$ giving

$$y(x+h) = y(x) + y'(x)\,h + \frac{1}{2}y''(x)\,h^2 + \frac{1}{3!}y'''(\xi)\,h^3 \tag{2.14}$$

and

$$y(x-h) = y(x) - y'(x)\,h + \frac{1}{2}y''(x)\,h^2 - \frac{1}{3!}y'''(\zeta)\,h^3 \tag{2.15}$$

then we can use this to define

$$y(x+h) - y(x-h) = 2\,y'(x)\,h + \mathcal{O}(h^3)\,,$$

which gives

$$y'(x) = \frac{y(x+h) - y(x-h)}{2h} - \mathcal{O}(h^2) \equiv \tilde{y}'_c(x) - \mathcal{O}(h^2)\,, \tag{2.16}$$

so the central difference scheme gives the derivative up to an $\mathcal{O}(h^2)$ error. Remember that h is a small step i.e. in suitable units (see later) it will satisfy the condition $h \ll 1$ so the error *decreases* with *increasing* order of magnitude in h.

You can understand why the CDS is more accurate than either FDS or BDS by looking at figure 2.2; the central difference gives a better estimate of the gradient (tangent line) at the point x than the forward or backward difference.

Higher order derivatives

It is possible to find numerical approximations to 2nd and higher order derivatives too. A 2nd order accurate version of d^2y/dx^2 is

$$\boxed{\tilde{y}''(x) \equiv \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} = y''(x) + \mathcal{O}(h^2)\ .} \tag{2.17}$$

This can be obtained by adding together the 4th order Taylor expansions for $y(x+h)$ and $y(x-h)$, i.e., equations (2.14) and (2.15) with one more term. Another way to get (2.17) is to take the central difference versions of 1st order derivative at $x + h/2$ and $x - h/2$, and then find the central difference of these;

$$\tilde{y}''(x) = \frac{\tilde{y}'_c(x+h/2) - \tilde{y}'_c(x-h/2)}{h}\ .$$

Three points, $y(x-h)$, $y(x)$ and $y(x+h)$, are needed to get $\tilde{y}''$, the finite difference approximation of y''. In general, to get the finite difference approximation $\tilde{y}^n$ requires $n+1$ points and going further away from x, e.g., $y(x+2h)$, $y(x-2h)$, $y(x+3h)$, $y(x-3h)$, etc.

2.3. Ordinary Differential Equations

In countless problems in physics we encounter the description of a physical system through a set of differential equations. This is because it is simpler to model the behaviour of a system (e.g. variables $\vec{y} \equiv \{u, v, w, \text{etc.}\}$) in terms of infinitesimal responses to infinitesimal changes of independent variable(s) (e.g. η). The system will be of the

form

$$\begin{aligned} \frac{du}{d\eta} &= f_u(\eta, \vec{y}, ...) \\ \frac{dv}{d\eta} &= f_v(\eta, \vec{y}, ...) \\ \frac{dw}{d\eta} &= ... \\ &\vdots \end{aligned} \tag{2.18}$$

Often the independent variable η is time t. Note that the functions defining the derivatives of the system variables are in general a function of the independent variable *and* all system variables, i.e., the system can be **coupled**. Only if the derivatives of each system variable are solely a function of that system variable is the system **uncoupled**.

The aim is to find a solution for the system variables at any value of the independent variable given a known initial condition for the system, e.g., solve for $u(\eta)$ given an initial condition $u(\eta_0)$ for all $\eta > \eta_0$.

To do this we need to integrate the differential equations. This can be done analytically for the simplest cases, e.g.,

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

then

$$\int_{y(0)}^{y(t_f)} y\,dy = \int_0^{t_f} t\,dt\,,$$

giving

$$y(t_f) = \sqrt{t_f^2 + 1}\,.$$

Most often however, if the function describing the derivative is not separable or if the system of many variables is coupled, the system cannot be integrated analytically and we have to take a numerical approach.

We can solve systems numerically if the equations have a unique, continuous solution and we know the initial (or boundary value) conditions for the system variables.

Euler method

The simplest approach to numerical integration of an ODE is obtained by looking at the Taylor expansion for a forward difference, again using time t as the independent variable,

$$\begin{aligned} y(t+h) &= y(t) + y'(t,y)\,h + \mathcal{O}(h^2) \\ &\approx y(t) + f(t,y)\,h\,. \end{aligned} \tag{2.19}$$

So if we know the value of the variable y at t we can use a finite step h to calculate the next value of y at $t+h$ up to $\mathcal{O}(h^2)$ accuracy. In terms of the discretised limit of the function we can write this as

$$\boxed{y_{n+1} = y_n + f(t_n, y_n)\,h \quad ,} \tag{2.20}$$

where we use the subscript n to denote values of the function y at **time step**

$$t_n = t_0 + (n-1)\,h \quad . \tag{2.21}$$

Higher order systems

A system described by an m^{th}-order set of ordinary differential equations can always be reduced to a set of m 1st-order ODEs. For example, consider the 3rd-order system

$$\frac{d^3y}{dt^3} + \alpha\frac{d^2y}{dt^2} + \beta\frac{dy}{dt} + \gamma\, y = 0 \quad .$$

We can introduce three new variables

$$u \equiv y, \quad v \equiv \frac{dy}{dt}, \quad \text{and} \quad w \equiv \frac{d^2y}{dt^2}\,,$$

such that the system can be written as three 1st-order equations

$$\begin{aligned} \frac{du}{dt} &= v \\ \frac{dv}{dt} &= w \\ \frac{dw}{dt} &= -\left(\gamma\, u + \beta\, v + \alpha\, w\right)\,, \end{aligned}$$

or in matrix notation

$$\frac{d}{dt}\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -\gamma & -\beta & -\alpha \end{pmatrix}\begin{pmatrix} u \\ v \\ w \end{pmatrix} \quad .$$

In general a **linear** ODE system can be written in terms of a matrix operator $\mathbf{L}$ as

$$\frac{d\vec{y}}{d\eta} = \mathbf{L}\,\vec{y} \quad , \tag{2.22}$$

with initial conditions $\vec{y}(\eta_0) = \vec{y}_0$. The Euler method is then $\vec{y}_{n+1} = \vec{y}_n + h\,\mathbf{L}\,\vec{y}_n$ so that

$$\boxed{\vec{y}_{n+1} = (\mathbf{I} + h\mathbf{L})\,\vec{y}_n = \mathbf{T}\,\vec{y}_n\,,} \tag{2.23}$$

where $\mathbf{T}$ is the update matrix.

A general, non-linear system is written as

$$\frac{d\vec{y}}{d\eta} = \mathcal{L}\,\vec{y}, \tag{2.24}$$

where $\mathcal{L}$ is a **non-linear operator**.

2.4. Natural units

It is very important to use the correct units when integrating systems numerically due to the limited range of numbers a computer can deal with. It also helps to understand the dynamics (and debugging) if we can identify dimensionless units for the system. For example, consider the dimensionfull system

$$\frac{dy}{dt} = \alpha\, y\,, \tag{2.25}$$

where α describes a rate of change and has dimensions $[\alpha] = s^{-1}$.

We can then define a new time variable $\tilde{t} = \alpha\, t$ such that

$$\frac{dy}{d\tilde{t}} \equiv \frac{dy}{dt}\frac{dt}{d\tilde{t}} = \alpha\, y\, \frac{1}{\alpha} = y\,. \tag{2.26}$$

This is the system in dimensionless units.

Changing to units where the independent variable is dimensionless means we don't have to worry about units in our integration;

- Step-size h; in these dimensions a small step is anything with $h < 1$. If we still had dimensions in the problem this would be a lot harder to define.
- Range of integration; the characteristic timescale in the dimensionless units is $\tilde{t} \sim 1$ so we know that integrating from $\tilde{t} = 0$ to $\tilde{\sim}$ a few will cover the entire dynamic range of interest.

One thing we *must* remember is to put the units back in when we present our results e.g. in plots of the integrated solution (or explicitly state what units the axis is in e.g. units of $[1/\alpha]$ in example above).

2.5. Solving Non-Linear Algebraic Equations

An important part of the computational physics 'toolkit' is to find roots of non-linear algebraic equations, i.e., the solution of

$$f(x) = 0 \ \ ,$$

where f involves transcendental functions or is even simply a polynomial of high order (i.e. $n > 4$), and a solution in closed form is not possible. An example of a non-linear equation is $\sin(x^2) - 1/(x + a) = 0$. This is just the sort of situation where numerical methods are essential. Non-linear root finders are typically **iterative methods**, where an initial guess is refined iteratively. Functions with discontinuities and/or infinities pose problems unless the initial guess is carefully made. Some key methods are given below.

Treatment of many variables $f(x, y, z) = 0$ will be covered in Section 7.

Bisection method

This is the simplest method and very robust, but is slow. One starts with two points x_l (the left point) and x_r (right point) which bracket the root. To bracket the root, $f(x_l)$ and $f(x_r)$ must have opposite sign. Then $f(x)$ must pass through zero (perhaps several times) between these points. Now get the mid point

$$x_m = \frac{x_l + x_r}{2}$$

and determine whether the pair x_l and x_m or x_m and x_r bracket the root. Update x_l or x_r to x_m accordingly and repeat until $\epsilon_i = x_r - x_l$ (where subscript i denotes the iteration number) is sufficiently small or $\max[f(x_l), f(x_r)]$ is sufficiently close to zero. (These are convergence criteria.) This method converges linearly, with the error ϵ_i (the uncertainty in where the root is) halving with each iteration;

$$\epsilon_{i+1} = \epsilon_i/2 \ .$$

Bisection will also find where discontinuous functions jump through zero.

Newton's method

Also known as the Newton-Raphson method. This iterative scheme uses the 1st derivative of the function

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \ .} \tag{2.27}$$

The Newton method can easily fail if it gets near maxima or minima, in which case x_{i+1} can shoot off 'to infinity'. It is also possible to be locked in a cycle where the method ping-pongs between the same two x points. The advantage of Newton's method is its speed of convergence; it converges quadratically,

$$\epsilon_{i+1} = \text{const} \times (\epsilon_i)^2 \ . \tag{2.28}$$

Secant method

This iterative methods is related to Newton's method, but works when an analytical expression for the derivative function $f'(x)$ is unknown. The tangent to the function $f'(x_i)$ is approximated using two points on the curve (which define the secant line)

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \ . \tag{2.29}$$

Inserting into Newton's method gives

$$\boxed{x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} \ .} \tag{2.30}$$

The speed of convergence of the secant method is somewhere between linear and quadratic.

2.6. Basic interpolation

Finding a value for a function between points where it is known, is another essential piece of a computational physicist's toolkit. One approach is to fit a function in the least-squares sense to the data points, as typically done with experimental data. This approximate fit will probably not go through any of the data points. Another approach, which is considered here, is to fit a polynomial that goes exactly through the points. This is interpolation.

Interpolation is an extensive subject. Here we briefly look at some very basic methods with which you can make do. (More efficient methods exist; see any text book!). Here is an example of when interpolation might be needed. In solving ODEs numerically, time is typically discretised (t_n). An idea of the solution at a time between these t_n might be required. As we will see later, in solving PDEs, space is discretised into a grid and the numerical method provides the (approximate) solution at these points. Again, the solution at other points in space is often needed.

Linear interpolation

This is simply fitting a straight line through adjacent points (x_i, f_i) and (x_{i+1}, f_{i+1}) to find f at a point x in between;

$$f(x) = \frac{(x_{i+1} - x)f_i + (x - x_i)f_{i+1}}{x_{i+1} - x_i} \quad . \tag{2.31}$$

Lagrange polynomials

Given $n + 1$ points (x_i, f_i) where $0 \leq i \leq n$, the Lagange polynomial is an n^{th} degree polynomial that goes exactly through these points;

$$P_n(x) = \sum_{i=0}^{n} \left(\prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \right) f_i \quad , \tag{2.32}$$

where in the product, j run over integers from 0 to n but misses out the value i. For example: for the $n = 2$ case

$$P_2(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} f_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} f_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} f_2 \quad . \tag{2.33}$$

Note that x_i do not need to be equally spaced. Polynomials fit this way can be very wavy.

Bi-linear

This works for a function of two variables $f(x, y)$. Imagine f is known on four points which are the corners of a rectangle in the x–y coordinate system; (x_i, y_k), (x_{i+1}, y_k), (x_i, y_{k+1}) and (x_{i+1}, y_{k+1}). We want f at an arbitrary point (x, y) within this square. Linear interpolation is first applied to f in the x direction, along the bottom ($y = y_k$) and top ($y = y_{k+1}$) of the square. For instance, along the bottom, equation (2.31) is used with $(x_i, f(x_i, y_k))$ and $(x_{i+1}, f(x_{i+1}, y_k))$ to obtain the intermediate value $f(x, y_k)$. Similarly at the top linear interpolation yields $f(x, y_{k+1})$. Now these two intermediate values are linearly interpolated in the y-direction, using an equation analogous to (2.31). Doing interpolation in y to get the intermediate values and then interpolation in x yields exactly the same value.

Multivariate interpolation is the generalisation to functions with more than one variable; $f(x, y, z, \ldots)$.

SECTION 3
Evaluating Finite Difference Methods

Outline of Section

- Consistency
- Accuracy
- Stability
- Convergence
- Efficiency

The Euler method, seen in section 2.3, is an example of a **finite difference method**. We will cover a number of more advanced methods for integrating ODEs numerically but before we do that we will look at how to evaluate the properties of any particular method.

3.1. Consistency

This is a check that the finite difference method (e.g. Euler method) reduces to the correct differential equation (e.g. $dy/dt = f(t, y)$) in the limit of step size $h \to 0$. Moreover, a consistency analysis of a finite difference equation reveals the actual differential equation that the finite difference method is effectively solving; the '*modified differential equation*'. The finite difference method samples the solution of the modified differential equation at the discretized time t_n (or the relevant discretized independent variable η_n). The consistency analysis also tells us the order of the method. This is useful if one is given a finite difference equation, but is not told what its order of accuracy is.

A consistency analysis is carried out by taking the equation for a finite difference method and Taylor expanding all terms about the base point (e.g. η_n). (This is essentially the reverse of the procedure used to derive the Euler method in the first place.) As an example, we take the Euler method

$$y_{n+1} = y_n + f_n\, h \quad , \tag{3.1}$$

where $f_n = f(\eta_n, y_n)$. Next Taylor expand y_{n+1} about the base point y_n. This yields

$$y_n + y'_n\, h + \frac{y''_n}{2}\, h^2 + \frac{y'''_n}{3!}\, h^3 + \ldots = y_n + f_n\, h \quad ,$$

(where $y' = dy/d\eta$ etc.) which can be rearranged into

$$y'_n = f_n - \frac{y''_n}{2}\,h - \frac{y'''_n}{3!}h^2 - \dots \quad . \tag{3.2}$$

Remember that the subscript n notation means evaluation of quantities at η_n. Allowing η_n to become continuous, i.e. $\eta_n \to \eta$ reveals that this is a differential equation

$$\boxed{y' = f(\eta, y) - \frac{y''}{2}\,h - \frac{y'''}{3!}h^2 - \dots} \quad . \tag{3.3}$$

This is the **modified differential equation** (MDE). The discrete points solved by the Euler method (η_n, y_n) lie on the continuous curve that solves the MDE (given the same initial condition of course!). Letting $h \to 0$, equation (3.3) becomes

$$y' = f(\eta, y) \quad , \tag{3.4}$$

which is the intended ODE the Euler method set out to approximate. Thus we say that (3.1) is consistent with the ODE $y' = f(\eta, y)$. The lowest order in h of the correction terms in the MDE, e.g. $-y''\,h/2 \sim \mathcal{O}(h)$ in this case, is the order of accuracy of the method. This is the same as the order of the *global error*, discussed in the following section.

Consistency analysis can be carried out with finite difference methods for partial differential equations too (see Section 10).

3.2. Accuracy

To determine the accuracy of a method we want to take the **local error** (truncation) and use it to estimate the global error (the error at the end of the calculation). Using the Euler method as an example we know that the local truncation error is $\mathcal{O}(h^2)$. y_{n+1}, the **true** value of the function at η_{n+1}, is related to the numerical approximation of the function $\tilde{y}_{n+1}$ by

$$\tilde{y}_{n+1} = y_n + f_n\,h \equiv y_{n+1} + \mathcal{O}(h^2)\,, \tag{3.5}$$

where we have assumed here that y_n is known exactly. Equation (3.5) shows us the error incurred in performing one step of the Euler method. However to integrate from η_0 to η_{end} would take $\mathcal{O}(1/h)$ steps. If we assume the local error at each of the steps simply adds up, in general, we will have

$$\boxed{\textbf{global error} \approx \text{local error} \times \text{number of steps}\,.} \tag{3.6}$$

In this case

$$\epsilon_{\text{end}} \approx \mathcal{O}(h^2) \times \mathcal{O}(1/h) \sim \mathcal{O}(h)\,. \tag{3.7}$$

In general **a method is called n^{th}-order** if its local error is of $\mathcal{O}(h^{n+1})$, i.e., **its global accuracy is of $\mathcal{O}(h^n)$**.

You might think that all we need to do is reduce the step size h (increase the number of steps) arbitrarily to increase the accuracy arbitrarily. Unfortunately the round–off error places a limit on how accurate any method can be. To see this consider an error

$\mathcal{O}(\mu)$ added to every step in the integration, then the global error after $\mathcal{O}(1/h)$ steps will be

$$\epsilon_{\rm end} \sim \frac{\mu}{h} + h\,. \tag{3.8}$$

Thus there is a minimum error (maximum accuracy) achievable for the Euler method when

$$h \sim \mu^{1/2}\,. \tag{3.9}$$

For double precision on most machines $\mu \sim 10^{-16}$ so the smallest useful step size is $h \sim 10^{-8}$ which will give a global accuracy of $\epsilon_{\rm end} \sim 10^{-8}$.

3.3. Stability

This is a crucial property of a method and describes how an error 'propagates' as the finite difference equation is iterated. If the error increases catastrophically then the method is unstable. If it doesn't grow or decreases, the method is stable. Imagine that the numerical solution at step n has an error ϵ_n

$$\tilde{y}_n = y_n + \epsilon_n \quad , \tag{3.10}$$

where y_n is the true solution of the ODE at $\eta = \eta_n$ and $\tilde{y}_n$ the numerical approximation. In taking one step of a finite difference method the numerical approximate solution and the error change and satisfy

$$\tilde{y}_{n+1} = y_{n+1} + \epsilon_{n+1} \quad . \tag{3.11}$$

For stability of *any method* we require that the **amplification factor**

$$\boxed{g \equiv \left|\frac{\epsilon_{n+1}}{\epsilon_n}\right| \leq 1 \quad ,} \tag{3.12}$$

otherwise the error in the finite difference method exponentially 'blows up'.

We consider the Euler method as an example. We can write it as

$$\tilde{y}_{n+1} = \tilde{y}_n + f(\eta_n, \tilde{y}_n)\, h \quad . \tag{3.13}$$

Substituting in the true values of the function we have

$$y_{n+1} + \epsilon_{n+1} = y_n + \epsilon_n + f(\eta_n, y_n + \epsilon_n)\, h \quad . \tag{3.14}$$

Stability analysis can only be carried out for linear ODEs. Luckily for non-linear ODEs, i.e., when f is a non-linear function of y, we can still do a stability analysis by assuming that the errors are small $|\epsilon_n|$, $|\epsilon_{n+1}| \ll |y_n|$ and **linearizing $\boldsymbol{f(\eta, y)}$**. To linearize, the function $f(\eta_n, y_n + \epsilon_n)$ is expanded around the point (η_n, y_n) in the variable y. To 1st-order in the expansion parameter ϵ_n the equation (3.14) becomes

$$\begin{aligned} y_{n+1} + \epsilon_{n+1} &= y_n + \epsilon_n + \left[f(\eta_n, y_n) + \left.\frac{\partial f}{\partial y}\right|_n \epsilon_n + \mathcal{O}(\epsilon_n^2) \right] h \quad , \\ &= y_n + f(\eta_n, y_n) h + \epsilon_n \left[1 + \left.\frac{\partial f}{\partial y}\right|_n h \right] + \mathcal{O}(\epsilon_n^2)\,. \end{aligned} \tag{3.15}$$

where $\partial f/\partial y|_n$ means $\partial f/\partial y$ evaluated at the base point $t = t_n$, $y = y_n$. Then since

$$y_{n+1} = y_n + f(\eta_n, y_n)h + \mathcal{O}(h^2)\,, \tag{3.16}$$

(i.e. Taylor expansion of the true value) and dropping terms of $\mathcal{O}(h^2)$, $\mathcal{O}(\epsilon_n^2)$ and higher we have

$$\boxed{\epsilon_{n+1} \approx \epsilon_n \left(1 + \left.\frac{\partial f}{\partial y}\right|_n h\right)\,,} \tag{3.17}$$

for the Euler method. Inserting $\epsilon_{n+1}/\epsilon_n$ into equation (3.12), and dropping the $|_n$ notation, we can get a condition on the step size h for stability:

$$g = \left|1 + \frac{\partial f}{\partial y}\,h\right| \leq 1 \quad \rightarrow \quad -2 \leq \frac{\partial f}{\partial y}\,h \leq 0\,,$$

then assuming $h > 0$ the Euler method is only stable if $\partial f/\partial y < 0$ and

$$\boxed{h \leq \frac{2}{|\partial f/\partial y|}\,.} \tag{3.18}$$

We say that the Euler method is **conditionally stable** for $\partial f/\partial y < 0$ and **unconditionally unstable** for $\partial f/\partial y > 0$ (i.e. no step size works).

For a non-linear ODE the amplification factor and step size for stability change with y. For a linear ODE, e.g., $dy/d\eta = f(\eta, y) = p(\eta)y + q(\eta)$, we have $\partial f/\partial y = p(\eta)$ and so g and the limiting step size are controlled by $p(\eta)$. (Note that $\partial f/\partial y$ means keeping η fixed.) For constant coefficients, e.g., the decay problem $y' = -\alpha y$ (with α positive), the stability conditions $g = |1 - \alpha\, h|$ and $h \leq 2/\alpha$ do not change over the calculation.

Note that this analysis does not tell us about the global error when the method is stable. Even when $g < 1$ there is still a global error that accumulates with each step. Global error (for a stable method) comes in through the $\mathcal{O}(h^2)$ term dropped in equation (3.16).

3.4. Stability Analysis of General (coupled) Linear Systems

A more general stability analysis can be carried out for m, coupled, linear 1-st order ODE equations by looking at the general matrix operator form for the method

$$\tilde{\vec{y}}_{n+1} = \mathbf{T}\,\tilde{\vec{y}}_n\,, \tag{3.19}$$

where $\mathbf{T}$ is the update matrix (e.g. $\mathbf{T} = (\mathbf{I} + \mathbf{L}\,h)$ for the Euler method). For a set of inhomogeneous ODEs $\tilde{\vec{y}}_{n+1} = \mathbf{T}\,\tilde{\vec{y}}_n + \vec{q}(\eta_n)$ the term $\vec{q}(\eta_n)$ does not influence numerical stability. Only the homogeneous part of the equation set, i.e., equation (3.19) need be analysed. We can relate the true solution vector $\vec{y}_n = \{y_n^1, y_n^2, \ldots, y_n^m\}$ (where y^i are each of the dependent variables in the coupled ODE set) to the numerical approximation $\tilde{\vec{y}}_n$ in an analogous way to before;

$$\tilde{\vec{y}}_n = \vec{y}_n + \vec{\epsilon}_n$$

where now $\vec{\epsilon}_n = \{\epsilon_n^1, \epsilon_n^2, ..., \epsilon_n^m\}$. Inserting this into the finite difference equation (3.19) yields

$$\vec{y}_{n+1} + \vec{\epsilon}_{n+1} = \mathbf{T}\vec{y}_n + \mathbf{T}\vec{\epsilon}_n\ .$$

We can Taylor expand $\vec{y}_{n+1}$ about $\vec{y}_n$

$$\vec{y}_{n+1} = \vec{y}_n + \vec{f}_n\, h + \mathcal{O}(h^2) = (\mathbf{I} + \mathbf{L}\, h)\, \vec{y}_n + \mathcal{O}(h^2) \approx \mathbf{T}\vec{y}_n \ ,$$

where $\vec{f}_n = \{f^1(\vec{y}_n), f^2(\vec{y}_n), \ldots, f^m(\vec{y}_n)\}$, i.e., the functions in each ODE $dy^i/d\eta = f^i(\eta, \vec{y})$. (This is basically equation (3.16) applied to each ODE in the set.) This yields a matrix equation for the error propagation;

$$\vec{\epsilon}_{n+1} = \mathbf{T}\, \vec{\epsilon}_n \ . \tag{3.20}$$

It turns out that condition for stability is that the modulus of all the eigenvalues λ^i of the update matrix $\mathbf{T}$ must be less than or equal to unity

$$\boxed{\left|\lambda^i\right| \leq 1 \quad \text{for} \quad i = 1, \ldots, m} \tag{3.21}$$

For real eigenvalues the condition then translates to

$$\lambda^i \leq 1 \ \text{ and } \ -\lambda^i \leq 1\,, \tag{3.22}$$

To show condition (3.21) it is useful to diagonalise the matrix equation for error propagation so that we are left with m *decoupled* equations which we can deal with individually. Any non-singular, quadratic $m \times m$ matrix that has m linearly independent eigenvectors can be diagonalised, i.e., written as

$$\boxed{\mathbf{T} = \mathbf{R}\,\mathbf{D}\,\mathbf{R}^{-1} \quad \text{with} \quad \mathbf{D} \equiv \text{diag}(\lambda^1, \lambda^2, ..., \lambda^m)} \ , \tag{3.23}$$

where λ^i are the eigenvalues of $\mathbf{T}$ and $\mathbf{R}$ is the matrix whose columns are the eigenvectors of $\mathbf{T}$. Substituting (3.23) into (3.20) and operating on both sides with a further $\mathbf{R}^{-1}$ we obtain the diagonal (uncoupled) system

$$\vec{\zeta}_{n+1} = \mathbf{D}\, \vec{\zeta}_n\,, \tag{3.24}$$

where we have defined the linear transformation $\vec{\zeta}_n = \mathbf{R}^{-1}\, \vec{\epsilon}_n$ which rotates the coupled vectors onto an uncoupled basis. This reduces to m uncoupled equations for the rotated errors ζ_n^i

$$\zeta_{n+1}^i = \lambda^i\, \zeta_n^i\,.$$

Since these are uncoupled we can now impose the constraint on their growth separately for each of them, i.e., for all $i = 1, ..., m$ we have the requirement;

$$g^i = \left|\frac{\zeta_{n+1}^i}{\zeta_n^i}\right| = |\lambda_n^i| \leq 1 \ ,$$

for a stable method. Since the coupled and uncoupled basis are related by a linear transformation any constraint applied in one basis is equivalent to one applied in the other. That is, the condition (3.4) holds for the original equation (3.20) too.

Non-linear Systems

The above analysis of the eigenvalues of the update matrix can be used for a coupled system of non-linear ODEs, if they are first linearized around the base point $(\eta_n, \vec{y}_n)$. This is necessary to get the update matrix $\mathbf{T}$ that applies at the current step in the iteration.

Considering the Euler method, we have a set of equations

$$\tilde{y}^i_{n+1} = \tilde{y}^i_n + f^i(\eta,\, \tilde{y}^1_n,\, \tilde{y}^2_n,\, \ldots\,,\, \tilde{y}^m_n)\, h \quad . \tag{3.25}$$

We substitute $\tilde{y}^i_n = y^i_n + \epsilon^i_n$ into $f^i(\eta, \tilde{y}^1_n, \ldots)$ and Taylor expand about the true solution to first order in the error, assumed to be small $|\epsilon^i_n| \ll |y^i_n|$;

$$f^i(\eta, y^1_n + \epsilon^1_n,\, y^2_n + \epsilon^2_n,\, \ldots\,,\, y^m_n + \epsilon^m_n) = f^i(\eta, \vec{y}_n) + \sum_{k=1}^{m} \left.\frac{\partial f^i}{\partial (y^k)}\right|_n \epsilon^k_n + \mathcal{O}((\epsilon_n)^2) \quad . \tag{3.26}$$

This is essentially what was done in section 3.3 with equation (3.15). Following the procedure used before we then get

$$\epsilon^i_{n+1} \approx \epsilon^i_n + h \sum_{k=1}^{m} \left.\frac{\partial f^i}{\partial (y^k)}\right|_n \epsilon^k_n \tag{3.27}$$

so that the elements of the update matrix $\mathbf{T}_n$ are

$$\boxed{T_{jk} = \delta_{jk} + h\, \frac{\partial f^j}{\partial (y^k)} \quad ,} \tag{3.28}$$

where δ_{jk} is the Kronecker delta function. (Note that the superscripts in y^j, f^i, etc., denote components and not 'raising to a power'.) The eigenvalues of this update matrix can then be analysed to determine the local stability of the method.

3.5. Convergence

If a finite difference method is both consistent with the ODE (system) being solved and stable, then the approximate solution $\tilde{y}^i_n$ converges to the true solution y_n in the limit as $h \to 0$ (in the absence of round-off error).

3.6. Efficiency

Another consideration when selecting a method is the number of computations required for each step. The Euler method requires 1 functional evaluation for each step. As we will see higher order methods will require more evaluations at each step but will in general be more stable and accurate. A compromise must be reached between efficiency and accuracy and stability for any particular system being solved.

SECTION 4
Methods for Integrating ODE Systems

Outline of Section

- Predictor-Corrector method
- Multi-step methods
- Runge-Kutta methods
- Implicit methods

So far we have only seen one finite difference method (FDM) for integrating ODEs; the Euler method

$$y_{n+1} = y_n + f_n(\eta_n, y_n)\, h \quad .$$

It uses a forward difference scheme (FDS) to approximate y' in $y' = f(\eta, y)$. Its properties are summarised as follows:

- It is a 1st-order method: the global error $\sim \mathcal{O}(h)$.
- The local truncation error (i.e. error per step/iteration) is $\mathcal{O}(h^2)$.
- It is a consistent method.
- It is conditionally stable; $h \leq 2/|\partial f/\partial y|$ for a single ODE.
- It is unstable if $\partial f/\partial y > 0$ (single ODE).
- It is an **explicit** method.
- It is a **single-step** method.

Explicit methods involve evaluating $f(\eta, y)$ using known values of y, e.g., y_n. There are also **implicit** methods which involve evaluating $f(\eta, y)$ using y_{n+1}. **Single-step** methods just need the solution at the current iteration (y_n) to get the next value. **Multi-step** methods require previous values too (e.g. y_{n-1}) in order to get y_{n+1}.

In this section we look at some more explicit FDMs, which are superior to explicit Euler. We also take a brief look at implicit FDMs, concentrating on the implicit Euler method.

In this section we will drop the '$\sim$' notation for denoting numerical approximate solutions, unless otherwise indicated.

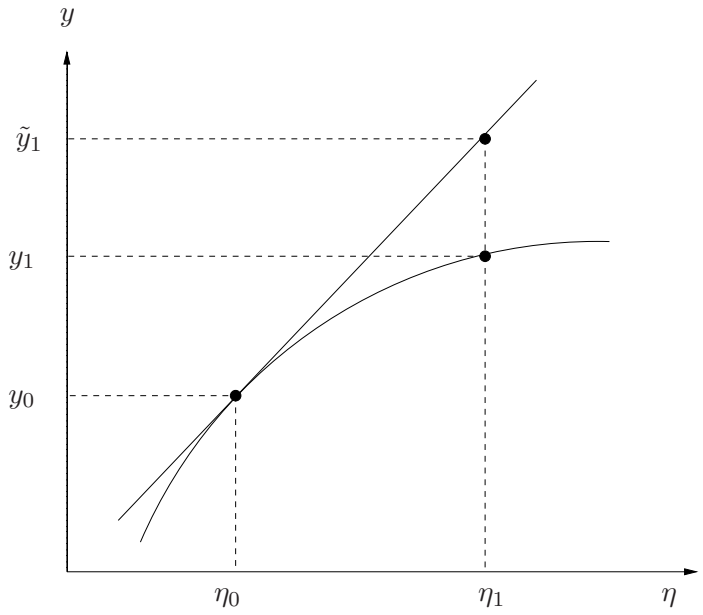

Figure 4.1. The Euler method applied to a non-linear function. The truncation error arises because the gradient is evaluated at the starting point and used to obtain the next point. Here, $\tilde{y}_1$ is the 'approximate' solution according to the Euler method and y_1 (on the curve) is the true solution to the ODE.

4.1. Predictor-Corrector Method

This is classified as an explicit, single-step method. It is a simple improvement to the Euler method. As shown in Fig. 4.1 the Euler method is inaccurate because it uses the gradient evaluated at the initial point to calculate the next point. This only gives a good estimate if the function is linear since the truncation error is quadratic in the step size.

To improve on this we could use the average gradient between the two points

$$y_{n+1} = y_n + \left(\frac{f_n + f_{n+1}}{2}\right) h\,, \tag{4.1}$$

where $f_n \equiv f(\eta_n, y_n)$ etc.

To show that this method is second order accurate (i.e. global error $\sim \mathcal{O}(h^2)$) we can start with the Taylor expansion of y_{n+1}

$$y_{n+1} = y_n + f_n\, h + f'_n\, \frac{h^2}{2} + f''_n(\xi)\, \frac{h^3}{3!}$$

where as usual $\eta_n < \xi < \eta_n + h$. Above, y_{n+1} is the exact value (at least until the remainder term is dropped). We can replace f'_n in the above Taylor expansion by a forward difference approximation plus a remainder term

$$f'_n = \frac{f_{n+1} - f_n}{h} - f''_n(\zeta)\frac{h}{2}\,,$$

to get

$$y_{n+1} = y_n + f_n\, h + \frac{f_{n+1} - f_n}{2h} h^2 - \left(f_n''(\zeta) \frac{h}{2} \right) \frac{h^2}{2} + \frac{1}{3!} f_n''(\xi)\, h^3 \,.$$

Expanding out and grouping terms of $\mathcal{O}(h^3)$ we get

$$y_{n+1} = y_n + \frac{f_{n+1} + f_n}{2}\, h + \mathcal{O}(h^3)\,, \tag{4.2}$$

So the local error (truncation) is 3rd order. Since the integration requires $\mathcal{O}(1/h)$ evaluations the global error (accuracy) will be 2nd order. Using (4.1) will give an approximate solution to the ODE since the remainder terms in the Taylor expansion of y_{n+1} and in using the forward difference approximation of f_n' have been discarded.

The only problem with this method is that we don't have the value y_{n+1} to use in $f_{n+1} = f(\eta_{n+1}, y_{n+1})$ in order to carry out the step. However we can use a first Euler step to *predict* y_{n+1} and use that to calculate f_{n+1} to use in the *corrected* Euler step.

	step 1 (predict):	$y^\star_{n+1} = y_n + f_n\, h \;\;,$
(4.3a)		$f^\star_{n+1} = f(\eta_{n+1}, y^\star_{n+1}) \;\;,$
(4.3b)	step 2 (correct):	$y^{\text{new}}_{n+1} = y_n + \dfrac{f^\star_{n+1} + f_n}{2}\, h \;\;.$

The above boxed equation is the algorithm for the predictor-corrector method, for a single 1st-order ODE. For a set of m coupled 1st-order ODEs, step 1 should be applied to each of the m ODEs to get the predicted value $(y^\star_{n+1})^i$ for each dependent variable y^i. Then $(f^\star_{n+1})^i = f^i(\eta_{n+1},\, \vec{y}^\star_{n+1})$ can be evaluated for each ODE. Then step 2 (corrected Euler) can be applied to each ODE.

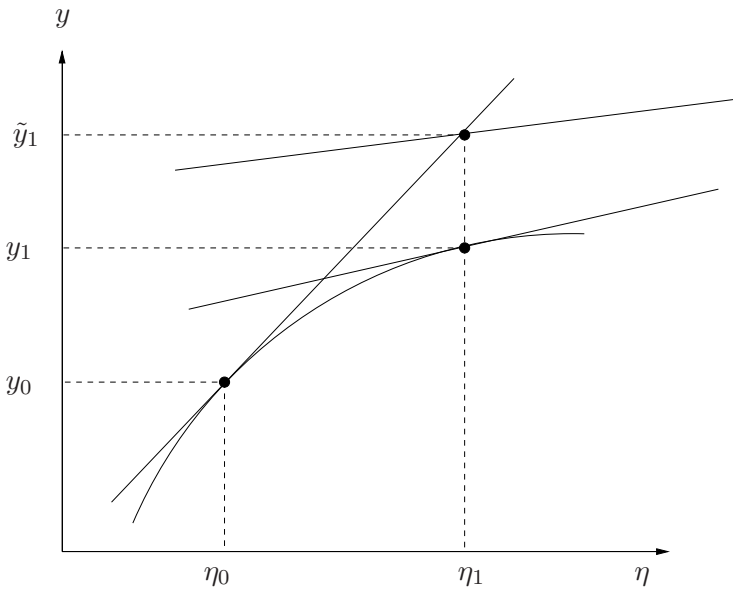

Figure 4.2. (Predictor-corrector) Ideally we would like to use the gradient at y_{n+1} to get the average gradient in the interval. However we do not have the true value y_{n+1} so we use the gradient calculated with the Euler estimate $\tilde{y}_{n+1}$.

The gradient calculated with the predicted point will not, in general, be exactly the correct value since it depends on both η and y but it will still be more accurate than the Euler method (see Fig. 4.2). The predictor-corrector method is $\mathcal{O}(h^2)$ accurate but involves two evaluations per step which is less efficient. It is conditionally stable for decaying solutions ($y' = -\alpha y$), but unstable for pure oscillating solutions ($y'' = -\omega^2 y$).

4.2. Multi-Step (Leapfrog) Method

We have seen how the use of a central gradient between two points is more accurate ($\mathcal{O}(h^2)$) than using the gradient at the first point ($\mathcal{O}(h)$). In the previous section we predicted the next point to take and average of the gradient between y_n and y_{n+1}. However since we have presumably evaluated values before y_n in previous steps we could use those to get an update. This requires storing a number of previous points.

The simplest of these methods is the Leapfrog method which uses the y_{n-1} value

$$\boxed{y_{n+1} = y_{n-1} + 2f_n\, h \quad ,} \tag{4.4}$$

and requires the storage of one previous step. In this method the point (η_n, y_n) is the midpoint between y_{n-1} and y_{n+1} where the gradient is used. The leapfrog algorithm essentially uses a central difference scheme to approximate the derivative y' in the ODE. The leapfrog method is an explicit FDM. The Leapfrog ($m = 1$ order multi-step) method is $\mathcal{O}(h^2)$ accurate and only requires one evaluation per step. This makes it more efficient than the Predictor-Corrector method. The method is stable for oscillating and growing solutions but is unstable for decaying solutions.

General multi-step

This kind of multi-step method can use any number of previous values, say 'm'. The Leapfrog is an $m = 1$ order multi-step method. In general an m^{th}-order multi-step method is equivalent to fitting $P_m(\eta)$, an m^{th}-degree polynomial, through the points and then approximating y' in the ODE by P'_m (for which an analytic expression can easily be obtained, once the coefficients of the interpolating polynomial have been found). However the drawback of multi-step methods is that $m + 1$ points are needed. So to apply a multi-step method from the outset, values before $\eta = \eta_0$ have to be guessed. If the guess is not a good one then the method can suffer from a initial value error.

Starting off

Typically, what is done is to use a single-step method for the first iteration(s). For instance to start off the leapfrog method, an Euler step can be used to get y_1 from the initial condition y_0. Then there are enough points to continue with the leapfrog scheme:

$$\begin{aligned} y_1 &= y_0 + f_0\, h\,, \\ y_2 &= y_0 + 2f_1\, h\,, \\ y_3 &= y_1 + 2f_2\, h\,, \\ &\cdots \\ y_{n+1} &= y_{n-1} + 2f_n\, h\,. \end{aligned} \tag{4.5}$$

To improve the accuracy of the starting step, a better single-step method could be used and/or smaller steps can be used (e.g. use k Euler steps with $h \to h/k$ to get y_1 for starting leapfrog).

Variable step size h

With single-step methods, it is easy to vary h during the calculation to, e.g., keep the error with a specified bounds. ('Adaptive step-size' algorithms can be found in any good numerical methods book; we won't cover them here.) Another disadvantage of multi-step methods, in comparison to single-step methods, is that varying h is more difficult (requiring back-calculating y_{n-1}, etc., using a good single-step method).

4.3. Runge-Kutta Methods

The Runge-Kutta methods are a family of explicit, single-step methods using more than one term in the Taylor series expansion of y_{n+1}. The method generalises the idea of using a weighted average of gradients calculated at m different points to estimate the change in y.

In general the RKm method equates to an m^{th}-order Taylor expansion, and is an m^{th}-order finite difference method. Thus the local error (truncation) is of $\mathcal{O}(h^{m+1})$ and the global accuracy of the method is $\mathcal{O}(h^m)$. A number of weighting schemes for the averaging of the gradients can be used for each RKm.

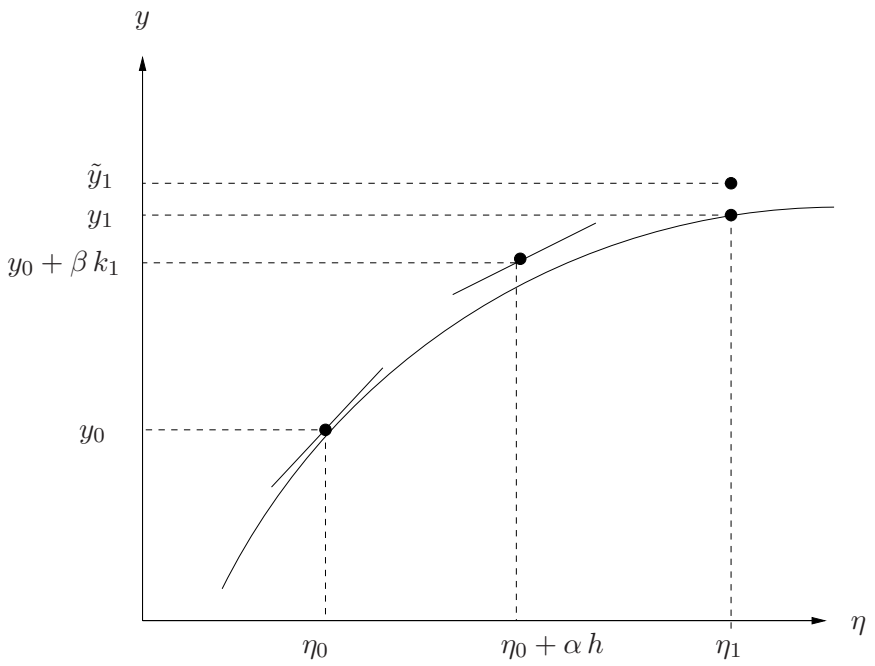

Figure 4.3. The RK2 method. A second gradient is calculated at a shifted position and averaged with the gradient at the starting position to obtain $\tilde{y}_1$, the next value of y. Comparing to fig. 4.1 you can see that $\tilde{y}_1$ is more accurate for RK2 compared to Euler.

RK2

The RK2 class of methods includes the Predictor-Corrector method discussed in section 4.1. The general RK2 scheme is;

$$y_{n+1} = y_n + a\,k_1 + b\,k_2 \quad , \tag{4.6a}$$
$$k_1 = h\,f(\eta_n, y_n) \quad , \tag{4.6b}$$
$$k_2 = h\,f(\eta_n + \alpha\,h, y_n + \beta\,k_1) \quad . \tag{4.6c}$$

Here a, b, α, and β are coefficients. Different choices for the coefficients give different methods

$$a = 0 \quad b = 1 \quad \alpha = \tfrac{1}{2} \quad \beta = \tfrac{1}{2} \qquad \text{:Single mid-point}$$
$$a = \tfrac{1}{2} \quad b = \tfrac{1}{2} \quad \alpha = 1 \quad \beta = 1 \qquad \text{:Predictor-Corrector} \tag{4.7}$$
$$a = \tfrac{1}{3} \quad b = \tfrac{2}{3} \quad \alpha = \tfrac{3}{4} \quad \beta = \tfrac{3}{4} \qquad \text{:Smallest error RK2}$$

The RK2 method is illustrated in Fig. 4.3. An Euler step is used to calculate a first shift in y. Then a second shift is calculated using the gradient at a point shifted by an amount $\alpha\,h$ in η and by an amount $\beta\,k_1$ in y. The two shifts are then averaged with weights a and b.

RK4

This uses four gradients to calculate an average one and is 4^{th}-order accurate. Each shift in y is a fraction of the previously calculated shift. The first one is an Euler step as before.

$$y_{n+1} = y_n + \frac{1}{6}\,(k_1 + 2\,k_2 + 2\,k_3 + k_4) \quad , \tag{4.8a}$$
$$k_1 = h\,f(\eta_n, y_n) \quad , \tag{4.8b}$$
$$k_2 = h\,f(\eta_n + \frac{1}{2}\,h, y_n + \frac{1}{2}\,k_1) \quad , \tag{4.8c}$$
$$k_3 = h\,f(\eta_n + \frac{1}{2}\,h, y_n + \frac{1}{2}\,k_2) \quad , \tag{4.8d}$$
$$k_4 = h\,f(\eta_n + h, y_n + k_3) \quad . \tag{4.8e}$$

The advantage of the RK4 method is that, although it requires four evaluations per step, it can take much larger values of h and is therefore more efficient than the Euler and Predictor-Corrector method. Higher order RK can be used but these require more (e.g. 11 for RK6) evaluations per step and are therefore slower. Only for $m < 4$ are $\leq m$ evaluations required per step.

One practical advantage of RK methods over multi-step methods is that we don't need to store any previous steps. This makes coding it a little easier.

4.4. Implicit Methods

Implicit methods for solving an ODE require evaluation of $f(\eta, y)$ using the yet unknown value y_{n+1}. The simplest is implicit Euler:

$$\boxed{y_{n+1} = y_n + f(\eta_{n+1}, y_{n+1})\, h \quad .} \tag{4.9}$$

Implicit Euler is still a 1st-order method, with global accuracy $\mathcal{O}(h)$, just like its explicit cousin. For a linear function $f(\eta, y) = p(\eta)y + q(\eta)$ equation (4.9) can easily be rearranged to give $y_{n+1} = g(\eta_{n+1}, y_n)$ which can easily be solved. For a non-linear function, (4.9) is a non-linear algebraic equation in y_{n+1}, that needs to be solved using something like the bisection, Newton or secant method (see section 2.5).

Stability

What is the advantage then? The advantage of implicit Euler over explicit Euler is that it is **unconditionally stable**. There is no critical step size h, above which numerical instability occurs. Of course, using a large h in implicit Euler will give a very inaccurate solution. The stability analysis carried out in section 3.3 (for explicit Euler) can be applied to implicit Euler, in much the same way, yielding

$$\frac{\epsilon_{n+1}}{\epsilon_n} \approx \left(1 - \frac{\partial f}{\partial y}\, h\right)^{-1} \tag{4.10}$$

so that for decaying problems ($\partial f/\partial y < 0$) we have $g \leq 1$ for ***any*** (positive) h. What happens to y_{n+1} for large h? It simply tends to zero; the same behaviour as the exact solution. Inspection of equation (4.9) for the linear decay problem ($f = -\beta\, y$) shows that $y_{n+1} = y_n/(1 + \beta\, h) \to 0$ as $h \to \infty$.

Just like we have seen more advanced explicit methods (than Euler), more advanced implicit methods exist and have superior stability characteristics to explicit methods.

Coupled ODEs

Implicit Euler works for coupled, linear, 1st-order ODEs. One has

$$\begin{aligned} & & \vec{y}_{n+1} &= \vec{y}_n + h\mathbf{L} \cdot \vec{y}_{n+1} \\ \therefore \quad & & (\mathbf{I} - h\mathbf{L}) \cdot \vec{y}_{n+1} &= \vec{y}_n \\ \therefore \quad & & \vec{y}_{n+1} &= (\mathbf{I} - h\mathbf{L})^{-1} \cdot \vec{y}_n = \bar{\mathbf{T}} \cdot \vec{y}_n \end{aligned} \tag{4.11}$$

where $\mathbf{L}$ is the same matrix operator as for explicit Euler. $\bar{\mathbf{T}}$ is the update matrix for implicit Euler and is the inverse of the matrix $(\mathbf{I} - h\mathbf{L})$. Hence implicit Euler works if $(\mathbf{I} - h\mathbf{L})$ is a non-singular matrix so that $\bar{\mathbf{T}}$ exists and can be found.

With non-linear coupled ODEs, things are not so easy. In principle one needs to find the solution of a set of (coupled) non-linear algebraic equations. This is at best tricky and at worst insoluble. A way out is to linearize the functions f^i in each ODE about the known 'point' $(\eta_n, \vec{y}_n)$. But then the update matrix $\bar{\mathbf{T}}$ has to be calculated at each step, since the matrix $\mathbf{L}$ obtained by linearization depends on $\vec{y}_n$. Thus implicit methods are generally less efficient than explicit methods.

SECTION 5
Matrix Algebra

Outline of Section

- Linear Algebraic Equations
- Direct Method - LU Decomposition
- Iterative Methods
- Eigensystems

In this section we will look at solving simultaneous linear algebraic equations of the form

$$\begin{aligned} a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 + ... + a_{1N}\,x_N &= b_1\,, \\ a_{21}\,x_1 + a_{22}\,x_2 + a_{23}\,x_3 + ... + a_{2N}\,x_N &= b_2\,, \\ ... &= ...\,, \\ a_{M1}\,x_1 + a_{M2}\,x_2 + a_{M3}\,x_3 + ... + a_{MN}\,x_N &= b_M\,, \end{aligned} \tag{5.1}$$

which can be written as the matrix operation

$$\mathbf{A}\cdot\vec{x} = \vec{b}\,, \tag{5.2}$$

where $\mathbf{A}$ is an $M \times N$ matrix and $\vec{x}$ is a vector with N components and $\vec{b}$ is a vector with M components. A given coefficients a_{ij} in the above simultaneous equations becomes the elements of the matrix located at row i (i.e. the first subscript) and column j (the 2nd subscript). We want to solve for the unknown variables $\vec{x}$ for a given linear operator $\mathbf{A}$ and right-hand side $\vec{b}$. M is the number of equations in the system and N is the number of unknowns we have to solve for.

The need to solve a matrix equation such as (5.2) arises frequently in many numerical methods. In section 4.4 we saw them in implicit finite difference methods for solving sets of coupled linear ODEs. In section 10 we will encounter them in solving PDEs via finite difference methods. The $N \times N$ matrices there will be very large, with N equal to the number of spatial grid points! Matrix equations will also occur in solving boundary value problems (section 6) and minimisation of functions (section 7). Of course matrix

equations arise directly in many physical problems too, such as quantum mechanics and classical mechanics.

The need to find eigenvalues of a matrix is also a common task both in numerical methods and in physics problems. In the former case, think back to the stability analysis of FDMs for coupled ODEs in section 3.4.

You will already know some **direct methods** for solving $\mathbf{A}\cdot\vec{x}=\vec{b}$, such as **Gaussian elimination** and possibly **Gauss-Jordan elimination**. These direct methods involve manipulating the matrix to eliminate elements so that the system can be easily solved. The manipulation involves, e.g., swapping rows, adding a multiple of one row to another, etc. For Gaussian elimination one manipulates the augmented matrix formed by $\left(\mathbf{A}|\vec{b}\right)$, in an effort to end up with $\mathbf{A}$ in upper triangular form. For Gauss-Jordan elimination one manipulates the augmented matrix formed by $(\mathbf{A}|\mathbf{I})$, until one has $(\mathbf{I}|\mathbf{B})$, which yields the inverse of $\mathbf{A}$ since it turns out that $\mathbf{B}=\mathbf{A}^{-1}$. This section of the course concentrates mainly on **iterative methods** for solving $\mathbf{A}\cdot\vec{x}=\vec{b}$ and finding eigenvalues. These turn out to be more suitable than direct methods for large matrices.

General Considerations

Before diving into various new methods it is worth discussing some general points about solving matrix equations. Consider the inhomogeneous matrix equation $\mathbf{A}\cdot\vec{x}=\vec{b}$. If $M=N$ and all equations are **linearly independent** then there should exist a unique solution for $\vec{x}$. However if some equations are degenerate (i.e. can be written as linear combination of other equations) then effectively the system has less equations than unknowns $M<N$ and there may not be a solution (or a unique solution). This will be evident in the matrix $\mathbf{A}$ being singular (has zero eigenvalues) and having a vanishing determinant. An approximate solution can be found using Singular Value Decomposition (SVD) not covered here.

Conversely if $M>N$ the system is over-determined. In general no solution exists in this case but an approximate one can usually be found by a linear least squares search for the solution fitting most equations closely.

If the matrix equation is homogeneous, i.e. $\mathbf{A}\cdot\vec{x}=\vec{0}$, then the solution is non-trivial only if $\det\mathbf{A}=0$.

In particular for the $M=N$ case the system can be extended to N unknown solutions for N right-hand sides i.e.

$$\mathbf{A}\cdot\mathbf{X}=\mathbf{B}\,, \tag{5.3}$$

where $\mathbf{X}$ and $\mathbf{B}$ are now $N\times N$ matrices too. Each column of $\mathbf{X}$ is one of the vectors $\vec{x}_i$ of the N equations. Similarly for $\mathbf{B}$.

If we can solve for $\mathbf{X}$ then we can also use the methods to find the inverse of a matrix since

$$\mathbf{A}\cdot\mathbf{A}^{-1}=\mathbf{I}\ , \tag{5.4}$$

so setting $\mathbf{B}$ to the identity matrix and solving yields $\mathbf{A}^{-1}$ As we will see we can also get the determinant of a matrix for free.

5.1. LU Decomposition

This is a direct method. We will focus on the case where $M = N$ i.e. $\mathbf{A}$ is a square matrix. We will assume that the matrix can be decomposed into upper and lower triangular matrices

$$\mathbf{A} \equiv \mathbf{L} \cdot \mathbf{U} \tag{5.5}$$

where $\mathbf{L}$ and $\mathbf{U}$ are of the form e.g.

$$\begin{pmatrix} \alpha_{11} & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & \beta_{33} \end{pmatrix} , \tag{5.6}$$

respectively. The algorithm to carry out LU decomposition is related to Gaussian elimination.

In that case we can write the system

$$\boxed{\mathbf{A} \cdot \vec{x} \equiv \mathbf{L} \cdot \mathbf{U} \cdot \vec{x} = \mathbf{L} \cdot \vec{y} = \vec{b} \ ,} \tag{5.7}$$

where $\vec{y} = \mathbf{U} \cdot \vec{x}$.

The trick here is that it is trivial to solve a triangular system i.e. one where each equation is a function of one more unknown than the previous one. To do this we first carry out a **forward substitution** to solve for $\vec{y}$.

$$y_1 = \frac{b_1}{\alpha_{11}} \tag{5.8}$$

$$y_i = \frac{1}{\alpha_{ii}} \left(b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right) \quad i = 2, 3, ..., N \quad \text{(in this order)} \ . \tag{5.9}$$

We then solve for $\vec{x}$ by carrying out a **backwards substitution**

$$x_N = \frac{y_N}{\beta_{NN}} \tag{5.10}$$

$$x_i = \frac{1}{\beta_{ii}} \left(y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right) \quad i = N-1, N-2, ..., 1 \ , \tag{5.11}$$

where the calculation in (5.11) must be carried out in the order shown. We will not look into the details of how the decomposition is carried out (see e.g. *Numerical Recipes*). It can be done using black-box routines in Linear Algebra packages (e.g. `LINPACK` and `LAPACK`). The decomposition requires $\mathcal{O}(N^3)$ operations.

If we want to find the inverse of $\mathbf{A}$ then we can decompose the matrix ($\mathcal{O}(N^3)$) once and solve equation (5.4) for each column of $\mathbf{A}^{-1}$ which requires $N \times \mathcal{O}(N^2)$ operations i.e. also $\mathcal{O}(N^3)$. Thus taking an inverse costs $\mathcal{O}(N^3)$ operations. This can become very slow even on the fastest computers when $N > \mathcal{O}(10^3)$.

Once the matrix is LU decomposed the determinant is easy to calculate since

$$\det \mathbf{A} = \prod_{j=1}^{N} \beta_{jj} \ . \tag{5.12}$$

However this will quickly lead to overflows so it is usually calculated as

$$\ln(\det \mathbf{A}) = \sum_{j=1}^{N} \ln \beta_{jj} \quad . \tag{5.13}$$

5.2. Iterative Solution - Jacobi Method

Given that inverting matrices explicitly can be prohibitive even on the fastest computers we can use iterative methods to converge onto a solution from an initial guess. This is a method which works if the matrix $\mathbf{A}$ is diagonally dominant, i.e.,

$$|A_{ii}| > \sum_{j \neq i} |A_{ij}| \ . \tag{5.14}$$

Sparse systems, where only a few variables appear in each equation, can often be rearranged into a diagonally dominant form such as a band diagonal matrix which looks like

$$\begin{pmatrix} \cdot & \cdot & & & & & \\ \cdot & \cdot & \cdot & & & & \\ & \cdot & \cdot & \cdot & & & 0 \\ & & \cdot & \cdot & \cdot & & \\ & 0 & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \\ & & & & & \cdot & \cdot \end{pmatrix} . \tag{5.15}$$

We can rearrange $\mathbf{A}$ as the sum of its diagonal elements and its lower and upper triangles

$$\boxed{\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U} \ .} \tag{5.16}$$

Not that the $\mathbf{L}$ and $\mathbf{U}$ matrices here are nothing to do with those from LU decomposition! (For a start, we are adding rather than multiplying to compose $\mathbf{A}$.) We then have

$$\mathbf{A} \cdot \vec{x} = (\mathbf{L} + \mathbf{D} + \mathbf{U}) \cdot \vec{x} = \vec{b},$$

giving

$$\mathbf{D} \cdot \vec{x} = -(\mathbf{L} + \mathbf{U}) \cdot \vec{x} + \vec{b}.$$

By multiplying by $\mathbf{D}^{-1}$ we can rearrange this to get

$$\vec{x} = -\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U}) \cdot \vec{x} + \mathbf{D}^{-1} \cdot \vec{b}.$$

This looks like an iterative equation if we identify the left and right-hand sides with the new and old value of $\vec{x}$ respectively

$$\boxed{\vec{x}_{n+1} = -\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U}) \cdot \vec{x}_n + \mathbf{D}^{-1} \cdot \vec{b}} \tag{5.17}$$

which we can use to find $\vec{x}$ starting from a guess $\vec{x}_0$. Equation (5.17) is the Jacobi method. In general this method will converge from any starting guess if the update matrix $\mathbf{T} \equiv \mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U})$ has eigenvalues $|\lambda^i| < 1$ as we saw in Section 4. This condition is equivalent to requiring that $\mathbf{A}$ be diagonally dominant. Note that the

inverse of $\mathbf{D}$ is easy to calculate since it is a diagonal matrix. For any diagonal matrix $\mathbf{D} = \text{diag}(d_1, d_2, \dots\ d_n)$ the inverse is

$$\mathbf{D}^{-1} = \text{diag}\left(\frac{1}{d_1}, \frac{1}{d_2}, \dots, \frac{1}{d_n}\right) \quad . \tag{5.18}$$

Proof of Jacobi convergence condition – The condition $|\lambda^i| < 1$ is simple to prove, using similar principles to those used previously in section 3.4 for the stability analysis of finite difference solution of a set of coupled linear ODEs. As before, we consider the errors at each iteration

$$\vec{\epsilon}_n = \vec{x}_n - \vec{x} \qquad \text{and} \qquad \vec{\epsilon}_{n+1} = \vec{x}_{n+1} - \vec{x}\,, \tag{5.19}$$

where $\vec{x}$ is the true solution. Then

$$\begin{aligned} \vec{\epsilon}_{n+1} = \vec{x}_{n+1} - \vec{x} &= -\mathbf{T}\cdot\vec{x}_n + \mathbf{D}^{-1}\cdot\vec{b} - \left(-\mathbf{T}\cdot\vec{x} + \mathbf{D}^{-1}\cdot\vec{b}\right) \quad , \\ &= -\mathbf{T}\cdot\vec{x}_n + \mathbf{T}\cdot\vec{x} \quad , \\ &= -\mathbf{T}\cdot(\vec{x}_n - \vec{x}) \quad , \\ \therefore \qquad \vec{\epsilon}_{n+1} &= -\mathbf{T}\cdot\vec{\epsilon}_n \quad . \end{aligned} \tag{5.20}$$

Conceptually, assuming we were to know the eigenvectors and eigenvalues of $\mathbf{T}$, the system can be diagonalised (as done in section 3.4)

$$\mathbf{T} = \mathbf{R}\cdot\mathbf{\Lambda}\cdot\mathbf{R}^{-1} \quad \text{with} \quad \mathbf{\Lambda} \equiv \text{diag}(\lambda^1, \lambda^2, ..., \lambda^m)\,. \tag{5.21}$$

We can then define a 'rotated' error $\vec{\zeta}_n = \mathbf{R}^{-1}\cdot\vec{\epsilon}_n$ such that

$$\vec{\zeta}_{n+1} = \mathbf{\Lambda}\cdot\vec{\zeta}_n \quad , \tag{5.22}$$

which allows us to impose a convergence criterion on each eigenvalue individually since

$$\left|\frac{\zeta^i_{n+1}}{\zeta^i_n}\right| \equiv \left|\lambda^i\right| < 1 \quad . \tag{5.23}$$

Iterative methods are faster than the direct inversion methods (such as LU decomposition) since we only need to carry out a matrix multiplication at each step ,i.e., the number of operations will be $\mathcal{O}(N^k \times N_{\text{iter}})$ where N_{iter} is the number of iterations required to reach a chosen level of convergence (fractional change in solution is less than a chosen tolerance) and the index k is 3 for a naive multiplication. However it can be shown that matrix multiplication can be done with $k < 3$, in particular the Coppersmith & Winegrad (1990) method scales with $k = 2.376$ and the commonly used `BLAS` routine scales with $k = 2.807$. Although these may seem close to $k = 3$ they make a big difference in computation time.

Checking for convergence

Typically, one checks that the fractional change in the norm of the solution vector

$$\varepsilon = \left|\frac{\|\vec{x}_{n+1}\| - \|\vec{x}_n\|}{\|\vec{x}_n\|}\right| \quad , \tag{5.24}$$

is below a specified tolerance; something a few orders of magnitude above the factional round-off error (e.g. $\varepsilon_{tol} \sim 10^{-14}$). There are many measures of the norm of a vector; the general ℓ-norm is

$$\|\vec{x}\|_\ell \equiv \left(\sum_{i=1}^{n} |x_i|^\ell \right)^{\frac{1}{\ell}} \quad . \tag{5.25}$$

Commonly the $\ell = 1$ (sum of $|x_i|$), the $\ell = 2$ (the familiar Euclidean sum of squares) or the $\ell = \infty$ (the largest component of the vector!) are used.

Another way to check for convergence is to substitute $\vec{x}_{n+1}$ into $\mathbf{A} \cdot \vec{x} = \vec{b}$ and check the fractional difference between the LHS and RHS. This is more costly to do every iteration.

5.3. Iterative Gauss - Seidel Method

An alternative method which converges faster than the Jacobi method is to take

$$\boxed{\vec{x}_{n+1} = -(\mathbf{L} + \mathbf{D})^{-1} \cdot \mathbf{U} \cdot \vec{x}_n + (\mathbf{L} + \mathbf{D})^{-1} \cdot \vec{b} \ ,} \tag{5.26}$$

where the update matrix is now $\mathbf{T} = (\mathbf{L} + \mathbf{D})^{-1} \cdot \mathbf{U}$.

The method converges in less steps but is a little harder to implement since we need to compute $(\mathbf{L} + \mathbf{D})^{-1}$. However due to the special nature of the diagonally dominant $(\mathbf{L} + \mathbf{D})$ operator, its inverse can be approximated efficiently so this is not as expensive as calculating the inverse of a general matrix.

5.4. Largest Eigenvalue of a Matrix

We have seen how finding the largest eigenvalue of a matrix would be useful in checking convergence criteria. The **method of powers** is a simple method to approximate the largest eigenvalue of any non-singular $N \times N$matrix with N linearly independent eigenvectors. It also yields the associated eigenvalue.

Consider a system of the type

$$\mathbf{A} \cdot \vec{x} = \lambda \, \vec{x} \, , \tag{5.27}$$

with $\mathbf{A}$ and $N \times N$ matrix. In general if $\mathbf{A}$ is non-singular (i.e. $\det \mathbf{A} \neq 0$) it will have N independent eigenvalues (λ_i) with associated eigenvectors ($\vec{e}_i$)

$$\mathbf{A} \cdot \vec{e}_i = \lambda_i \, \vec{e}_i \, . \tag{5.28}$$

The linearly independent eigenvectors form a basis in which any vector can be expanded

$$\vec{v} = \sum_{i=1}^{N} c_i \, \vec{e}_i \, . \tag{5.29}$$

We start with a random choice of vector $\vec{v}$ and act on it with $\mathbf{A}$ an number of times

$$\begin{aligned} \mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\mathbf{A}\cdot\vec{e}_i = \sum_{i=1}^{N} c_i\,\lambda_i\,\vec{e}_i \ , \\ \mathbf{A}\cdot\mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^2\,\vec{e}_i \ , \\ \mathbf{A}^n\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^n\,\vec{e}_i \ , \\ \therefore \quad \lim_{n\to\infty} \mathbf{A}^n\cdot\vec{v} &\to c_j\,\lambda_j^n\,\vec{e}_j \ , \end{aligned} \tag{5.30}$$

where λ_j is the largest eigenvalue. Since any multiple of an eigenvector is an eigenvector itself we have found the largest eigenvector. We can normalise the vector we have just found as

$$\boxed{\vec{\omega}_j = \frac{\mathbf{A}^n\cdot\vec{v}}{|\mathbf{A}^n\cdot\vec{v}|} \equiv \frac{\vec{e}_j}{|\vec{e}_j|} \ ,} \tag{5.31}$$

such that ${\vec{\omega}_j}^T\cdot\vec{\omega}_j = 1$. Then it is simple to calculate the eigenvalue itself since

$$\boxed{{\vec{\omega}_j}^T\cdot\mathbf{A}\cdot\vec{\omega}_j = {\vec{\omega}_j}^T\cdot(\lambda_j\vec{\omega}_j) = \lambda_j \ .} \tag{5.32}$$

Smallest eigenvalue

We can also use the method of powers to find the smallest eigenvalue of the system, by using $\mathbf{A}^{-1}$ instead of $\mathbf{A}$ in equations (5.31) and (5.31). This works because, the inverse of a matrix has the same eigenvectors but with the inverse eigenvalues. This can be shown since

$$\vec{e}_i = \mathbf{A}^{-1}\cdot\mathbf{A}\cdot\vec{e}_i = \mathbf{A}^{-1}\cdot(\lambda_i\,\vec{e}_i) = \lambda_i\,\mathbf{A}^{-1}\cdot\vec{e}_i \quad ,$$

thus

$$\mathbf{A}^{-1}\cdot\vec{e}_i = (\lambda_i)^{-1}\,\vec{e}_i \ .$$

So the power method on $\mathbf{A}^{-1}$ finds its largest eigenvalue which will be the smallest eigenvalue of the original $\mathbf{A}$.

5.5. Other Eigenvalues

There are many methods for finding the remaining eigenvalues of a matrix but most use the **shift method** by finding the eigenvalue closest to a given value α. To see this define the shifted matrix

$$\mathbf{A}' \equiv \mathbf{A} - \alpha\,\mathbf{I}\,, \tag{5.33}$$

such that

$$\mathbf{A}'\cdot\vec{e}_i = \mathbf{A}\cdot\vec{e}_i - \alpha\,\mathbf{I}\cdot\vec{e}_i = \lambda_i\,\vec{e}_i - \alpha\,\vec{e}_i = (\lambda_i-\alpha)\,\vec{e}_i\,. \tag{5.34}$$

So the shifted matrix has the same eigenvectors but shifted eigenvalues. Thus finding the largest eigenvalue of $(\mathbf{A}')^{-1}$ (e.g. by the power method) will give the eigenvalue closest

to the value α. But how to get $(\mathbf{A}')^{-1}$? In principle, we can solve equation $\mathbf{A}' \cdot \mathbf{X} = \mathbf{I}$ for $\mathbf{X}$ using an efficient method like Jacobi or Gauss-Seidel (or LU decomposition for a relatively small matrix). Then $\mathbf{X}$ will be $(\mathbf{A}')^{-1}$ that we seek.

A crucial point is then to chose a suitable value for α. One way is to use **Gerschgorin's Theorem** which states that for any eigenvalue λ_i the following inequality is satisfied

$$|\lambda_i - A_{ii}| \leq \sum_{j \neq i} |A_{ij}| \ , \tag{5.35}$$

i.e. the eigenvalue shifted by a diagonal element lies within a circle given by the sum of the absolute value of off-diagonal elements.

This is particularly useful if $\mathbf{A}$ is diagonally dominant since the radius will be small and any algorithm based on the theorem will be more efficient. For example take the following matrix[1]

$$\mathbf{A} \equiv \begin{pmatrix} 1 & 0.1 & 0 \\ 0.1 & 5 & 0.2 \\ 0.1 & 0.3 & 10 \end{pmatrix} . \tag{5.36}$$

We then have that $0.9 \leq \lambda_1 \leq 1.1$, $4.7 \leq \lambda_2 \leq 5.3$, or $9.6 \leq \lambda_3 \leq 10.4$. And we can find the exact values by setting α to 1, 5, or 10 in the shift method.

[1] To make life easier I have chosen one where the ranges of each row do not overlap...

SECTION 6
Boundary Value Problems

Outline of Section

- Boundary Value Problems
- Shooting Method
- Finite differences
- Eigensystems

Consider the general, linear, second order equation

$$\alpha \frac{d^2y}{d\eta^2} + \beta \frac{dy}{d\eta} + \gamma\, y = k\,. \tag{6.1}$$

Normally we would require initial conditions specifying the value of the solution at some initial point $\eta = a$, i.e. $y(a)$ and its derivative $y'(a)$, to integrate the system to a final point.

But what if we want to find the solution that matches an initial and final condition e.g. $y(a)$ and $y(b)$? Since we are giving two conditions and we have two degrees of freedom (second order ODE) we should be able to find a consistent solution.

6.1. Shooting Method

A first method we can use is the **shooting method** where we start at a with $y(a) = A$, make a guess for $y'(a) = C_1$ and find a solution $y_1(\eta)$. This can be obtained via a finite difference method (from section 4), or algebraically if we are lucky. In general the solution will not end at the desired point $y(b) = B$ say, but will end instead at $y_1(b) = B_1$.

We then make another guess starting from the same point $y(a) = A$ but with $y'(a) = C_2$ and find a new solution $y_2(\eta)$ which ends at $y_2(b) = B_2$. Since the system is **linear** a sum of two solutions will itself be a solution of the system so we can introduce a weighted average of y_1 and y_2 as a new solution

$$\boxed{y_c(\eta) = c\, y_1(\eta) + (1-c)\, y_2(\eta)\ \ ,} \tag{6.2}$$

where c is an unknown constant which makes y_c the required solution which ends at $y(b) = B$.

We can check that

$$y_c(a) = c\,A + (1-c)\,A = A\,,$$

as required and the condition that

$$y_c(b) = c\,B_1 + (1-c)\,B_2 = B\,,$$

gives us a solution for c

$$\boxed{c = \frac{B - B_2}{B_1 - B_2}\ .} \tag{6.3}$$

So finding the required solution to a linear system is relatively straightforward using the shooting method. However it still requires solving the system twice to get a single solution. Next we will look at another method which solves the system explicitly via matrix methods.

6.2. Finite Difference Method

Consider the system

$$\frac{d^2y}{d\eta^2} = k\,. \tag{6.4}$$

We can re-write the second derivative as a central difference and discretise the solution with m intervals of size h as in Section 2.2 (see also Problem Sheet 1)

$$\frac{y_{i-1} - 2\,y_i + y_{i+1}}{h^2} = k\,. \tag{6.5}$$

Then, since we have $y_0 = A$ and $y_m = B$, the system will look like

$$\begin{aligned} A - 2\,y_1 + y_2 &= k\,h^2\,,\\ y_1 - 2\,y_2 + y_3 &= k\,h^2\,,\\ &\ \ \vdots\\ y_{m-2} - 2\,y_{m-1} + B &= k\,h^2\,, \end{aligned} \tag{6.6}$$

i.e. $m-1$ linear equations. The system can be written in matrix form as

$$\begin{pmatrix} -2 & 1 & 0 & 0 & .... & 0 & 0\\ 1 & -2 & 1 & 0 & .... & 0 & 0\\ & & & \vdots & & & \\ 0 & 0 & 0 & 0 & .... & 1 & -2 \end{pmatrix} \begin{pmatrix} y_1\\ y_2\\ \vdots\\ y_{m-1} \end{pmatrix} = \begin{pmatrix} k\,h^2 - A\\ k\,h^2\\ \vdots\\ k\,h^2 - B \end{pmatrix}\,. \tag{6.7}$$

Since this is a diagonally dominated system it can be solved using the Jacobi or Gauss-Seidel method introduced in Sections 5.2 and 5.3. Figure 6.1 illustrates the finite difference method applied to solving a boundary value problem.

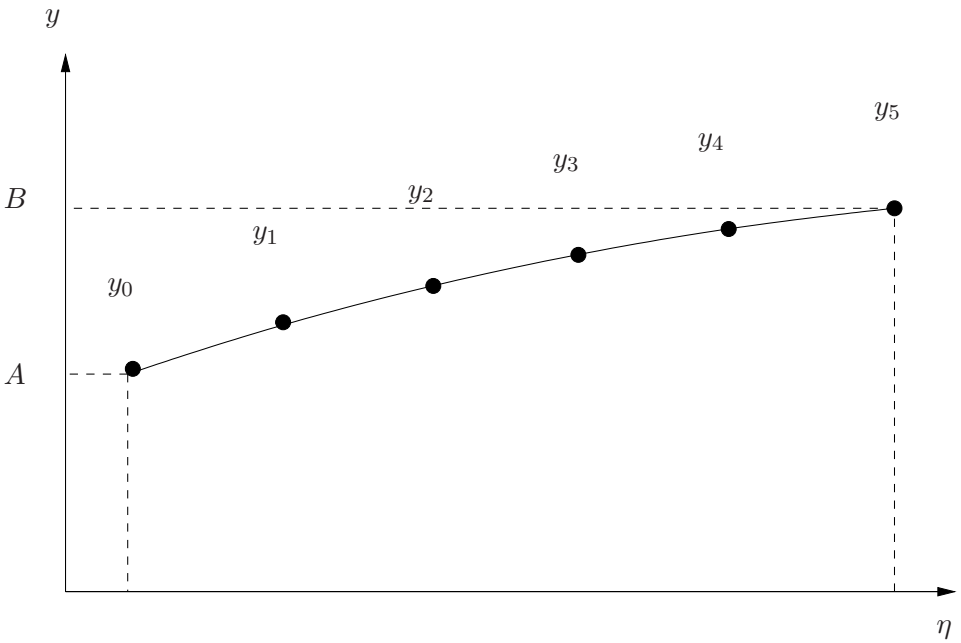

Figure 6.1. Finite difference method, for solution of boundary value problem. Example of discrete points with $m = 5$.

6.3. Derivative Boundary Conditions

On occasion we know the derivative at one of the boundaries but not the initial condition for the variable. Considering the same system as in (6.4) we could have boundary conditions

$$y'(a) = C \quad \text{and} \quad y(b) = B\,. \tag{6.8}$$

We can adapt the finite difference method by taking a central difference for the derivative at the boundary

$$y_0'(a) \approx \frac{y_1 - y_{-1}}{2h} = C\,, \tag{6.9}$$

and extending the system by one interval so that the first equation reads

$$y_{-1} - 2\,y_0 + y_1 = k\,h^2\,, \tag{6.10}$$

which, given (6.9), is

$$-\,2\,y_0 + 2y_1 = k\,h^2 + 2\,h\,C\,. \tag{6.11}$$

The system can then be written as m linear equations

$$\begin{pmatrix} -2 & 2 & 0 & 0 & .... & 0 & 0 \\ 1 & -2 & 1 & 0 & .... & 0 & 0 \\ & & & \cdot & & & \\ & & & \cdot & & & \\ & & & \cdot & & & \\ 0 & 0 & 0 & 0 & .... & 1 & -2 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_{m-1} \end{pmatrix} = \begin{pmatrix} k\,h^2 + 2\,h\,C \\ k\,h^2 \\ \cdot \\ \cdot \\ \cdot \\ k\,h^2 - B \end{pmatrix}. \tag{6.12}$$

6.4. Eigenvalue Problems

For the particular case where the differential equations are homogeneous and linear we can view the problem as an eigensystem. For example, consider the wave equation

$$\frac{d^2y}{dx^2} + k^2 y = 0\,, \tag{6.13}$$

with boundary conditions $y(0) = 0$ and $y(1) = 0$. This describes the vibrations on a string of length 1 and fixed at the endpoints.

The general solution to this system is easily found to be

$$y(x) = A\,\sin(k\,x) + B\,\cos(k\,x)\,. \tag{6.14}$$

The boundary conditions imply that $B = 0$ and that $k = \pm n\,\pi$.

The solution describes fundamental modes of vibrations on the string where $n = 1, 2, 3, ...$ etc.

The system (and more complicated ones in particular) can be solved using finite differences

$$\frac{y_{i-1} - 2\,y_i + y_{i+1}}{h^2} + k^2\,y_i = 0\,, \tag{6.15}$$

which gives the eigensystem

$$\begin{pmatrix} +2 & -1 & 0 & 0 & .... & 0 & 0 \\ -1 & +2 & -1 & 0 & .... & 0 & 0 \\ & & & \cdot & & & \\ & & & \cdot & & & \\ & & & \cdot & & & \\ 0 & 0 & 0 & 0 & .... & -1 & +2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_{m-1} \end{pmatrix} = h^2\,k^2 \begin{pmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_{m-1} \end{pmatrix}, \tag{6.16}$$

where $\lambda = h^2\,k^2$ are the eigenvalues of the system. It is often useful to find the eigenvector corresponding to the largest (or smallest) eigenvalues. These define the fundamental modes of the system and the power method can be used to find these.

SECTION 7
Minimisation and Maximisation of Functions

Outline of Section

- One–dimensional minimisations
- Multi-dimensional minimisations

This section concerns finding local or global minima (maxima) of a function of one variable $f(x)$ and of many independent variables $f(x_1, x_2, \ldots, x_N)$, written more succinctly as $f(\vec{x})$ where $\vec{x}$ is an N-dimensional vector. Both the position of a minimum $\vec{x}_\star$ and the value of the function there $f(\vec{x}_\star)$ may be of interest. The function can be non-linear. The methods covered will find minima; maxima can be found by applying these methods to $F(\vec{x}) = -f(\vec{x})$. Root-finding of non-linear functions $f(x)$ was introduced in section 2.5. In principle roots of a non-linear function involving many variables $f(\vec{x})$ can be found by finding minima $\vec{x}_\star$ of $|f(\vec{x})|^2$ where $f(\vec{x}_\star) = 0$. There exist iterative matrix solvers that are more efficient than Jacobi and Gauss-Seidel which are based on minimisation of a function based on the matrix $\mathbf{A}$, the vectors $\vec{x}$ and $\vec{b}$. (These methods are not covered in this course, but it is worth knowing that you have the tools to understand them if you wished!)

We will deal with **unconstrained** minimisation here. ***Constrained*** optimisation exists, but will not be covered. We will also limit ourselves to real valued, scalar functions $f(\vec{x}) \in \mathbb{R}$ and real valued variables; $x \in \mathbb{R}$ and $\vec{x} \in \mathbb{R}^N$.

One setting where finding the minimum of a complicated non-linear function occurs is the optimisation of parameters in a model when fitting to data. Consider the observed data $d_i^{\rm obs}$ with errors σ_i in figure 7.1. The model we would like to fit is a linear one

$$d_i^{\rm model} = a + \alpha\, X_i\,. \tag{7.1}$$

The normal procedure is to find the minimum χ^2 with respect to the parameters a and α

$$\chi^2(a,\alpha) = \sum_{i=1}^{i=N^{\rm data}} \frac{\left(d_i^{\rm obs} - d_i^{\rm model}\right)^2}{\sigma_i^2}\,. \tag{7.2}$$

The assumption here is that the data is distributed as independent Gaussian random numbers and we are **maximising the likelihood**

$$L(a,\alpha) \propto e^{-\frac{1}{2}\chi^2(a,\alpha)}\,, \tag{7.3}$$

which is equivalent to finding the minimum in the χ^2. The result of the minimisation may look something like figure 7.2.

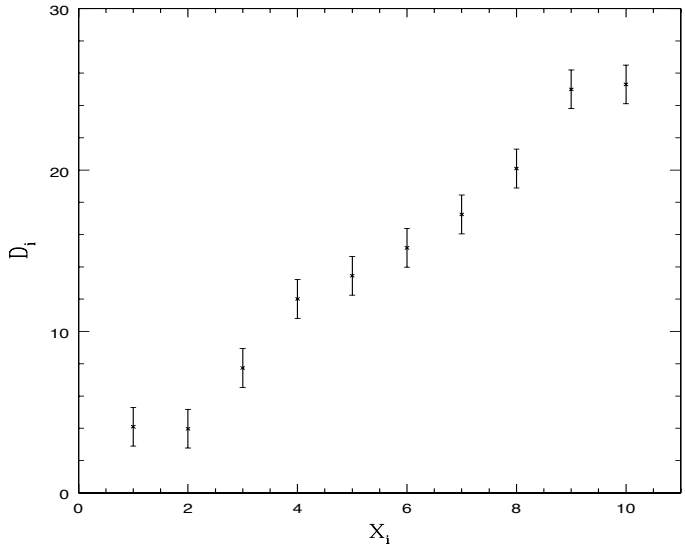

Figure 7.1. Some data with errors.

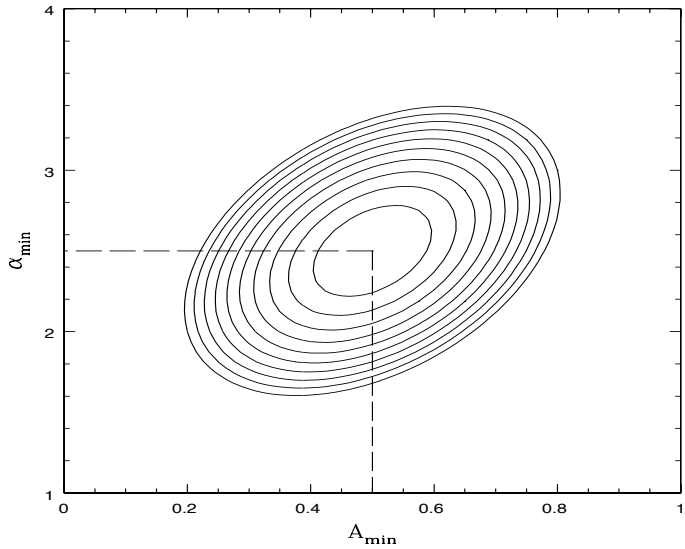

Figure 7.2. Contour plot of the χ^2 around the minimum. $a_{\min}$ and $\alpha_{\min}$ are the maximum likelihood values for the parameters in the model. The curvature around the minimum is related to the error in the parameters.

7.1. One–dimensional minimisation

Naively we might think that finding the minima of a function is trivial. All we need to do is differentiate the function and find the roots of the resulting equation. However there are many cases where the analytical method is not applicable, e.g., where the derivative in the function is discontinuous or the function has a regular (possibly infinite) set of minima. For a function of many, many variables, it can be too computationally costly to evaluate $\partial f/\partial x_i$ for all variables or simply too laborious to implement them all in code!

In practice this is often the case when evaluating a function of some scattered data where the analytical dependence of the likelihood with respect to the parameters is not known *a priori* and must be evaluated numerically. In this case we have to choose between a method that searches for minima/maxima using just the function values or (more optimal) ones that also use approximations for the derivatives of the function.

Parabolic Method

Functions almost always approximate a parabola near a minimum (except for some pathological cases). A very simple method to find a local minimum of a function exploits this. The method works when the curvature of $f(x)$ is positive everywhere in the interval we are looking at. The parabolic method consists of selecting three points along the function $f(x)$, e.g., x_0, x_1, and x_2 where

$$f(x_0) = y_0, \quad f(x_1) = y_1, \quad \text{and} \quad f(x_2) = y_2 \tag{7.4}$$

and then fitting $P_2(x)$, the 2nd-order Lagrange polynomial, through the points (see section 2.6). The location of the minimum of the interpolating parabola $P_2(x)$ is found using equation (7.6) below; this value is x_3. This procedure is illustrated in fig. 7.3. We then keep the three lowest points out of $f(x_0)$, $f(x_1)$, $f(x_2)$, and $f(x_3)$ and repeat the procedure. At each successive iteration the point x_3 will converge towards $x_\star$ where the minimum of the function $f(x)$ is located. The iterations can be stopped once the change in x_3 is less than a desired value.

The quadratic interpolating the 3 points is given by

$$P_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2 \ , \tag{7.5}$$

[c.f. equation (2.33)]. We want to find the minimum of $P_2(x)$, which will be a better estimate of the minimum's position (than the middle point of our trio). Differentiating $P_2(x)$ gives

$$\frac{dP_2}{dx} = \frac{[(x-x_1)+(x-x_2)]\,(x_2-x_1)}{d}y_0 + \frac{[(x-x_0)+(x-x_2)]\,(x_0-x_2)}{d}y_1 + \frac{[(x-x_0)+(x-x_1)]\,(x_1-x_0)}{d}y_2 \ ,$$

where $d = (x_1-x_0)(x_2-x_0)(x_2-x_1)$. Then setting $dP_2/dx = 0$ and solving for x gives (after some algebra) a new estimate of the minimum x_3

$$\boxed{x_3 = \frac{1}{2}\,\frac{(x_2^2-x_1^2)y_0 + (x_0^2-x_2^2)y_1 + (x_1^2-x_0^2)y_2}{(x_2-x_1)y_0 + (x_0-x_2)y_1 + (x_1-x_0)y_2}} \ . \tag{7.6}$$

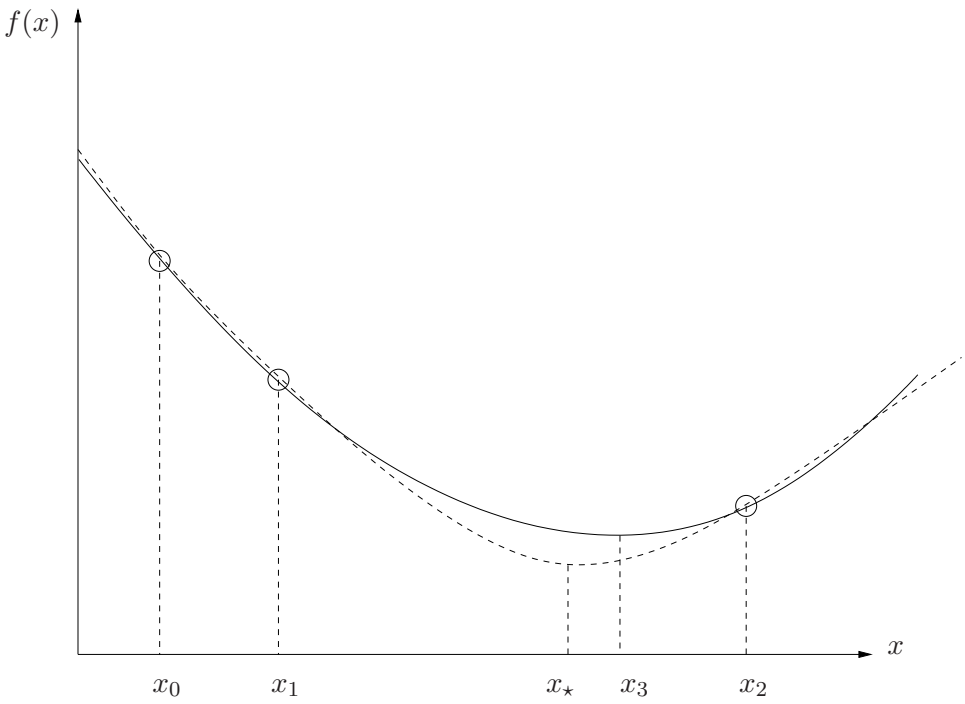

Figure 7.3. The parabolic minimum search. The function being minimised $f(x)$ is the dashed curve. A parabola (solid curve) is fit to the three points at x_0, x_1, x_3 and the minimum $x = x_3$ is found. Then the highest of the four points x_0, x_1, x_2, and x_3 is discarded and the method repeated. The minima of successive parabola approximations converges to the minimum of the function $(x_\star, f(x_\star))$.

This method works because any minimum can be approximated by a parabola and the approximation becomes more and more accurate as you get closer to the minimum.

For the case where the curvature is not positive throughout the initial interval we can first evaluate the function at two points inside the interval. We then keep the interval which includes the lowest point and then use the parabolic method to find the minimum within the new interval.

7.2. Multi-Dimensional methods

Univariate method

For the case where the function is multi-dimensional, i.e., $f(\vec{x})$ we can extend the one-dimensional parabolic method. This can be done by searching for the minimum in each direction successively and iterating. However this is a very inefficient method which is not useful in most cases. Univariate search is illustrated by the black arrows in figure 7.4. Contours are shown for a 2D function $f(x, y)$. The univariate search spirals into the minimum, converging slowly.

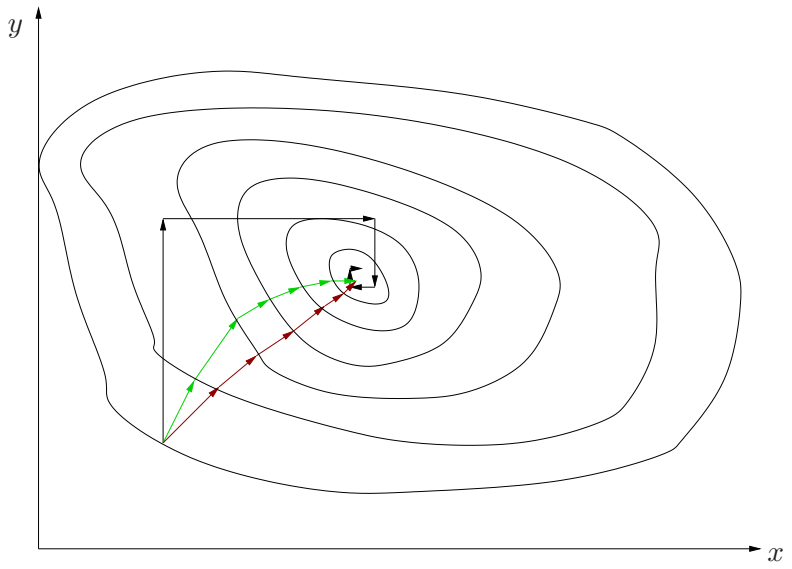

Figure 7.4. Multi-dimensional minimisation - for a function $f(x, y)$. Three methods are depicted: univariate method (black arrows), gradient method (green arrows) and Newton's method (red arrows). [Nb: this is not quite accurate; the green arrows are all supposed to be perpendicular to the contours!]

Gradient method

We can follow the steepest descent towards the minimum. This is achieved by calculating the local gradient and following its (negative) direction. The gradient of $f(\vec{x})$ at point $\vec{x}_n$ is given by the vector

$$\vec{d}(\vec{x}_n) = \vec{\nabla} f(\vec{x}_n) \qquad \text{where} \qquad \vec{\nabla} f = \begin{bmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \\ \vdots \\ \partial f/\partial x_N \end{bmatrix} \tag{7.7}$$

and is perpendicular to the local contour line. We can take a small step in the direction **opposite** to gradient i.e.

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - \alpha\, \vec{d}(\vec{x}_n) \ ,} \tag{7.8}$$

where $\alpha \ll 1$ and positive. If α is small enough then

$$f(\vec{x}_{n+1}) < f(\vec{x}_n) \ .$$

We can iterate this to find the minimum. (See green arrows in fig. 7.4.)

The gradient $\vec{\nabla} f$ can be found analytically or using a finite difference approximation; e.g. via a FDS applied in each variable x_i for $i = 1, \ldots, N$,

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, x_2, \ldots, x_i + \Delta, \ldots, x_N) - f(x_1, x_2, \ldots, x_i, \ldots, x_N)}{\Delta} \ . \tag{7.9}$$

Newton's method

A more efficient method to find the minimum is achieved by including the local curvature at each step. Consider starting at a location $\vec{x}_0$ which is displaced from the minimum by $\vec{\delta}$. The Taylor series for the function around the minimum is then

$$f(\vec{x}_0+\vec{\delta}) = f(\vec{x}_0) + [\vec{\nabla} f(\vec{x}_0)]^T \cdot \vec{\delta} + \frac{1}{2}\vec{\delta}^{\,T} \cdot \mathbf{H}(\vec{x}_0) \cdot \vec{\delta} + \mathcal{O}(|\vec{\delta}|^3) \ , \tag{7.10}$$

where **H** is the **Hessian** or **Curvature** matrix (an $N \times N$ matrix)

$$H_{ij}(\vec{x}) = \frac{\partial^2 f(\vec{x})}{\partial x_i \partial x_j} \ . \tag{7.11}$$

(c.f. equation (2.4) in section 2.1.) The notation $\mathbf{H}(\vec{x}_0)$ means (7.11) evaluated at $\vec{x} = \vec{x}_0$. An extremum of the function is obtained when $\vec{\nabla} f(\vec{x}_0+\vec{\delta}) = \vec{0}$. To be a minimum (rather than a maximum) the Hessian needs to be **positive definite**, i.e.,

$$\vec{\delta}^{\,T} \cdot \mathbf{H} \cdot \vec{\delta} > 0 \qquad (\text{for } \textit{all } \vec{\delta} \neq \vec{0}\ ) \ . \tag{7.12}$$

How to determine $\vec{\delta}$? Using the Taylor expansion (7.10) to 2nd-order and taking its gradient gives

$$\begin{aligned} \vec{\nabla} f(\vec{x}_0+\vec{\delta}) &= \vec{\nabla} f(\vec{x}_0) + \vec{\nabla}\left(\vec{\nabla} f(\vec{x}_0)\cdot\vec{\delta}\right) = \vec{0} \ , \\ &= \vec{\nabla} f(\vec{x}_0) + \mathbf{H}(\vec{x}_0)\cdot\vec{\delta} = \vec{0} \ , \end{aligned}$$

thus we can solve the following matrix-equation for $\vec{\delta}$,

$$\mathbf{H}(\vec{x}_0)\cdot\vec{\delta} = -\vec{\nabla} f(\vec{x}_0) \ . \tag{7.13}$$

If $f(\vec{x})$ is exactly 'parabolic', i.e.,

$$f(\vec{x}) = \vec{x}^{\,T} \cdot \mathbf{A} \cdot \vec{x} + \vec{b}^{\,T} \cdot \vec{x} + c \ , \tag{7.14}$$

then there are no terms $\mathcal{O}(|\vec{\delta}|^3)$ in the Taylor expansion, and $\vec{x}_1 = \vec{x}_0 + \vec{\delta}$ is the exact position of the minimum.

If the function is not well approximated by a quadratic then the vector $\vec{\delta}$ will not take us directly to the minimum so we iterate this until we get arbitrarily close to it, i.e., $\vec{x}_{n+1} = \vec{x}_n + \vec{\delta}$ which can be written as

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - [\mathbf{H}(\vec{x}_n)]^{-1} \cdot \vec{\nabla} f(\vec{x}_n) \ .} \tag{7.15}$$

This is illustrated by the red arrows in fig. 7.4. This method can be very efficient with **quadratic convergence**, $|\vec{\delta}|_{n+1} \sim |\vec{\delta}|_n^2$, however in problems with large dimensions the calculation of the inverse Hessian can become very slow. In addition calculating the Hessian using finite differences often yields a matrix which is not strictly positive definite.

Aside – What we have done in obtaining equation (7.13) is completely analogous to what would be done for finding the extremum of a simple quadratic $f(x) = ax^2 + bx + c$ if we know its gradient and curvature at a point x_0. The turning point $f' = 2ax + b = 0$ occurs at $x = -b/2a \equiv x_\star$. Also $f'' = 2a$. The exact Taylor expansion is, $f(x_0 + h) =$

$f(x_0) + hf'|_{x_0} + \frac{1}{2}h^2 f''|_{x_0} = (ax_0^2 + bx_0 + c) + h(2ax_0 + b) + \frac{1}{2}h^2(2a)$. The gradient of the Taylor expansion is

$$f'|_{x_0+h} = f'|_{x_0} + h\, f''|_{x_0} = (2x_0 + b) + h(2a) + h^2(0) \ .$$

From this we get the value of h to take us from x_0 to the turning point ($f'|_{x_0+h} = 0$);

$$h = -f'|_{x_0}/f''|_{x_0} = -x_0 - b/2a \ .$$

This can be re-expressed as $h = x_\star - x_0$, demonstrating that getting h from the gradient of the Taylor expansion does indeed work for a parabola. For the multi-dimensional case equation (7.13) corresponds to the expression for h above.

Quasi-Newton Method

A disadvantage of Newton's method is that we need to calculate both first and second derivative of the multi-dimensional function, and the inverse of the curvature matrix which takes $\mathcal{O}(N^3)$.

The Quasi-Newton method gets around this by approximating the inverse Hessian using the local gradient. The basic iteration step is

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - \alpha\, \mathbf{G}_n \cdot \vec{\nabla} f(\vec{x}_n) \ ,} \tag{7.16}$$

where $\mathbf{G}_n$ is an approximation of $\mathbf{H}^{-1}(\vec{x}_n)$ and $\alpha \ll 1$.

For the first iteration $\mathbf{G}_0$ is set to the identity matrix $\mathbf{I}$, which makes this iteration the same as a gradient search. To update $\mathbf{G}_n \to \mathbf{G}_{n+1}$ we use the updates in $\vec{x}$ and $\vec{\nabla} f$;

$$\vec{\delta}_n = \vec{x}_{n+1} - \vec{x}_n \qquad , \qquad \vec{\gamma}_n = \vec{\nabla} f(\vec{x}_{n+1}) - \vec{\nabla} f(\vec{x}_n) \ .$$

Then comparing the above expression for the gradient update $\vec{\gamma}_n$ with the gradient of the Taylor expansion of $f(\vec{x})$, to linear order,

$$\vec{\nabla} f(\vec{x}_{n+1}) \approx \vec{\nabla} f(\vec{x}_n) + \vec{\nabla}\left(\vec{\nabla} f(\vec{x}_n) \cdot \vec{\delta}_n\right)$$

we see that to linear order

$$\mathbf{H}_n^{-1} \cdot \vec{\gamma}_n = \vec{\delta}_n \ .$$

(This is equivalent to equation (7.13) when $\nabla f(\vec{x}_0 + \vec{\delta}) \neq 0$.) The trick is then to update $\mathbf{G}$ to satisfy

$$\mathbf{G}_n \cdot \vec{\gamma}_n = \vec{\delta}_n \, . \tag{7.17}$$

so that $\mathbf{G}_n$ mimics the inverse Hessian. A number of methods exist for this update, each with different efficiency and convergence characteristics. One of the most common is the DFP (Davidon–Fletcher–Power) algorithm where the update is

$$\mathbf{G}_{n+1} = \mathbf{G}_n + \frac{(\vec{\delta}_n \otimes \vec{\delta}_n)}{\vec{\gamma}_n \cdot \vec{\delta}_n} - \frac{\mathbf{G}_n \cdot (\vec{\delta}_n \otimes \vec{\delta}_n) \cdot \mathbf{G}_n}{\vec{\gamma}_n \cdot \mathbf{G}_n \cdot \vec{\gamma}_n} \, , \tag{7.18}$$

where $(\vec{u} \otimes \vec{v})_{ij} \equiv u_i v_j$ is the outer product of $\vec{u}$ and $\vec{v}$.

The advantage of this scheme is that it only involves multiplications, i.e., $\mathcal{O}(N^k)$ operations (where $2.37 \leq k < 3$ for optimised multiplication algorithms) at each iteration and the resulting (approximated) Hessian is positive definite by construction.

Global minimum search

These methods do not allow for the fact that the function may have multiple minima (local minima) in the interval we are interested in. There is no fool-proof method to find the global minimum (lowest minimum) of a function and we often need to restart algorithms from different starting points to check we have not found a local minimum.

Minimisation & solving matrix equations (non-examinable)

The value of $\vec{x}$ that minimises the ***quadratic form***

$$f(\vec{x}) = \frac{1}{2}\vec{x}^T \cdot \mathbf{A} \cdot \vec{x} - \vec{b}^T \cdot \vec{x} \ , \tag{7.19}$$

happens to be the solution to

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

for a symmetric, positive-definite matrix $\mathbf{A}$. This is because the gradient of this quadratic form is $\vec{\nabla} f(\vec{x}) = \mathbf{A} \cdot \vec{x} - \vec{b}$. At the minimum $\vec{x} = \vec{x}_\star$, $\vec{\nabla} f(\vec{x_\star}) = 0$ hence $\mathbf{A} \cdot \vec{x}_\star = \vec{b}$. (Note the similarity of this quadratic form with equation (7.19) seen earlier.)

So to solve the matrix equation, one of the multi-dimensional iterative methods seen in section 7.2 can be used with the quadratic form above. The best iterative methods, such as "conjugate gradient" and "bi-conjugate gradient" will (in the absence of round-off error) get the exact solution within N iterations. This is better than Jacobi, G-S & SOR, where $N_{iter} \sim \mathcal{O}(N^2)$ for some problems. With large sparse matrices, the cost of computing $f(\vec{x})$ is just $\mathcal{O}(N)$. One step of the iteration is also $\mathcal{O}(N)$. Thus one can solve matrix equations in $\mathcal{O}(N^2)$ or better. Convergence can be further accelerated by using a technique called ***pre-conditioning***. This effectively pushes the contours of $f(\vec{x})$ towards being spherical (in N dimensions), rather than very elongated ellipsoids (i.e. imagine fig. 7.2, but more elongated), so that a descent along the local gradient gets close to the global minimum.

To show that $\vec{\nabla} f = \mathbf{A} \cdot \vec{x} - \vec{b}$, consider the k-th component of this gradient;

$$\begin{aligned}
(\vec{\nabla} f)_k &= \frac{\partial}{\partial x_k}\left[\frac{1}{2}\sum_i x_i \left(\sum_j A_{ij}x_j\right) - \sum_i b_i x_i\right] \\
&= \frac{1}{2}\sum_i \left[\delta_{ki}\left(\sum_j A_{ij}x_j\right) + x_i\left(\sum_j A_{ij}\delta_{kj}\right)\right] - b_k \\
&= \frac{1}{2}\left[\left(\sum_j A_{kj}x_j\right) + x_k A_{ik}\right] - b_k \\
&= \frac{1}{2}\left[\mathbf{A} \cdot \vec{x}\right]_k + \frac{1}{2}\left[\mathbf{A}^T \cdot \vec{x}\right]_k - b_k \ = \ \left[\mathbf{A} \cdot \vec{x}\right]_k - b_k
\end{aligned}$$

where the last line makes use of the fact that $\mathbf{A}$ is symmetric.

SECTION 8
Random Numbers

Outline of Section

- Pseudo-Random Numbers
- Transformation Method
- Rejection Method

8.1. Random Numbers

There are countless uses for random numbers in computational physics. Generally we need random numbers when we are simulating stochastic systems, e.g., Brownian motion, thermodynamical systems, state realisations and when we use Monte Carlo methods to calculate the variability of stochastic systems (more on this later).

A **uniform deviate** is a random number lying within a range where any number has the same probability as any other. The range is usually $0 \leq x \leq 1$. Here x is a *random variable* and the deviates are the values that x can randomly take.

Most `C` implementations come with a uniform random number generator that returns **pseudo-random numbers**. This means that the same sequence of (seemingly) random numbers can be regenerated by using the same **seed** number to start the sequence. This is a useful feature for debugging codes and reproducing results.

A simple implementation of a uniform deviate generator is the **linear congruential generator**

$$\boxed{I_{n+1} = (a\, I_n + c)\%m \ ,} \tag{8.1}$$

where I_n are the random numbers, a is a multiplier, c is the increment, and m is the modulus. ($a\%b$ denotes modulo division; i.e. the remainder of dividing a by b.) The parameters a, c and m are all integers. This will generate pseudo-random **integers** in the (closed) interval between 0 and $m-1$ (usually stored as `RAND_MAX` in `C` implementations).

We can turn the results into a real number by dividing by m

$$x_n = \frac{I_n}{m} = \frac{I_n}{\texttt{RAND\_MAX} + 1} \qquad \text{where} \qquad 0 \leq x_n < 1 \ . \tag{8.2}$$

One problem with this method is that the sequence of numbers will repeat itself with periodicity which is at most m, and for unfortunate choices of a, c & m the sequence length is much less than m (with a significant portion of integers in the range $0 \leq < I < m$ being skipped). So generally we want a large m and a choice of a and c that ensures that the period is equal to m. You should be wary of the `rand()` function supplied with a compiler, especially `C` and `C++` compilers. The ANSI C standard specifies that `rand()` return type `int`, which can be as small as 2-bytes on some systems (i.e. `RAND_MAX`=32767). This is is not a very long period for Monte Carlo applications! These typically require millions of random numbers.

Another problem with linear congruential generators is that there is correlation between successive numbers; if k successive random numbers are used to plot points in k-dimensional space, then the points do not fill up the space, but lie on distinct planes (of dimension $k-1$).

It is best to use random number generators other than the standard `C` one. There are good ones in the GSL libraries. Of note is the "Mersenne Twister" generator with an exceptionally long period of $m = 2^{19937} - 1 \sim 10^{6000}$, and the "RANLUX" generators which provides the most reliable source of uncorrelated numbers.

8.2. Non-uniform distributions - Transformation Method

Often we need random deviates with a distribution other than uniform. The most common requirement is for a Gaussian distribution which is ubiquitous due to the Central Limit Theorem.

Given a generator that returns a uniform deviate x in the range 0 to 1, we can use the **transformation method** to obtain a new deviate (random variable) y which is distributed according to a *probability density function* (PDF) $P(y)$. Recall the basic properties of a PDF;

- A PDF must be positive, i.e., $P(y) \geq 0$.
- A PDF must be normalised, i.e., $\int_{-\infty}^{\infty} P(y)\, dy = 1$.

We are given x with PDF

$$U(x)\, dx = \begin{cases} dx & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases} \quad . \tag{8.3}$$

which is also normalised; $\int_{-\infty}^{\infty} U(x)\, dx = 1$. We want our new random variable y to be a function of x, i.e., $y(x)$. The fundamental transformation law of probabilities then relates the PDFs of each variable via

$$|P(y)\, dy| = |P(x)\, dx| \ , \tag{8.4}$$

where $P(x) = U(x)$ in our case. This satisfies the requirement that both PDFs integrate up to unity. Assuming $dy/dx \geq 0$ (so that the $|\ldots|$ can be discarded) we have

$$\frac{dx}{dy} = P(y) \ ,$$

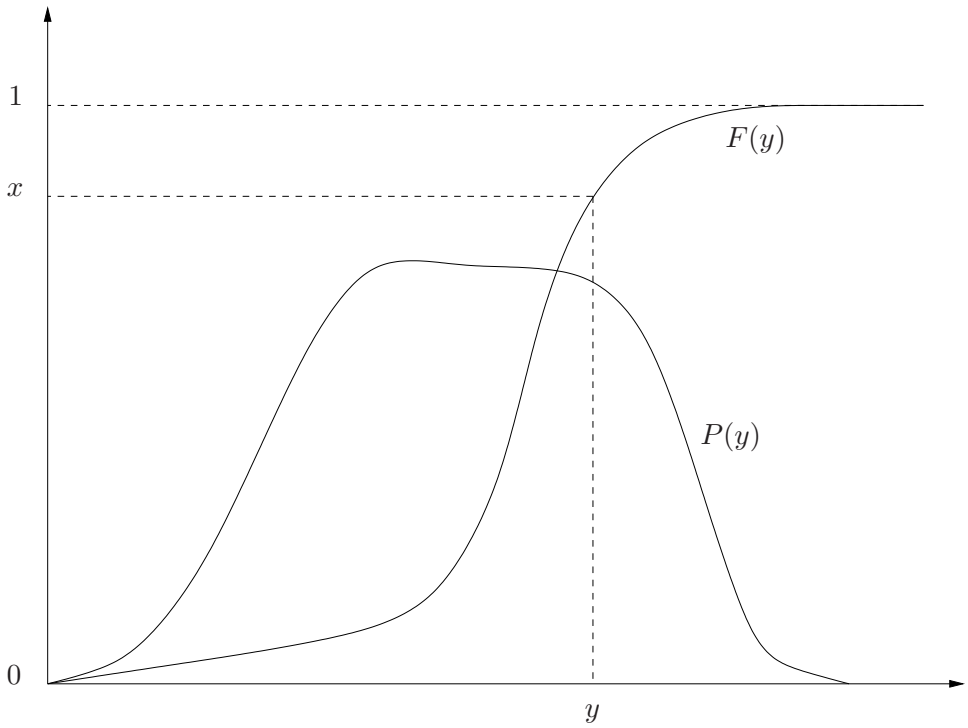

Figure 8.1. The transformation method. The Cumulative Distribution function $F(y) = \int_{-\infty}^{y} P(\tilde{y})d\tilde{y}$ is inverted to find a deviate y for every uniform random deviate x.

since $P(x) = U(x) = 1$. We can then integrate up and define the function

$$F(y) = \int_{-\infty}^{y} P(\tilde{y})\, d\tilde{y} = \int_{-\infty}^{y} \frac{d\tilde{x}}{d\tilde{y}}\, d\tilde{y} = \int_{0}^{x} d\tilde{x} = x \quad , \tag{8.5}$$

So if $F(y)$ is an invertible function, i.e.,

$$y = F^{-1}(x) \quad , \tag{8.6}$$

we can map each x given by a uniform deviate generator into a new variable y which will have the desired PDF $P(y)$. Note that $F(y)$ is just the **Cumulative Distribution Function** (CDF) of y. Since the PDF is normalised, the range of $F(y)$ is $0 \leq F(y) \leq 1$. The transformation method is illustrated in fig. 8.1.

Example consider the *triangular* distribution $P(y) = 2y$ with $0 < y < 1$. The CDF is $F(y) = y^2 = x$ such that the transformation $y = \sqrt{x}$ gives deviates with the PDF $P(y)$.

8.3. Non-uniform distributions - Rejection Method

If the CDF is not a computable or invertible function then we can use the **rejection method** to obtain deviates with the required distribution.

We first draw a new function $f(y)$ called the ***comparison function*** which lies above $P(y)$ for all y. For simplicity let's define

$$f(y) = C \ , \tag{8.7}$$

where $C > P(y)$ everywhere. The procedure is then as follows:

(a) Pick a random number y_i, uniformly distributed in the range between $y_{\min}$ and $y_{\max}$.
(b) Pick a second random number p_i, uniformly distributed in the range between 0 and C.
(c) If $P(y_i) < p_i$ then reject y_i and go back to step (a). Otherwise accept.

Accepted y_i will have the desired distribution $P(y)$.

To prove that the accepted y_i follows $P(y)$, consider the probability of creating an acceptable y_i in the range between y and $y + dy$; this is

$$\text{Prob} = \frac{dy}{y_{\max} - y_{\min}} \times \frac{P(y)}{C} = \text{const} \times P(y)\, dy \, . \tag{8.8}$$

The efficiency of the method is obtained by integrating the probability of acceptance which, since $P(y)$ is normalised, gives

$$\text{Eff} = \frac{1}{C} \frac{1}{y_{\max} - y_{\max}} \ . \tag{8.9}$$

1/Eff is the average number of times the steps (a)–(c) have to be carried out to get a y_i value.

The efficiency can be improved by using a comparison function $f(y)$ that conforms more closely to the desired $P(y)$ (but still lies above it) and is suitable for use in the transformation method (i.e. it has invertible definite integral $\int_{y_{min}}^{y} f(\tilde{y}) d\tilde{y}$). Step (a) then becomes; Using the transformation method, pick a random deviate y_i in the range between $y_{\min}$ and $y_{\max}$, distributed according to the PDF obtained by normalising $f(y)$. Since $\int_{y_{min}}^{y_{max}} P(y) dy = 1$ and $f(y) \geq P(y)$ then the area under $f(y)$ is bigger than unity, so $f(y)$ needs to be normalised to be a PDF.

SECTION 9
Monte Carlo Methods

Outline of Section

- MC Integration
- MC Minimisation

Monte Carlo Methods are methods where random numbers are used to statistically solve deterministic problems. They replace exact methods in the case where analytical solutions do not exist or take too long to calculate. Generally they are useful for systems of many variables or dimensions.

9.1. Monte Carlo Integration

Consider calculating an integral of a d-dimensional function $f(\vec{x})$ over some volume V defined by boundaries a_i and b_i

$$I = \int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 \ldots f(\vec{x}) = \int_V f(\vec{x})\, d\vec{x} \ . \tag{9.1}$$

The integral I is related to the mean value of the function in the volume. This can be written as

$$\langle f \rangle = \frac{1}{V} \int_V f(\vec{x})\, d\vec{x} \ .$$

Then I can be obtained from the mean of the function as

$$I \equiv V \langle f \rangle \ . \tag{9.2}$$

This means we can *estimate* I by taking N random samples of the function in the volume

$$\hat{I} = \frac{V}{N} \sum_{i=1}^{N} f(\vec{x}_i) \ . \tag{9.3}$$

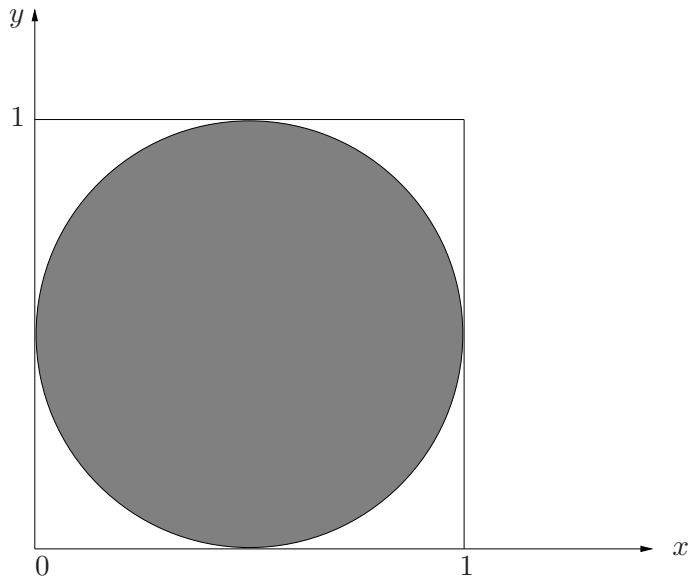

Figure 9.1. Circle of radius 1/2. The function $f(x, y) = 1$ inside the circle and $f(x, y) = 0$ outside

We can also derive an estimate of the error in $\hat{I}$ by considering the estimate of the parent variance of the samples $f(\vec{x}_i)$, i.e.,

$$\sigma_{f_i}^2 = \frac{1}{N-1} \sum_{i=1}^{i=N} \left(f(\vec{x}_i) - \langle f \rangle \right)^2 \quad . \tag{9.4}$$

Given this, the variance of the mean $\langle f \rangle$ is $\sigma_{\langle f \rangle}^2 = \sigma_{f_i}^2 / N$ and given (9.2), we can write the variance in the estimate of I as

$$\sigma_{\hat{I}}^2 = V^2 \sigma_{\langle f \rangle}^2 = \frac{V^2}{N} \sigma_{f_i}^2 \quad . \tag{9.5}$$

Therefore we can state the estimate of I as

$$\boxed{\hat{I} = \frac{V}{N} \sum_{i=1}^{N} f(\vec{x}_i) \pm \frac{V}{\sqrt{N}} \sigma_{f_i} \quad .} \tag{9.6}$$

Example – Consider the integral of the function shown in Fig. 9.1, a uniform circle of radius 1/2;

$$f(x, y) = \begin{cases} 1 & \text{if } (x - \frac{1}{2})^2 + (y - \frac{1}{2})^2 < (\frac{1}{2})^2 \\ 0 & \text{otherwise} \end{cases} \quad . \tag{9.7}$$

In this example, we have the luxury of knowing that

$$I = \int f(x, y)\, dx\, dy = \text{circle area} = \pi r^2 = \frac{1}{4} \pi \quad . \tag{9.8}$$

If we sample N random, uniformly distributed points in the range $0 \leq x < 1$ and $0 \leq y < 1$ then and assign a *success* if $f(x_i, y_i) = 1$ and a *fail* if $f(x_i, y_i) = 0$ then the ***probability of success***, p, for any given sample is the fraction of the square (area $A = \int_0^1 \int_0^1 dx\, dy = 1$) covered by the circle

$$p \equiv \frac{I}{A} \ . \tag{9.9}$$

The successes should follow a binomial distribution. We can define an estimator for p as

$$\hat{p} = \frac{N_1}{N} \ ,$$

where N_1 is the number of successes. We know that for a binomial distribution the variance of N_1 is

$$\sigma^2_{N_1} = N\, p\, (1-p) \ ,$$

such that

$$\sigma^2_{\hat{p}} = \frac{\sigma^2_{N_1}}{N^2} = \frac{p\,(1-p)}{N} \ .$$

We can then have our estimate for the value of the integral I

$$\boxed{\hat{I} = \hat{p}\, A = \hat{p} = \frac{N_1}{N} \pm \sqrt{\frac{p\,(1-p)}{N}} \ .} \tag{9.10}$$

Again we see that the error on the integral scales as the square root of the number of sample.

Comparison with the trapezium method – The trapezium method for calculating the integral of a function is equivalent to a linear expansion (interpolation) of the function in each interval $x_i \to x_i + h$. Being a linear expansion, we know that the truncation error goes as $\mathcal{O}(h^2)$. In general for d-dimensions we will evaluate the function at $N \sim h^{-d}$ points to calculate the integral. (Exactly $N = h^{-d}$ if the integration range in each dimension has unit length.) Therefore

$$h \sim N^{-1/d} \ , \tag{9.11}$$

such that the error in the integral is

$$\epsilon = \mathcal{O}(h^2) = \mathcal{O}(N^{-2/d}) \ . \tag{9.12}$$

Thus for 4 or more dimensions the Monte Carlo method is as accurate or more accurate than the trapezium method for the same number of samples.

Error scalings $\longrightarrow$ (9.13)

d	Trapezium	MC
1	N^{-2}	$N^{-1/2}$
2	N^{-1}	$N^{-1/2}$
3	$N^{-2/3}$	$N^{-1/2}$
4	$N^{-1/2}$	$N^{-1/2}$
5	$N^{-2/5}$	$N^{-1/2}$
6	$N^{-1/3}$	$N^{-1/2}$

9.2. Minimisation - Simulated Annealing

Another application of the Monte Carlo approach is to the problem of minimisation (optimisation) of functions that we looked at in section 7. The simulated annealing and related methods are particularly useful in cases where;

- The degrees of freedom in the system are so many that a direct search for an optimal configuration is too time consuming.
- Many quasi-optimal solutions (local minima) exist in which direct search methods can get stuck instead of finding the optimal solution (global minimum).

The basic idea is to introduce some randomness in the search for a minimum in the function being analysed. The analogy is with thermodynamics where a system can temporarily move to a less optimal configuration due to fluctuations. The analogy is quantified in the use of the **Boltzmann Probability Distribution**

$$P(E)\,dE \sim \exp\left(-\frac{E}{kT}\right)\,dE \quad , \tag{9.14}$$

where 'energy' E encodes a **cost function** and 'temperature' T is a variable which determines the probability of changes in the energy of the system. In particular, even for low temperatures, it allows for the system to move to higher energies with very low probability. This is what can allow the system to get 'unstuck' from local minima and continue the search for a global minimum.

In practice the method involves the use of a **Metropolis Algorithm** where at each step in the iteration the configuration of the system is changed randomly. The energy of the system before (E_1) and after (E_2) the change are calculated to get the change in the system's energy $\Delta E = E_2 - E_1$. The step is **accepted** with a probability p_{acc} given by the simple algorithm

$$\boxed{p_{acc} = \begin{cases} 1 & \text{if} \quad \Delta E \leq 0 \\ \exp(-\Delta E/kT) & \text{if} \quad \Delta E > 0 \end{cases}} \tag{9.15}$$

In functional minimisation the energy is simply the value of the function $E = f(\vec{x})$ and the temperature T is slowly lowered to 0, hence the analogy with annealing of metals.

SECTION 10
Partial Differential Equations

Outline of Section

- Classification of PDEs
- Elliptic
- Hyperbolic
- Parabolic

10.1. Classification of PDEs

For partial differential equations (PDEs) we have partial derivatives with respect to more than one variable in each equation in the system. This is the most common form of equation encountered when, e.g., we are modelling the spatial (x) and dynamical (t) characteristics of a system or a static system in more than one dimension, e.g., (x, y). We will consider the two-dimensional case as an example in this section of the course.

A general PDE can be classified through the coefficients multiplying the various derivatives. The general form for a PDE involving two independent variables x and y with solution $u(x, y)$ takes the form

$$A\,\frac{\partial^2 u}{\partial x^2} + B\,\frac{\partial^2 u}{\partial x \partial y} + C\,\frac{\partial^2 u}{\partial y^2} + f(u, x, y, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}) = G(x, y) \quad , \tag{10.1}$$

where $f(\ldots)$ is an arbitrary function involving anything other than 2nd derivatives (but involving u somehow). $G(x, y)$ does not depend on u and makes (10.1) into an inhomogeneous PDE.

Note that the coefficients of the 2nd derivatives A, B and D, can depend on x, y, u, $\partial u/partialx$ and/or $\partial u/partialy$. But (10.1) must be linear in 2nd derivatives. In analogy with the conic sections (in geometry)

$$A\,x^2 + B\,xy + C\,y^2 + D\,x + E\,y + F = 0 \quad ,$$

(where A, $\ldots$, F are constants) we define the **discriminant**

$$Q = B^2 - 4\,A\,C \quad , \tag{10.2}$$

and mathematically classify PDEs as

$$\boxed{\begin{array}{ll} Q<0 & : \text{Elliptic,} \\ Q=0 & : \text{Parabolic,} \\ Q>0 & : \text{Hyperbolic.} \end{array}} \tag{10.3}$$

For conic sections, $Q < 0$ yields an elliptical curve, $Q = 0$ a parabola and $Q > 0$ a hyperbola.

The classification can be done for 3 or more independent variables (e.g. $u(t, x, y, z)$) but is outside the scope of this course. For common PDEs occurring in physics, the classification above carries over into more spatial dimensions, in the obvious way, as will be discussed below. The presence or absence of the inhomogeneous term $G(x, y)$ does not affect the nature of the PDE.

In terms of a more physical picture, basic example of the three types are

(a) **Poisson Equation (Elliptic)**:

$$\nabla^2 u \equiv \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x, y) \ , \tag{10.4}$$

i.e. how a solution (potential) reacts to a source (charge density). If $\rho(x, y) = 0$ then the Poisson equation becomes the **Laplace Equation**. Both the homogeneous (Laplace) and inhomogeneous (Poisson) equations are 'elliptic'. We reference to the general equation (10.1), for Poisson's equation $A = 1$, $B = 0$, $C = 1$ hence $Q = -1 < 0$. The 3D version of the Poisson & Laplace equations, where

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \ ,$$

are still elliptic PDEs.

(b) **Diffusion Equation (Parabolic)**:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D \frac{\partial u}{\partial x} \right) \ , \tag{10.5}$$

where D is the diffusion coefficient. Here, $A = 1$, $B = C = 0$ hence $Q = 0$. Again, moving to 2 or 3 spatial dimensions we have

$$\frac{\partial u}{\partial t} = \nabla \cdot (D \nabla u)$$

which is still a parabolic PDE. The diffusion coefficient can even be a function of u, $D(u)$, such that

$$\frac{\partial u}{\partial t} = D(u)\nabla^2 u + \frac{\partial D(u)}{\partial u} |\nabla u|^2$$

and the PDE is non-linear. This is still a parabolic PDE though.

(c) **Wave Equation (Hyperbolic)**:

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \ , \tag{10.6}$$

where v is the (constant) speed of a wave. Here, $A = v^2$, $B = 0$ and $C = -1$ hence $Q = 4v^2 > 0$. The 3D wave equation

$$\frac{\partial^2 u}{\partial t^2} = v^2 \nabla^2 u \ ,$$

is still a hyperbolic PDE. The wave speed v can depend on position and even on the wave displacement u and the PDE is still hyperbolic.

Types of physical problems – these are governed a combination of the PDE and boundary conditions (BCs) and can be classified into the following.

- **Boundary value problems** are relevant to **elliptic** PDEs, as in the case of the Poisson equation where we are solving for the static solution $u(x, y)$ requiring constraints on the boundary. The BCs are specified on a **closed surface**. These problems effectively involve *integrating in from the boundaries.* and are solved numerically by **relaxation methods**.
- **Initial value problems** are relevant to **hyperbolic** and **parabolic** PDEs, e.g., as in the diffusion and wave cases where the dynamical solution $u(t, x)$ is sought. In this case we require constraints on the initial state of the solution for all x and we solve this problem by *integrating forward in time* using a **marching methods**. We may also impose conditions on the spatial boundaries in this case too.

Types of boundary conditions –

u defined on boundaries	: Dirichlet conditions,
$\vec{\nabla} u$ defined on boundaries	: Neumann conditions,
both u and $\vec{\nabla} u$ defined on boundaries	: Cauchy conditions.
u or $\vec{\nabla} u$ applied on diff. parts of boundary	: mixed conditions.
$u(x_r) = u(x_l + L) = u(x_l)$	: Periodic BC (e.g. in x).
$u(x_l - x) = u(x - x_l)$	: Reflective BC (e.g. about x_l).

where, e.g., x_l and x_r denote the left and right edges of domain in the x-direction. Since the scope of this section is PDEs with 2 independent variables (i.e. x, y or t, x), for 'boundary surface' we really mean a boundary curve. For elliptic problems, this closed curve could be arbitrarily shaped (e.g. outline of a potato). However we will limit ourselves to a rectangular shaped boundary, and to Cartesian coordinates. For hyperbolic/parabolic initial-value problems, Cauchy conditions are applied only at $t = t_0$. $t \to \infty$ is known as an **open boundary** with nothing being imposed/specified there. For $\vec{\nabla} u$ we mean $\partial u / \partial \zeta$ where $\zeta \to x$, y, z as appropriate. We will not consider subtleties in defining boundaries and applying BCs when there are 3 (or more) independent variables. A reflective BC is equivalent to having, e.g., $\partial u / \partial x = 0$ at a boundary, so is a special case of a Neumann condition.

We have already seen Dirichlet conditions in section 6.1 and mixed conditions in 6.3 when solving boundary value problems for ODEs. We have effectively seen Cauchy conditions in section 2.3, and in Project A, when solving equations of motion (2nd-order ODEs), where the initial position & velocity are specified.

10.2. Elliptic Equations

To solve an elliptical equation we take a similar approach to the finite difference method of section 6.2 by discretising the system onto a 2D lattice of points and using finite difference approximations to the partial derivatives. The lattice is also known as a **mesh** or a **grid**. As an example we start with Laplace's equation

$$\nabla^2 u(x,y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \ , \tag{10.7}$$

with Dirichlet boundary conditions, i.e., u specified on the boundary. We will get a matrix equation $\mathbf{A}\cdot\vec{u} = \vec{b}$. This can be solved for the values of u on the 2D grid (packaged into a vector $\vec{u}$) using a **relaxation method**; an iterative matrix solver such as Jacobi, Gauss-Seidel or SOR. The vector $\vec{b}$ holds the boundary conditions.

We consider a rectangular domain extending $0 \le x \le L_x$ and $0 \le y \le L_y$. There are $nx+2$ regularly spaced points in the x-direction including the edges and similarly $ny+2$ in the y-direction. To keep things simple the grid spacing will be taken as h in both directions. (The numerical equations below can readily be generalised for different spacings, e.g., $\Delta x \neq \Delta y$. It does not change the approach.) The location of the grid points are defined as

$$x_i = i\,h \qquad (i = 0,\,1,\,\ldots\,,nx+1) \qquad L_x = (nx+1)\,h \tag{10.8}$$

$$y_j = j\,h \qquad (j = 0,\,1,\,\ldots\,,ny+1) \qquad L_y = (ny+1)\,h \ . \tag{10.9}$$

Thus $i=0$ corresponds to the left-boundary, $i = nx+1$ the right-boundary. Similarly $j=0$ and $j = ny+1$ correspond to the bottom and top boundaries, respectively. There are $m = nx \times ny$ 'interior' points.

Setting up the FD matrix equation – To illustrate the method we take $nx = 3$ and $ny = 3$ and discretise the system using the following scheme;

$$\begin{array}{ccccc}
C_{0,4} & C_{1,4} & C_{2,4} & C_{3,4} & C_{4,4} \\
* & * & * & * & * \\
C_{0,3} & u_{1,3} & u_{2,3} & u_{3,3} & C_{4,3} \\
* & \circ & \circ & \circ & * \\
C_{0,2} & u_{1,2} & u_{2,2} & u_{3,2} & C_{4,2} \\
* & \circ & \circ & \circ & * \\
C_{0,1} & u_{1,1} & u_{2,1} & u_{3,1} & C_{4,1} \\
* & \circ & \circ & \circ & * \\
C_{0,0} & C_{1,0} & C_{2,0} & C_{3,0} & C_{4,0} \\
* & * & * & * & *
\end{array} \tag{10.10}$$

where

$$u_{i,j} = u(x_i, y_j) \ . \tag{10.11}$$

We need to solve for the nine unknown internal points and the boundary conditions are set as $u_{0,0} = C_{0,0}$, $u_{0,1} = C_{0,1}$, etc. For the second order partial derivatives in the Laplacian operator we apply the finite difference approximation seen in section 2.2, i.e.,

equation (2.17), which yields

$$\frac{\partial^2 u_{i,j}}{\partial x^2} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \mathcal{O}(h^2) \quad , \tag{10.12}$$

and

$$\frac{\partial^2 u_{i,j}}{\partial y^2} = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} + \mathcal{O}(h^2) \quad , \tag{10.13}$$

giving the Laplacian

$$\boxed{\nabla^2 u_{i,j} = \frac{u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1} - 4u_{i,j}}{h^2} + \mathcal{O}(h^2) \quad .} \tag{10.14}$$

(The $u_{i,j}$ in (10.14) are FD approximations to the true solution, but we dispense with the $\tilde{u}$ notation here.) The Laplacian can be represented as a **pictorial operator** acting on each unknown lattice point

$$\nabla^2 \tilde{u}_{i,j} \approx \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} \tilde{u}_{i,j} \quad . \tag{10.15}$$

This visually shows how the 4 points points to the left, right, above and below the centre point (where the Laplacian is being determined) are involved and their weights (+1). The weight of the centre point is -4. Equation (10.14) applied at each internal grid point forms one of $m = nx \times ny$ simultaneous equations. We can represent the system of simultaneous equations as a matrix equation involving an $m \times m$ matrix (i.e. 9×9 in our example)

(10.16)

$$\left(\begin{array}{ccc|ccc|ccc} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{array}\right) \left(\begin{array}{c} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ \hline u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ \hline u_{3,1} \\ u_{3,2} \\ u_{3,3} \end{array}\right) = - \left(\begin{array}{c} C_{0,1} + C_{1,0} \\ C_{0,2} \\ C_{0,3} + C_{1,4} \\ \hline C_{2,0} \\ 0 \\ C_{2,4} \\ \hline C_{3,0} + C_{4,1} \\ C_{4,2} \\ C_{4,3} + C_{3,4} \end{array}\right) \quad .$$

Each unknown is one element of the solution vector $\vec{u}$. The mapping from the spatial indices i, j into the row index p of the solution vector $\vec{u}$ is in general

$$p = j + ny \times (i - 1) \quad . \tag{10.17}$$

An example of this is a 2D-array to 1D-array indexing scheme; for our case with $nx = ny = 3$, unknown $u_{2,3}$ with $i = 2$ and $j = 3$ is located at row $p = 3 + 3 \times (2 - 1) = 6$. (Nb: for 3 spatial dimensions, a 3D-array to 1D-array scheme – an extension of (10.17)

– needs to be used.) The horizontal lines in (10.16) serve to visually vertically-partition the matrix and vectors into nx blocks. Each block has ny rows.

Solving the FD matrix equation – We can use the Jacobi method to solve this system, i.e., $\mathbf{A}\cdot\vec{u}=\vec{b}$ since the matrix is diagonally dominant. Starting with some initial guess $\vec{u}^0$ we can iterate with

$$\vec{u}^{n+1} = -\mathbf{D}^{-1}\cdot(\mathbf{L}+\mathbf{U})\cdot\vec{u}^n + \mathbf{D}^{-1}\cdot\vec{b}\ . \tag{10.18}$$

In this case $D^{-1}_{i,j} = -\delta_{i,j}\,\frac{1}{4}$ and we have

$$\vec{u}^{n+1} = \frac{1}{4}\mathbf{I}\cdot\left[(\mathbf{L}+\mathbf{U})\cdot\vec{u}^n - \vec{b}\right]\ . \tag{10.19}$$

However this can be represented using the pictorial operator

$$u^{n+1}_{i,j} = \frac{1}{4}\begin{bmatrix} & 1 & \\ 1 & 0 & 1 \\ & 1 & \end{bmatrix} u^n_{i,j} - b_{i,j}\ , \tag{10.20}$$

where $b_{i,j}$ are the elements b_p of $\vec{b}$, and p is calculated using the 2D-array $\to$ 1D-array indexing scheme (10.17). This pictoral operator averages each of the nearest neighbour for each of the internal points. For internal points adjacent to the boundary $b_{i,j}\neq 0$ and so the boundary conditions are applied here too. Thus the algorithm to solve the system is

- Initialise all lattice points including boundary values.
- Operate on all internal points with the pictorial operator with, e.g., the Jacobi method.
- Iterate until the solution has converged.

(See section 5.2, page 37 for details on checking for convergence).

Consistency and Accuracy – The order of accuracy of this FD scheme is $\mathcal{O}(h^2)$. For PDEs, the order is most readily determined from the modified differential equation (MDE). If Taylor expansions for $u_{i,j\pm1}$ $u_{i\pm1,j}$ are substituted into (10.14), the FD equivalent of Laplace's equation, then the resulting MDE can be shown to be

$$\nabla^2 u = -\frac{1}{12}\left(\frac{\partial^4 u}{\partial x^4}+\frac{\partial^4 u}{\partial y^4}\right)h^2 + \mathcal{O}(h^4) \tag{10.21}$$

demonstrating **consistency** and **2nd-order accuracy**. The order is the rate at which the numerical solution converges to the true solution as the step sizes tend to zero.

Poisson – It is simple to show that the extension to the Poisson case

$$\nabla^2 u = \rho(x,y)\,, \tag{10.22}$$

requires the minor modification

$$u^{n+1}_{i,j} = \frac{1}{4}\begin{bmatrix} & 1 & \\ 1 & 0 & 1 \\ & 1 & \end{bmatrix} u^n_{i,j} - b_{i,j} - \frac{h^2}{4}\rho_{i,j}\ , \tag{10.23}$$

where the source term is discretised on an identical lattice.

Physical interpretation of scheme – The pictorial operator shows how the centre point is locally coupled to 4 surrounding points. But these local couplings connect up giving a global coupling between all points; everything is interconnected. Hence we end up with a matrix equation to be solved. Physically, information at one point in space instantaneously propagates everywhere else. This is okay, as 'time' does not come into the equations; no need for causality.

Derivative boundary conditions

Consider the case where we have conditions on the derivatives in the solution u at the boundaries

$$u'_{i,j} = Q_{i,j} \tag{10.24}$$

i.e. Neumann boundary conditions. As we did in section 6.2 we will approximate the derivative as a central difference at the boundary involving a fictitious point. Below, we illustrate derivative boundary conditions in y, i.e., applied at $y = 0$ and $y = L_y$. Dirichlet conditions are still used in x. With different conditions holding on different sides of our rectangular boundary, we thus would classify this as mixed BCs. The setup is as follows:

$$\begin{array}{ccccc}
 & u'_{1,4} = Q_{1,4} & u'_{2,4} = Q_{2,4} & u'_{3,4} = Q_{3,4} & \\
 & * & * & * & \\
C_{0,3} & u_{1,3} & u_{2,3} & u_{3,3} & C_{4,3} \\
* & \circ & \circ & \circ & * \\
C_{0,2} & u_{1,2} & u_{2,2} & u_{3,2} & C_{4,2} \\
* & \circ & \circ & \circ & * \\
C_{0,1} & u_{1,1} & u_{2,1} & u_{3,1} & C_{4,1} \\
* & \circ & \circ & \circ & * \\
 & u'_{1,0} = Q_{1,0} & u'_{2,0} = Q_{2,0} & u'_{3,0} = Q_{3,0} & \\
 & * & * & * &
\end{array} , \tag{10.25}$$

(notice that we have omitted the corner points as they do not affect the solution at this order). We would create fictitious points next to the top and bottom boundaries

$$
\begin{array}{ccccc}
 & u_{1,5} & u_{2,5} & u_{3,5} & \\
 & * & * & * & \\
C_{0,4} & u_{1,4} & u_{2,4} & u_{3,4} & C_{4,4} \\
* & \circ & \circ & \circ & * \\
C_{0,3} & u_{1,3} & u_{2,3} & u_{3,3} & C_{4,3} \\
* & \circ & \circ & \circ & * \\
C_{0,2} & u_{1,2} & u_{2,2} & u_{3,2} & C_{4,2} \\
* & \circ & \circ & \circ & * \\
C_{0,1} & u_{1,1} & u_{2,1} & u_{3,1} & C_{4,1} \\
* & \circ & \circ & \circ & * \\
C_{0,0} & u_{1,0} & u_{2,0} & u_{3,0} & C_{4,0} \\
* & \circ & \circ & \circ & * \\
 & u_{1,-1} & u_{2,-1} & u_{3,-1} & \\
 & * & * & * &
\end{array}
\qquad (10.26)
$$

This system has $m = nx \times (ny + 2)$ unknowns. The fictitious points then become the boundaries and can be initialised using the definition of the central difference scheme

$$
\left.\frac{\partial u}{\partial y}\right|_{i,0} = \frac{u_{i,1} - u_{i,-1}}{2h} = Q_{i,0} \qquad \text{giving} \qquad u_{i,-1} = u_{i,1} - 2h\,Q_{i,0} \quad . \qquad (10.27)
$$

This system of simultaneous equations forms an $m \times m$ matrix equation which can then be solved iteratively as before taking care to update the fictitious points at each iteration.

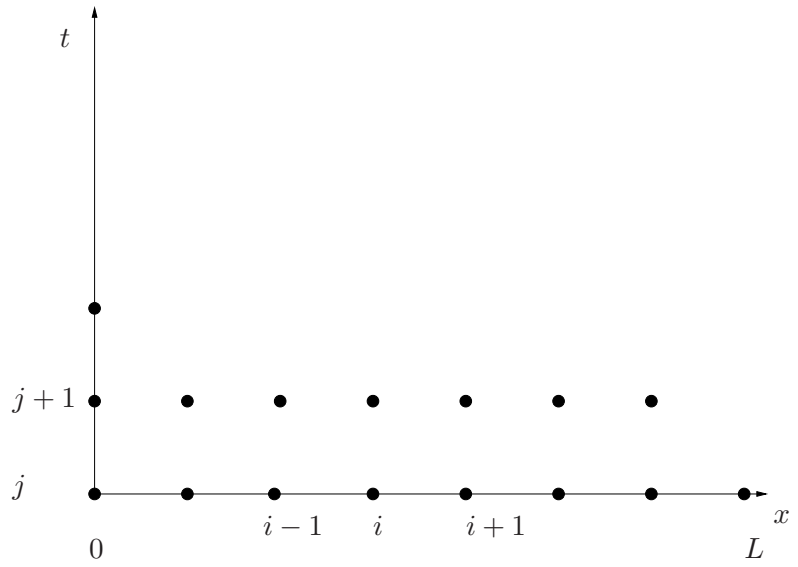

Figure 10.1. Initial value problem in (x, t) and the 1+1 discretisation scheme. The solution is integrated forward in time from an initial solution with boundary conditions at $x = 0$ and $x = L$.

10.3. Hyperbolic Equations

The wave equation involves second order derivatives in both time and space and is

$$\frac{\partial^2 u}{\partial t^2} - v^2 \frac{\partial^2 u}{\partial x^2} = 0 \quad , \tag{10.28}$$

in the 1D, linear case. We are looking to solve for $u(x, t)$ with $0 \le x \le L$ and $t_i \le t \le t_f$, i.e., an initial value problem. The boundary conditions for this system have to be provided for all x at t_i (i.e. the initial condition) and at the edges ($x = 0$, $x = L$) for $t_i < t \le t_f$. We can discretise the solutions using h as the spatial step and Δt as the time step. The location of the grid points in (x, t) are defined as

$$x_i = i\,h \qquad (i = 0,\, 1,\, \ldots\, , nx) \qquad h = L/nx \tag{10.29}$$

$$t^j = t_i + j\,\Delta t \qquad (j = 0,\, 1,\, \ldots\, , nt) \qquad \Delta t = (t_f - t_i)/nt \quad . \tag{10.30}$$

This approach is depicted in figure 10.1. The notation for $u(x_i, t^j)$ is

$$u_i^j = u(x_i, t^j)$$

i.e. the subscript & superscript denote the space and time index, respectively.

We could try and solve (10.28) numerically 'as is' by using second order finite difference approximations in both the space and time derivatives. We would effectively end up with a multi-point method since $\partial^2 u/\partial t^2$ would link 3 time steps; t^{j-1}, t^j and next (unknown) time t^{j+1}. However, this is not the best way. Some physical insight points to a better way, not involving the raw wave-equation.

Advection equations

This wave equation actually splits into two uncoupled 1st-order PDEs,

$$\frac{\partial f}{\partial t} + v\,\frac{\partial f}{\partial x} = 0 \quad , \qquad \frac{\partial g}{\partial t} - v\,\frac{\partial g}{\partial x} = 0 \ , \tag{10.31}$$

where $v \geq 0$ is the phase speed. They have general solutions $f(x,t) = F(x - vt)$ (arbitrary forward propagating disturbance) and $g(x,t) = G(x + vt)$ (arbitrary function propagating backwards), respectively. These are known as the **advection equations**, or more correctly the 'constant velocity advection' (CVA) equations. The term "convection" equation is also often used. The general solution of the linear wave-equation is thus $u(x,t) = F(x - vt) + G(x + vt)$. In 3D the CVA equation looks like

$$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla f = 0 \ .$$

The solution is $f(\vec{r},t) = f(t - \vec{v} \cdot \vec{r}/v^2)$, in a Cartesian coordinate system.

Coupled conservation-law + equation of motion

Physically, wave equations often arise from combining a *conservation law* (e.g. conservation of mass) with an *equation of motion*, e.g., for an acoustic wave in a gas

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u}) = 0 \ , \qquad \text{(mass conservation law)} \tag{10.32}$$

$$\frac{\partial\,(\rho \vec{u})}{\partial t} + \vec{u} \cdot \nabla(\rho \vec{u}) = -\nabla P \ , \qquad \text{(eqn. of motion)} \tag{10.33}$$

where ρ is mass density, $\vec{u}$ is the fluid velocity and P is the pressure. These are coupled 1st-order PDEs. For example, using the adiabatic gas law ($PV^\gamma = $ const.) which is $P\rho^{-\gamma} = P_o \rho_o^{-\gamma}$ (where P_o and ρ_o are the unperturbed values in the absence of a wave) and then ignoring the non-linear term $\vec{u} \cdot \nabla(\rho \vec{u})$ yields the wave equation

$$\frac{\partial^2 \rho}{\partial t^2} = \frac{\gamma P_o}{\rho_o} \nabla^2 \rho$$

when the the adiabatic law is linearised. We identify $c = \sqrt{\gamma P_o / \rho_o}$ as the *sound speed*.

In 1D linearised versions of the mass conservation law and equation of motion are

$$\frac{\partial \rho}{\partial t} + \rho_o \frac{\partial u}{\partial x} = 0 \ , \tag{10.34}$$

$$\frac{\partial u}{\partial t} + \frac{c^2}{\rho_o} \frac{\partial \rho}{\partial x} = 0 \ , \tag{10.35}$$

i.e. coupled which can be combined into a single PDE of conservative form,

$$\frac{\partial \vec{U}}{\partial t} = -\frac{\partial(\mathbf{F} \cdot \vec{U})}{\partial x} \ . \tag{10.36}$$

In our example

$$\vec{U} \equiv \begin{bmatrix} \rho(x,t) \\ u(x,t) \end{bmatrix} \qquad \text{and} \qquad \mathbf{F} \equiv \frac{1}{\rho_o} \begin{bmatrix} 0 & \rho_o^2 \\ c^2 & 0 \end{bmatrix} \ , \tag{10.37}$$

and **F** is a **flux operator**. Equation (10.36) looks like an advection equation for the a vector of functions, when we bear in mind that **F** is a matrix (tensor) of constants so can come outside the spatial derivative. For EM waves in vacuum (plane waves propagating in the x-direction), $\vec{U} = [E_y, cB_z]^T$ and the off-diagonal elements of **F** would become c (speed of light in vacuum).

So the solution of the wave equations reduces to solving an advection equation for a vector of 2 functions. Therefore, most of the effort in numerically solving hyperbolic PDEs concentrates on solving the advection equation

$$\frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} = 0 \ , \tag{10.38}$$

where now v_x is a velocity, so that its sign determines the direction e.g. forward or backward) of propagation. In practical terms, what we will be doing for can be considered in as an extension of the ODE methods (for initial value problems) seen in sections 2–4, to solve

$$\frac{du}{dt} = f(t, u) \ ,$$

but where now the function f depends on the spatial gradient of u. We also have to integrate the solution forward in time, simultaneously all across the spatial domain.

10.3.1. Upwind method

We take inspiration from the physics of advection and chose a one-sided finite difference approximation (FDA) for $\partial u/\partial x$ in the direction from which information propagates from. For $v_x > 0$ we have

$$\frac{\partial u}{\partial x} = \frac{u_i^j - u_{i-1}^j}{h} + \mathcal{O}(h) \ . \tag{10.39}$$

We chose the simple, forward difference scheme (FDS) for the partial time derivative

$$\frac{\partial u}{\partial t} = \frac{u_i^{j+1} - u_i^j}{\Delta t} + \mathcal{O}(\Delta t) \ . \tag{10.40}$$

The FDE equation is thus

$$\boxed{u_i^{j+1} = u_i^j - \frac{|v_x|\Delta t}{h}\left(u_i^j - u_{i-1}^j\right)} \ , \tag{10.41}$$

which is known as the **upwind method** (also known as the "donor cell method"). The factor

$$a = |v_x|\Delta t/h \ ,$$

is known as the **advection number**, an important dimensionless parameter.

For $v_x < 0$ we have

$$\frac{\partial u}{\partial x} = \frac{u_{i+1}^j - u_i^j}{h} + \mathcal{O}(h^2) \ ,$$

and thus the FDE

$$u_i^{j+1} = u_i^j + a(u_{i+1}^j - u_i^j) \ . \tag{10.42}$$

The upwind method is classed as an **explicit** scheme since $\partial u/\partial x$ depends on the known values $u(t^j)$ rather than the unknown ones $u(t^{j+1})$. Because the speed of information is finite, explicit schemes are best for hyperbolic PDEs; implicit schemes are more computationally intensive for the same accuracy. They offer no benefits for hyperbolic PDEs.

Consistency & accuracy – The MDE obtained from (10.41) is

$$\frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} = -\frac{1}{2}\frac{\partial^2 u}{\partial t^2}\Delta t + \mathcal{O}(\Delta t^2) + \frac{1}{2}\frac{\partial^2 u}{\partial x^2} v_x h + \mathcal{O}(h^2) \ . \tag{10.43}$$

The RHS vanishes as $\Delta t \to 0$ and $h \to 0$ which demonstrates that the upwind method is consistent with the advection PDE. The leading error terms show that it is 1st-order accurate in time and in space; it is a "$\boldsymbol{\mathcal{O}(\Delta t) + \mathcal{O}(h)}$ **scheme**". The MDE for $v_x < 0$ is almost exactly the same as (10.43) , but with $|v_x|$ multiplying the second partial spatial derivative.

Numerical diffusion – The above MDE can be rewritten as

$$\begin{aligned} \frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} &= \frac{|v_x| h}{2}(1-a)\,\frac{\partial^2 u}{\partial x^2} + \mathcal{O}(\Delta t^2) + \mathcal{O}(h^2) \\ &\approx D_{\text{num}} \frac{\partial^2 u}{\partial x^2} \ , \end{aligned} \tag{10.44}$$

which is really interesting. It shows that the upwind method is really giving us the solution of the advection equation with some non-physical diffusion (the leading term on the RHS). This "numerical diffusion" has a diffusion coefficient D_{num} that grows, the larger h is. A pulse with sharp features (e.g. top hat function) will blur out as it propagates along. The higher order error terms further modify the physics but are weaker than numerical diffusion.

Matrix equation – Practical implementation of (10.41) does not (and should not!) be done via matrices. However formulating the equations formed by the FDE applied at each spatial grid point, as a matrix equation is useful both conceptually and for, e.g., undertaking a stability analysis. We assume ***periodic spatial boundary conditions***, which is

$$u^j_{nx} = u^j_0 \ , \tag{10.45}$$

on our choice of spatial grid. Although there are $nx+1$ spatial points, this leaves us with $m = nx$ unknowns. The matrix equation is then

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{nx} \end{pmatrix}^{j+1} = \begin{pmatrix} (1-a) & 0 & 0 & \dots & 0 & a \\ a & (1-a) & 0 & \dots & 0 & 0 \\ 0 & a & (1-a) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a & (1-a) \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{nx} \end{pmatrix}^{j} \ ,$$

$$\vec{u}^{\,j+1} = \mathbf{T} \cdot \vec{u}^{\,j} \ , \tag{10.46}$$

where **T** is the update matrix (analogous to those seen with ODE FD methods in sections 2–4).

10.3.2. Stability analysis – von Neumann (Fourier) method

Von Neumann stability analysis checks whether the FDE causes a Fourier mode to grow exponentially in amplitude (instability) or not. It is arguably more physically intuitive than the matrix method (coming up soon), giving us insight about how the FD grid affects the phase speed and highlighting the presence of **numerical dispersion**. It only works for periodic boundary conditions.

The procedure is to substitute a Fourier mode for $u(x,t)$

$$\boxed{u_i^j = \exp[i(kx_i - \omega t^j)] = (g)^j \, \exp(ikx_i) \ ,} \tag{10.47}$$

into the FDE. This mode is sampled at discrete spatial and temporal points. Here the amplification factor g is raised to the power j (the time index). k and ω are the wave-number and angular frequency of the mode. The relation between u_i^j and adjacent points on the FD grid is

$$u_i^{j+1} = g\, u_i^j \qquad , \qquad u_{i\pm1}^j = u_i^j \, \exp(\pm i\, k\, h) \ . \tag{10.48}$$

Substituting into the upwind method FDE (10.41) yields

$$g \cancel{u_i^j} = \cancel{u_i^j}(1-a) + \cancel{u_i^j}\, a\, e^{-ikh} \ ,$$

and then separating out into real and imaginary parts gives

$$g = (1-a) + a\cos(kh) \ - \ i\, a \sin(hk) \ ,$$

After some algebra we get

$$|g|^2 = 1 - 2a(1-a)[1-\cos(kh)] \ ,$$

and then using the identity $1 - cos\theta = 2\sin^2\theta$ yields

$$|g|^2 = 1 - 4a(1-a)\sin^2(kh) \ . \tag{10.49}$$

The maximum wave-number allowable on the grid is

$$k_{max} = \frac{\pi}{h} \qquad (\text{i.e. } \lambda_{min} = 2h \ , \tag{10.50}$$

so that $\sin^2(kh)$ ranges from 0 to 1. The condition for stability is

$$|g| \le 1 \ ,$$

i.e., $1 - 4a(1-a) \le 1$ which yields

$$\boxed{a \le 1 \qquad \text{or equivalently} \qquad \Delta t \le h/|v_x| \ .} \tag{10.51}$$

This is known as the **CFL condition** (named after Courant, Friedrich & Lewy). The CFL condition is equivalent to

$$|v_x| \le v_{\text{grid}} \ , \tag{10.52}$$

i.e. physical information propagation speed must be less than the *grid speed* $v_{\text{grid}} = h/\Delta t$. If violated, then the domain of dependence of a given grid point is larger than the FD scheme can reach, which leads to problems.

Note that in practice the time step needs to be quite a bit smaller than that given by the Courant condition to get good accuracy.

[ASIDE] Numerical damping & dispersion – From equation (10.47) we find that

$$g = \exp(-i\omega\Delta t)$$

but we also know $g = |g|\exp(i\theta)$ hence

$$\omega = i\frac{\ln|g|}{\Delta t} - \frac{\theta}{\Delta t} \quad . \tag{10.53}$$

The imaginary part of ω gives the **numerical damping** (or growth when unstable!) while the real part described **numerical dispersion**, the fact that different wavelengths travel with different speeds although they should have the same phase speed $|v_x|$. The grid (or 'numerical') phase-speed is

$$v_{ph-g} = \frac{w}{k} = -\frac{\theta(kh,a)}{\Delta t} \quad . \tag{10.54}$$

In principle, an explicit expression for v_{ph-g} in terms of kh and a can be found; it would show $v_{ph-g} = |v_x|$ as $k \to 0$ which matches the physical phase speed. However at short wavelengths, approaching k_{max}, the numerical phase speed will unphysically dilate reaching v_{grid} at $k = k_{max}$.

10.3.3. Stability analysis – matrix method

This is the same as the approach used for coupled ODEs in section 3.4. In this context the coupled variables are u_i's at each spatial grid point.

We could define $\tilde{\vec{u}}^j = \vec{u}^j + \vec{\epsilon}^j$ relating the FD approximation ($\tilde{\vec{u}}^j$) to the true solution of the exact PDE ($\vec{u}^j$) via an error ($\vec{\epsilon}^j$). Ignoring terms responsible for gradual, increasing loss of accuracy, as before, we are left with $\vec{\epsilon}^{\,j+1} = \mathbf{T}\cdot\vec{\epsilon}^{\,j}$ upon inserting $\tilde{\vec{u}}^j = \vec{u}^j + \vec{\epsilon}^j$ into the matrix version of the FDE (i.e. (10.46)).

For stability we require all eigenvalues λ_i of the update matrix $\mathbf{T}$ satisfy

$$|\lambda_i| \le 1 \qquad (\text{all } i) \quad .$$

Again, the bounds of the eigenvalues can be assessed by Gerschgorin's theorem

$$|\lambda_i - T_{ii}| \le \sum_{i\neq j} |T_{ij}| \quad .$$

Example: for the upwind method (and periodic BCs) all rows of the matrix are 'similar' so

$$|\lambda - (1-a)| \le a \quad ,$$

where a is a positive real value. So λ is lies within (or on the edge of) a disk of radius a in the complex plane, with this disk centred at $z_o = (1-a)$, i.e., on the real axis. The 'right edge' of this disk is always at $z_{right} = +1$. The 'left edge' lies at $z_{left} = 1-2a$. For $a > 1$ we have $z_{left} < -1$ and so $|\lambda| > 1$ (i.e. instability) is possible. (Note that all other points on the circle edge break-out beyond unit distance from the origin when

this happens, except the point exactly on the +ve real axis.) Hence $a \leq 1$ is required for $|\lambda| \leq 1$ and therefore stability.

The advantage of the matrix method over Von-Neumann's method is its ability to deal with (i) BCs other than periodic, (ii) cases where the phase speed changes across the grid.

10.3.4. FTCS method – (one not to use!)

This illustrates one sort of pitfall that exist when trying to come up with numerical FD schemes for PDEs. To increase the accuracy of the spatial derivative, centred differencing might seem like a good idea;

$$\frac{\partial u}{\partial x} = \frac{u_{i+1}^j - u_{i-1}^j}{2h} + \mathcal{O}(h^2) \ .$$

This leads to the innocuous looking FDE (for any v_x)

$$u_i^{j+1} = u_i^j - \frac{v_x \Delta t}{2h}\left(u_{i+1}^j - u_{i-1}^j\right) \ . \tag{10.55}$$

It may surprise you that this is **unconditionally unstable!**. A stability analysis (by either method) yield the following impossible condition for stability $|v_x|\Delta t/(2h) \leq 0$. (Impossible for positive Δt and h.) However, this FTCS scheme is a consistent, $\mathcal{O}(\Delta t) + \mathcal{O}(\Delta h^2)$ FD scheme but spoiled by instability that cannot be dodged!

10.3.5. Lax method

This illustrates a second sort of pitfall that exist when trying to come up with numerical FD schemes for PDEs; inadvertently coming up with an **inconsistent scheme**. The Lax method takes FTCS and substitutes a spatial average for the current point,

$$u_i^j = \frac{1}{2}\left(u_{i+1}^j + u_{i-1}^j\right) \ ,$$

giving the FDE

$$u^{j+1} = \frac{1}{2}\left(u_{i+1}^j + u_{i-1}^j\right) - \frac{|v_x|\,\Delta t}{2h}\left(u_{i+1}^j - u_{i-1}^j\right) \ , \tag{10.56}$$

which turns out to have the usual CFL stability condition, $a < 1$. However the MDE can be shown to be

$$\frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} \ = \ -\frac{1}{2}\frac{\partial^2 u}{\partial t^2}\,\Delta t - \frac{1}{6}\frac{\partial^3 u}{\partial x^3}\,v_x h^2 + \frac{1}{2}\frac{\partial^2 u}{\partial x^2}\,\frac{h^2}{\Delta t} + \mathcal{O}(\Delta t^2) + \mathcal{O}(h^2) \ . \tag{10.57}$$

The third term on the RHS will not vanish $h\,,\ \Delta t \to 0$ unless h and Δt are constrained as $\Delta t \propto h^p$ with $p < 2$, and will diverge for $p > 2$ (swamping the intended PDE!). In practice this method will work, but suffers from numerical diffusion.

10.3.6. Lax-Wendroff

This is FTCS with a little extra

$$\boxed{u_i^{j+1} = u_i^j - \frac{v_x \Delta t}{2h}\left(u_{i+1}^j - u_{i-1}^j\right) + \frac{(v_x \Delta t)^2}{2h^2}\left(u_{i+1}^j - 2u_i^j + u_{i-1}^j\right) \ ,} \tag{10.58}$$

which can be shown to be a consistent, $\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta h^2)$ scheme. If has the usual CFL condition $a = |v_x|\Delta t/h \leq 1$. Also Lax-Wendroff doesn't have two different versions of the FDE as for the one-sided upwind method.

10.3.7. Coupled equations

The Lax-Wendroff and Lax methods can be applied to the coupled 1st-order PDEs seen earlier, e.g., equation (10.35) for ρ (mass density) and u (fluid velocity). Using the simpler Lax scheme to illustrate;

$$\begin{aligned} \rho_i^{j+1} &= \tfrac{1}{2}\left(\rho_{i+1}^j + \rho_{i-1}^j\right) + \tfrac{\rho_o\,\Delta t}{2h}\left(u_{i+1}^j - u_{i-1}^j\right) \quad , \\ u_i^{j+1} &= \tfrac{1}{2}\left(u_{i+1}^j + u_{i-1}^j\right) + \tfrac{c^2\,\Delta t}{2\rho_o\,h}\left(\rho_{i+1}^j - \rho_{i-1}^j\right) \quad . \end{aligned} \tag{10.59}$$

10.4. Parabolic Equations

Consider the Diffusion equation

$$\frac{\partial u}{\partial t} = D\,\nabla^2 u \quad . \tag{10.60}$$

with D constant.

Explicit method

Explicit methods are ones that obtain the next time-step i.e. u_i^{j+1} *only* using the solution at the current time step i.e. u_i^j. We choose the, first order, forward difference scheme for the time derivative

$$\frac{\partial u}{\partial t} = \frac{u_i^{j+1} - u_i^j}{\Delta t} + \mathcal{O}(\Delta t) \quad ,$$

as we did with schemes for hyperbolic PDEs. For the spatial derivative we choose the simplest approximation, the second order, central difference scheme,

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i-1}^j - 2\,u_i^j + u_{i+1}^j}{h^2} + \mathcal{O}(h^2) \quad . \tag{10.61}$$

Inserting these approximations into the diffusion equation (10.60) we get

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} = D\left(\frac{u_{i-1}^j - 2\,u_i^j + u_{i+1}^j}{h^2}\right) \quad , \tag{10.62}$$

giving the FDE

$$\boxed{u_i^{j+1} = (1 - 2\,d)\,u_i^j + d\,(u_{i-1}^j + u_{i+1}^j) \quad ,} \tag{10.63}$$

where

$$d = \frac{D\,\Delta t}{h^2} \quad , \tag{10.64}$$

is the **diffusion number**, a dimensionless parameter.

This is a consistent, $\mathcal{O}(\Delta t) + \mathcal{O}(h^2)$ scheme.

Either the Von-Neumann method (section 10.3.2) or the matrix method (section 10.3.3) can be used to show that the condition for this FDE above (10.63) to be stable is

$$\boxed{d = \frac{D\,\Delta t}{h^2} < \frac{1}{2} \qquad \text{or equivalently} \qquad \Delta t < \frac{h^2}{2D} \quad .} \tag{10.65}$$

This bound has a physical implication in that it tells us that the maximum time step size is limited to roughly half the **diffusion time**

$$\tau \sim \frac{h^2}{D} \quad , \tag{10.66}$$

which describes the time it takes for information to propagate between the sites in the lattice.

This is actually quite restrictive since we are typically trying to look at processes on scales much larger than the discretisation scale

$$\ell \gg h\,, \tag{10.67}$$

which means we will have to calculate many time steps to observe the dynamics of the system.

Crank-Nicolson Method - (Implicit Method)

To improve on this stringent constraint on the time-step and also increase the accuracy of the method, we go to a second order approximation for the time derivative. The Crank-Nicholson method is obtained by inserting a fictitious layer of grid points at $j+\frac{1}{2}$, i.e., half way between the original time steps. The second order, central difference scheme in time for the $j+1/2$ step is

$$\left.\frac{\partial u}{\partial t}\right|_i^{j+1/2} = \frac{u_i^{j+1}-u_i^j}{\Delta t} + \mathcal{O}(\Delta t^2)\ . \tag{10.68}$$

For the spatial derivative we can take the average of those at j and $j+1$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_i^{j+1/2} = \frac{1}{2}\left[\left.\frac{\partial^2 u}{\partial x^2}\right|_i^{j} + \left.\frac{\partial^2 u}{\partial x^2}\right|_i^{j+1}\right]\ . \tag{10.69}$$

Rearranging all $j+1$ terms on the LHS we get the system

$$\boxed{(1+2\,d)\,u_i^{j+1} - d\,(u_{i-1}^{j+1}+u_{i+1}^{j+1}) = (1-2\,d)\,u_i^j + d\,\left(u_{i-1}^j + u_{i+1}^j\right)\ ,} \tag{10.70}$$

or in matrix form

$$\mathbf{M}\cdot\vec{u}^{\,j+1} = \vec{b}^{\,j}\ , \tag{10.71}$$

where

$$\mathbf{M} \equiv \begin{pmatrix} (1+2\,d) & -d & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ -d & (1+2\,d) & -d & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & d & (1+2\,d) & -d & \cdot & \cdot & \cdot & 0 \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & -d & (1+2\,d) \end{pmatrix}\ , \tag{10.72}$$

and

$$\vec{b}^{\,j} \equiv \begin{pmatrix} (1-2\,d) & d & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ d & (1-2\,d) & d & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & d & (1-2\,d) & d & \cdot & \cdot & \cdot & 0 \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & d & (1-2\,d) \end{pmatrix}\cdot\vec{u}^{\,j} = \mathbf{A}\cdot\vec{u}^{\,j}\ . \tag{10.73}$$

where Dirichlet BCs of $u_0^j = u_{nx}^j = 0$ have been applied in getting the above forms of $\mathbf{M}$ and $\vec{b}$. Hence the number of unknowns is $m = nx - 1$ (i.e. $1 \leq i \leq nx - 1$).

Crank-Nicholson is a consistent, $\mathcal{O}(\Delta t^2) + \mathcal{O}(h^2)$ scheme.

It can be shown that this method is actually stable for all values of d (i.e. any Δt for a given D and h). So we can substitute a time integration with an iterative solution of a linear system using, e.g., a Jacobi, Gauss-Seidel or SOR iterator starting with the initial condition $\vec{u}^0$.

SECTION 11
Fourier Transforms

Outline of Section

- Fourier Transforms – recap
- Discrete FTs (DFTs)
- Sampling & Aliasing
- Spectral methods – Solving the Poisson Equation as an example
- Fast Fourier Transform (FFT) algorithm (non-examinable)

11.1. Fourier Transforms – Recap

The continuous Fourier Transform (FT), both forward and backward, of a function $f(t)$ are defined as

$$\tilde{f}(\omega) = \int_{-\infty}^{\infty} e^{i\,\omega\,t} f(t) dt = \mathcal{F}(f(t)) \ , \tag{11.1}$$

$$\text{and} \qquad f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{-i\,\omega\,t} \tilde{f}(\omega) d\omega = \mathcal{F}^{-1}(\tilde{f}(\omega)) \ , \tag{11.2}$$

for the case of time t and angular frequency $\omega = 2\pi\nu$ (where ν is frequency). Time t and ω are reciprocal 'coordinates' or variables. $\tilde{f}(\omega)$ is the (angular) frequency spectrum of the function $f(t)$. The FT is an expansion on plane waves $\exp(-i\,\omega\,t)$ which constitute an orthonormal basis, and $\tilde{f}(\omega)$ can be thought of as the "coefficients" of each component wave of angular frequency ω. (Notation note: don't confuse with the tilde '$\sim$' notation used earlier in the course to denote numerical approximation!) In general $\tilde{f} \in \mathbb{C}$, even for a real function $f(t)$, with $|\tilde{f}(\omega)|$ being the amplitude and $\arg(\tilde{f}(\omega))$ the phase of the component wave $\exp(-i\,\omega\,t)$. For $f(t) \in \mathbb{R}$ the **reality condition** applies in reciprocal space (also referred to as 'Fourier space')

$$\tilde{f}(-\omega) = \tilde{f}^*(\omega) \ , \tag{11.3}$$

so that the negative frequencies are degenerate; there is no extra information in them. However, for an arbitrary $f(t) \in \mathbb{C}$, $\tilde{f}(-\omega)$ is unrelated to $\tilde{f}^*(\omega)$.

The "FT pairs" $f(t)$ and $\tilde{f}(\omega)$ are different representations of the same thing in the time and frequency domain, respectively, and the relationship between such FT pairs is

often written as

$$f(t) \rightleftharpoons \tilde{f}(\omega) \quad . \tag{11.4}$$

$\mathcal{F}^{-1}(\tilde{f}(\omega))$ is often called the inverse Fourier transform. Note that there are different conventions for defining the FT operations, e.g., which operation gets the $1/2\pi$ factor or do both share it (i.e. $1/\sqrt{2\pi}$ in front of each), the sign in the transform kernel (i.e. $\exp(-i\,\omega\,t)$ or $\exp(i\,\omega\,t)$) , etc.

For spatial FTs in one dimension (e.g. x) the reciprocal variables become $t \to x$ and $\omega \to k$ where $k = 2\pi/\lambda$ is the wavenumber and λ is wavelength. The FTs defined above apply with these changes of variables. In multiple spatial dimensions the wavenumber becomes a wave vector

$$\vec{x} \equiv (x,\, y,\, z) \qquad \leftrightarrow \qquad \vec{k} \equiv (k_x,\, k_y,\, k_z) \quad , \tag{11.5}$$

and the FTs are modified accordingly

$$\tilde{f}(\vec{k}) \;=\; \int_{-\infty}^{\infty} e^{i\,\vec{k}\cdot\vec{x}} f(\vec{x})\, d\vec{x} \;=\; \mathcal{F}(f(\vec{x})) \quad , \tag{11.6}$$

$$f(\vec{x}) \;=\; \frac{1}{(2\pi)^3} \int_{-\infty}^{\infty} e^{-i\,\vec{k}\cdot\vec{x}} \tilde{f}(\vec{k})\, d\vec{k} \;=\; \mathcal{F}^{-1}(\tilde{f}(\vec{k})) \quad . \tag{11.7}$$

Derivatives and FTs

FTs can be very useful in manipulating differential equations. This is because spatial or time derivatives become algebraic operations in Fourier space. As an example consider the following equation in real space

$$\frac{d}{dx} u(x) = v(x) \quad .$$

Taking the FT of this equation yields

$$-\,i\,k\tilde{u}(k) = \tilde{v}(k) \quad . \tag{11.8}$$

This can be shown as follows: replace $u(x)$ and $v(x)$ by their inverse FTs

$$\frac{d}{dx}\frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-i\,k\,x}\tilde{u}(k)dk = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-i\,k\,x}\tilde{v}(k)dk \quad . \tag{11.9}$$

The only bits containing x are the transform kernel (i.e. the plane wave part $\exp(-i\,k\,x)$ so we can bring the derivative into the integral and operate on the kernel yielding

$$\frac{1}{2\pi}\int_{-\infty}^{\infty} -i\,k\,e^{-i\,k\,x}\tilde{u}(k)dk = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-i\,k\,x}\tilde{v}(k)dk \quad .$$

Equating the integrands on both sides we have (11.8).

This can be generalised to higher derivatives:

$$\frac{d^n}{dx^n} f(x) \;\rightleftharpoons\; (-i\,k)^n\, \tilde{f}(k) \quad . \tag{11.10}$$

It can be generalised to 3D too:

$$\begin{aligned} \nabla f(\vec{x}) &\rightleftharpoons -i\,\vec{k}\,\tilde{f}(\vec{k}) \quad , & \nabla^2 f(\vec{x}) &\rightleftharpoons -|\vec{k}|^2\,\tilde{f}(\vec{k}) \quad , \\ \nabla\cdot\vec{E}(\vec{x}) &\rightleftharpoons -i\,\vec{k}\cdot\tilde{E}(\vec{k}) \quad , & \nabla\times\vec{E}(\vec{x}) &\rightleftharpoons -i\,\vec{k}\times\tilde{E}(\vec{k}) \quad , \end{aligned} \tag{11.11}$$

where $\nabla^2 f = \nabla \cdot (\nabla f)$ has been used.

11.2. Discrete FTs

DFTs are the equivalent of complex Fourier series, which is defined on a finite length domain, but for a **discretely sampled function** rather than a continuous function. We illustrate here with time and angular frequency.

Time domain – We assume the function is sampled by N equally spaced samples (with N even for simplicity, see later) of a time domain of length $T = N\Delta t$

$$f_n \equiv f(t_n) \qquad \text{with} \qquad n = 0, 1, 2, ..., N-1 \qquad \text{and} \qquad t_n = n\,\Delta t \ . \tag{11.12}$$

The function $f(t)$ is ***assumed to be periodically extended*** beyond the domain $0 \leq t \leq T$ so that $f(t + m\,T) = f(t)$ for integer m. For the sampled function, periodicity means $f_N = f_0$ hence we discard f_N in our sequence of samples.

Frequency domain – What does the corresponding (angular) frequency domain look like? The longest wave that can fit exactly into the time domain has an angular frequency $\omega_{min} = 2\pi/T \equiv \Delta\omega$. The shortest wave that can be *recognised* by the grid and fits periodically into the time domain has duration $2\Delta t$ (i.e. one peak and one trough in just two time samples) so that $\omega_{max} = 2\pi/(2\Delta t) = \pi/\Delta t$. This maximum frequency is known as the **Nyquist frequency**

$$\boxed{\omega_{max} = \frac{\pi}{\Delta t} = \Delta\omega\ \frac{N}{2} \ .} \tag{11.13}$$

Allowing for negative angular frequencies too, these frequencies define the discrete, finite, angular frequency grid on which $\tilde{f}$ is sampled;

$$\boxed{\tilde{f}_p \equiv \tilde{f}(\omega_p) \quad \text{with} \quad p = -\frac{N}{2}, \ldots, 0, \ldots, \frac{N}{2} \quad \text{and} \qquad \omega_p = p\,\Delta\omega = p\frac{2\pi}{N\,\Delta t} \ .} \tag{11.14}$$

Each discrete frequency corresponds to a wave that fits exactly an integer number of times 'p' into the finite domain. Note that the integer index p seems to run over $N+1$ values but in fact the $-N/2$ and $N/2$ samples are degenerate, i.e.,

$$\tilde{f}_{-N/2} \equiv \tilde{f}_{N/2} \ .$$

so there are only N degrees of freedom. (See section 11.3.)

The DFT – is the analogue of a continuous FT, but with the continuous integral $\int \ldots dt$ replaced by a finite sum $\sum \ldots \Delta t$;

$$\boxed{\tilde{f}_p = \sum_{n=0}^{N-1} f_n\, e^{i\,\omega_p t_n} = \sum_{n=0}^{N-1} f_n\, e^{i\,2\pi p n/N} \ ,} \tag{11.15}$$

where $\omega_p\, t_n = \left(\frac{2\pi\, p}{N\Delta t}\right)\, (n\,\Delta t)$ has been used in obtaining the second form. The backwards transform is similarly defined as

$$f_n = \frac{1}{N} \sum_{p=-N/2+1}^{N/2} \tilde{f}_p\, e^{-i\,2\pi pn/N} \quad . \tag{11.16}$$

The different normalisation factor of $1/N$ accounts for the discrete sampling. This comes from $dt \to \Delta t$ and $d\omega \to \Delta\omega = 2\pi/(N\Delta t)$. (Note that in going from the FS (11.1) to the DFT (11.15) a factor of Δt has been absorbed into $\tilde{f}_p$, i.e., $\tilde{f}_p \approx \tilde{f}(\omega_p)/\Delta t$.) The backward DFT (11.16) is usually defined as

$$\boxed{f_n = \frac{1}{N} \sum_{p=0}^{N-1} \tilde{f}_p\, e^{-i\,2\pi pn/N} \quad ,} \tag{11.17}$$

which is more symmetrical with the forward DFT. Why this is possible will be seen in section 11.3.

Relation to complex Fourier series – Discrete FTs (DFTs) are intimately related to complex Fourier series (CFS). Complex Fourier series represents the discrete spectrum of a ***continuous function*** $f(t)$, also on a domain of finite length. The allowed angular frequencies have the same spacing $\Delta\omega$, but are now infinitely many.

FFTs – Fast Fourier transforms

The DFTs above, equations (11.15) and (11.16), are easily implemented on a computer, however the naive method of implementing it requires $\mathcal{O}(N^2)$ operations. With typical samples volumes of $N \sim 10^6$ in modern applications this becomes prohibitive even on the fastest computers. The DFT can actually be done in $\boldsymbol{\mathcal{O}(N \log_2 N)}$ **operations**; the algorithm for this is known as the **fast Fourier transform** or just **FFT**. For $N = 10^6$, the speed up is $N/log_2 N \approx 50,000$ times! FFTs work best when $N = 2^m$ with integer m, and works by recursively splitting the DFT into separate sums over only the even or odd indices. ***The FFT is what is used in practice.*** If $N \neq 2^m$, then it is usual to ***pad out*** the samples with zeros until the size is $N = 2^m$. (Non-examinable – The FFT algorithm is outlined at the end of this section.)

There are countless examples of uses of FFT

- Noise suppression – Data analysis
- Image compression – JPEGS
- Music compression – MPEGS
- Medical imaging
- Interferometric imaging
- Modelling of optical systems
- Solution of periodic boundary value problems

to name a few.

DFT in 2D

For a function $f_{n,m} = f(x_n, y_m)$ defined on a 2D grid $x_n = m\Delta x$ for $0 \le n \le N-1$ and $y_m = m\Delta y$ for $0 \le n \le M-1$ its DFT is given by

$$\tilde{f}_{p,q} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} f_{n,m}\, e^{i\,2\pi pn/N}\, e^{i\,2\pi qm/M} \quad . \tag{11.18}$$

The backward DFT is

$$f_{n,m} = \frac{1}{N\,M} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} \tilde{f}_{p,q}\, e^{-i\,2\pi pn/N}\, e^{-i\,2\pi qm/M} \quad . \tag{11.19}$$

Going to 3-dimensions just adds another nested sum (with a new, independent index) and another exponential wave factor (incorporating the new dimension) into the transform kernel.

11.3. Sampling & Aliasing

The discrete sampling of the function to be FT'd, given by equation (11.12), introduces some sampling effects and care has to be taken in allowing for them. There is a minimum sampling rate that needs to be used so that we don't lose information. If the function to be sampled fits into the finite length time (or spatial) domain of length T, and is **bandwidth limited** to below the Nyquist frequency then all frequency content of the function is exactly captured by the discrete sampling process. If the function has $\tilde{f}(\omega) \neq 0$ for $|\omega| > \omega_{max}$ then the frequency content for $|\omega| > \omega_{max}$ will be **aliased** down into the range $|\omega| < \omega_{max}$ and distort the DFT compared to the exact FT of $f(t)$. If the original function is bandwidth limited but is longer than T, then it will be ***clipped*** which will introduce a sharp cutoff at $t = 0$ and/or T. This will effectively increase the bandwidth of the clipped function and aliasing will occur again during sampling of it.

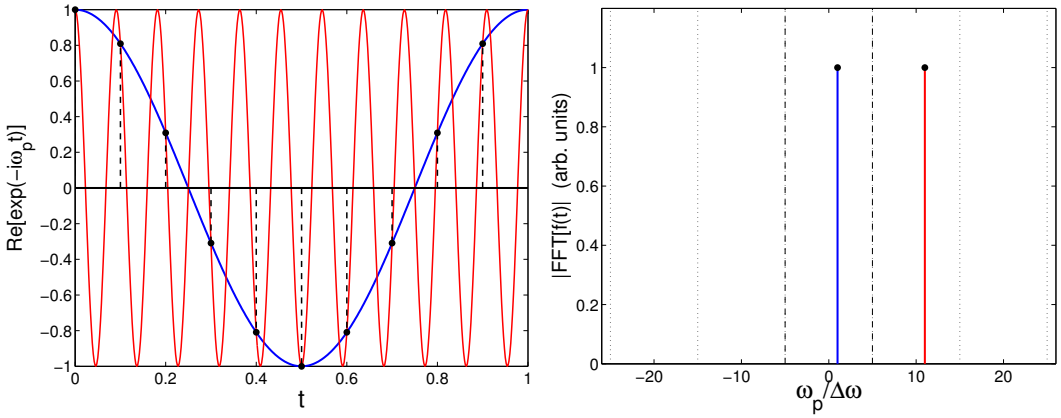

Figure 11.1. (Left) Indistinguishable harmonic waves below and above the Nyquist frequency, for $N = 10$. Blue line: $\omega = \omega_1 = \Delta\omega$. Red line: $\omega = \omega_{N+1} = \Delta\omega + 2\omega_{max}$. (Waves are periodically extended beyond range shown.) (Right) Corresponding (angular) frequency spectrum. The vertical dashed lines lie at $\pm$ the Nyquist frequency.

Aliasing & indistinguishable frequencies – For any wave with a frequency ω_p lying in the frequency domain captured by the DFT, there are higher frequency waves with $\omega_{p'} = \omega_p + m\Omega$ (where $m \in \mathbb{Z}$ and $\Omega = 2\omega_{max}$ is the width of the frequency domain) that look exactly the same to the discrete time-sampling grid. This is illustrated in figure 11.1. This can be understood by considering the kernel of the DFT, i.e., $\exp(i\,\omega_{p'} t_n)$.

$$\begin{aligned}\exp\left[i(\omega_p + m\Omega)t_n\right] &= \exp\left[i\left(\frac{2\pi\,pn}{N} + m\frac{2\pi}{\Delta t}n\Delta t\right)\right] = \exp\left(i\frac{2\pi\,pn}{N} + i2\pi\,m\,n\right)\\ &= \exp\left(i\frac{2\pi\,pn}{N}\right) = \exp\left(i\omega_p\,t_n\right) \quad .\end{aligned}$$

This has the following implications;

(a) It explains why $\tilde{f}_{-N/2} \equiv \tilde{f}_{N/2}$.

(b) It also explains why equation (11.17) for the backwards DFT 'works'. The "negative" frequency part of the spectrum $p = \left\{-\frac{N}{2}+1, \ldots, -1\right\}$ maps to the positive spectrum at $p' = p + N = \left\{\frac{N}{2}+1, \frac{N}{2}+2, \ldots, N-1\right\}$ lying beyond the Nyquist frequency. In other words, although the $\sum_{p=N/2+1}^{N-1}$ parts of (11.17) are above the Nyquist frequency, they happen to capture the desired negative frequencies within the Nyquist range.

(c) Aliasing:

$$\tilde{f}_p = \sum_m \mathcal{F}(f)|_{\omega_p + m\Omega} \quad , \tag{11.20}$$

i.e., the DFT (the LHS) folds in the Fourier components from the *exact FT of the continuous function* from the indistinguishable set of waves $\omega'_p = p + m\Omega$.

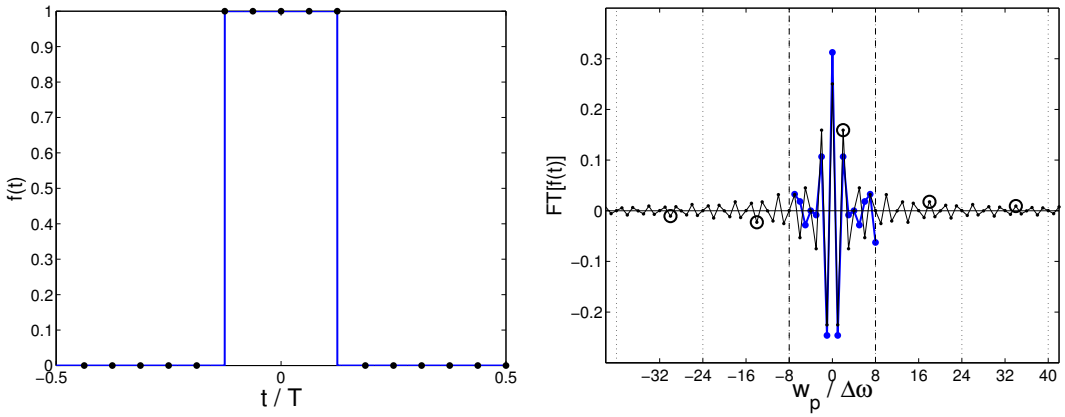

Figure 11.2. Aliasing: (Left) $f(t)$ in the time domain; centred "rect" function of width $T/4$. Sampled with $N = 16$. (Right) The DFT (blue, solid line + markers), and exact FT (black, thin solid line) in the frequency domain. The open circles denote Fourier components outside the range $-\omega_{max} \leq \omega \leq \omega_{max}$ that are aliased into that range during the DFT. The real part of the DFT & FT are shown. The DFT been divided by Δt.

Figure 11.2 shows the phenomenon of aliasing for a top-hat function

$$f(t) = \begin{cases} 1 & |t| < a/2 \\ 0 & |t| > a/2 \end{cases} \quad .$$

11.4. Spectral methods – Solving the Poisson Equation by FFT

Methods using FTs to solve PDEs are known as **spectral methods**. As an example we consider the general Poisson equation

$$\nabla^2 u(\vec{x}) = \rho(\vec{x}) \quad , \tag{11.21}$$

where ρ is some source density and u is some potential we are seeking to solve for. For electrostatic problems this specialises to

$$\nabla^2 \phi(\vec{x}) = -\rho_c(\vec{x})/\epsilon_o \quad ,$$

where ϕ and ρ_c are the electric potential and charge-density, respectively, and for (Newtonian) gravitational problems

$$\nabla^2 \Phi(\vec{x}) = 4\pi G\, \rho(\vec{x})$$

where Φ is gravitational potential, G is Newton's constant and ρ is the mass density of matter.

We have seen how to solve this system using an iterative method in Section 10.2. The iterative method works for any BC; periodic, fixed, etc. For periodic BCs, using FTs is much more efficient.

The exact analytical solution can easily be written in terms of FTs as follow. As we have seen the Laplacian is replaced by the factor $-|\vec{k}|^2$ in Fourier space so that exact solution if Fourier space is

$$\tilde{u}(\vec{k}) = -\frac{\tilde{\rho}(\vec{k})}{|\vec{k}|^2} \quad . \tag{11.22}$$

To obtain the exact solution in real (coordinate) space we just need to inverse transform the above

$$u(\vec{x}) = \mathcal{F}^{-1}\left[-\frac{\tilde{\rho}(\vec{k})}{|\vec{k}|^2}\right] \quad . \tag{11.23}$$

In practice for general $\rho(\vec{x})$, tractable expressions for the Fourier transform integrals will be difficult (even impossible) to find.

Equipped with a discrete Fourier transform, the obvious way to solve (11.21) numerically (in, e.g., 2D) would seem to be to take a FFT of the gridded source density $\rho_{n,m}$ to get $\tilde{\rho}_{p,q}$, divide by $|\vec{k}_{p,q}|^2$ (where $(k_x)_p = p\pi/\Delta x$ and $(k_y)_q = q\pi/\Delta y$) and then take the backwards DFT to get $u_{n,m}$. However, this does not yield the same thing as solving the discretised PDE, as we did in section 10.2 (i.e. equation (10.23)). The reason is that we have not yet taken into account the FD approximation of the Laplacian.

11.4.1. Application to discrete system – 1-D

We discretise the system as usual with grid spacing h and use a second order finite difference scheme for the Laplacian

$$\frac{u_{j+1} - 2\,u_j + u_{j-1}}{h^2} = \rho_j\,. \tag{11.24}$$

Expanding each term through the backwards DFT we have

$$u_j = \frac{1}{N}\sum_p \tilde{u}_p\, e^{-i\,2\pi\,pj/N}\,, \tag{11.25}$$

and

$$u_{j\pm1} = \frac{1}{N}\sum_p \tilde{u}_p\, e^{-i\,2\pi\,p(j\pm1)/N} = \frac{1}{N}\sum_p \tilde{u}_p\, e^{\mp i\,2\pi\,p/N}\, e^{-i\,2\pi\,pj/N}\quad. \tag{11.26}$$

Putting these into (11.24) gives

$$\frac{1}{N}\sum_p \tilde{u}_p\, e^{-i\,2\pi\,pj/N}\left[e^{+i\,2\pi\,p/N} + e^{-i\,2\pi\,p/N} - 2\right] = \frac{h^2}{N}\sum_p \tilde{\rho}_p\, e^{-i\,2\pi\,pj/N}\quad. \tag{11.27}$$

Equating the bits inside the sum we have

$$\tilde{u}_p = \frac{h^2\,\tilde{\rho}_p}{\left[e^{+i\,2\pi\,p/N} + e^{-i\,2\pi\,p/N} - 2\right]}\quad, \tag{11.28}$$

and since

$$e^{i\,\theta} = \cos\theta + i\,\sin\theta\,, \tag{11.29}$$

the denominator simplifies to

$$\left[e^{+i\,2\pi\,p/N} + e^{-i\,2\pi\,p/N} - 2\right] = \left[e^{+i\,\pi\,p/N} - e^{-i\,\pi\,p/N}\right]^2 = (2i)^2\,\sin^2(\pi\,p/N)\quad, \tag{11.30}$$

giving

$$\tilde{u}_p = -\frac{h^2\,\tilde{\rho}_p}{4\sin^2(\pi\,p/N)} \tag{11.31}$$

Notice that the 'monopole' $\tilde{u}_0$ diverges if $\tilde{\rho}_0 \neq 0$. (Note that $\tilde{\rho}_0$ is the average source density on the finite, periodically repeated, domain.) Physically, potential is only defined to within a constant so we can safely set $\tilde{\rho}_0 = 0$ to make the potential in real space zero when for a system where all the 'mass' is uniformly spread out. We then obtain the solution in coordinate space by taking the inverse DFT of (11.31)

$$u_j = -\frac{h^2}{4\,N}\sum_p \frac{\tilde{\rho}_p}{\sin^2(\pi\,p/N)}\, e^{-i\,2\pi\,pkj/N}\quad. \tag{11.32}$$

In general the DFTs are replaced by black-box FFT routines from standard libraries and the algorithm can be summarised as

- Generate source lattice ρ_j.
- FFT source lattice $\rho_j \to \tilde{\rho}_p$.
- Solve for $\tilde{u}_p$ (and remove monopole).
- Inverse FFT $\tilde{u}_p \to u_j$.

The method can be easily extended to higher dimensions, e.g., 2D or 3D lattice.

11.5. Fast Fourier Transforms - FFTs (non-examinable)

The FFT algorithm for doing a discrete Fourier transform in $\mathcal{O}(N \log N)$ operation was described by Daniel & Lanczos in 1942 but rediscovered and popularised in the computer age by Cooley & Tukey in 1965.

The trick is to split the DFT into separate sums involving only even (e) and odd (o) indices

$$\begin{aligned} \tilde{f}_k = \sum_{n=0}^{N-1} f_n \, e^{i\,2\pi kn/N} &= \sum_{n=0}^{N/2-1} f_{2n} \, e^{i\,2\pi k 2n/N} + \sum_{n=0}^{N/2-1} f_{2n+1} \, e^{i\,2\pi k(2n+1)/N} \,, \\ &= \sum_{n=0}^{N/2-1} f_{2n} \, e^{i\,2\pi kn/(N/2)} + e^{i\,2\pi k/N} \sum_{n=0}^{N/2-1} f_{2n+1} \, e^{i\,2\pi kn/(N/2)} \,, \qquad (11.33) \\ &= \tilde{f}^e_k + e^{i\,2\pi k/N} \, \tilde{f}^o_k \,. \end{aligned}$$

(11.34)

The two functions $\tilde{f}^e_k$ and $\tilde{f}^o_k$ are then periodic over $N/2$ i.e. half the range of the original function. This splitting requires N operations. We can iterate this process and split both $\tilde{f}^e_k$ and $\tilde{f}^o_k$ into even and odd parts themselves

$$\tilde{f}^e_k = \tilde{f}^{ee}_k + e^{i\,2\pi k/N} \, \tilde{f}^{eo}_k \,, \qquad (11.35)$$

and

$$\tilde{f}^o_k = \tilde{f}^{oe}_k + e^{i\,2\pi k/N} \, \tilde{f}^{oo}_k \,, \qquad (11.36)$$

with the new functions being periodic over the range $N/4$. The second splitting requires $N/2$ operations.

Assuming $N = 2^p$ (i.e. even), after p even/odd splitting we are left with N functions of period 1 which are each trivially FT'd; the FT of each is the same as itself

$$\tilde{f}^{eooeeoeoeo\ldots e}_k = f_n \,, \qquad (11.37)$$

for some n.

All we have left to do is identify which combination of $eoeeo\ldots e$ corresponds to each n - this can be done by assigning the binary value $e \equiv 0$ and $o \equiv 1$ to each element in the sequence and then **reversing** the sequence. The series of 1s and 0s obtained is a binary representation of n.

The complete method requires

$$N \times p = \frac{N \log N}{\log 2} \sim \mathcal{O}(N \log N) \,, \qquad (11.38)$$

operations. As an example consider a problem with $N \sim 10^4$ samples, in this case the FFT is faster than the naive DFT by a factor of about 750.