Chapter 1

Course Description

Welcome to the Computational Physics course. 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 that 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 that you will acquire will be relevant for your future work in theoretical and experimental physics, in mathematical modelling, and in broader quantitative settings in industry.

In the lectures we will focus in turn on each of the most common types of numerical problems that you might encounter, explaining the issues that need to be dealt with in order to approach such a problem with a computer, and outlining the basic algorithms available for dealing with each class of problem. The best way to learn these algorithms is to code them yourself, so we will also provide some relatively simple problems that you can use to test your ability to implement the algorithms. Combining the lectures and lecture notes with your own efforts at coding the algorithms should prepare you well for the assignment, project and exam—and for your future work in any field requiring quantitative skills.

1.1 Overview

The course will cover the following topics.

- Basics of numerical programming: Variable types, floating point and integer arithmetic, numerical accuracy and error propagation.
- Algebraic equations: Linear matrix algebra; direct and iterative methods for solution of linear equations. Non-linear equations; iterative methods for one or more variables.
- Interpolation: How to represent a functional form in a computer, given a finite number of data points.
- Numerical calculus: Estimating first and higher order derivatives. Numerical integration with deterministic rules. Evaluation of improper and badly-behaved integrals.
- Fourier transform methods: Fast Fourier transforms and their use in data processing.
- Ordinary differential equations: Theory and analysis of the accuracy and stability of numerical methods for solving differential equations. Finite-difference methods. Solution of initial-value and boundary-value problems using finite-difference methods. Integration using ODE methods.
- Random numbers: How random number generators work. How to use them to generate non-uniform random number distributions. Monte Carlo methods for integration. Minimisation and simulation of equilibria of ensembles of particles.

- Optimisation methods: Finding the minima and maxima of general multi-dimensional functions.
- Partial Differential Equations. Solution of initial-value elliptic, parabolic and hyperbolic partial differential equations using finite difference methods.

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

- Identify fundamental problem types in computational physics (root-finding, interpolation, matrix inversion, optimisation, integration, differential equations, etc.).
- Find the roots of an algebraic equation numerically.
- Invert matrices and solve matrix equations numerically.
- Interpolate regularly-spaced data in one or more dimensions.
- Evaluate definite integrals numerically, including improper examples.
- Select suitable random number generators and use them for simulation and integration.
- Know how Fourier transform methods can be used for data processing.
- Design and implement Monte Carlo methods to simulate statistical physics problems.
- Find the minimum of a given function for a number of input variables.
- Select, assess (in terms of accuracy, stability and efficiency) and implement finite difference methods to solve differential equations in physics.
- Formulate and solve initial value and boundary value problems.
- Use any of the above techniques to design and write computer programs that are easy for human beings to read, run correctly on computers, and solve physics problems.

Example problems will be drawn from a wide range of areas of physics such as mechanics, fluid dynamics, electromagnetism, quantum physics, statistical physics, climate physics, astrophysics and high energy physics.

1.2 Course components

The course comprises the following components:

- A live introductory lecture in Teams outlining the organisation of the course.
- A series of video lectures, covering the theory of numerical methods and their applications in physics. These will be accompanied by lecture notes on the same material. Table 1.1 shows the video lecture schedule.
- Two additional seminars on "Computational Physics in Action", to put the course material into context.
- A series of non-assessed problem sheets each week, featuring problems that will help you understand the lecture material.
- In Weeks 4–11 of term, two consultation sessions each week. These will be used to discuss a specific problem from the previous week's problem sheet. They will also allow more general discussions on the course material. Later in the term, they can include discussion on the assignment and your project.
- An assignment, with a number of questions that you will need to answer using techniques learnt from the lectures.
- A project, where you will study one particular computational topic in depth.
- A two-hour written exam.

The *consultation sessions* will be held using Teams on **Monday and Wednesday** mornings during Weeks 4–11 of term. On Mondays these will be from **10am to 1pm** and on Wednesdays from **9am to 12 noon**. Each demonstrator group will be scheduled for one hour in each of these periods.

Table 1.1: Schedule of material covered in video lectures

Topic	Week
Introduction to numerical calculations	1
Algebraic equations	2
Interpolation and numerical calculus	3
Numerical Fourier transforms	4
Ordinary differential equations	4
Ordinary differential equations (cont)	5
Random numbers and Monte Carlo methods	6
Minimisation and maximisation	6
Partial Differential Equations	7
Computational Physics in Action	8-10
Revision lecture	*TBC*

1.3 Assessment

The course will be assessed in three units:

- **Assignment** (assessed problems): To be submitted by **12 noon** on Monday of Week 7, i.e. **Monday, 16th November 2020**. This will be worth **15%** of the marks for the course. There will be no choice of questions in the assignment.
- **Project**: To be submitted by **12 noon** on the first Monday after the last day of term, i.e. **Monday, 21st December 2020**. There will be a choice of several topics for this. The project will be worth **45%** of your final grade for the course.
- **Written Exam**: To be held at **10 am** on the first Monday of Term 2, i.e. **Monday, 11th January 2021**. The exam is worth **40%** of the marks. This exam will test your understanding of the numerical methods and algorithms. It will not involve writing real code, but you will be expected to explain some methods using pseudocode. The exam will cover all material from the lectures, and all examinable material from these notes. The exam will be held remotely although the exact format is not yet defined.

See table 1.2 for a summary of the submission deadlines for the assignment and project.

Table 1.2: Assessment deadlines.

Day and date	Time	Topic
Monday November 16th	**12 noon**	**Assignment submission deadline**
Monday December 21th	**12 noon**	**Project submission deadline**
Monday January 11th	**10 am**	**Exam**

The notes will include both examinable and non-examinable material.

Where we include additional, non-examinable material in these notes, we will mark it like this, with a blue mark in the margin. This material will not be tested in the final exam, but you may find it helpful for your project or for expanding your understanding of the examinable material.

You will work individually on the coursework for both the assignment and the project, submitting your own written answers (assignment), report (project) and source code for each.

The assignment will be a series of assessed problems, each designed to get you familiar with the core methods discussed in the first few lectures. Further details on submission and assessment will be available in a separate document, later on in the course.

For your project, you will be able to choose between several topics. The options will be reviewed briefly in the lectures and relatively detailed scripts will be provided for each. Your report should cover the physics problem addressed, the numerical methods used to solve it, the results of your investigation, their analysis and interpretation and your conclusions. Further details on the projects on offer will be available in November.

When writing up your assignment and project reports, remember that when working with a computer, you will rarely be working in isolation; numerical results are often worthless if you cannot convince other people that your code is working correctly. You need to write and present your code in a professional way, keeping in mind that you are coding and writing for your fellow human beings, as well as a computer. **This means that you must comment your code extensively!** Marks will be awarded for good documentation.

All coursework will be assessed by both a first and second marker.

In addition to the assessed work, we will provide some problem sheets, which will not be assessed. They will contain some theoretical and some programming problems to help you better understand the lecture material and get started on the assignment and project problems afterwards, along with a few examples of applications to physics. These problems will be relevant to the exam paper and to the projects, and form a key part of the course material. Please do not devote all your time in practicals to the assessed work, but use the problems that we supply to help understand the algorithms outlined in the lectures. This will ultimately make your experience with the assessed work a bit more pleasant, as the problems sheets are also designed to 'ease you in' to the assignment and projects.

1.4 Course website

The course is **Computational Physics (2020-21)** on **Blackboard**: https://bb.imperial.ac.uk.

Lecture notes, course handouts, problem and solution sheets, and reference material for coding will be made available there. Lecture notes and videos will be uploaded on a weekly basis, usually before Monday for the material covered in the coming week. Problem sheets will also be uploaded weekly, with solution sheets being released two weeks later. The assignment (consisting of the answers to the questions along with the programs that you wrote in order to be able to answer them) and project (consisting of the report and the associated programs) should be submitted electronically though the course website on Blackboard Learn. Coursework will be checked by anti-plagiarism software; TurnItIn for the written report and a code similarity checker for the source code.

1.5 Consultation sessions

Unlike first- and second-year computing, you will be working much more on your own and in a less structured way. The consultation sessions in Teams on Monday and Wednesday mornings are 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 syntax errors on your own. You are not *required* to call in for your practical sessions; the teaching assistants 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.

The most commonly-used platform for practical work is Python, but any language which can be run by the assessors is acceptable; see below. Given the limitations on using the PCs in the computing suite, you are encouraged to use your own laptop and programming environment, or use College computing resouces remotely.

Please note that the teaching assistants are not there to provide technical help for setting up your laptop, but rather to assist on your understanding the concepts and algorithms that are being taught in the course.

1.6 Programming language

The choice of programming language that you use for the assignment and project is your own (subject to a few caveats). The main requirement is that it be a low-level language which allows you to directly access the individual values held in memory for arrays and matrices. We will discuss this in the first few lectures. We would imagine that the Imperial MSci/BSc students on this course will prefer to use Python, as this is what you learned in years 1 & 2. If you wish to work in another language, such as C, C++, or FORTRAN, this is welcome but you must check yourself that the software is available to you, and that you are able to submit it in a way that other will find easy to compile and run. In particular, any graphical output must be straightforward for someone else to produce. One important condition is that, unless indicated otherwise in the project script, you must implement numerical methods yourself rather than using a package (e.g. an ODE solver or function minimiser) provided by a maths library or built into the 'language'. The course is not a programming course as such, but we will provide some advice to help you improve your overall programming style. The quality of your programming will influence the grade that you achieve in the assessment of the projects at a minor level.

For graphs and tables you can use whatever tool you are familiar with to make effective plots for your assignment and/or report, e.g., matplotlib, (c)tioga, Matlab, GnuPlot, Origin, ROOT, Excel. (Of course, as with any other form of academic writing, make sure axes are labelled and that multiple curves are distinguishable and labelled by a legend or in the figure caption.)

1.7 Plagiarism

All the work that you submit for assessment—the prose in the report, the code, the results and plots—must be your own. Any help from your colleagues or others must be clearly acknowledged in your submitted work. Occurrences of plagiarism are taken very seriously and can have dire consequences. See the Departmental Policy on Plagiarism. We are aware that reports and code from previous years have been posted online by graduated students, and are sometimes passed directly on by earlier cohorts. These sources are in the plagiarism detection systems. Do not risk copying from them! Unfortunately, a small minority of students have been caught out in recent years. We hope that this will not happen this year.

Please do not make your work for this course publicly accessible during or after the course via GitHub, BitBucket or similar; if you want to share it with a prospective employer, please give them private access to your repository.

1.8 Contact

The primary forum for questions and discussions on the course content will be the scheduled consultation sessions (see above) in Teams. You will be divided into groups of around ten students and assigned to a course teaching assistant; this assignment and the assistants' names and contact details will be provided later in the term. At these sessions, the teaching assistants will be available to answer questions. At least one of the lecturers will normally be available online during these sessions to answer questions about the course in general.

If you would like to raise a question which could be of interest to other students, please use the Discussion Board feature on Blackboard; this is our secondary forum for questions. If you identify errors or other problems with the course material, or have general comments or suggestions, please again post this on the forum too. The reason we ask you to do this rather than just emailing us the question is that it allows other students to also benefit from the answers that we give—and in many cases, it allows you to get an even more rapid response from another person in the class than one of us is able to provide.

We will also be holding online office hours during the weeks when we are teaching. For Dr Scott, these will be Tuesdays at 10 am from Week 3 onwards, and for Prof Dauncey, they will be Thursdays at 10 am from Week 2 onwards.

1.9 Literature

This course is bespoke and does not exactly follow any one textbook. However the following textbooks are strongly recommended:

- 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. There are versions for different languages, but any of them are appropriate, since the logic of the algorithms does not depend on the language.
- C. Gerald and P. Wheatley, *Applied Numerical Analysis*, International Edition, 7th edition, (Pearson, 2004), ISBN 0-321-19019-X.
- 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:

- N. J. Giordano and H. Nakanishi. *Computational Physics*, Second Edition, (Pearson 2006), ISBN 0-13-146990-8. Good as background for some projects.
- E. Süli and D. Mayers. *An introduction to Numerical Analysis*, Cambridge University Press, ISBN-13 978-0-521-00794-8. A very formal & advanced book.
- E. W. Weisstein, *MathWorld–A Wolfram Web Resource.* http://mathworld.wolfram.com. Mathematics reference.
- 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.
- M. Galassi et. al., *GSL - GNU Scientific Library.* Recommended numerical library for those programming in C++ or C. It is installed on the PCs in the UG Computer Suite. Source code and full documentation available at http://www.gnu.org/software/gsl/.

You will find many online sources 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 to reference any sources (books, reports, websites, etc.) that you use in writing your assignment and/or project report.

Acknowledgements

These lecture notes are adapted from previous course notes developed by C. Contaldi, U. Egede, R.J. Kingham, E. van Sebille, P. Scott and Y. Uchida.

Chapter 2

Introduction to Numerical Calculations

Outline of Section

- How information is stored in a computer
- A review of the Taylor series
- The nature of computational errors
- Using natural units

2.1 Numerical Formats and Types

Here we discuss how computers store information at the hardware level. Computer memory is formed, in general, as a succession of "bits" that can take one of two values: "off" and "on", or equivalently, "0" and "1". Since only a finite number of bits are available to represent the information, a way of representing this information efficiently has to be chosen.

And whilst "information" can take many forms, ultimately it is numbers that are handled inside a computer—any other information, such as letters of an alphabet, are converted into numbers using what are effectively look-up tables such as the Unicode Standard.

We say "hardware-level" numerical formats, since it is these that are directly manipulated by the CPUs that access the memory, and the entire system (CPU and memory) can be optimised for speed and efficiency for the basic mathematical operations that are needed, such as addition, subtraction, multiplication and division. Other numerical formats exist which are built in software, at a higher level than the hardware—these include those that manipulate rational numbers, internally representing them using pairs of integers.

The two basic types of numbers which are normally stored directly in computers are *integers* and *real* numbers. These are the ones most used in this course and each is described below.

Integer numbers

Integers are relatively simple—a binary representation of any integer, optionally with its sign, carries all the information possessed by the original number, so it is only the number of bits (called the *precision* or *width*) that are allocated to store the number that can vary. Modern systems often use 32-bit or 64-bit integers, with these limiting the size of the integers that can be stored. How to access these different integer *types*, and the number of bits that are available, are hardware- and language-dependent. In particular, note that python3 dynamically allocates memory to provide as many bits as required for an integer.

It is up to the user to make sure enough bits are being used to their specific purposes. An unsigned integer is stored in terms of bits as a straightforward binary number. Specifically, N bits can store unsigned integers from 0 to $2^N - 1$. Signed integers use one bit to indicate the sign, so the non-negative values they can represent only go from 0 to $2^{N-1} - 1$. However, they can also go down to -2^{N-1}, which means for N bits, there are the same number of signed integers as unsigned, i.e. 2^N in total for both. The negative values are represented in bits using "two's complement", which means taking the bit pattern of the equivalent unsigned value, inverting all the bits (so all 0's go to 1, and all 1's go to 0) and adding 1. For example, four binary bits gives $2^4 = 16$ possible values and allows unsigned decimal integers from 0 to $2^4 - 1 = 15$ or signed decimal integers from $-2^3 = -8$ to $2^3 - 1 = 7$ to be represented. The table shows all the possible 16 values in binary and decimal representations.

Binary		Unsigned decimal		Signed decimal
0000	=	0	=	0
0001	=	1	=	1
0010	=	2	=	2
0011	=	3	=	3
0100	=	4	=	4
0101	=	5	=	5
0110	=	6	=	6
0111	=	7	=	7
1000	=	8	=	−8
1001	=	9	=	−7
1010	=	10	=	−6
1011	=	11	=	−5
1100	=	12	=	−4
1101	=	13	=	−3
1110	=	14	=	−2
1111	=	15	=	−1

Real numbers

Storing a real number is a much more complicated affair, and the computing world has settled on the IEEE-754 standard, which describes how "floating-point" numbers can be represented in a computer.

The concept of floating-point representations of real numbers is similar to the exponential notation that a compact way to write down numbers with a certain number of significant digits and an arbitrary magnitude: for example 1.541×10^{-8} or -9.34031×10^{22}, which would require much more space to write out in full.

Here too, 32 and 64 bits are normally used to store numbers in single and double precision respectively with the bits being allocated to the *sign*, *mantissa* and the *exponent* according to some agreed standard, where the numbers are stored in base 2.

	Sign	Mantissa	Exponent
Single (32 bits)	1 bit	23 bits	8 bits
Double (64 bits)	1 bit	52 bits	11 bits

(2.1)

If we consider the mantissa as a binary value with one bit before the point, e.g. 1.01101001, then the exponent is stored as an 8-bit binary value ranging between −126 and +127 for single precision and an 11-bit binary value ranging from −1022 to +1023 for double precision. The number of bits allocated to the mantissa affects the number of significant digits that can be represented. Thus the range of numbers allowed are roughly $10^{\pm 38}$ and $10^{\pm 308}$ for single and double precision respectively; i.e. $2^{\pm 127} \sim 10^{\pm 38}$ and $2^{\pm 1023} \sim 10^{\pm 308}$. Numbers smaller or greater than these will cause an **underflow** or **overflow** respectively. Underflows usually result in the result being set to zero. Overflows are replaced by the symbol `Inf` in some languages. The IEEE standard uses `NaN`, i.e., Not-a-Number, to replace 0/0, $\sqrt{-1}$, etc.

It important to be aware of how floating-point numbers are represented on computers, how computers operate on them to perform subtraction, addition, multiplication and division, and what the consequences are for usable accuracy, rounding and truncation, including the concept of "machine epsilon" (or machine accuracy) which is a measure of the relative error introduced by rounding an arbitrary real number to one of the representable floating point values. What is given here is a mere introduction, and more examples are given in the lectures.

One of the best places to read more about that is in "What Every Computer Scientist Should Know About Floating-Point Arithmetic", by David Goldberg, 1991. Online full text is available on the Library web site.

2.2 Review of Taylor series

Functions and their derivatives are often approximated as truncated series. The function $f(x)$ can be expanded around the point a to n^{th} order as

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + ... + \frac{1}{n!}f^{n\prime}(a)(x-a)^n, \tag{2.2}$$

where $f^{n\prime}(a)$ 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

$$\boxed{f(x) = \sum_{i=0}^{i=n} \frac{1}{i!}f^{i\prime}(a)(x-a)^i + R_n(x),} \quad \text{where } R_n(x) \equiv \sum_{i=n+1}^{i=\infty} \frac{1}{i!}f^{i\prime}(a)(x-a)^i \,. \tag{2.3}$$

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

$$\boxed{R_n(x) = \frac{1}{(n+1)!}f^{n+1\prime}(\xi)(x-a)^{n+1} \qquad \text{where} \qquad (a \le \xi \le x)} \tag{2.4}$$

Note the remainder looks like the $(n+1)^{\text{th}}$ term of the expansion but the derivative is evaluated at the point ξ, not x. 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).

To make use of this in practice, 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, the Taylor series can be written as

$$f(x+h) \approx f(x) + f'(x)\,h + \frac{1}{2}f''(x)\,h^2 + ... + \frac{1}{n!}f^{n\prime}(x)\,h^n \,. \tag{2.5}$$

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

$$f(x+h, y+k) \approx \sum_{i=0}^{i=n} \frac{1}{i!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^i f(x,y) \tag{2.6}$$

where the differential operator $(\ldots)^i$ can understood by expanding as a binomial expansion. E.g. for $i = 2$

$$\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^2 = h^2\frac{\partial^2}{\partial x^2} + 2hk\frac{\partial^2}{\partial x \partial y} + k^2\frac{\partial^2}{\partial y^2} \tag{2.7}$$

This operates on the function and the result is then evaluated at (x, y). The remainder term is similar to Eq. (2.4), in that it involves $(n+1)^{\text{th}}$ order partial derivatives of f evaluated at the point (ξ, η), where $x \le \xi \le x+h$ and $y \le \eta \le y+k$. Eq. (2.6) can be generalised to more independent variables by adding the relevant partial derivatives (e.g. $\partial/\partial z$ or $\partial/\partial t$) into the $(\ldots)^i$ term and using a multinomial expansion.

2.3 Numerical Accuracy and Errors

There are many sources which lead to a loss of accuracy and other errors that are inevitably introduced when performing mathematical calculations using numbers with finite representations. Here we describe the basic categories that these fall into.

Truncation errors

The Taylor series discussed above cannot be used with an infinite number of terms so if we discard higher terms, i.e. truncate the series, it will mean some inaccuracy which is called a truncation error. However, this can also happen even without us knowing. Even the most precise computer often evaluates functions internally using approximations. These approximations are typically in the form of a series and so can also lead to truncation errors. Most programming languages use some form of power expansion to approximate standard mathematical functions. For example, the sine function is often calculated using a series. The Taylor expansion of $\sin(x)$ around $x = 0$ is

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \ldots . \tag{2.8}$$

The computer will typically keep a reasonable number of terms so as to keep the truncation error comparable with the representation accuracy, but it cannot be eliminated completely.

Round-off errors

Even in the absence of any truncation error a round-off error is inherent to all digital computers. This is due to the fact that the 'floating point' format used by computers stores any number with only a finite precision. The numbers are rounded off to the closest value at the computer's precision. With the `Python` programming language, the intrinsic (built in) floating point numerical type '**float**' is accurate to at least 15 significant decimal digits of precision. (With the 'right' number, it can be accurate to 17 significant figures.) In `C++` there is both 'single' precision (accurate to 6–8 significant figures) and 'double' precision (accurate to 15–17 significant figures; same as Python's 'float'). A perhaps surprising example of the effect of rounding error is subtracting 1/9 away from 1, nine times, e.g., in `Python`;

```
n=9;  x=1.0; dx=x/n
for i in range(n):
    x=x-dx
print x
```

Rather than obtaining zero, the output (on my Linux machine) is **−1.665e−16** to 3 d.p. This round-off phenomenon occurs because the value $1/9 = 0.111\ldots$ is only stored to about 16 significant figures.

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 quickly with time (or whatever the independent variable is). Chaotic systems are by their nature particularly sensitive with respect to initial conditions.

Propagation errors

Propagation errors are closely related to initial condition errors. This error is what 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.

If the propagated error increases 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.

2.4 Natural units

It is very important to use sensible units when integrating systems numerically due to the limited range of numbers a computer can deal with. It also helps in understanding the dynamics if we can scale the variables relative to appropriate characteristic units of measure that encapsulate important physical properties of the system. This in turn greatly aids debugging.

The scaling process removes SI units from the variables (leaving them in 'natural' or 'dimensionless' units) and for this reason is also known as equation nondimensionalisation. Thus **natural units**, **scaling** and **nondimensionalisation** all refer to the same concept.

All the independent variables should be scaled. Scaling of the dependent variable is optional, but often insightful. Initial values or asymptotic values can be good choices. For example, consider the dimensionful system for exponential decay with a source

$$\frac{\mathrm{d}y}{\mathrm{d}t} + \alpha\, y \;=\; g \tag{2.9}$$

where α is a decay rate and g a constant growth/source term. Writing the dependent variable SI dimensions as $[y]$, then $[\alpha] = \mathrm{s}^{-1}$ and $[g] = [y]/\mathrm{s}$. We can then define a scaled dimensionless time variable $\hat{t} = \alpha\, t$ such that [1]

$$\frac{\mathrm{d}}{\mathrm{d}t} = \frac{\mathrm{d}\hat{t}}{\mathrm{d}t}\frac{\mathrm{d}}{\mathrm{d}\hat{t}} = \alpha\frac{\mathrm{d}}{\mathrm{d}\hat{t}} \tag{2.10}$$

and the equation becomes

$$\frac{\mathrm{d}y}{\mathrm{d}\hat{t}} + y \;=\; \frac{g}{\alpha} \equiv G, \tag{2.11}$$

where $[G] = [y]$. We could choose to go further and scale y too. Given that the steady-state solution (i.e. where the time derivative is zero) is $y_s = g/\alpha = G$, a good choice is $\hat{y} = y/y_s = y/G = \alpha y/g$, which makes $\hat{y}$ also dimensionless, and the equations becomes

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}\hat{t}} + \hat{y} \;=\; 1. \tag{2.12}$$

This is the original equation but with time scaled to the exponential decay time and the amplitude scaled to the final, steady-state value.

Changing to units where the independent variable is dimensionless means we do not have to worry about units in our integration of the above equation. Specifically

- Step-size h; in these natural units 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 $\hat{t} \sim 1$ so we know that integrating from $\hat{t} = 0$ to $\hat{t}_f$ (where $\hat{t}_f \sim$ a few) will cover the entire dynamic range of interest.

One thing we *must* remember is to put the SI units back in when we present our results, e.g., in plots of the integrated solution, or explicitly state what natural units are (to give meaning to the numbers). For example, in a plot of $\hat{y}(\,\hat{t}\,)$ would label the horizontal axis in units of $[1/\alpha]$ and the vertical in units of $[g/\alpha]$.

[1] There are many notations for scaled variables. Common alternatives include using a prime (e.g. t'), a tilde, (e.g $\tilde{t}$, which unfortunately clashes with our use of '$\sim$' to distinguish numerical approximations of things (derivatives, solutions of DE) from exact versions), using Greek symbols (e.g. τ) and using capitalised symbols (e.g. T).

Chapter 3

Solving Algebraic Equations

Outline of Section

- Introduction
- Linear Algebraic Equations
- Direct Methods: Gaussian and Gauss-Jordan elimination
- Direct Methods: LU Decomposition
- Iterative Methods: general concepts
- Iterative Methods: Jacobi and Gauss-Seidel
- Eigensystems
- Non-linear algebraic equations of one variable
- Non-linear algebraic equations of multiple variables

3.1 Introduction

In this section we will look at solving simultaneous algebraic equations. By "algebraic", we mean equations which do not involve derivatives; those are handled later in the course.

Consider the case of one unknown x where we generally need one equation to define the solution. By gathering all terms on the left hand side, any equation can be written in the form

$$f(x) = 0 \tag{3.1}$$

For N unknowns x_i, where $i = 0, N-1$, we will often use the notation where the x_i are written as a vector $\vec{x}$. For N unknowns we generally need N equations which again can be written as

$$f_i(\vec{x}) = 0 \qquad \text{where} \qquad i = 0, N-1 \tag{3.2}$$

Hence, all algebraic problems can be expressed as function root-finding problems.

This chapter first considers linear equations and later non-linear equations. Linear equations have terms with powers of at most x_i^1. The general one variable case can be written as $Ax = b$ so

$$f(x) = Ax - b = 0 \qquad \text{giving} \qquad x = \frac{b}{A} = A^{-1}b \tag{3.3}$$

There is one and only one solution, except when $A = 0$. For N variables and M equations, the general case is

$$\begin{aligned} A_{00}\,x_0 + A_{01}\,x_1 + A_{02}\,x_2 + ... + A_{0\,N-1}\,x_{N-1} &= b_1, \\ A_{10}\,x_0 + A_{11}\,x_1 + A_{12}\,x_2 + ... + A_{1\,N-1}\,x_{N-1} &= b_2\,, \\ ... &= ..., \\ A_{M-1\,0}\,x_0 + A_{M-1\,1}\,x_1 + A_{M-1\,2}\,x_2 + ... + A_{M-1\,N-1}\,x_{N-1} &= b_{M-1}, \end{aligned} \tag{3.4}$$

which can be written as the matrix operation

$$\sum_j A_{ij}x_j = b_i \qquad \text{or} \qquad \mathbf{A}\cdot\vec{x} = \vec{b}, \tag{3.5}$$

where $\mathbf{A}$ is an $M \times N$ matrix, $\vec{x}$ is a vector with N components and $\vec{b}$ is a vector with M components. Any coefficient A_{ij} in the above simultaneous equations becomes the element of the matrix located at row i (i.e. the first subscript) and column j (the second 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.

3.2 General considerations for linear equations

Before diving into various new methods it is worth discussing some general points about solving matrix equations.

The need to solve a matrix equation such as (3.5) arises frequently in many numerical methods. In section 8.6 we will see them in implicit finite difference methods for solving sets of coupled linear ODEs. In section 12 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 9) and minimisation of functions (section 11). Of course matrix equations arise directly in many physical problems too, such as quantum mechanics and classical mechanics.

Consider the inhomogeneous matrix equation $\mathbf{A} \cdot \vec{x} = \vec{b}$. In the following, we will focus on the case where $M = N$, i.e., $\mathbf{A}$ is a square matrix. 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 fewer equations than unknowns (i.e., $M < N$) and there may not be a solution (or not a unique solution). This will be evident in the matrix $\mathbf{A}$ being singular (some eigenvalues are zero) and having a vanishing determinant. An approximate solution can be found using Singular Value Decomposition (SVD), which is 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 the equations most closely. Finally, if the matrix equation is homogeneous, i.e., $\mathbf{A} \cdot \vec{x} = \vec{0}$ (where $\mathbf{A}$ is an $N \times N$ matrix), then the solution is non-trivial (i.e. not $\vec{x} = 0$) only if $\det \mathbf{A} = 0$.

A particular feature of the $M = N$ case is that the system can be extended to N unknown solutions for N right-hand sides. Consider two equations

$$\mathbf{A} \cdot \vec{x} = \vec{b} \qquad \text{and} \qquad \mathbf{A} \cdot \vec{y} = \vec{c} \tag{3.6}$$

We can form a two-column matrix from $\vec{x}$ and $\vec{y}$, usually written as $(\vec{x}|\vec{y})$ and do the same for $\vec{b}$ and $\vec{c}$, written as $\left(\vec{b}|\vec{c}\right)$. Both equations can then be written as one equation

$$\mathbf{A} \cdot (\vec{x}|\vec{y}) = (\vec{b}|\vec{c}) \tag{3.7}$$

Extending this to N vectors gives an $N \times N$ matrix $\mathbf{X} = (\vec{x}|\vec{y}|\vec{z}| \dots)$ to be solved for, and another known matrix $\mathbf{B} = \left(\vec{b}|\vec{c}|\vec{d}| \dots \right)$. The equation can then be written as

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B}, \tag{3.8}$$

Each column of $\mathbf{X}$ is one of the vectors like $\vec{x}$ of the N equations (where $0 \le i \le N-1$). Similarly, each column of $\mathbf{B}$ is a right hand-side vector, like $\vec{b}$, of one of the equations. If we can solve the original equation for $\vec{x}$, we can repeat this solution N times to find all the columns in $\mathbf{X}$. In addition, if we can solve for $\mathbf{X}$ then we can also use the method to find the inverse of a matrix since

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}. \tag{3.9}$$

Hence, setting $\mathbf{B}$ to the identity matrix, so the matrix equations become $\mathbf{A} \cdot \mathbf{X} = \mathbf{I}$, and solving yields as the solution $\mathbf{X} = \mathbf{A}^{-1}$.

We will also briefly look at eigenvalue equations of a matrix, which are of the form

$$\mathbf{A} \cdot \vec{v} = \lambda \vec{v}, \tag{3.10}$$

where λ is the eigenvalue to be found and $\vec{v}$ is the eigenvector. Eigenvalues are important for convergence in some equation-solving methods. The need to find eigenvalues of a matrix as given by (3.10) is also a common task both in numerical methods and in physics problems. We will encounter one such case in the stability analysis of finite difference methods for coupled ODEs in section 7.5.

3.3 Gaussian and Gauss-Jordan elimination

Direct methods estimate the matrix without using approximations, so if there were no calculational errors due to floating point rounding, etc., the result would be exact.

All methods may involve manipulating the matrix to shift around elements so that the system can be more easily solved. These manipulations involve swapping rows and scaling rows; these 'row-swapping' and 'row-scaling' operations are known as 'pivoting', and the best approach can in general be tricky to spot.

Actual methods for solving the linear equations can involve finding the inverse of $\mathbf{A}$ explicitly, such as the **cofactor** method. Others, such as **Gaussian elimination** and **Gauss-Jordan elimination** are basically what you would do if you needed to solve the matrix equation using pen and paper. These work by adding (or subtracting) multiples of different rows to each other (in addition to any pivoting needed).

For Gaussian elimination, one manipulates an "augmented" matrix formed by added another column to the right of $\mathbf{A}$ made from $\vec{b}$, which gives an $(N+1)\times N$ matrix $\left(\mathbf{A}|\vec{b}\right)$. By reducing the $\mathbf{A}$ part of this augmented matrix to a unit matrix, the extra column will become the solution for $\vec{x}$, i.e. it is $\left(\mathbf{I}|\mathbf{A}^{-1}\vec{b}\right) = (\mathbf{I}|\vec{x})$. For Gauss-Jordan elimination one manipulates the augmented matrix formed by $(\mathbf{A}|\mathbf{I})$, until one has $(\mathbf{I}|\mathbf{C})$, which yields the inverse of $\mathbf{A}$ since it turns out that $\mathbf{C} = \mathbf{A^{-1}}$. In both methods, the entries in $\mathbf{A}$ inform your (or your code's) choices about what manipulations to perform, and $\vec{b}$ (in Gaussian elimination) or $\mathbf{I}$ (in Gauss-Jordan elimination) 'shadow' those moves, and thereby become transformed to the answer.

3.4 LU Decomposition

LU decomposition is another direct method. It uses lower ($\mathbf{L}$) and upper ($\mathbf{U}$) diagonal matrices, where a lower diagonal matrix has all elements above the diagonal equal to zero. Similarly, the upper diagonal matrix has all elements below the diagonal equal to zero.

First, consider the case where the matrix $\mathbf{A}$ happens to be a lower diagonal matrix, e.g.

$$\mathbf{A}\cdot\vec{x} = \begin{pmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}. \tag{3.11}$$

Writing the first equation out explicitly gives

$$L_{00}x_0 = b_0 \qquad \text{so} \qquad x_0 = \frac{b_0}{L_{00}} \tag{3.12}$$

Given we now know x_0, the second equation gives

$$L_{10}x_0 + L_{11}x_1 = b_1 \qquad \text{so} \qquad x_1 = \frac{b_1 - L_{10}x_0}{L_{11}} \tag{3.13}$$

Similarly

$$L_{20}x_0 + L_{21}x_1 + L_{22}x_2 = b_2 \qquad \text{so} \qquad x_2 = \frac{b_2 - L_{20}x_0 - L_{21}x_1}{L_{22}} \tag{3.14}$$

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. For a general $N \times N$ lower diagonal matrix, the solution is

$$x_0 = \frac{b_0}{L_{00}} \qquad \text{and} \qquad x_{i>0} = \frac{1}{L_{ii}} \left(b_i - \sum_{j=0}^{i-1} L_{ij} x_j \right) , \tag{3.15}$$

where, because we need to ensure the x_j used to calculate later values of x_i have been correctly evaluated, the x_i must be evaluated in the order of increasing i. This is called **forward substitution**.

As might be expected, a similar method can be applied if $\mathbf{A}$ happens to be an upper diagonal matrix.

$$\mathbf{A} \cdot \vec{x} = \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} . \tag{3.16}$$

The critical difference is that we have to start from the last equation, not the first, and evaluate x_i in descending order of i. The equations are then

$$x_{N-1} = \frac{b_{N-1}}{U_{N-1\,N-1}} \qquad \text{and} \qquad x_{i<N-1} = \frac{1}{U_{ii}} \left(b_i - \sum_{j=i+1}^{N-1} U_{ij} x_j \right) . \tag{3.17}$$

This is called **backward substitution**. Of course, it is rare that we would be lucky enough to have a matrix $\mathbf{A}$ which is either a lower or an upper diagonal matrix. However, if we assume that the matrix can be factorised into upper and lower triangular matrices

$$\mathbf{A} \equiv \mathbf{L} \cdot \mathbf{U} \tag{3.18}$$

then we can write the equation as

$$\boxed{\mathbf{A} \cdot \vec{x} = \mathbf{L} \cdot \mathbf{U} \cdot \vec{x} = \mathbf{L} \cdot \vec{y} = \vec{b},} \tag{3.19}$$

where we have defined $\vec{y} = \mathbf{U} \cdot \vec{x}$. We can now carry out a forward substitution to solve $\mathbf{L} \cdot \vec{y} = \vec{b}$ for $\vec{y}$ and then use a backward substitution to solve $\mathbf{U} \cdot \vec{x} = \vec{y}$ for $\vec{x}$. Hence, if we can factorise $\mathbf{A}$, we can solve for $\vec{x}$.

Here we briefly outline how such a decomposition can be performed. Details can be found in the recommended reading material. The matrix equation 3.18 will be something like

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{pmatrix} \tag{3.20}$$

which represents $N \times N$ individual equations, with $N^2 + N$ unknown variables (the L_{ij} and U_{ij} terms, where the diagonals are counted twice). This means that N of these variables can be specified arbitrarily. A common choice (called "Doolittle") chooses to fix the diagonal L_{ii} values to 1. An alternative choice ("Crout") fixes the U_{ii} to 1. Given one of these, the above equations can be solved directly; this solution is called "Crout's algorithm" (not to be confused with the Crout choice for the diagonal; the algorithm can be adapted to either choice). Choosing the Doolittle $L_{ii} = 1$, then for each column $j = 0, 1, 2, \ldots, N-1$ in turn starting from 0, we need first to solve for the U_{ij} in that column; these have $i \leq j$ so we solve for $i = 0, 1, 2, \ldots, j$ using

$$U_{ij} = A_{ij} - \sum_{k=0}^{i-1} L_{ik} U_{kj} \tag{3.21}$$

We must then solve for the L_{ij} in the same column; these have $i \geq j$ but L_{ii} is already known, so for $i = j+1, j+2, \ldots, N-1$ using

$$L_{ij} = \frac{1}{U_{jj}} \left(A_{ij} - \sum_{k=0}^{j-1} L_{ik} U_{kj} \right) . \tag{3.22}$$

When all the U_{ij} and L_{ij} in the column are evaluated, the next column must be processed in the same way. The above procedure must be done in the stated order as it determines all the **L** and **U** values by the time that they are needed in the algorithm. From a computational point of view, the fact that each element of **A** is only used once lends itself to the storage of the newly-found **L** and **U** values in-situ, i.e. in the same matrix that **A** was held in.

The algorithm to carry out LU decomposition could alternatively be done by a process identical to Gaussian elimination but just setting all the elements below the diagonal to zero, i.e. to obtain an upper diagonal matrix. Applying the operations which achieve this to $\left(\mathbf{A}|\vec{b}\right)$ results in $\left(\mathbf{U}|\mathbf{L}^{-1}\vec{b}\right) = (\mathbf{U}|\vec{y})$ and hence gives the vector $\vec{y}$ defined previously. Using the resulting **U** and $\vec{y}$ allows us to solve for $\vec{x}$ using backward substitution as before.

As with many of these algorithms, stability is a critical requirement, and the division by U_{jj} is an operation that can introduce instability if the value of U_{jj} is small. Crout's algorithm *with partial pivoting* is an enhancement of this above procedure that helps make use of the freedom to choose which element becomes the diagonal element U_{jj} to improve the algorithm's stability.

In `Python` it can be carried out using the `SciPy` linear algebra function `scipy.linalg.lu(...)`. For other languages, black-box routines in Linear Algebra packages (e.g. `LINPACK` and `LAPACK`) can be used. The decomposition requires $\mathcal{O}(N^3)$ operations.

A significant benefit of the LU method is that it only depends on **A**; if $\mathbf{A} \cdot \vec{x} = \vec{b}$ is to be solved many times for different choices of $\vec{b}$, it is advantageous to spend the time decomposing **A** once first.

If we want to find the inverse of **A** then we can decompose the matrix ($\mathcal{O}(N^3)$) once and solve equation (3.9) 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 because (quoted here without proof)

$$\det \mathbf{A} = \prod_j U_{jj} \tag{3.23}$$

3.5 Iterative Methods

The next few sections concentrate on **iterative methods** for solving $\mathbf{A}\cdot\vec{x} = \vec{b}$. 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.

These are called "indirect methods" and will include truncation errors as we can never iterate to get all terms of an infinite series. However, it is often the case that the truncation errors are smaller than the accumulated computational errors for direct methods. For large matrices, the indirect methods turn out to be more suitable (faster or with smaller memory footprints) than direct methods.

The basic idea is that an approximation to **A** is made and this is used to find an approximate solution. The remaining part of **A**, which was not included in the approximation, is then used to make an iterative correction. The methods in principle only differ in terms of what approximation they are making.

3.6 Iterative Jacobi Method

The Jacobi method is *guaranteed* to work if the matrix **A** is strictly **diagonally dominant**, i.e.,

$$|A_{ii}| > \sum_{j\neq i} |A_{ij}|\,, \qquad \text{or} \qquad |A_{jj}| > \sum_{i\neq j} |A_{ij}|\,, \tag{3.24}$$

for all rows i or columns j of the matrix. The Jacobi method can converge for a matrix which is not diagonally dominant, if the eigenvalues of its associated update matrix are suitable, as shown shortly. [1] In particular, **sparse systems**, where only a few variables appear in each equation, can often be rearranged using row manipulations (i.e. pivoting) into a diagonally dominant form such as a band diagonal matrix which looks like

$$\begin{pmatrix} \cdot & \cdot & & & & & \\ \cdot & \cdot & \cdot & & & & \\ & \cdot & \cdot & \cdot & & 0 & \\ & & \cdot & \cdot & \cdot & & \\ & 0 & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \\ & & & & & \cdot & \cdot \end{pmatrix}. \tag{3.25}$$

It is convenient to consider $\mathbf{A}$ as the sum of three matrices; one containing its diagonal elements $\mathbf{D}$, one containing the elements in the triangle below the diagonal $\mathbf{T}_L$ and the third containing the elements in the triangle above the diagonal $\mathbf{T}_U$.

$$\boxed{\mathbf{A} = \mathbf{D} + \mathbf{T}_L + \mathbf{T}_U.} \tag{3.26}$$

This will be something like

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} A_{00} & 0 & 0 \\ 0 & A_{11} & 0 \\ 0 & 0 & A_{22} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ A_{10} & 0 & 0 \\ A_{20} & A_{21} & 0 \end{pmatrix} + \begin{pmatrix} 0 & A_{01} & A_{02} \\ 0 & 0 & A_{12} \\ 0 & 0 & 0 \end{pmatrix} \tag{3.27}$$

Note that the $\mathbf{T}_L$ and $\mathbf{T}_U$ 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

$$\mathbf{A} \cdot \vec{x} = (\mathbf{D} + \mathbf{T}_L + \mathbf{T}_U) \cdot \vec{x} = \vec{b}, \tag{3.28}$$

The Jacobi method corresponds to taking $\mathbf{D}$ as the approximation to $\mathbf{A}$, such that the solution is approximated to $\mathbf{D}^{-1} \cdot \vec{b}$. This means $\mathbf{T}_L + \mathbf{T}_U$ is the correction which is applied iteratively. This leads to

$$\mathbf{D} \cdot \vec{x} = -(\mathbf{T}_L + \mathbf{T}_U) \cdot \vec{x} + \vec{b}\,. \tag{3.29}$$

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

$$\vec{x} = -\mathbf{D}^{-1} \cdot (\mathbf{T}_L + \mathbf{T}_U) \cdot \vec{x} + \mathbf{D}^{-1} \cdot \vec{b}\,. \tag{3.30}$$

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

$$\boxed{\vec{x}^{(n+1)} = -\mathbf{D}^{-1} \cdot (\mathbf{T}_L + \mathbf{T}_U) \cdot \vec{x}^{(n)} + \mathbf{D}^{-1} \cdot \vec{b}} \tag{3.31}$$

which we can use to find $\vec{x}$ starting from a guess $\vec{x}^{(0)}$. Note that the inverse of $\mathbf{D}$ is easy to calculate since it is a diagonal matrix. For any diagonal matrix $\mathbf{D} = \mathrm{diag}(d_1, d_2, \ldots d_N)$ the inverse is

$$\mathbf{D}^{-1} = \mathrm{diag}\left(\frac{1}{d_1}, \frac{1}{d_2}, \ldots, \frac{1}{d_N}\right). \tag{3.32}$$

Equation (3.31) is the Jacobi method.

In the above, the lower and upper triangular matrices only appear together as a sum. We could have used a single matrix to represent this sum and only used that, but we shall see later in the chapter that keeping them separate can be useful.

[1] In particular, matrices with $|A_{ii}| \geq \sum_{j \neq i} |A_{ij}|$, i.e., some (but not all) rows not *strictly* diagonally dominant, will often work with the Jacobi method.

In general this method will converge from any starting guess if all the eigenvalues of the update matrix $\mathbf{H} \equiv \mathbf{D}^{-1} \cdot (\mathbf{T}_L + \mathbf{T}_U)$ satisfy $|\lambda_i| < 1$. [2] We now need to introduce some new terminology: the **spectral radius** of **H**. This must be less than unity. The spectral radius ρ of a matrix **M** is defined as the largest modulus of any of its eigenvalues, i.e., $\rho(\mathbf{M}) = \max\{|\lambda_i|\}$. The condition $\rho(\mathbf{H}) < 1$ is guaranteed for any matrix **A** that is strictly diagonally dominant. [3]

Proof of Jacobi convergence condition: The condition $|\lambda_i| < 1$ is simple to prove, using similar principles to those used later in section 7.5 for the stability analysis of finite difference solutions of a set of coupled linear ODEs. We consider the errors at each iteration

$$\vec{\epsilon}^{(n)} = \vec{x}^{(n)} - \vec{x} \qquad \text{and} \qquad \vec{\epsilon}^{(n+1)} = \vec{x}^{(n+1)} - \vec{x}, \tag{3.33}$$

where $\vec{x}$ is the exact solution. For convergence, we need $\vec{\epsilon}^{(n)}$ to vanish as $n \to \infty$. Using these expressions for the errors and the Jacobi iteration formula, we can show that

$$\begin{aligned} \vec{\epsilon}^{(n+1)} = \vec{x}^{(n+1)} - \vec{x} &= -\mathbf{H} \cdot \vec{x}^{(n)} + \mathbf{D}^{-1} \cdot \vec{b} - \left(-\mathbf{H} \cdot \vec{x} + \mathbf{D}^{-1} \cdot \vec{b}\right) \\ &= -\mathbf{H} \cdot \vec{x}^{(n)} + \mathbf{H} \cdot \vec{x} \\ &= -\mathbf{H} \cdot \left(\vec{x}^{(n)} - \vec{x}\right), \\ \therefore \qquad \vec{\epsilon}^{(n+1)} &= -\mathbf{H} \cdot \vec{\epsilon}^{(n)}. \end{aligned} \tag{3.34}$$

If **H** is a diagonalisable matrix so that it can be factorised using the eigen-decomposition

$$\mathbf{H} = \mathbf{R} \cdot \mathbf{\Lambda} \cdot \mathbf{R}^{-1} \quad \text{with} \quad \mathbf{\Lambda} \equiv \text{diag}(\lambda_0, \lambda_1, ..., \lambda_{N-1}), \tag{3.35}$$

then we can map to a 'rotated' error $\vec{\zeta}^{(n)} = \mathbf{R}^{-1} \cdot \vec{\epsilon}^{(n)}$ such that

$$\vec{\zeta}^{(n+1)} = \mathbf{\Lambda} \cdot \vec{\zeta}^{(n)}, \tag{3.36}$$

and therefore the components of ζ are uncoupled. This allows us to impose a convergence criterion on each eigenvalue individually since

$$\left| \frac{\zeta_i^{(n+1)}}{\zeta_i^{(n)}} \right| \equiv |\lambda_i| < 1. \tag{3.37}$$

(Note that eigen-decomposition is just used here to prove the requirements for Jacobi iteration to converge; we don't actually have to explicitly eigen-decompose **H** to implement the Jacobi method!) What we see is that repeated application of the Jacobi iteration 'erodes' away the error vector (present in our initial guessed solution $\vec{x}_0$). The slowest possible rate of erosion, and therefore the slowest rate of convergence to the solution of $\mathbf{A} \cdot \vec{x} = \vec{b}$, is governed by the spectral radius of the Jacobi update matrix.

Benefits of iterative methods

Firstly, iterative methods are usually faster than the direct inversion methods (such as LU decomposition), especially for sparse matrix equations. This is because we only need to carry out one matrix–vector multiplication (i.e. $\mathbf{H} \cdot \vec{x}^{(n)}$) at each step. (The $\mathbf{D}^{-1} \cdot \vec{b}$ calculation need only be done once, then added on at each step which carries little extra computational cost in comparison to the multiplication.) The total number of operations needed to obtain the solution 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 N^k accounts for multiplication. For a sparse matrix the number of non-zero elements can be a few N (i.e. $3N - 2$ for a tri-diagonal matrix) so that $k = 1$, i.e., $\mathbf{H} \cdot \vec{x}_n$ takes $\mathcal{O}(N)$ operations. For a full matrix, $k = 2$. As long as N_{iter}

[2] Note that a real-valued matrix can have complex eigenvalues.

[3] If **A** is not diagonally dominant then the spectral radius of its Jacobi update matrix will need to be checked, to determine whether or not the Jacobi method will converge with **A**.

is substantially less than N, an iterative solution scales much better than the $\mathcal{O}(N^3)$ needed for direct inversion methods, even for a full matrix. (It should be noted that LU decomposition can be carried out faster than $\mathcal{O}(N^3)$ for a band diagonal matrix by using a bespoke algorithm tailored to the specific structure of that particular matrix.) The key thing with iterative methods is to get quick convergence so that N_{iter} is as small as possible.

A second advantage of iterative methods is that in practice, they often yield more accurate solutions than exact (i.e. direct) methods because they are much less susceptible to round-off error. This is especially true the larger the matrix.

Checking for convergence

Typically, one checks that the fractional change of one of the generalised "norms" of the solution vector

$$\varepsilon^{(n)} = \left| \frac{\|\vec{x}^{(n+1)}\| - \|\vec{x}^{(n)}\|}{\|\vec{x}^{(n)}\|} \right| \tag{3.38}$$

is below a specified tolerance; something a few orders of magnitude above the fractional round-off error due to finite precision arithmetic (e.g. $\varepsilon_{\text{tol}} \sim 100 \times 10^{-16} \sim 10^{-14}$). There are many measures of the norm of a vector; the general ℓ-norm is

$$\|\vec{x}\|_\ell \equiv \left(\sum_i |x_i|^\ell \right)^{\frac{1}{\ell}}. \tag{3.39}$$

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 monitor the **residual** which is $\vec{r} = \mathbf{A} \cdot \vec{x}^{(n+1)} - \vec{b}$ which gives us $\mathbf{A} \cdot \vec{\epsilon}^{(n+1)}$, i.e., the matrix acting on the error vector. One would then monitor $\varepsilon = \|\vec{r}\|/\|\vec{b}\|$. This is more costly to do every iteration.

3.7 Iterative Gauss-Seidel Method

Gauss-Seidel is an alternative method which often converges faster than the Jacobi method. The approximation to $\mathbf{A}$ is $\mathbf{D} + \mathbf{T}_L$, which leaves only $\mathbf{T}_U$ as the iterated correction. This gives

$$\boxed{\vec{x}^{(n+1)} = -(\mathbf{D} + \mathbf{T}_L)^{-1} \cdot \mathbf{T}_U \cdot \vec{x}^{(n)} + (\mathbf{D} + \mathbf{T}_L)^{-1} \cdot \vec{b},} \tag{3.40}$$

where the update matrix is now $\mathbf{H} = (\mathbf{D} + \mathbf{T}_L)^{-1} \cdot \mathbf{T}_U$. This algorithm is derived similarly to the Jacobi method, except that $\mathbf{T}_U \cdot \vec{x}$ (rather than $(\mathbf{T}_L + \mathbf{T}_U) \cdot \vec{x}$) is moved over to the RHS to be with $\vec{b}$. The improved convergence speed stems from the fact that the spectral radius of the Gauss-Seidel update matrix is less than that of the Jacobi update matrix (for a given $\mathbf{A}$), i.e., $\rho(\mathbf{H}_{GS}) < \rho(\mathbf{H}_J)$.

At first glance it looks like we have to calculate $(\mathbf{D} + \mathbf{T}_L)^{-1}$, so that Gauss-Seidel (G-S) would seem to be more computationally costly than Jacobi. However the sum of the diagonal and below-diagonal elements is a lower diagonal matrix, as discussed for LU decomposition. (Again, note this will not be the *same* lower diagonal matrix as would be found using that method.) We rewrite (3.40) as

$$(\mathbf{D} + \mathbf{T}_L) \cdot \vec{x}^{(n+1)} = -\mathbf{T}_U \cdot \vec{x}^{(n)} + \vec{b}. \tag{3.41}$$

Thanks to the shape of the matrix $\mathbf{D} + \mathbf{T}_L$, the G-S iteration has the same form as $\mathbf{L} \cdot \vec{x} = \vec{b}$ so we can use **forward substitution** to solve for $\vec{x}^{(n)}$ in each iteration.

The reason this performs better than the Jacobi method can be understood by considering the iteration equation in a different form. Writing

$$\mathbf{D} \cdot \vec{x}^{(n+1)} = -(\mathbf{T}_L \cdot \vec{x}^{(n+1)} + \mathbf{T}_U \cdot \vec{x}^{(n)}) + \vec{b}. \tag{3.42}$$

then we have

$$\vec{x}^{(n+1)} = -\mathbf{D}^{-1} \cdot (\mathbf{T}_L \cdot \vec{x}^{(n+1)} + \mathbf{T}_U \cdot \vec{x}^{(n)}) + \mathbf{D}^{-1} \cdot \vec{b}. \tag{3.43}$$

Comparing this to the Jacobi iteration equation 3.31, it can be seen that we are using some of the updated (improved) values from $\vec{x}^{(n+1)}$ in the RHS, rather than only $\vec{x}^{(n)}$, and so would expect a more accurate result.

3.8 Iterative Successive Over-Relaxation Method

Abbreviated to SOR, this can be thought of as a modified version of Gauss-Seidel with superior speed of convergence. Introducing a parameter ω, the original equation is multiplied by this parameter to give $\omega\mathbf{A} \cdot \vec{x} = \omega\vec{b}$ and the matrix is now considered as the sum

$$\omega\mathbf{A} = \mathbf{D} + \omega\mathbf{T}_L + \omega\mathbf{T}_U + (\omega - 1)\mathbf{D} \tag{3.44}$$

The approximation to $\mathbf{A}$ is now taken as $\mathbf{D} + \omega\mathbf{T}_L$, leaving $\omega\mathbf{T}_U + (\omega - 1)\mathbf{D}$ as the correction. Note, this reduces to the G-S method for $\omega = 1$. For any ω, the iteration equation is then

$$(\mathbf{D} + \omega\mathbf{T}_L) \cdot \vec{x}^{(n+1)} = -[\omega\mathbf{T}_U + (\omega - 1)\mathbf{D}] \cdot \vec{x}^{(n)} + \omega\vec{b}. \tag{3.45}$$

The matrix $\mathbf{D} + \omega\mathbf{T}_L$ is still lower diagonal so we can again use forward substitution to iterate. The new parameter ω is called the **relaxation parameter**, and a high speed of convergence can be achieved by tuning it, which affects the spectral radius of SOR's update matrix. With $1 < \omega \leq 2$, SOR gives faster convergence than G-S. However, the optimum value of ω is problem-specific. In simple cases it can be determined analytically (not covered in this course), otherwise it must be found by trial and error. For $\omega > 2$, SOR fails, while $\omega < 1$ corresponds to under-relaxation, which is not beneficial to speed of convergence.

3.9 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 eigenvector.

Consider a system of the type

$$\mathbf{A} \cdot \vec{x} = \lambda\,\vec{x}, \tag{3.46}$$

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

$$\mathbf{A} \cdot \vec{e}_i = \lambda_i\,\vec{e}_i\,. \tag{3.47}$$

The linearly independent eigenvectors form a basis in which any vector (say $\vec{v}$) can be expanded

$$\vec{v} = \sum_{i=1}^{N} c_i\,\vec{e}_i\,. \tag{3.48}$$

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

$$\begin{aligned}
\mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\mathbf{A}\cdot\vec{e}_i = \sum_{i=1}^{N} c_i\,\lambda_i\,\vec{e}_i, \\
\mathbf{A}\cdot\mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^2\,\vec{e}_i, \\
\mathbf{A}^n\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^n\,\vec{e}_i, \\
\therefore \quad \lim_{n\to\infty} \quad \mathbf{A}^n\cdot\vec{v} &\to c_j\,\lambda_j^n\,\vec{e}_j, \qquad (3.49)
\end{aligned}$$

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

$$\boxed{\vec{\omega}_j = \frac{\mathbf{A}^n\cdot\vec{v}}{|\mathbf{A}^n\cdot\vec{v}|} \equiv \frac{\vec{e}_j}{|\vec{e}_j|},} \qquad (3.50)$$

such that $\vec{\omega}_j^{\,T}\cdot\vec{\omega}_j = 1$. (Here $\vec{\omega}_j^{\,T}$ is the transpose of vector $\vec{\omega}_j$.) Then it is simple to calculate the eigenvalue itself since

$$\boxed{\vec{\omega}_j^{\,T}\cdot\mathbf{A}\cdot\vec{\omega}_j = \vec{\omega}_j^{\,T}\cdot(\lambda_j\vec{\omega}_j) = \lambda_j.} \qquad (3.51)$$

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 (3.50) and (3.51). This works because the inverse of a matrix has the same eigenvectors but with the inverse eigenvalues. This can be shown since

$$\vec{e}_i = \mathbf{A}^{-1}\cdot\mathbf{A}\cdot\vec{e}_i = \mathbf{A}^{-1}\cdot(\lambda_i\,\vec{e}_i) = \lambda_i\,\mathbf{A}^{-1}\cdot\vec{e}_i \qquad (3.52)$$

thus

$$\mathbf{A}^{-1}\cdot\vec{e}_i = (\lambda_i)^{-1}\,\vec{e}_i. \qquad (3.53)$$

So using the power method on $\mathbf{A}^{-1}$ finds its largest eigenvalue which will be the smallest eigenvalue of the original $\mathbf{A}$. We find $\mathbf{A}^{-1}$ using one of the methods described earlier in this section.

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

$$\mathbf{A}' \equiv \mathbf{A} - \alpha\,\mathbf{I}, \qquad (3.54)$$

such that

$$\mathbf{A}'\cdot\vec{e}_i = \mathbf{A}\cdot\vec{e}_i - \alpha\,\mathbf{I}\cdot\vec{e}_i = \lambda_i\,\vec{e}_i - \alpha\,\vec{e}_i = (\lambda_i - \alpha)\,\vec{e}_i\,. \qquad (3.55)$$

So the shifted matrix has the same eigenvectors but shifted eigenvalues, $\lambda_i' = \lambda_i - \alpha$. Thus finding the largest eigenvalue of $(\mathbf{A}')^{-1}$ (e.g. by the power method) will give the the smallest of $\mathbf{A}'$ and thus the eigenvalue of $\mathbf{A}$ closest to the value α. Again, we obtain $(\mathbf{A}')^{-1}$ using the methods already described.

A crucial point is then to chose a suitable value for α. It turns out that the values of the diagonal elements are good choices, particularly if $\mathbf{A}$ is diagonally dominant. This follows from **Gerschgorin's Theorem** which states that for any eigenvalue λ_i the following inequality is satisfied

$$|\lambda_i - A_{ii}| \le \sum_{j\neq i} |A_{ij}|\,, \qquad (3.56)$$

i.e. the eigenvalue lies within a circle/disk in the complex plane of radius $\sum_{j \neq i} |A_{ij}|$ centred on the value of the diagonal element A_{ii}. [4] Equivalently, each 'Gerschgorin disc' will have an eigenvalue in it (and possibly more than one if discs overlap and if eigenvalues happen to fall on such overlaps).

This is particularly useful if $\mathbf{A}$ is diagonally dominant since the radius will be small and therefore the values of the diagonal elements will be close to the eigenvalues, allowing the algorithm to work quickly and efficiently.

For example take the following matrix[5]

$$\mathbf{A} \equiv \begin{pmatrix} 1 & 0.1 & 0 \\ 0.1 & 5 & 0.2 \\ 0.1 & 0.3 & 10 \end{pmatrix} . \tag{3.57}$$

We then have that $0.9 \leq \lambda_0 \leq 1.1$, $4.7 \leq \lambda_1 \leq 5.3$, or $9.6 \leq \lambda_2 \leq 10.4$. We can then find the exact values by setting α to 1, 5, or 10 in the shift method.

Finally, it is worth noting that Gerschgorin's Theorem provides a convenient way of assessing upper bounds for the spectral radius of the update matrices of the iterative solvers seen earlier (sections 3.6, 3.7, 3.8), and thus determining the suitability of a matrix $\mathbf{A}$ for solution by, e.g., Jacobi iteration.

3.10 Non-Linear Algebraic Equations of one variable

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

$$f(x) = 0, \tag{3.58}$$

where f involves transcendental functions or is even simply a polynomial of high order (i.e. $n > 4$), and therefore 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.

Bisection method

This is the simplest method and is very robust, but 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 calculate the mid point

$$x^{(0)} = \frac{x_l + x_r}{2} \tag{3.59}$$

which gives us a first estimate of the solution. To iterate on this, determine whether the pair x_l and $x^{(0)}$ or $x^{(0)}$ and x_r now bracket the root. Update x_l or x_r using the value of $x^{(0)}$ accordingly and repeat until $\epsilon^{(n)} = x_r - x_l$ (where superscript n denotes the iteration number) is sufficiently small or $\max[|f(x_l)|, |f(x_r)|]$ is sufficiently close to zero. (These are the convergence criteria.) This method converges linearly, with the error $\epsilon^{(n)}$ (the uncertainty in where the root is) halving with each iteration;

$$\epsilon^{(n+1)} = \epsilon^{(n)}/2.$$

Beware: the bisection method will also converge on where discontinuous functions jump through zero.

[4]The role of 'i' in equation (3.56) needs careful qualification. 'i' specifies a certain row of the matrix as would be expected, but which of the N eigenvalues is λ_i? It turns out that λ_i belongs to the eigenvector with element i being the largest in magnitude (in that eigenvector).

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

Newton-Raphson method

This is also known as the Newton's method. This iterative scheme uses the function and its first derivative for the current iterative estimate $x^{(n)}$ to find the next iterative estimate $x^{(n+1)}$

$$\boxed{x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})}.} \tag{3.60}$$

This trivially follows from applying the first order Taylor expansion at $x^{(n)}$ to estimate the change in x required to get to the zero of the function

$$f(x^{(n+1)}) \approx 0 \approx f(x^{(n)}) + [x^{(n+1)} - x^{(n)}]f'(x^{(n)})\,. \tag{3.61}$$

The Newton method can easily fail if it gets near maxima or minima, in which case the divide by the derivative $f' \approx 0$ can cause $x^{(n+1)}$ to shoot off 'to infinity' (but not 'beyond'). 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

$$\epsilon^{(n+1)} \propto [\epsilon^{(n)}]^2. \tag{3.62}$$

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 $f'(x^{(n)})$ is approximated using the last two estimates (which define the "secant line", hence the method name)

$$f'(x^{(n)}) \approx \frac{f(x^{(n)}) - f(x^{(n-1)})}{x^{(n)} - x^{(n-1)}}. \tag{3.63}$$

Inserting into Newton's method gives

$$\boxed{x^{(n+1)} = x^{(n)} - f(x^{(n)})\frac{x^{(n)} - x^{(n-1)}}{f(x^{(n)}) - f(x^{(n-1)})}.} \tag{3.64}$$

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

Brent's method

This is the Rolls Royce of 1D root-finding. It combines bisection with inverse quadratic interpolation *and* the secant method, plus careful bracketing and error tracking—to give what is essentially the last algorithm you will ever need for solving equations in one variable. It has the downside of being quite fiddly to code though, as it flips back and forth between the bisection, inverse quadratic and secant methods according to a series of different criteria, depending on how it is tracking at the time. You can read a good account of it, and see some example code, in Numerical Recipes.

3.11 Non-Linear Algebraic Equations of multiple variables

As stated previously, to solve for N unknowns $\vec{x}$, we need to consider N equations. This requires N functions $f_i(\vec{x}) = 0$, sometimes written as $\vec{f}(\vec{x}) = 0$.

The basic method to use is the generalisation of the Newton-Raphson method to N variables. The Taylor series to first order becomes

$$f_i(\vec{x}^{(n+1)}) \approx 0 \approx f_i(\vec{x}^{(n)}) + \sum_{j=0}^{N-1} [x_j^{(n+1)} - x_j^{(n)}] \frac{\partial f_i}{\partial x_j}(\vec{x}^{(n)})\,. \tag{3.65}$$

The matrix of the $N \times N$ partial derivatives is the Jacobian matrix; writing this as

$$\mathbf{A} = \begin{pmatrix} \partial f_0/\partial x_0 & \partial f_0/\partial x_1 & \partial f_0/\partial x_2 & \dots \\ \partial f_1/\partial x_0 & \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \dots \\ \partial f_2/\partial x_0 & \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}, \tag{3.66}$$

the iteration equation becomes

$$\sum_{j=0}^{N-1} A_{ij}[x_j^{(n+1)} - x_j^{(n)}] = -f_i(\vec{x}^{(n)}) \tag{3.67}$$

and is seen to be a linear matrix equation, i.e. $\mathbf{A} \cdot \Delta\vec{x} = -\vec{f}$. This means each step of the generalised Newton-Raphson method requires solving a matrix equation, using any of the methods discussed earlier in the chapter which are appropriate for the size of the matrices being handled in a particular problem.

The secant method cannot be easily generalised. To estimate the $N \times N$ derivatives needed for the Jacobian matrix requires more than just the current and previous iterations. Hence, if the derivatives cannot be calculated analytically, then each derivative must be estimated specifically for each iteration using a small step ϵ applied to each component of $\vec{x}$ in turn. For example

$$\frac{\partial f_0}{\partial x_0} \approx \frac{f_0(x_0 + \epsilon, x_1, x_2, \dots) - f_0(x_0, x_1, x_2, \dots)}{\epsilon}. \tag{3.68}$$

This must then be repeated for all remaining combinations of the N variables and functions. meaning N^2 function evaluations are needed. This makes the method slower than the Newton-Raphson generalisation although the overall speed of the method is often dominated by the matrix equation solver, which is common to both methods.

Chapter 4

Interpolation

4.1 Introduction

Finding the value of a function at an arbitrary point in a range, given a set of known points that represent it, is another essential piece of a computational physicist's toolkit. This is not a fully-determined problem—the information is not there to completely specify the values between the data points—and therefore additional assumptions are needed. It is up to the physicist to decide what is adequate for the purpose.

One approach is to fit a function to the data points, as is often done with experimental data. This approximate fit will probably not go right through any of the data points, which is often what is desired, particularly if there are uncertainties associated to the data points themselves. This will be discussed in later chapters.

Another approach, which is considered here, is to make a function that connects the known points. This is interpolation. When might interpolation be needed? One example is in solving ODEs numerically, where time is typically discretised (t_n). The value of the solution at a time between these t_n might be required. As we will see later, when 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, and can only be obtained by interpolation.

Interpolation is an extensive subject. Here we briefly look at a few of the simplest methods. These simple methods will get you surprisingly far, and can still be found in most numerical software suites today. More advanced and powerful methods certainly exist too. One of our favourites is splines under tension,[1] where one uses an additional parameter to interpolate the interpolation scheme between linear and cubic splines. Neural networks and other machine learning algorithms are, at the end of the day, also just fancy multi-dimensional interpolators. A machine-learning algorithm uses functions trained on some known data points to produce a good guess at a quantity somewhere between the data points on which it has been trained.

As with any other numerical method, it is up to the physicist to show that the chosen algorithm is appropriate for their purposes, as we shall see.

4.2 Linear interpolation

Linear interpolation is simply to connect adjacent points with a straight line, connect-the-dots style. If two, adjacent, known points are (x_i, f_i) and (x_{i+1}, f_{i+1}), to find f at a point x in between we simply move along the straight line between the two points:

$$\boxed{f(x) = \frac{(x_{i+1} - x)f_i + (x - x_i)f_{i+1}}{x_{i+1} - x_i}.} \tag{4.1}$$

This is the simplest form of interpolation, and is useful if the density of the provided data points is high, or if you somehow know that the function is linear in the regions between the points. Its

[1]See e.g. TSPACK, http://dl.acm.org/citation.cfm?id=151277

behaviour is certainly more predictable than other, more involved forms of interpolation. However, in the more general case, it is unlikely that a true function looks like anything that is made up of only a succession of straight segments, and the limitations of linear interpolation will be quite clear.

4.3 Bi-linear interpolation

Bi-linear interpolation works for a function of two variables $f(x, y)$. Actually it is just linear interpolation in two dimensions. You can probably work it out for yourself already. 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 both the bottom ($y = y_k$) and top ($y = y_{k+1}$) of the square. For instance, along the bottom, equation (4.1) 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 (4.1). Doing interpolation in y to get the intermediate values and then interpolation in x yields exactly the same value.

Of course you can do this in as many dimensions as you like; multivariate interpolation is the generalisation to functions with more than one variable; $f(x, y, z, \ldots)$. In this case, you need to use a (hyper-)box around your desired position, with 2^D vertices, where D is the dimension of your problem.

4.4 Lagrange polynomials

Starting with $n + 1$ distinct points (x_i, f_i) where $0 \leq i \leq n$ and $x_i < x_j$, it is clear that there is a unique n^{th} degree polynomial (i.e. with $n + 1$ degrees of freedom) that goes exactly through these points. This is called the Lagrange polynomial for these points, and can be written:

$$\boxed{P_n(x) = \sum_{i=0}^{n} \left(\prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \right) f_i,} \tag{4.2}$$

where in the product, j run over integers from 0 to n but misses out the value i. Eq. 4.1 is the case of $n = 1$. For the $n = 2$ case

$$P_2(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} f_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} f_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} f_2. \tag{4.3}$$

Note that the x_i do not need to be equally spaced.

Such a polynomial does meet the requirements of being a smooth, well-behaved function that passes through all the data points, but unless it is known that the data follow a polynomial form of the correct degree (which is unlikely), the Lagrange polynomial is likely to “contort” itself to make it go through the data points, resulting in unnaturally wavy behaviour, making it a poor interpolant.

For specific cases, when the data points and the physics problem at hand suit it, the Lagrange polynomial can be extremely useful.

4.5 Cubic splines

Linear interpolation is nice, because it is easy and fast, and does not display the odd behaviour that the Lagrange polynomial can if you are not careful. But it is not ideal—the resulting approximation to the underlying function is clearly not a very good representation of reality in most cases, as it has zero second derivative and a discontinuous first derivative at every data point. No self-respecting function has discontinuous derivatives, especially not one that purports to represent a physical quantity. The data points given to us are usually not accompanied by the statement "these points are also where the first derivatives can be discontinuous".

What can be done about this? We can introduce non-zero higher derivatives to our interpolating function, and force them to be continuous across the data-points, but to do that we need to use polynomials of higher order than one. The lowest-order option is to make all our interpolating functions quadratics. This lets us have continuous first derivatives, and non-zero second derivatives—but the second derivatives will still be discontinuous across the data points. The lowest order polynomial interpolant that allows for continuous first and second derivatives is cubic.

As a general term, a function that matches polynomials, in piecewise fashion along the data points, is called a "spline".

However, once we start introducing additional terms in the interpolating functions, we will have more free parameters to constrain for each piece of the approximating function, so we are going to have to start using more data points at a time to fit those parameters. We therefore need to start making the interpolant somewhat non-local, such that it is not just fit to the two data points either side of it, but more points, further away in each direction. The more continuous derivatives you want, the higher-order your interpolant needs to be, and the more data points you must use to constrain it.

For the cubic spline, we need to use four points at a time; one at each end of the interval being interpolated, and one more each on either side of the interval. We can start by taking the expression for the linear interpolant:

$$f(x) = A(x)f_i + B(x)f_{i+1}, \tag{4.4}$$

with

$$A(x) \equiv \frac{x_{i+1} - x}{x_{i+1} - x_i}, \tag{4.5}$$

$$B(x) \equiv 1 - A(x) = \frac{x - x_i}{x_{i+1} - x_i} \tag{4.6}$$

and supplementing it with some additional corrections, proportional to the second derivatives at each of the adjacent points:

$$f(x) = A(x)f_i + B(x)f_{i+1} + C(x)f''_i + D(x)f''_{i+1}, \tag{4.7}$$

with

$$\begin{aligned} C(x) &\equiv \tfrac{1}{6}(A(x)^3 - A(x))(x_{i+1} - x_i)^2 && (4.8)\\ D(x) &\equiv \tfrac{1}{6}(B(x)^3 - B(x))(x_{i+1} - x_i)^2 && (4.9) \end{aligned}$$

At this point, Eq. 4.7 seems pretty useless—how do we know the values of the second derivative of the function at the data points? We only have data on the value of the function itself, not its derivatives. The key here is to solve for these second derivatives at the points $\{x_0, x_2, x_3, \ldots, x_n\}$.

But how? First we need to impose continuity of the *first* derivative across each data point. Using

$$\frac{\mathrm{d}A}{\mathrm{d}x} = -\frac{\mathrm{d}B}{\mathrm{d}x} = -\frac{1}{x_{i+1} - x_i}, \tag{4.10}$$

$$\frac{\mathrm{d}C}{\mathrm{d}x} = \frac{1 - 3A^2}{6}(x_{i+1} - x_i) \tag{4.11}$$

$$\frac{\mathrm{d}D}{\mathrm{d}x} = \frac{3B^2 - 1}{6}(x_{i+1} - x_i) \tag{4.12}$$

we get

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} - \frac{3A^2 - 1}{6}(x_{i+1} - x_i)f''_i + \frac{3B^2 - 1}{6}(x_{i+1} - x_i)f''_{i+1}. \tag{4.13}$$

Imposing the continuity of $\frac{\mathrm{d}f}{\mathrm{d}x}$ at x_i (i.e. where intervals $[x_{i-1}, x_i]$ and $[x_i, x_{i+1}]$ meet)

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x\to x_i^-} = \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x\to x_i^+} \tag{4.14}$$

then gives the fundamental expression

$$\frac{x_i - x_{i-1}}{6} f''_{i-1} + \frac{x_{i+1} - x_{i-1}}{3} f''_i + \frac{x_{i+1} - x_i}{6} f''_{i+1} = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} - \frac{f_i - f_{i-1}}{x_i - x_{i-1}}. \tag{4.15}$$

This equation is valid for $i = 1\,..\,n{-}1$. This means that it constitutes a system of $n-1$ equations in $n+1$ unknowns ($f''_{0..n}$). We know from linear algebra that this means it has a two-dimensional solution space, implying that we need two more constraints to solve the system. Those constraints come from the chosen **boundary conditions**, which are up to the user of the algorithm to choose as they please. These may take the form of a specified value of the first or second derivative at each of the two end-points of the region being interpolated. The choice $f''_0 = f''_n = 0$ has a special name: the 'natural' spline. This is because if there is a thin massless and frictionless beam that is fixed such that it passes through each data point, it will take on the form that is given by the natural spline. One can also choose to use the two degrees of freedom to fix the first derivatives at the end points to pre-determined values.

The simultaneous equations for $f''_{0..n}$ that are given by this can be set up as a matrix equation, and the matrix be inverted to solve for all the second derivative values. Once you have these, you can use Eq. 4.7 to evaluate your cubic spline at any and as many values of x as you want, at your leisure—without ever having to re-evaluate the second derivatives.

With any interpolation scheme, it is important that validity of the scheme for your set of data is checked, on a case-by-case basis. With that caveat, the cubic spline is a excellent algorithm for general use, with continuity and smoothness of the interpolant being guaranteed with a small number of free parameters, which is often what one needs in a physics context.

Extensions of the cubic spline into multiple dimensions, as well as more sophisticated algorithms do exist, and are implemented in many external library packages. However, from a physicist's point of view, the decision whether or not to interpolate, or to rather fit a function to the data, is often the most important—and we will look at function fitting in a later lecture.

Chapter 5

Numerical Calculus

Outline of Section

- Numerical Differentiation
- Numerical Integration
- The Trapezoidal Rule
- Simpson's Rule
- Romberg integration
- Improper Integrals

5.1 Numerical Differentiation

We are used to defining a derivative, e.g., $\frac{\mathrm{d}y}{\mathrm{d}x}$ or $y'(x)$, as

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} \equiv \lim_{h\to 0} \frac{y(x+h)-y(x)}{h} \equiv \lim_{\Delta x\to 0} \frac{\Delta y}{\Delta x}\,. \tag{5.1}$$

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

$$\boxed{\tilde{y}'_f(x) \equiv \frac{y(x+h)-y(x)}{h}\,.} \tag{5.2}$$

The FDS is an example of a **finite difference approximation**.

Notation: in the this part of the course a tilde ($\sim$) over a quantity signifies that it is an approximated version, which is subject to truncation error.

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

$$y(x+h) = y(x) + y'(x)\,h + \frac{y''(x)}{2}\,h^2 + \frac{y'''(x)}{3!}\,h^3 + ... \tag{5.3}$$

We can define the truncation error as

$$\epsilon = \frac{y''(\xi)}{2}\,h^2, \tag{5.4}$$

where ξ is an unknown value which lies in the interval $x < \xi < x+h$. The correct choice of ξ results in the following exact relation

$$y(x+h) = y(x) + y'(x)\,h + \frac{y''(\xi)}{2}\,h^2\,, \tag{5.5}$$

Thus we can write the first derivative as

$$y'(x) = \frac{y(x+h)-y(x)}{h} - \frac{y''(\xi)}{2}\,h = \tilde{y}'_f(x) - \mathcal{O}(h)\,. \tag{5.6}$$

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.

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

$$\boxed{\tilde{y}'_b(x) \equiv \frac{y(x) - y(x-h)}{h}\,,} \tag{5.7}$$

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 and is 2nd order accurate,

$$\boxed{\tilde{y}'_c(x) \equiv \frac{\tilde{y}'_f(x) + \tilde{y}'_b(x)}{2} = \frac{y(x+h) - y(x-h)}{2h}\,.} \tag{5.8}$$

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

$$y(x+h) = y(x) + y'(x)\,h + \frac{1}{2}y''(x)\,h^2 + \frac{1}{3!}y'''(\xi)\,h^3 \tag{5.9}$$

and

$$y(x-h) = y(x) - y'(x)\,h + \frac{1}{2}y''(x)\,h^2 - \frac{1}{3!}y'''(\zeta)\,h^3. \tag{5.10}$$

Crucially the y'' terms cancel when subtracting these

$$y(x+h) - y(x-h) = 2\,y'(x)\,h + \mathcal{O}(h^3), \tag{5.11}$$

which gives

$$y'(x) = \frac{y(x+h) - y(x-h)}{2h} - \mathcal{O}(h^2) \equiv \tilde{y}'_c(x) - \mathcal{O}(h^2), \tag{5.12}$$

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 5.1; 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

$$\boxed{\tilde{y}''(x) \equiv \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} = y''(x) + \mathcal{O}(h^2).} \tag{5.13}$$

This can be obtained by adding together the 4th order Taylor expansions for $y(x+h)$ and $y(x-h)$, i.e., equations (5.9) and (5.10) with one more term. Another way to get (5.13) 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 at least $n+1$ points and going further away from x, e.g., $y(x+2h)$, $y(x-2h)$, $y(x+3h)$, $y(x-3h)$, etc.

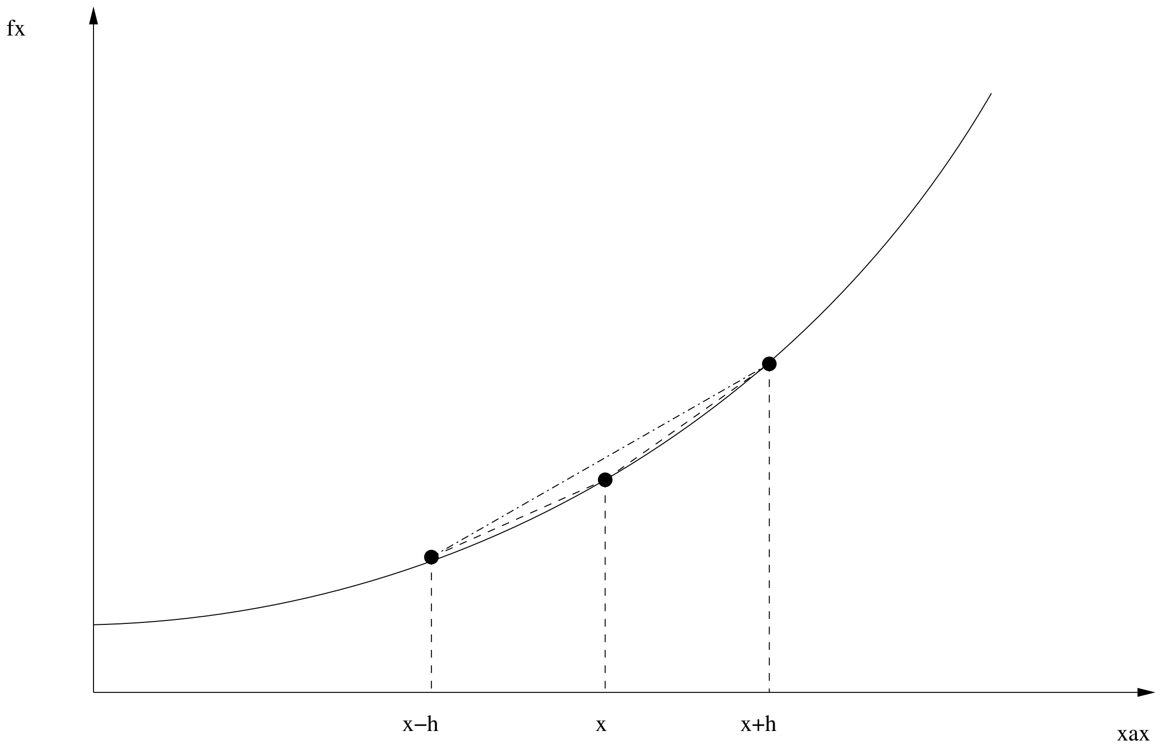

Figure 5.1: Forward, backward, and central difference schemes for approximating the derivative of the function $y(x)$ at the point x

5.2 Numerical Integration

One of the most commonly-encountered problems in everyday physics research is to efficiently evaluate a definite integral

$$I = \int_a^b f(x)\mathrm{d}x. \tag{5.14}$$

Numerical integration is an entire sub-field of numerical analysis, with numerous sophisticated methods and codes available. These basically all fall into three main categories: quadrature, ODE and Monte Carlo methods. We will cover ODE methods in Chapter 8 and Monte Carlo integration in Chapter 10. In this Chapter, we will deal predominantly with Newton-Cotes quadrature methods. We will also see how to transform badly-behaved integrals into a form that is more conducive to numerical evaluation.

5.2.1 Quadrature Methods

The basic idea of quadrature methods is to divide the integral into a number of sub-regions, and integrate over them independently. In a single dimension, this boils down to approximating the integral as a number of rectangle-like sub-regions, as shown in Fig. 5.2 – basically a fancy Riemann sum. The reason we say ‘fancy’ here is that, unlike in a Riemann sum, the tops of the rectangles are generally not taken to be flat, but rather slanted or with an even more complicated shape, in order to better approximate the behaviour of the integrand between samples.

There are thus three main decisions to be made in designing a quadrature algorithm:

(a) The number of samples to take of the integrand $f(x)$
(b) The distribution of the samples over the integration domain, i.e. the stepsize between samples h, and its variation with x
(c) The interpolating function to assume at the tops of the rectangle-like shapes.

The simplest methods in this class are the *Newton-Cotes rules*, which use a constant stepsize h; quadrature methods with a variable h across the integration domain are referred to as *Gaussian*

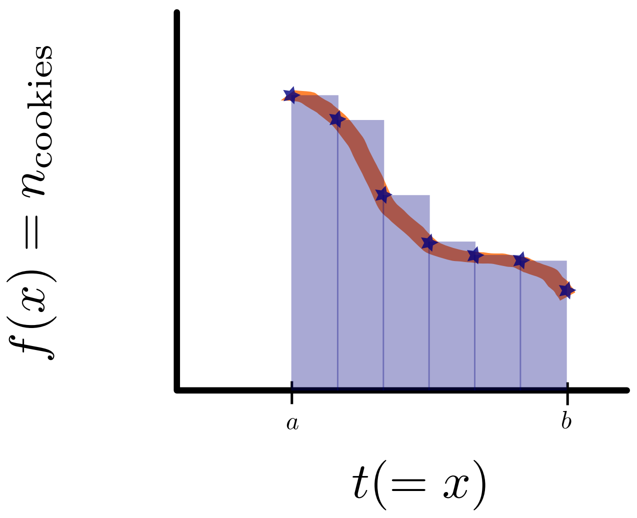

Figure 5.2: Basic anatomy of numerical integration by quadrature: the integral is divided up into rectangle-like shapes, the integrand is sampled at their edges, and the final integral is obtained by assuming some interpolating function passing between the sampled points. Cookies for dramatic effect only.

quadrature. The example shown in Fig. 5.2 is therefore in fact a Newton-Cotes method. Newton-Cotes rules take a number of equal-spaced samples of the integrand, and then estimate the overall integral using a weighted sum of the samples. The different weightings correspond to different interpolating functions across the tops of the rectangles. The next three subsections deal with three different Newton-Cotes methods; the difference between them is essentially in the choice of interpolating function.

The Trapezoidal Rule

Apart from a basic Riemann sum, the simplest way to approximate the integrand between samples is to interpolate between them linearly. This is the so-called *Trapezoidal Rule*. The Trapezoidal Rule is a *two-point* rule, in that it estimates the integral in a region between $x = x_i$ and $x = x_{i+1}$ using the value of the integrand at only two points, namely the two integration limits:

$$\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x = h\left[\frac{1}{2}f(x_i) + \frac{1}{2}f(x_{i+1})\right] + O\left(h^3\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right). \tag{5.15}$$

To compute an entire integral however, we need to patch together many instances of the Trapezoidal Rule into the *Extended Trapezoidal Rule*:

$$\int_{x_0}^{x_{n-1}} f(x)\,\mathrm{d}x = h\left[\frac{1}{2}f(x_0) + f(x_1) + f(x_2) + \ldots + f(x_{n-2}) + \frac{1}{2}f(x_{n-1})\right] + O\left(\frac{(x_{n-1}-x_0)^3}{N^2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right), \tag{5.16}$$

where the stepsize $h = \frac{(x_{n-1}-x_0)}{N}$ for N samples.

This rule provides an estimate of the integral using n samples of the integrand, all connected via linear interpolation according to the Trapezoidal Rule. Turning this into a useful integration algorithm is then just a matter of wrapping it in a loop that steadily increases the number of samples until some convergence criterion is reached.

So, the basic algorithm for computing a definite integral using the Trapezoidal Rule to some desired relative accuracy ϵ is

(a) Evaluate $f(a)$ and $f(b)$

(b) Use these as a first estimate $I_1 = h_1\frac{1}{2}\left[f(a) + f(b)\right]$

(c) Evaluate the midpoint $f(\frac{a+b}{2})$

(d) Use this to update your estimate $I_2 = h_2\left[\frac{1}{2}f(a) + f(\frac{a+b}{2}) + \frac{1}{2}f(b)\right]$

(e) If $\left|\frac{I_2-I_1}{I_1}\right| < \epsilon$, terminate (convergence has been reached).

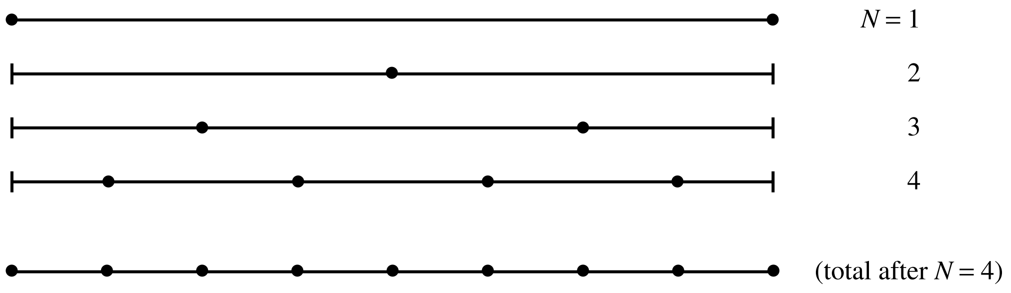

Figure 5.3: Sampling strategy for successive iterations of the Extended Trapezoidal Rule. Each new step fills in the midpoints between the previous points, never evaluating the integrand at the same value of x twice.

(f) If not, keep filling in intermediate points and making more trapezoids until the convergence criterion is satisfied.

Note in particular the use of the word 'intermediate' – there is no reason to ever evaluate the integrand twice at the same value of x. The idea is to decrease h by a factor of 2 at each iteration (Fig. 5.3), so that each successive iteration of the Extended Trapezoidal Rule can be built up from the previous one by simply reweighting the old estimate to account for the change in h, and adding in the new samples at $x = x_1, x_3, x_5, \ldots, x_{n-2}$ (remembering that our samples are indexed from 0 to $n-1$, not 1 to n):

$$T_{j+1} = \frac{1}{2}T_j + \frac{1}{2}h_j \sum_{i=1}^{2^{j-1}} f\left(x_0 + \frac{2i-1}{2}h_j\right), \tag{5.17}$$

$$h_{j+1} = \frac{h_j}{2}. \tag{5.18}$$

Simpson's Rule

The next level of sophistication is to add in an additional point to the basic Newton-Cotes rule, and build an extended rule from this instead. The rule goes by the name of *Simpson's Rule*, and looks like

$$\int_{x_i}^{x_{i+2}} f(x)\,\mathrm{d}x = h\left[\frac{1}{3}f(x_i) + \frac{4}{3}f(x_{i+1}) + \frac{1}{3}f(x_{i+2})\right] + O\left(h^5\frac{\mathrm{d}^4 f}{\mathrm{d}x^4}\right). \tag{5.19}$$

We can see immediately that this three-point rule is no longer doing linear interpolation. In fact, with the additional point, we can now uniquely define a polynomial of one degree higher, i.e. a quadratic. Simpson's Rule therefore improves on the Trapezoidal Rule by approximating the tops of the equal-width sub-integrals with quadratic curves rather than straight lines.

The corresponding *Extended Simpson's Rule* can then be build up in the same manner as for the Extended Trapezoidal Rule, by patching together successive copies of the basic 3-point Simpson's Rule:

$$\begin{aligned}\int_{x_0}^{x_{n-1}} f(x)\,\mathrm{d}x =& h\Bigg[\frac{1}{3}f(x_0) + \frac{4}{3}f(x_1) + \frac{2}{3}f(x_2) + \frac{4}{3}f(x_3)\ldots \\ &\ldots + \frac{4}{3}f(x_{n-4}) + \frac{2}{3}f(x_{n-3}) + \frac{4}{3}f(x_{n-2}) + \frac{1}{3}f(x_{n-1})\Bigg] &(5.20)\\ &+ O\left(\frac{(x_{n-1}-x_0)^5}{N^4}\frac{\mathrm{d}^4 f}{\mathrm{d}x^4}\right), &(5.21)\end{aligned}$$

where again the stepsize $h = \frac{(x_{n-1}-x_0)}{N}$ for N samples. Note that the 3-point sub-rules here do not overlap at all, and are simply patched together end-to-end.

The implementation of the Extended Simpson's Rule in a full integration algorithm follows that for the Trapezoidal Rule fairly closely, except in two important ways. The first is pretty obvious

– one needs to start with 3 points in the initial step, at the two limits of integration and the midpoint. The second is more subtle, and interesting. Simpson's Rule is specifically designed so that the higher-order interpolant results in a higher-order method, i.e. so that the error terms of order h^3 and h^4 from the trapezoidal rule cancel. This means that it can be achieved by taking two successive iterations of the Trapezoidal Rule and combining them with the appropriate weights to cancel the lower-order errors induced by the two lower-order steps:

$$S_j = \frac{4}{3}T_{j+1} - \frac{1}{3}T_j. \tag{5.22}$$

This is quite cute, as it means that an implementation of the Extended Simpson's Rule can be easily piggybacked onto an existing implementation of the Extended Trapezoidal Rule. This gives a more accurate result, more or less 'for free' in terms of computational time.

Romberg integration

Unsurprisingly, successive iterations of Simpson's Rule can also be combined in such a way as to cause the $\mathcal{O}(h^5)$ and even higher-order errors to cancel. The trade-off of this sort of exercise though is that the interpolating function across the tops of the sub-integrals becomes steadily more non-local the higher order one goes to. Much as higher-order spline interpolation starts to become a bit dicey due to non-local effects causing substantial higher derivatives to produce large excursions and 'ringing', higher-order Newton-Cotes schemes can also start to suffer stability problems with rapidly-varying integrands. This isn't catastrophic, but it does offset any real speed gains that one can achieve from them, as it can mean that they need smaller h than one might naively expect, to avoid non-local effects.

Romberg integration is an algorithm that extrapolates the Newton-Cotes strategy to arbitrarily high-order accuracy, at the same time as extrapolating the stepsize h to zero. This sounds almost too good to be true, and it basically is: except for extremely smooth functions, Romberg integration doesn't usually provide much speedup compared to a simple Simpson's Rule integrator – and it adds quite a lot more complexity. For moderately-varying integrands, variable-h methods such as ODE integration usually provide more significant speedup than Romberg integration.

5.2.2 Improper Integrals

Everything we have seen so far in this Chapter corresponds to a *closed* Newton-Cotes rule. This means that the integrand is evaluated directly at the edge of every sub-interval. However, it is also possible to construct *open* rules, which evaluate the integral in an interval without directly evaluating it at the limits. One such rule is the *midpoint rule*,

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = h\left[f\left(\frac{x_0+x_1}{2}\right)\right], \tag{5.23}$$

which simply does a central Reimann sum, approximating the integrand in the interval by its value at the midpoint of the interval.

Open rules can be particularly useful when it is difficult or impossible to evaluate the integrand at one of the limits of integration. This may happen if the integrand is undefined at the limit, e.g. $\frac{0}{0}$, or if the integral diverges at the limit.

Notice that the midpoint rule only covers a small sub-domain of integration, just like the 2-point Trapezoidal and 3-point Simpson's Rules introduced earlier. To use it for doing a real integration, we need to patch it together with many other basic rules to make an extended rule, and put it inside an iterative algorithm that increases the number of sub-domains until we can get a convergent result. However, there is nothing that forces us to patch together only rules of exactly the same kind when creating our extended rules. For dealing with an integrand that is well-behaved right up to the limit of integration, but undefined exactly on the limit, a good approach is therefore simply to patch a single copy of the midpoint rule (at the limit where the integrand is undefined) on to an extended Simpson's rule (for use in the interior and at the other limit).

In the case where an integrand diverges as one approaches one of the limits, or where one end of the integral extends to infinity, a little more care is needed.

Take the case of integrating to infinity first. Here, we have an integral

$$I = \int_a^b f(x)\mathrm{d}x. \tag{5.24}$$

where $a = -\infty$ or $b = \infty$. The best thing to do is to transform the asymptotic part of the integral via the transformation $x \to \frac{1}{t}$, which gives

$$\int_a^b f(x)\,\mathrm{d}x = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right)\,\mathrm{d}t, \tag{5.25}$$

i.e. making the previously badly-behaved limit correspond to $t = 0$. The integrand still can't be evaluated at this limit, but the asymptotic behaviour has been compressed into a finite range, allowing us to employ an open rule on the transformed integral, and simply avoid evaluating the integrand at exactly $t = 0$. Note that this trick only works for one limit at a time, and only when a and b have the same sign; otherwise, you will need to split the integral at some opportune location (often at $x = 0$) and do the two parts separately. This trick is also inefficient if the entire integrand is not asymptotically decreasing at least as quickly as x^{-2}; in this case it also pays to split the integral, so as to isolate the part where it *is* dropping faster than x^{-2}, and use the transformation only on that part (and a regular Simpson's Rule on the rest).

The case of a divergent integral requires even more thought. In this case, we have an integrable singularity at some special value of x; call this x_s. If we know the value of x_s, we can proceed fairly safely: split the integral at the singularity. Now you have two integrals each with singularities at the edges of their domains. We can then transform the nasty parts of each integral as $x \to \alpha$, with

$$\begin{aligned} \alpha &= t^{\frac{1}{1-\gamma}} + x_s \quad \text{for lower limit } x_s & (5.26)\\ \alpha &= x_s - t^{\frac{1}{1-\gamma}} \quad \text{for upper limit } x_s. & (5.27) \end{aligned}$$

This gives (with either $a' = x_s$ or $b' = x_s$)

$$\int_{a'}^{b'} f(x)\,\mathrm{d}x = \frac{1}{1-\gamma}\int_0^{(b'-a')^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(\alpha)\,\mathrm{d}t, \tag{5.28}$$

which we can happily integrate with any extended rule that is open at $t = 0$. Here γ is a power-law index that we are basically free to choose to be anything between 0 and 1. Note that the most efficient integration will result if we choose γ to match the slope of our divergence as closely as possible. That is, if our function behaves like $f(x) \to (x - x_s)^{-\beta}$ as $x \to x_s$, then the most efficient choice is $\gamma = \beta$.

If you run into a situation where you know that there is a singularity somewhere in $a < x_s < b$, but don't actually know the value of x_s, then you're in significantly more trouble. In this case, you need to try to somehow 'sneak past' the singularity using variable-stepsize methods such as ODE integration, Gaussian quadrature or Monte Carlo methods. The challenge of course is that the area around $x = x_s$ generally contributes significantly to the total integral, so you *also* need to sample this area fairly densely to get a converged result – but at the same time, you need to avoid evaluating the integrand so close to the pole that the result overflows the floating-point type you're using. It's not a lot of fun. In most cases, it's worth putting some time into instead trying to transform your problem so that the singularity either goes away, or sits at some known value of some suitably transformed integration variable.

Chapter 6

Fourier Transforms

Outline of Section

- Fourier Transforms—recap
- Discrete FTs (DFTs)
- Sampling & Aliasing
- Fast Fourier Transform (FFT) algorithm
- Pseudocode

In the Chapter 4 we looked at how to take a set of data points and produce a function that returns a value for any point in the range of these data points, interpolating between them. We now turn towards another way in which such a set of data points can be processed, which is to perform a Fourier Transform on them, to extract the frequency-based properties of the data set.

We focus on how to numerically calculate the Fourier transform of *regular,* discrete, samples of a function. The **Discrete Fourier Transform** (DFT) is used for this, and is the discrete equivalent of the Fourier transform. The sampled function could be the function found by solving an ODE or PDE using finite difference methods. Or it might be data sampled from, e.g., an oscilloscope or a microphone. Considerations and limitations caused by the sampling procedure will be discussed.

The **Fast Fourier Transform** (FFT)—an efficient implementation on the DFT—is used extensively, e.g., for

- Noise suppression—Data analysis
- Image compression—JPEG formats
- Audio and video compression—MPEG formats
- Medical imaging
- Interferometric imaging
- Modelling of optical systems
- Solution of periodic boundary value problems

to name a few examples.

"Pseudocode" is a concept that is very useful and is widely-used in discussions of computer algorithms. While this has no direct connection with Fourier Transforms, we make the use of the opportunity to describe the Fast Fourier Transform algorithm in this chapter to introduce the concept of writing and using pseudocode.

6.1 Fourier Transforms—Recap

The continuous Fourier Transform (FT), both forward and backward, of a function $f(t)$ are defined as

$$\tilde{f}(\omega) = \int_{-\infty}^{+\infty} e^{i\omega t} f(t)\,\mathrm{d}t = \mathcal{F}(f(t)), \tag{6.1}$$

$$\text{and} \qquad f(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-i\omega t}\tilde{f}(\omega)\,\mathrm{d}\omega = \mathcal{F}^{-1}(\tilde{f}(\omega)), \tag{6.2}$$

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: do not confuse this with the tilde '$\sim$' notation mentioned elsewhere in the course, such as to denote numerical approximation or for scaled variables!) 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')

$$\tilde{f}(-\omega) = \tilde{f}^*(\omega), \tag{6.3}$$

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 $f(t)$ 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

$$f(t) \rightleftharpoons \tilde{f}(\omega). \tag{6.4}$$

$\mathcal{F}^{-1}(\tilde{f}(\omega))$ 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 whether both share it (i.e. $1/\sqrt{2\pi}$ in front of each), and 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. In multiple spatial dimensions the wavenumber becomes a wave vector

$$\vec{x} \equiv (x, y, z) \qquad \leftrightarrow \qquad \vec{k} \equiv (k_x, k_y, k_z)\,, \tag{6.5}$$

and the FTs are modified accordingly:

$$\tilde{f}(\vec{k}) = \int_{-\infty}^{+\infty} e^{i\vec{k}\cdot\vec{x}} f(\vec{x}) d\vec{x} = \mathcal{F}(f(\vec{x})), \tag{6.6}$$

$$f(\vec{x}) = \frac{1}{(2\pi)^3}\int_{-\infty}^{+\infty} e^{-i\vec{k}\cdot\vec{x}} \tilde{f}(\vec{k}) d\vec{k} = \mathcal{F}^{-1}(\tilde{f}(\vec{k})). \tag{6.7}$$

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

$$\frac{\mathrm{d}}{\mathrm{d}x} u(x) = v(x). \tag{6.8}$$

Taking the FT of this equation yields

$$-ik\tilde{u}(k) = \tilde{v}(k). \tag{6.9}$$

This can be shown as follows: replace $u(x)$ and $v(x)$ by their inverse FTs

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-ikx}\tilde{u}(k)\,\mathrm{d}k\right) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-ikx}\tilde{v}(k)\,\mathrm{d}k\,. \tag{6.10}$$

The only bits containing x are the transform kernel (i.e. the plane wave part $\exp(-ikx)$). Therefore using Leibnitz's integral rule yields

$$\frac{1}{2\pi}\int_{-\infty}^{+\infty} -ike^{-ikx}\tilde{u}(k)\,\mathrm{d}k = \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-ikx}\tilde{v}(k)\,\mathrm{d}k\,. \tag{6.11}$$

Equating the integrands on both sides we have (6.9).

This can be generalised to higher derivatives:

$$\frac{\mathrm{d}^n}{\mathrm{d}x^n} f(x) \rightleftharpoons (-ik)^n \tilde{f}(k). \tag{6.12}$$

It can be generalised to 3D too:

$$\nabla f(\vec{x}) \rightleftharpoons -i\vec{k}\tilde{f}(\vec{k}) \quad , \qquad \nabla^2 f(\vec{x}) \rightleftharpoons -|\vec{k}|^2 \tilde{f}(\vec{k}) \quad ,$$
$$\nabla \cdot \vec{E}(\vec{x}) \rightleftharpoons -i\vec{k} \cdot \tilde{\vec{E}}(\vec{k}) \quad , \qquad \nabla \times \vec{E}(\vec{x}) \rightleftharpoons -i\vec{k} \times \tilde{\vec{E}}(\vec{k}), \tag{6.13}$$

where $\nabla^2 f = \nabla \cdot (\nabla f)$ has been used.

6.2 Discrete FTs

DFTs are the equivalent of complex Fourier series, which are 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 on a time domain of length $T = N\Delta t$ as illustrated in figure 6.1 ;

$$f_n \equiv f(t_n) \qquad \text{with} \qquad n = 0, 1, 2, ..., N-1 \qquad \text{and} \qquad t_n = n\Delta t. \tag{6.14}$$

The function $f(t)$ is ***assumed to be periodically extended*** beyond the domain $0 \le t \le T$ so that $f(t + mT) = f(t)$ for integer m. For the sampled function, periodicity means $f_N = f_0$. This does not mean that the value of the function at $t \to T$ has to be the same as that at $t = 0$; but it needs to be single-valued, and for discretising it one would choose the value for $t = 0$.

If one is only focused on the time domain $0 \le t < T$, the periodicity of the function does not affect things directly.

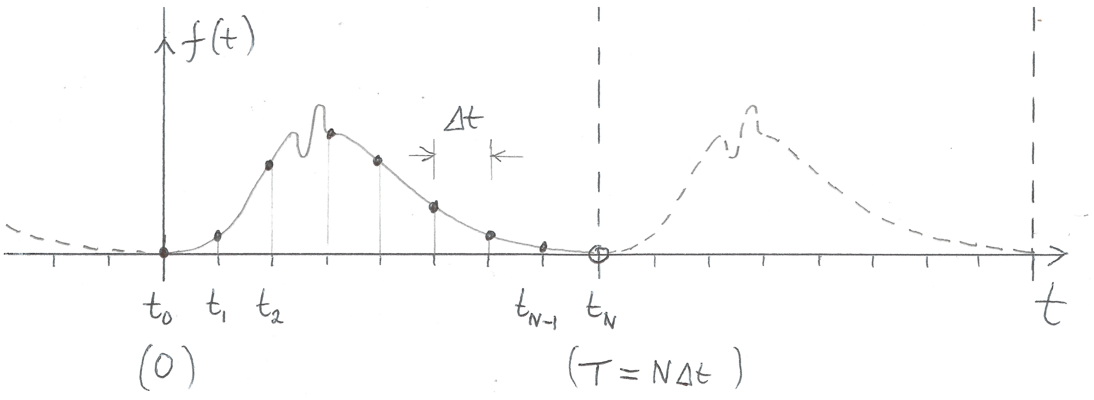

Figure 6.1: Illustration of a signal in the time domain. $N = 8$ here. The signal/function is assumed to be periodically extended beyond $0 \le t \le T$.

Frequency domain— Figure 6.2 illustrates the frequency domain and discrete spectrum of the signal sampled in figure 6.1. 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**

$$\boxed{\omega_{max} = \frac{\pi}{\Delta t} = \Delta\omega \frac{N}{2}.} \tag{6.15}$$

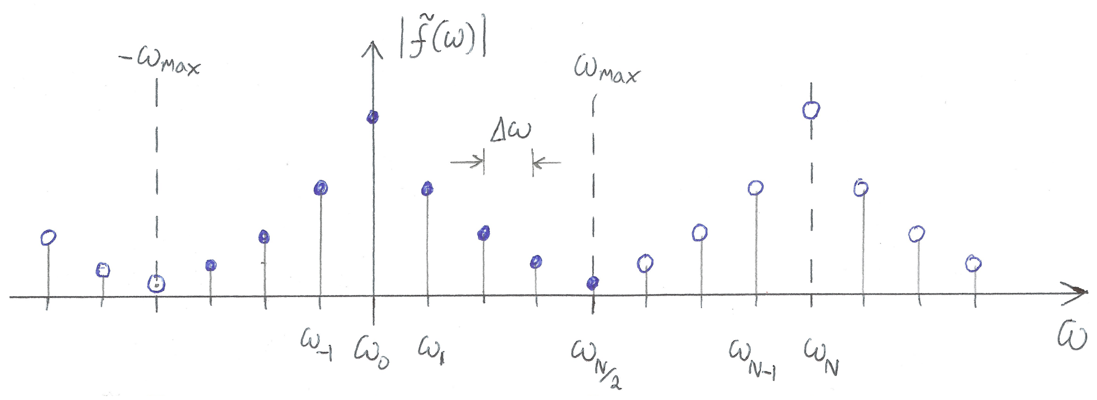

Figure 6.2: Frequency domain and discrete spectrum corresponding to figure 6.1. The discrete spectrum is also periodically extended (beyond $|\omega_{max}|$ in this case). The modulus of $\tilde{f}(\omega)$ is depicted.

Allowing for negative angular frequencies too, these frequencies define the discrete, finite, angular frequency grid on which $\tilde{f}$ is sampled;

$$\boxed{\tilde{f}_p \approx \frac{1}{\Delta t}\tilde{f}(\omega_p) \quad \text{with} \quad p = -\frac{N}{2}, \dots, 0, \dots, \frac{N}{2} \quad \text{and} \qquad \omega_p = p\Delta\omega = p\frac{2\pi}{N\Delta t}.} \tag{6.16}$$

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 6.3.)

Why does $\tilde{f}_p \approx \tilde{f}(\omega_p)/\Delta t$ rather than $\tilde{f}_p = \tilde{f}(\omega_p)/\Delta t$ in equation (6.16) above? This is because $\tilde{f}_p$ represents the DFT which is not always exactly $\tilde{f}(\omega)$, the true spectrum of $f(t)$, simply picked out at discrete frequencies ω_p. We assume that $f(t)$ *is* exactly, discretely sampled. If there is no aliasing (see 6.3) then $\tilde{f}_p = \tilde{f}(\omega_p)/\Delta t$. If there is aliasing, then the DFT is a distorted, discrete, version of the true spectrum. (The $1/\Delta t$ factor is just the convention used in the definition of the DFT, see below.)

The DFT is the analogue of a continuous FT, but with the continuous integral $\int \dots dt$ replaced by a finite sum $\sum \dots \Delta t$:

$$\boxed{\tilde{f}_p = \sum_{n=0}^{N-1} f_n e^{i\omega_p t_n} = \sum_{n=0}^{N-1} f_n e^{i2\pi pn/N},} \tag{6.17}$$

where $\omega_p t_n = \left(\frac{2\pi p}{N\Delta t}\right)(n\Delta t)$ has been used to obtain the second form. The backwards transform is similarly defined as

$$f_n = \frac{1}{N}\sum_{p=-N/2+1}^{N/2} \tilde{f}_p e^{-i2\pi pn/N}. \tag{6.18}$$

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 FT (6.1) to the DFT (6.17) 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 (6.18) is usually defined as

$$\boxed{f_n = \frac{1}{N}\sum_{p=0}^{N-1} \tilde{f}_p e^{-i2\pi pn/N},} \tag{6.19}$$

which is more symmetrical with the forward DFT. Why this is possible will be seen in section 6.3.

Relation to complex Fourier series – Discrete FTs (DFTs) are intimately related to complex Fourier series (CFS).

$$c_k = \frac{1}{T}\int_0^T f(t)e^{i\omega_k t}\,\mathrm{d}t\,, \tag{6.20}$$

$$f(t) = \sum_{k=-\infty}^{+\infty} c_k e^{-i\omega_k t}. \tag{6.21}$$

Complex Fourier series represent 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 (6.17) and (6.18), 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 **$\mathcal{O}(N\log_2 N)$ operations**; the algorithm for this is known as the **fast Fourier transform** or just **FFT**. For $N = 10^6$, the speed-up factor N/log_2N is approximately 50,000. 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 implementation of the DFT 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 $N = 2^m$.

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

$$\tilde{f}_{p,q} = \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} f_{n,m}e^{i2\pi pn/N}e^{i2\pi qm/M}. \tag{6.22}$$

The backward DFT is

$$f_{n,m} = \frac{1}{NM}\sum_{p=0}^{N-1}\sum_{q=0}^{M-1} \tilde{f}_{p,q}e^{-i2\pi pn/N}e^{-i2\pi qm/M}. \tag{6.23}$$

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.

6.3 Sampling & Aliasing

The discrete sampling of the function to be FT'd, given by equation (6.14), 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 do not 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.

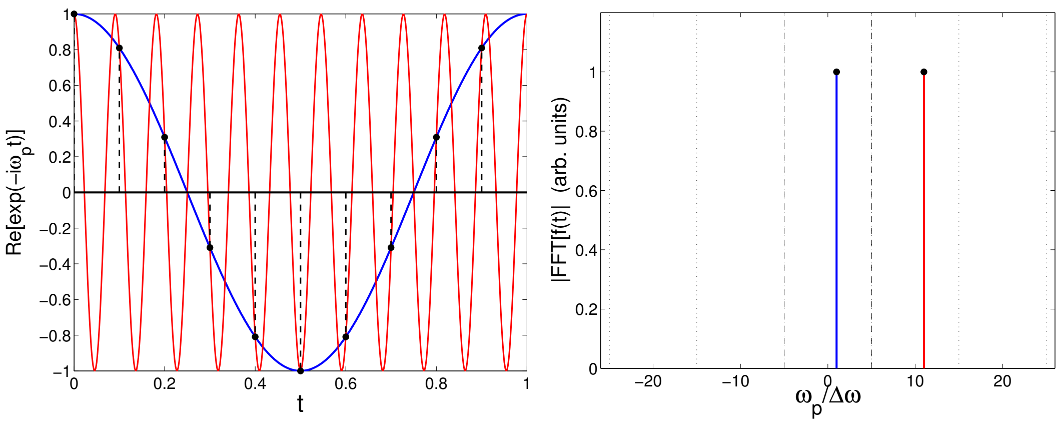

Figure 6.3: (Left) Indistinguishable harmonic waves below and above the Nyquist frequency, for $N = 10$. Blue line: $\omega = \omega_1 = \Delta\omega$. Red line: $\omega = \omega_{N+1} = \Delta\omega + 2\omega_{max}$. (Waves are periodically extended beyond range shown.) (Right) Corresponding (angular) frequency spectrum. The vertical dashed lines lie at ± 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 6.3. This can be understood by considering the kernel of the DFT, i.e., $\exp\bigl(i\omega_{p'}t_n\bigr)$.

$$\begin{aligned}\exp\left[i(\omega_p + m\Omega)t_n\right] &= \exp\left[i\left(\frac{2\pi pn}{N} + m\frac{2\pi}{\Delta t}n\Delta t\right)\right] = \exp\left(i\frac{2\pi pn}{N} + i2\pi mn\right)\\ &= \exp\left(i\frac{2\pi pn}{N}\right) = \exp\left(i\omega_p t_n\right).\end{aligned}$$

This has the following implications;

(a) It explains why $\tilde{f}_{-N/2} \equiv \tilde{f}_{N/2}$.

(b) It also explains why equation (6.19) 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 (6.19) are above the Nyquist frequency, they happen to capture the desired negative frequencies within the Nyquist range.

(c) Aliasing:

$$\tilde{f}_p = \sum_m \mathcal{F}(f)|_{\omega_p + m\Omega}, \tag{6.24}$$

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 6.4 shows the phenomenon of aliasing for a top-hat function

$$f(t) = \begin{cases} 1 & |t| < a/2 \\ 0 & |t| > a/2 \end{cases}. \tag{6.25}$$

6.4 Fast Fourier Transforms—FFTs

The FFT algorithm for doing a discrete Fourier transform in $\mathcal{O}(N \log N)$ operation was described by Daniel & Lanczos in 1942, and earlier versions existed in the 1800s (indeed, the first example of

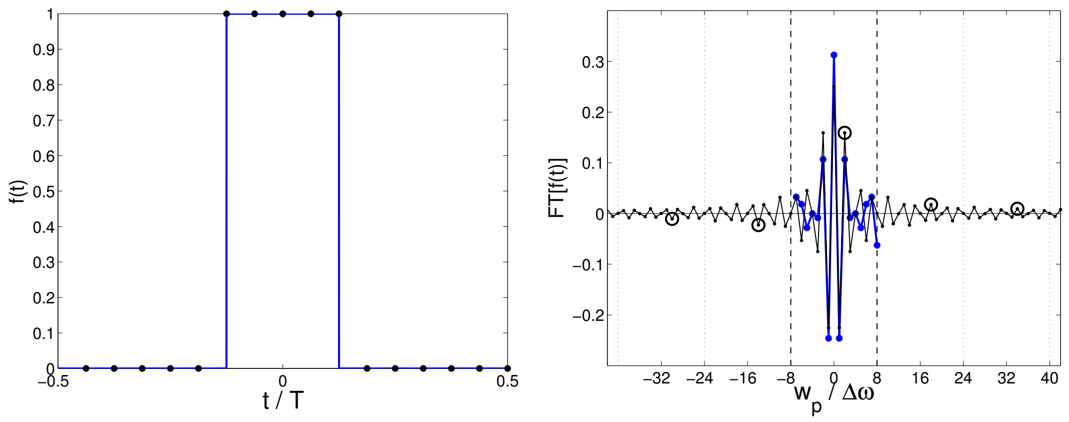

Figure 6.4: Aliasing: (Left) $f(t)$ in the time domain; centred "rect" function of width $T/4$. Sampled with $N = 16$. (Right) The DFT (blue, solid line + markers), and exact FT (black, thin solid line) in the frequency domain. The open circles denote Fourier components outside the range $-\omega_{max} \leq \omega \leq \omega_{max}$ that are aliased into that range during the DFT. The real part of the DFT & FT are shown. The DFT been divided by Δt.

the algorithm was given by Gauss in 1805), but was rediscovered and popularised in the computer age by Cooley & Tukey in 1965.

Directly coding the DFT (as we did last time) results in an $\mathcal{O}(N^2)$ algorithm. But this can be reduced to $\mathcal{O}(N \log_2 N)$ with the Fast Fourier Transform, for $N = 2^m$. The central idea is that an N-sample DFT can be split into two $N/2$-sample ones:

$$\begin{aligned}\tilde{f}_p &= \sum_{n=0}^{N-1} f_n e^{i2\pi pn/N} \\ &= \sum_{n=0,2,\ldots}^{N-2} f_n e^{i2\pi pn/N} + \sum_{n=1,3,\ldots}^{N-1} f_n e^{i2\pi pn/N} \\ &= \tilde{f}_p^{\text{even}} + e^{i2\pi p/N} \times \tilde{f}_p^{\text{odd}}\end{aligned}$$

Each half of these is an $\mathcal{O}(\frac{N}{2}\log\frac{N}{2})$ problem, and this can be done recursively.

Here, $\tilde{f}_p^{\text{even, odd}}$ are $N/2$-sample DFTs of the even and odd samples of the original f_n. These are each periodic such that $\tilde{\boldsymbol{f}}_{\boldsymbol{p+N/2}}^{\text{even, odd}} = \tilde{\boldsymbol{f}}_{\boldsymbol{p}}^{\text{even, odd}}$ This results in, for $0 \leq p < N/2$:

$$\begin{aligned}\tilde{f}_p &= \tilde{f}_p^{\text{even}} + e^{i2\pi p/N} \times \tilde{f}_p^{\text{odd}} \\ \tilde{f}_{p+N/2} &= \tilde{f}_p^{\text{even}} - e^{i2\pi p/N} \times \tilde{f}_p^{\text{odd}}.\end{aligned}$$

This leads to a simple algorithm for implementation, which we can discuss using pseudocode.

6.4.1 Pseudocode

We often need to explain an algorithm to each other, in a way that clearly shows how you would code it up (in any language). In this case, it helps not to be burdened by the (language-specific) overhead that real code brings with it (as well as a need to be syntactically perfect). This to say that it helps to omit constructs such as:

```
"import numpy as np"
"tarray = np.zeros(N)"
"#include <stdio.h>"
"COMMON/CRZT/IXSTOR,IXDIV,IFENCE(2),LEV,LEVIN,BLVECT(LURCOR)"
```

and the positioning of colons and semicolons etc., and allow ourselves to focus on the logic of the algorithm that is being presented. We call this “Pseudocode”; it is intended to be for humans, but has a code-like structure because of the logic of algorithms. While attempts have been made in the past to standardise pseudocode, these have failed because this runs counter to the benefits of using pseudocode; therefore there is no formal syntax; it is up to the author to make things clear for their purpose and their audience. One is allowed to use language-specific elements if that is not confusing (e.g., logic statements from Python or C etc.).

Examples of pseudocode of all sorts of styles are readily available online.

6.4.2 Fast Fourier Transforms in Pseudocode

The following is an example of the FFT algorithm, in pseudocode. The recursive nature of the algorithm is made clear through “calls” to the FFT() function itself:

```
def FFT(f): # f is an array
    N = size of array f
    if N == 1:
        # for a sample size of 1, the FT is the value itself
        return f[0]
    if N is not a power of 2, exit

    # recursive calls
    farray_even = FFT(even entries of f) # size N/2
    farray_odd  = FFT(odd  entries of f) # size N/2

    # return an N-element array made from
    for p = 0..N/2-1
        farray[p] = farray_even[p]
                       + exp(i2pi p/N) * farray_odd[p]
        farray[p+N/2] = farray_even[p]
                       - exp(i2pi p/N) * farray_odd[p]

    return farray
```

Note the use of expressions such as a) `N = size of array f`, b) `if N is not a power of 2, exit` and c) `odd entries of f`. These are of course not syntactically correct in any programming language, but are easy to parse for a human being, without being confusing for people who do not speak a particular language. These examples could be written as a) `N = $#f + 1;`, b) `if(x&(x-1)) exit;` and c) `f[1::2]`, none of which can necessarily be said to be easy to understand if one is not aware of the programming language that is in use.

The conversion of the above pseudocode into real code is left for the problem sheets.

Recursive function calls, whilst being logically clear, can require substantial overheads when the code is executed. In the past, when coding in BASIC and FORTRAN recursive function calls were not even part of the standards for the languages.

In addition to the above, further shortcuts exist which can speed up the code by sorting the elements of the calculations in a clever way.

Contracting the “even” and “odd” superscripts in Equation 6.26 to “e” and “o” yields:

$$\begin{aligned}\tilde{f}_p &= \tilde{f}_p^{\mathrm{e}} + e^{i2\pi p/N} \times \tilde{f}_p^{\mathrm{o}}\\ \tilde{f}_{p+N/2} &= \tilde{f}_p^{\mathrm{e}} - e^{i2\pi p/N} \times \tilde{f}_p^{\mathrm{o}}.\end{aligned}$$

Iterating once again, the quantity $\tilde{f}^{\mathrm{e}}$ can be written, for $p = 0, \ldots, N/4 - 1$:

$$\begin{aligned}\tilde{f}_p^{\mathrm{e}} &= \tilde{f}_p^{\mathrm{ee}} + e^{i2\pi p/(N/2)} \times \tilde{f}_p^{\mathrm{eo}}\\ \tilde{f}_{p+N/4}^{\mathrm{e}} &= \tilde{f}_p^{\mathrm{ee}} - e^{i2\pi p/(N/2)} \times \tilde{f}_p^{\mathrm{eo}}.\end{aligned}$$

Assuming $N = 2^m$ (i.e. even), after m even/odd splitting we are left with N functions of period 1 which are each trivially Fourier Transformed; the transform of each is the same as itself

$$\tilde{f}_p^{eooeeoeoeo...e} = f_n, \tag{6.26}$$

for some n.

All we have left to do is identify which combination of *eoeeo...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.

This can be seen empirically in the following, where an input function represented by a series of 2^3 samples is broken up in to two even and odd series, continuing recursively into single-sample elements:

$$\begin{array}{cccccccc}
||f_0, & f_1, & f_2, & f_3, & f_4, & f_5, & f_6, & f_7|| \\
||f_0, & f_2, & f_4, & f_6|| & ||f_1, & f_3, & f_5, & f_7|| \\
||f_0 & f_4|| & ||f_2 & f_6|| & ||f_1 & f_5|| & ||f_3 & f_7|| \\
||f_0|| & ||f_4|| & ||f_2|| & ||f_6|| & ||f_1|| & ||f_5|| & ||f_3|| & ||f_7||
\end{array}$$

This results in the sequence of indices: 0, 4, 2, 6, 1, 5, 3, 7. This sequence can be compared with the original sequence, in binary form:

Original		Transformed	
Decimal	Binary	Decimal	Binary
0	000	0	000
1	001	4	100
2	010	2	010
3	011	6	110
4	100	1	001
5	101	5	101
6	110	3	011
7	111	7	111

Here, the binary representation is made of of bits that are reversed between the original and transformed order of indices. The "bit-reversing" procedure may not be particularly fast in software, but it is easy to implement directly in signal-processing chips etc.

The complete method requires

$$N \times p = \frac{N \log N}{\log 2} \sim \mathcal{O}(N \log N), \tag{6.27}$$

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.

The FFT algorithm leads naturally to the fast convolution of functions, filtering, and finding the correlation between functions etc.

Chapter 7

Ordinary Differential Equations

Outline of Section

- Introduction to solving ODEs
- The Euler method
- Consistency
- Accuracy
- Stability

7.1 Introduction

In countless problems in physics we encounter the description of a physical system through a set of ordinary differential equations (ODE). This is because it is simpler to model the behaviour of a system in terms of infinitesimal responses to infinitesimal changes of independent variable(s).

While these could be equations for any variables, in physics we often want to solve for changes as a function of time t, so this would act as the independent variable for our equations. (However, the methods we use could be applied to any variable type; the mathematics doesn't care what the variables correspond to physically.) Hence, in discussing differential equations, we will often talk about 'timesteps' and 'time-stepping' in solving systems of differential equations. This simply refers to increments in the value of the independent variable (whether that independent variable actually physically corresponds to time or not). We will write something like $x(t)$ as the function we are trying to solve for. We will be looking at first-order ordinary differential equations, which for one variable can be generally written in the form

$$\frac{dx}{dt} = f(t, x) \tag{7.1}$$

There are various convenient compact notations for a derivative; a time derivative is often written as $\dot{x}$ although we will use x' as this is more general for use in systems where the independent variable is not t. Hence we have $x' = f(t, x)$. Higher derivatives are indicated by more primes, e.g. x'' for the second derivative or $x^{n\prime}$ for the n^{th} derivative.

We can have more than one variable; e.g. for three variables, the general form is

$$\frac{dx}{dt} = x' = f_x(t, x, y, z), \qquad \frac{dy}{dt} = y' = f_y(t, x, y, z), \qquad \frac{dz}{dt} = z' = f_z(t, x, y, z) \tag{7.2}$$

and we can write for a general m equation system

$$\frac{d\vec{x}}{dt} = \vec{x}' = \vec{f}(t, \vec{x}) \tag{7.3}$$

where x_i for $i = 0$ to $m-1$ is an m component vector and each equation has a function $f_i(t, \vec{x})$. Note that the functions defining the derivatives of the system variables are in general a function of the independent variable t *and* all the system variables (x, y, z, etc.), 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**.

To solve these equations, we are effectively doing integrations. The explicit connection between numerical integration and ODE solutions will be discussed in section (8.7). However, it should be clear that, as for integrations, we will need to know the constants of integration to solve the ODE. There must be one constant for every first-order equation, typically specifying the value of x at some time t_0. Clearly, the solution can be found analytically for the simplest cases but if the function describing the derivative is not separable or if the system of many variables is coupled, the equations cannot be integrated analytically and we have to take a numerical approach. We can solve systems numerically if the equations have a continuous solution and we know the information required for the constants of integration.

Higher-order equations

Many equations in physics involve higher derivatives, notably Newton's classical laws of motion which we can write as $x'' = F(t, x, x')/m$ for a general force F. As shown, these can include a velocity dependence, e.g. for a charged particle in a magnetic field. It would be possible to use methods specifically derived for this type of equation. However, it is always possible to use the velocity $v = x'$ as another dependent variable such that we have two coupled first-order equations

$$\frac{dx}{dt} = x' = v = f_x(t, x, v), \qquad \frac{dv}{dt} = v' = \frac{F(t, x, v)}{m} = f_v(t, x, v) \tag{7.4}$$

where the x derivative function $f_x(t, x, v) = v$, and so fits into our approach, with a function which happens to have a particularly simple form.[1] A similar approach could be done for a third-order equation, using the acceleration $a = v'$ as another dependent variable. In fact, *any* m^{th}-order equation can always be expressed as m first-order equations.

However, note the converse is not true. Consider the general case of two variables

$$\frac{dx}{dt} = x' = f_x(t, x, v), \qquad \frac{dy}{dt} = y' = f_y(t, x, v) \tag{7.5}$$

which we want to express as a single second-order equation for x only. We can do

$$\frac{d^2x}{dt^2} = \frac{df_x}{dt} = \frac{\partial f_x}{\partial t} + \frac{\partial f_x}{\partial x}x' + \frac{\partial f_x}{\partial y}y' = \frac{\partial f_x}{\partial t} + \frac{\partial f_x}{\partial x}f_x + \frac{\partial f_x}{\partial y}f_y \tag{7.6}$$

Note the RHS is a function of t, x and y. To remove y, we need to invert the equation $x' = f_x(t, x, y)$ to get $y = g(t, x, x')$ and then substitute this into the above. However, this inversion is not always practical or even possible. (For the above case which did come from a second-order equation, $x' = v$ so the inversion is trivial.) Hence, the methods we will study have been developed for solving m first-order equations as these cover the most general case, but can also be applied to higher-order equations as well.

7.2 Finite difference methods

In this course, we will look at **finite difference methods**. In these methods, we do not attempt to solve to obtain a continuous function $x(t)$, but we solve for values of the dependent variables at discrete points in time $x(t_0)$, $x(t_1)$, etc. If a smooth continuous function is required, then interpolation (as described in section 4) can be done between the discrete points afterwards. In all the methods described in detail here, the time steps are taken to be equal and denoted by h so the time values will be $t_0 + nh$ for integer n. However, varying the step size occurs in more sophisticated methods.

The basic concept for the one variable case is that the constant of integration gives a value of $x = x_0$ at some time t_0 and we then step forward and/or backward in time until we cover the required range of time for which we want the solution. Because we can calculate the derivative

[1] The classical physics Lagrangian and Hamiltonian approach can be considered as a formal method for taking the Euler-Lagrange second-order equations and converting them into twice as many Hamiltonian first-order equations.

at t_0 using $x_0' = f(t_0, x_0)$, we can make some estimate of the function at some small time step h away, i.e. at $t_1 = t_0 + h$. All the methods discussed in this and the next section are about how to make such estimates. Having got an estimate of e.g. $x_1 = x(t_1)$, we can use this value to find $x_1' = f(t_1, x_1)$ and so project further in time for the next estimate and so on. For clarity we will not consider the most general stepping in both directions for an arbitrary starting time. We will assume we only want to step forwards from a time t_0. Hence, the known value of $x_0 = x(t_0)$ is the starting value. This means these problems are solved using what are called **initial value methods**.

The multiple variable case is more complicated. If the values of the variables are all defined for the same time t_0, i.e. $\vec{x}(t_0)$ is known, then the same concept as for the one variable case can be applied as we can calculate the derivatives $\vec{x}' = \vec{f}(t_0, \vec{x}_0)$ and hence can proceed as before; this is also an initial value problem. However, if any (or all!) are specified at different times, then we do not have enough information to calculate the derivatives at any single time in general, as they all depend on all components of $\vec{x}$. In this case we cannot use initial value methods to solve within the region between the earliest and latest time at which any of the values are specified. Instead we have to use different approaches called **boundary value methods**, so-called as there will be at least one variable specified at each of the two ends of the time period, i.e. on the "boundaries" of the time period. Having solved for this time range, initial value methods can be used to extend in time beyond the ends of the range in either or both directions. Again, for clarity, we will just consider solving the boundary value range by itself; these methods are discussed in section (9).

All methods for solving these equations have been developed to keep down the number of evaluations of the functions which give the derivatives. This is because these functions can be non-trivial; they could be the result of a significant numerical calculation themselves. Hence, the methods are optimised to squeeze as much accuracy as possible out of the evaluations that are done. They are all based on Taylor series expansions.

7.3 The Euler method

The simplest initial value method for numerical integration of an ODE is obtained by looking again at the Taylor expansion as done for a forward difference. Using time t as the independent variable,

$$\begin{aligned} x(t+h) &= x(t) + h\,x'(t,x) + \mathcal{O}(h^2) \\ &\approx x(t) + h\,f(t,x)\,. \end{aligned} \tag{7.7}$$

So if we know the value of the variable x at t we can use a finite step h to calculate the next value of x at $t + h$ up to $\mathcal{O}(h^2)$ accuracy. This is illustrated in figure (8.1).

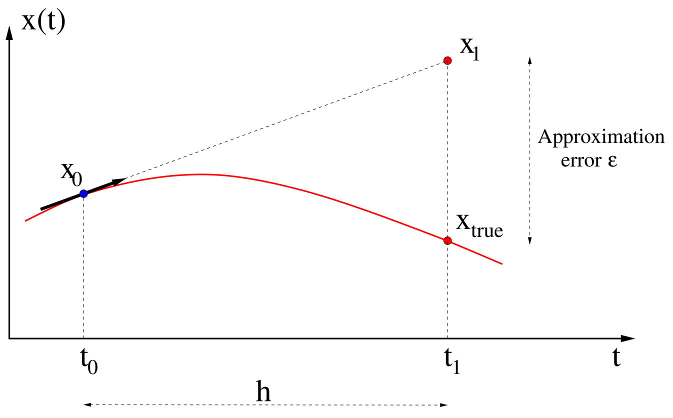

Figure 7.1: The Euler method applied to a non-linear function shown in red. The first step from t_0 to t_1 is made by projecting along the derivative at t_0, as shown by the black arrow. The approximation error arises because the gradient is evaluated at the starting point and used to obtain the next point, while the true function is non-linear.

This Euler step can be iterated, stepping the solution forward h each time; we can write this as

$$\boxed{x_{n+1} = x_n + h\, f(t_n, x_n),} \tag{7.8}$$

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

$$t_n = t_0 + n\, h. \tag{7.9}$$

The Euler method can also be applied to systems with multiple variables. The method is effectively identical and the equations are

$$\vec{x}_{n+1} = \vec{x}_n + h\, \vec{f}(t_n, \vec{x}_n), \tag{7.10}$$

The main thing to note is that in general the functions $\vec{f}$ each depend on *all* the variables $\vec{x}$ so you need to make sure all are updated to step n before moving on to calculating any of the $\vec{x}_{n+1}$ values.

Linear systems

A linear system with one variable is generally written as

$$x' = f(t, x) = a(t)x + b(t) \tag{7.11}$$

where a and b can in general be functions of time. This can in fact be divided into two independent integrals for each of the two terms. The $b(t)$ integral is straightforward to do by Euler's method, so consider only the first

$$x' = a(t)x \tag{7.12}$$

If a is not time-dependent, the solution is easily found; $x = x_0 e^{at}$. Otherwise, writing $a(t_n) = a_n$, the Euler method gives

$$x_{n+1} = x_n + h\, a_n x_n = (1 + h\, a_n)x_n \tag{7.13}$$

Hence, the value of x_n is "updated" by a factor of $1 + h\, a_n$. With multiple variables, the equivalent equations are

$$\vec{x}' = \mathbf{A}(t) \cdot \vec{x} \tag{7.14}$$

where $\mathbf{A}$ is an $m \times m$ matrix for m variables. If this matrix is not a function of time, the formal solution is $\vec{x} = e^{\mathbf{A}t} \cdot \vec{x}_0$ where the exponential of a matrix is defined through its series expansion

$$e^{\mathbf{A}t} = \mathbf{I} + \mathbf{A}t + \mathbf{A}^2 \frac{t^2}{2!} + \mathbf{A}^3 \frac{t^3}{6!} + \ldots \tag{7.15}$$

Writing $\mathbf{A}(t_n) = \mathbf{A}_n$, the Euler method gives

$$\vec{x}_{n+1} = \vec{x}_n + h\, \mathbf{A}_n \cdot \vec{x}_n = (\mathbf{I} + h\, \mathbf{A}_n) \cdot \vec{x}_n = \mathbf{H}_n \cdot \vec{x}_n \tag{7.16}$$

where $\mathbf{H}_n$ is called the update matrix. This has a very similar structure to the iteration matrix inversion equations discussed in sections (3.6) and (3.7).

7.4 Properties of initial values methods

The Euler method is an example of a **finite difference method** (FDM). We will cover a number of more advanced methods for solving ODEs numerically in section (8). We will find they are all of the form

$$x_{n+1} = x_n + h\, \overline{x}'_n \tag{7.17}$$

where $\overline{x}'_n$ is an estimate of the average gradient within the period of the step. This is a function of the input information which each method uses but typically includes t_n and x_n. For the Euler method, $\overline{x}'_n = f(t_n, x_n)$, i.e. the average gradient is approximated to be the gradient at the start of the step period.

Before looking at other methods in detail, in this section we will look at how to evaluate the properties of any particular method. We will look at **efficiency**, **accuracy** and **convergence**.

Efficiency is basically a measure of how much computational power the method needs. A method which takes weeks to calculate a solution is usually not feasible. A significant consideration when selecting a method is therefore the number of computations required for each step. Typically, the step computation is dominated by evaluating the derivative function $f(t, x)$, which can be complicated or even itself a numerical calculation. The Euler method requires one functional evaluation for each step and so is efficient in terms of CPU usage. As we will see, higher-order methods will require more evaluations at each step but will in general be more stable and accurate. However, as they are more accurate, higher-order methods can work well with bigger step sizes and hence a smaller number of steps. A compromise must be reached between efficiency, accuracy and stability for any particular system being solved.

Convergence means that the numerical solution x_n will converge to the true solution $\tilde{x}_n$ in the limit of zero step size $h \to 0$ (in the absence of computational errors). If a finite difference method is both **consistent** with the ODE (system) being solved and the errors in the method are **stable**, then it will converge.

The properties of **accuracy**, **consistency** and **stability** are discussed in more detail below.

7.5 Accuracy

To determine the accuracy of a method we want to find the **local** or **step** error (the error incurred in each step) and use it to estimate the **global error** (the error at the end of the calculation).

Local errors

The local error can be split into three contributions; the **truncation** or **approximation** local error, the **computational** local error and the **propagation** error amplification.

Using the Euler method as an example we know that the local truncation error is $\mathcal{O}(h^2)$, as shown by equation (7.7). This means $\tilde{x}_{n+1}$, the **true** value of the function at t_{n+1}, is related to the numerical approximation of the function by

$$x_{n+1} \equiv x_n + h\, f(t_n, x_n) = \tilde{x}_{n+1} + \mathcal{O}(h^2), \tag{7.18}$$

where we have assumed here that x_n is known exactly. Equation (7.18) shows us the approximation error a incurred in performing one step of the Euler method. Higher-order methods by definition have truncation errors with higher powers of h; we will see methods with truncation errors up to $a \sim \mathcal{O}(h^5)$.

The computational error μ cannot be quantified in a general way. It depends on the method but is typically dominated by the calculations of the derivative function $f(t, x)$. If this is an analytic function, using 64-bit floating point values can give a relative local computational error of $\mathcal{O}(10^{-17})$ to $\mathcal{O}(10^{-15})$, depending on the function complexity. Functions which are the result of numerical calculations can be significantly less accurate. Note, the above numbers are the fractional accuracy; the error μ is the product of this with the function value.

The propagation error arises because in reality x_n is not known exactly; it has an error. The inexact value of x_n is used in the method to calculate x_{n+1} so the error on x_n propagates through the method calculations and contributes to the error on x_{n+1}. This is quantified by the amplification factor g_n. If the total error on x_n is ϵ_n and the error on x_{n+1} due to ϵ_n only (i.e. in the absence of truncation or computational errors) is ϵ_{n+1}, then the local amplification factor is defined to be

$$g_n = \frac{\epsilon_{n+1}}{\epsilon_n} \tag{7.19}$$

The effect of the error in increased if $g_n > 1$ so this must be avoided. This is what is meant by stability, which is discussed below. We will assume $g \approx 1$ for the discussion on global errors.

The total error after the step is the combination of the three sources above so

$$\epsilon_{n+1} = g_n \epsilon_n + a_n + \mu_n \tag{7.20}$$

Global errors

Assuming the amplification factors are $g \approx 1$, then the total error at any step is effectively the combination of all the truncation and computational errors from the previous steps. The **global error** is the total error at the end of the solution, after all N steps. To solve for a fixed time period from t_0 to t_N takes $N = (t_N - t_0)/h$ steps; hence the number of steps is proportional to $1/h$.

If we conservatively assume the local truncation error at each of the steps simply adds up, in general, we will have

$$\boxed{\textbf{global truncation error} \propto \text{local truncation error} \times \text{number of steps}\,.} \tag{7.21}$$

In the case of the Euler method, the global error ϵ_N is

$$\epsilon_N \propto \mathcal{O}(h^2) \times 1/h \sim \mathcal{O}(h)\,. \tag{7.22}$$

In general **a method is called $\boldsymbol{m}^{\textbf{th}}$-order** if it keeps terms up to $\mathcal{O}(h^m)$ in each step, so its local truncation error is of $\mathcal{O}(h^{m+1})$ and its **global truncation error is of $\boldsymbol{\mathcal{O}(h^m)}$.**

You might think that all we need to do is reduce the step size h (by increasing the number of steps) to increase the accuracy arbitrarily. Even ignoring the computation time to calculate a huge number of steps, unfortunately the computational error places a limit on how accurate any method can be. Unlike the truncation errors, the computational errors tend to be more like random variables and so are typically added in quadrature. In this case, we have

$$\boxed{\textbf{global computational error} \propto \text{local computational error} \times \sqrt{\text{number of steps}}\,.} \tag{7.23}$$

Hence, considering both truncation and computational errors, the global error for an m^{th}-order method will be

$$\epsilon_N \sim \mathcal{O}\left(\frac{\mu}{\sqrt{h}}\right) + \mathcal{O}(h^m)\,. \tag{7.24}$$

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

$$h \sim \mathcal{O}(\mu^{1/(m+1/2)})\,. \tag{7.25}$$

For double precision on most machines the relative computational error is $\mathcal{O}(10^{-16})$. Assuming natural units so the x_n are all $\mathcal{O}(1)$, for the Euler method the smallest useful step size is $h \sim 10^{-8}$ which will give a global error of $\epsilon_N \sim 10^{-8}$. However, it is unusual to be able to work with a step size this small, so the Euler method error is typically dominated by the truncation error. In contrast, for a higher-order method, such as fourth-order, the equivalent expression is $h \sim \mathcal{O}(\mu^{2/9}) \sim 10^{-4}$ which is quite feasible and gives a global error $\epsilon_N \sim 10^{-13}$.

7.6 Consistency

This is a check that the finite difference method (e.g. the Euler method) reduces to the correct differential equation (i.e. $dx/dt = f(t, x)$) in the limit of vanishingly small 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 actually samples the exact solution of the modified differential equation at the discretised time t_n (or the relevant discretised independent variable).

A consistency analysis is carried out by taking the equation for a finite difference method and comparing it to a Taylor expansion about the step starting point (e.g. t_n). We can express all the methods we will study as

$$x_{n+1} = x_n + h\,\overline{x}'_n \tag{7.26}$$

where $\overline{x}'_n$, the estimate of the average gradient with the step period, is calculated differently for each method. The result is an approximation to the true value $\tilde{x}_{n+1}$. We can Taylor expand to express the true $\tilde{x}_{n+1}$ in terms of derivatives at t_n. It is a time h forward from t_n so this yields

$$\tilde{x}_{n+1} = x_n + x'_n\,h + \frac{x''_n}{2!}\,h^2 + \frac{x'''_n}{3!}\,h^3 + \cdots = x_n + h\,f(t_n, x_n) + \frac{x''_n}{2!}\,h^2 + \frac{x'''_n}{3!}\,h^3 + \ldots, \tag{7.27}$$

For consistency, we need $x_{n+1} \to \tilde{x}_{n+1}$ as $h \to 0$. This means we need

$$\overline{x}'_n \to f(t_n, x_n) + h\,\frac{x''_n}{2!} + h^2\,\frac{x'''_n}{3!} + \ldots \tag{7.28}$$

and since all the higher derivative terms vanish when $h = 0$ this becomes

$$\boxed{\overline{x}'_n \to f(t_n, x_n)} \tag{7.29}$$

Hence, for any method the average gradient should become the true gradient when the step shrinks to zero width. All the methods we will study have this property. This is obviously true of the Euler method as $\overline{x}'_n = f(t_n, x_n)$ for this method, even with non-zero h.

As stated above, another way to consider this is by considering the **modified differential equation** (MDE). Let the solution of the MDE be $y(t)$. We can do a Taylor expansion as above but in terms of $y(t)$ which by definition should give the same value of x_{n+1} as the method does. We need to start the expansion from the same point, i.e. $y_n = x_n$ so we have

$$x_{n+1} = x_n + y'_n\,h + \frac{y''_n}{2!}\,h^2 + \frac{y'''_n}{3!}\,h^3 + \cdots = x_n + h\,\overline{x}'_n \tag{7.30}$$

This can be rearranged into

$$y'_n = \overline{x}'_n - \frac{y''_n}{2!}\,h - \frac{y'''_n}{3!}h^2 - \ldots. \tag{7.31}$$

Allowing y to become continuous, i.e. $y_n = y(t_n) \to y(t)$, and noting that $x_n = y_n$ so $\overline{x}'_n \to \overline{y}'_n$, this is converted into a differential equation for y

$$\boxed{y' = \overline{y}' - h\,\frac{y''}{2!} - h^2\,\frac{y'''}{3!} - \ldots}. \tag{7.32}$$

This is the MDE for the method. Note, we could have specified that consistency requires the MDE to become the original equation in the $h \to 0$ limit. This would mean $y' = f(t, y)$ and hence the consistency requirement reduces to $f(t, y) \to \overline{y}'$, which is the same condition as previously. The discrete points solved by the method (t_n, x_n) lie on the continuous curve of $y(t)$ that solves the MDE (given the same initial condition of course!). The Euler method has $\overline{y}' = f(t, y)$ so its MDE is

$$y' = f(t, y) - h\,\frac{y''}{2!} - h^2\,\frac{y'''}{3!} - \ldots \tag{7.33}$$

All the higher-order derivatives remain in the MDE, and form the difference between the true equation and the MDE. This is as expected as the step is a second-order approximation. For higher-order methods, the form of $\overline{y}'$ is such that some of the higher-order derivatives are cancelled. This makes $y(t)$ a better approximation to the true $x(t)$.

Consistency analysis can be carried out with finite difference methods for partial differential equations too, as discussed in Section (12).

7.7 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 with iteration 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

$$x_n = \tilde{x}_n + \epsilon_n, \tag{7.34}$$

where $\tilde{x}_n$ is the true solution of the ODE at $t = t_n$ and x_n is the numerical approximation. In taking one step of a finite difference method the numerical approximate solution and the error change and satisfy

$$x_{n+1} = \tilde{x}_{n+1} + \epsilon_{n+1}. \tag{7.35}$$

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

$$\boxed{|g_n| \equiv \left|\frac{\epsilon_{n+1}}{\epsilon_n}\right| \leq 1,} \tag{7.36}$$

otherwise the error in the finite difference method exponentially 'blows up'. Note that this is evaluating the absolute error; sometimes the relative error is more relevant but this is not covered here.

We consider the Euler method as an example. Substituting in the true values of the function to the iteration step, we have

$$\tilde{x}_{n+1} + \epsilon_{n+1} = \tilde{x}_n + \epsilon_n + h\, f(t_n, \tilde{x}_n + \epsilon_n). \tag{7.37}$$

Stability analysis can only be carried out precisely for linear ODEs. Luckily for non-linear ODEs, i.e. when f is a non-linear function of x, we can still do a stability analysis by assuming that the errors are small, i.e. $|\epsilon_n|$ and $|\epsilon_{n+1}| \ll |x_n|$, and **linearising $\boldsymbol{f(t,x)}$**. To linearise, the function $f(t_n, \tilde{x}_n + \epsilon_n)$ is expanded around the point $(t_n, \tilde{x}_n)$ in the variable x. To first-order in the expansion parameter ϵ_n the equation (7.37) becomes

$$\begin{aligned} \tilde{x}_{n+1} + \epsilon_{n+1} &= \tilde{x}_n + \epsilon_n + h\left[f(t_n, \tilde{x}_n) + \left.\frac{\partial f}{\partial x}\right|_n \epsilon_n + \mathcal{O}(\epsilon_n^2)\right], \\ &= \tilde{x}_n + h\, f(t_n, \tilde{x}_n) + \epsilon_n\left[1 + h\left.\frac{\partial f}{\partial x}\right|_n\right] + \mathcal{O}(\epsilon_n^2)\,. \end{aligned} \tag{7.38}$$

where $\partial f/\partial x|_n$ means the partial derivative is evaluated at the base point t_n, $\tilde{x}_n$. Then since

$$\tilde{x}_{n+1} = \tilde{x}_n + h\, f(t_n, \tilde{x}_n) + \mathcal{O}(h^2), \tag{7.39}$$

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

$$\boxed{\epsilon_{n+1} \approx \epsilon_n \left(1 + h\left.\frac{\partial f}{\partial x}\right|_n\right)\,,} \tag{7.40}$$

for the Euler method. Inserting $\epsilon_{n+1}/\epsilon_n$ into equation (7.36), and dropping the $|_n$ notation, we can get a condition on the step size h for stability:

$$|g| = \left|1 + h\,\frac{\partial f}{\partial x}\right| \leq 1 \quad \rightarrow \quad -2 \leq h\,\frac{\partial f}{\partial x} \leq 0\,. \tag{7.41}$$

Assuming $h > 0$ the Euler method is only stable if

$$\boxed{\frac{\partial f}{\partial x} < 0} \qquad \text{and} \qquad \boxed{h \leq \frac{2}{\left|\frac{\partial f}{\partial x}\right|}.} \tag{7.42}$$

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

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 truncated $\mathcal{O}(h^2)$ term dropped in equation (7.39) and through computational errors.

Stability of linear systems

For a linear one-variable ODE, i.e. $f(x,y) = a(t)x + b(t)$, we have $\partial f/\partial x = a(t)$ and so g and the limiting step size are a function of t only, and controlled by $a(t)$. Specifically

$$g = 1 + h\,\frac{\partial f}{\partial x} = 1 + h\,a(t) \tag{7.43}$$

which is the update factor for the Euler method. Note, the stability can change with time. For constant coefficients, e.g., the decay problem $x' = -\alpha x$ (with α positive), the stability conditions $|g| = |1 - \alpha\,h| \le 1$ and $h \le 2/a$ do not change over the calculation.

A more general stability analysis can be carried out for N coupled, linear first-order ODEs by looking at the general matrix operator form for the method

$$\vec{x}_{n+1} = \mathbf{H}\cdot\vec{x}_n, \tag{7.44}$$

where $\mathbf{H}$ is the update matrix (e.g. $\mathbf{H} = \mathbf{I} + h\mathbf{A}$ for the Euler method). We can relate the true solution vector $\tilde{\vec{x}}_n$ to the numerical approximation $\vec{x}_n$ in an analogous way to before;

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

where now $\vec{\epsilon}_n = \{\epsilon^0_n,\, \epsilon^1_n,\, ...,\, \epsilon^{m-1}_n\}$. Inserting this into the finite difference equation (7.44) yields

$$\tilde{\vec{x}}_{n+1} + \vec{\epsilon}_{n+1} = \mathbf{H}\cdot\tilde{\vec{x}}_n + \mathbf{H}\cdot\vec{\epsilon}_n.$$

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

$$\tilde{\vec{x}}_{n+1} = \tilde{\vec{x}}_n + h\,\vec{f}_n + \mathcal{O}(h^2) = (\mathbf{I} + h\,\mathbf{A})\cdot\tilde{\vec{x}}_n + \mathcal{O}(h^2) \approx \mathbf{H}\cdot\tilde{\vec{x}}_n,$$

where $\vec{f}_n = \{f^0(\tilde{\vec{x}}_n), f^1(\tilde{\vec{x}}_n), \ldots, f^{m-1}(\tilde{\vec{x}}_n)\}$, i.e., its components are the functions in each ODE in $d\vec{x}/dt = \vec{f}(t,\vec{x})$. This yields a matrix equation for the error propagation;

$$\vec{\epsilon}_{n+1} = \mathbf{H}\cdot\vec{\epsilon}_n. \tag{7.45}$$

The terms of $\mathcal{O}(h^2)$, discarded in obtaining at equation (7.45), are the source of global error. However, they are not relevant to the question of stability. It turns out that condition for stability is that the modulus of all the eigenvalues λ_i of the update matrix $\mathbf{H}$ must be less than or equal to unity

$$\boxed{|\lambda_i| \le 0 \quad \text{for all } i} \tag{7.46}$$

This means the eigenvalue with the largest magnitude must be less than 1. This is an identical condition to the requirement for convergence when solving linear algebraic equations using the Jacobi or Gauss-Seidel methods as described in sections (3.6) and (3.7) and the same techniques for find the largest eigenvalue can be applied here.

To show condition (7.46) 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 $m \times m$ matrix that has m linearly independent eigenvectors can be diagonalised, i.e., written as

$$\boxed{\mathbf{H} = \mathbf{R}\cdot\mathbf{D}\cdot\mathbf{R}^{-1} \quad \text{with} \quad \mathbf{D} \equiv \text{diag}(\lambda_0, \lambda_1, ...),} \tag{7.47}$$

where λ_i are the eigenvalues of $\mathbf{H}$ and $\mathbf{R}$ is the matrix whose columns are the eigenvectors of $\mathbf{H}$. Substituting equation (7.47) into (7.45) and operating on both sides with a further $\mathbf{R}^{-1}$ we obtain the diagonal (uncoupled) system

$$\vec{\zeta}_{n+1} = \mathbf{D}\cdot\vec{\zeta}_n, \tag{7.48}$$

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

$$\zeta_{n+1}^i = \lambda_i \, \zeta_n^i \, . \tag{7.49}$$

Since these are uncoupled we can now impose the constraint on their growth separately for each of them, i.e., for all $i = 0, 1, \ldots, m-1$ we have the requirement;

$$g_n^i = \left| \frac{\zeta_{n+1}^i}{\zeta_n^i} \right| = |\lambda_{i\,n}| \leq 1, \tag{7.50}$$

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 in equation (7.50) holds for the original equation (7.45) too.

Stability of non-linear systems

For a non-linear ODE the amplification factor and step size for stability change with both x and t. However, 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 linearised around the base point $(t_n, \vec{x}_n)$. This is necessary to get the equivalent of the update matrix $\mathbf{H}$ that applies at the current step in the iteration.

Considering the Euler method, we have a set of equations

$$\vec{x}_{n+1} = \vec{x}_n + h \, \vec{f}(t, \, \vec{x}_n). \tag{7.51}$$

We set $\vec{x}_n = \tilde{\vec{x}}_n + \vec{\epsilon}_n$ to put this in terms of the true solution. We substitute this into $\vec{f}(t, \tilde{\vec{x}}_n)$ and Taylor expand about the true solution to first-order in the error, assumed to be small $|\epsilon_n^i| \ll |\tilde{x}_n^i|$;

$$f^i(t, \tilde{x}_n^1 + \epsilon_n^1, \, \tilde{x}_n^2 + \epsilon_n^2, \, \ldots \, , \, \tilde{x}_n^m + \epsilon_n^m) = f^i(t, \tilde{\vec{x}}_n) + \sum_{k=1}^{m} \left. \frac{\partial f^i}{\partial x^k} \right|_n \epsilon_n^k + \mathcal{O}((\epsilon_n)^2). \tag{7.52}$$

This is essentially what was done in section (7.7) with equation (7.38). Following the procedure used before we then get

$$\epsilon_{n+1}^i \approx \epsilon_n^i + h \sum_{k=0}^{m-1} \left. \frac{\partial f^i}{\partial x^k} \right|_n \epsilon_n^k \tag{7.53}$$

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

$$\boxed{H_{jk} = \delta_{jk} + h \, \frac{\partial f^j}{\partial x^k},} \tag{7.54}$$

where δ_{jk} is the Kronecker delta function. (Note that the superscripts in x^j, f^i, etc., denote components and not 'raising to a power'.) The partial derivatives form the matrix $\mathbf{J}$, the Jacobian of the derivative functions with respect to the dependence variables, so this is

$$\mathbf{H} = \mathbf{I} + h\mathbf{J} \tag{7.55}$$

The eigenvalues of this update matrix can then be analysed to determine the local stability of the method. In principle this needs to be done for every step as the Jacobian matrix depends on t and $\vec{x}$.

Chapter 8

ODEs: Initial Value Methods

Outline of Section

- Higher-order Taylor methods
- Multi-step methods
- Implicit methods
- Runge-Kutta methods
- Relationship to interpolation
- Relationship to integration

8.1 Introduction

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

$$x_{n+1} = x_n + f_n(t_n, x_n)\, h. \tag{8.1}$$

Its properties are summarised as follows:

- The local truncation error (i.e. the error per step) is $\mathcal{O}(h^2)$. This means it is a first-order method: the global error is $\mathcal{O}(h)$.
- It is a consistent method: it solves the original equation in the limit of the step size $h \to 0$.
- It is conditionally stable: for a single ODE with a test function $x' = ax$, the step size h must satisfy $|a|h \leq 2$ and since h is positive, it is unstable if $a > 0$.

In this section we look at some more finite difference methods which are superior to Euler. As shown in Fig. 8.1 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. Hence, to improve on the Euler method, we generally need to go to a higher-order truncation error per step to improve the accuracy, although we also need to be careful about stability. To do this requires using more information, specifically from the function $f(t, x)$ or from other values of t and x.

These improved methods can be considered as extensions of the Euler method and the different methods do this in different ways. There are two main categories for methods:

- **Single-step** methods only use the solution at the current iteration (x_n) to get the next value x_{n+1}. In contrast, **multi-step** methods use values from other steps as well (e.g. x_{n-1}) in order to get x_{n+1}. Euler is a single-step method.
- **Explicit** methods involve evaluating $f(t, x)$ using only known values of x, e.g. x_n. There are also **implicit** methods which use $f(t_{n+1}, x_{n+1})$ as part of the method. Since the resulting value will be used to find x_{n+1}, these require iterating or some other manipulations to find the solution. Multi-step implicit methods use $f(t, x)$ both at known and unknown x values. Euler is an explicit method.

Typically, higher-order methods give better accuracy but take more computational time, which can be balanced by adjusting the step side h. The appropriate choice depends on the particular problem being solved, but generally fourth-order methods give a reasonable balance of accuracy and efficiency. Explicit formulae for fourth-order methods are given where appropriate below.

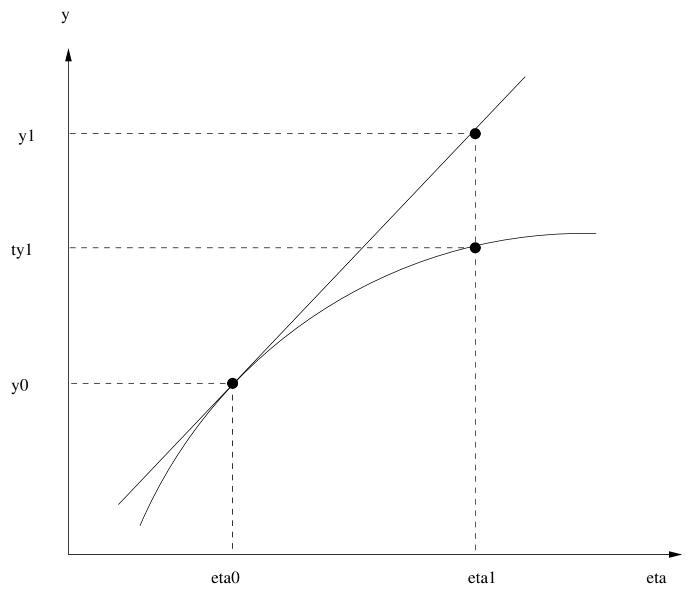

Figure 8.1: The Euler method applied to a non-linear function. The truncation error arises because the gradient is evaluated at the starting point and used to obtain the next point. Here, $y1$ is the 'approximate' solution (i.e. x_1) according to the Euler method and $ty1$ (on the curve) is the true solution (i.e. $\tilde{x}_1$) to the ODE.

To put the Euler method in context, all the initial value methods we will look can be considered to have the form

$$x_{n+1} = x_n + h\,\overline{x}'_n \tag{8.2}$$

where $\overline{x}'_n$ is some estimate of the average of the derivative function over the step period. For (almost) all methods discussed, this is a weighted average of one or more function evaluations somewhere within the step

$$\overline{x}'_n = \sum_a w_a\, f(t_n + \delta t_a, x_n + \delta x_a) \qquad \text{where} \qquad \sum_a w_a = 1 \tag{8.3}$$

Since all the arguments of the function evaluations are within the step, then they all go to t_n and x_n in the limit of $h \to 0$. The average derivative also becomes the derivative at t_n. The sum becomes

$$\sum_a w_a\, f(t_n + \delta t_a, x_n + \delta x_a) \to f(t_n, x_n) \sum_a w_a = f(t_n, x_n) \tag{8.4}$$

This means the fact that the weights sum to 1 ensures that the methods are consistent; as shown in section (7.6), this requires $\overline{x}'_n \to f(t_n, x_n)$ in this limit.

Hence, the Euler method corresponds to approximating the average gradient by the gradient at the initial time of the step. This means there is one evaluation of the function at t_n, i.e. using $\delta t = 0$, and using the value at that point x_n, so $\delta x = 0$. Since there is only one evaluation, it must therefore have a weight of 1. For explicit comparison with other methods, we can write the Euler method as

$$\boxed{\begin{aligned} f_a &= f(t_n, x_n), \\ x_{n+1} &= x_n + h\, f_a \end{aligned}} \qquad \begin{aligned} &(8.5a) \\ &(8.5b) \end{aligned}$$

8.2 Higher-order Taylor methods

We want to add higher-order terms to improve the step accuracy. The Euler method is based on the first two terms of the Taylor series around x_n

$$x_{n+1} = x_n + x'_n h + x''_n \frac{h^2}{2!} + x'''_n \frac{h^3}{3!} + \dots \tag{8.6}$$

where $x'_n = f(t_n, x_n)$ and the terms of h^2 and higher are ignored. The most obvious extension of the Euler method is to use more terms in the Taylor expansion it is based on. This can be done if the derivative function $f(t,x)$ is known analytically. In principle, this can then be differentiated to give the second derivative of x

$$x'' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x} x' = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x} f \tag{8.7}$$

and this can be continued to give $x''' = dx''/dt$, etc. This is called a **higher-order Taylor method**. Hence, the extra information needed in this method comes from our knowledge of the function $f(t,x)$.

This can be interpreted to mean the average gradient across the step $\overline{x}'_n$ is given by

$$\overline{x}'_n = x'_n + x''_n \frac{h}{2!} + x'''_n \frac{h^2}{3!} + \dots \tag{8.8}$$

which is indeed exact if all terms are included. In this particular method, unlike all the others, we are not using a weighted sum of derivative estimates to find $\overline{x}'_n$. However, it is clearly consistent since when $h \to 0$, the only remaining term is $x'_n = f(t_n, x_n)$. For a test function $x' = ax$, the Taylor methods give the error amplification factor

$$g = 1 + ah + \frac{a^2 h^2}{2!} + \frac{a^3 h^3}{3!} + \dots \tag{8.9}$$

The higher-order Taylor methods are all stable for decaying exponentials, and stable (or nearly stable) for oscillating solutions.

The higher-order Taylor methods are limited by the complexity of handling the differentiation required (both analytically and in terms of coding) and eventually by the rounding error which will prevent the higher-order h^m terms from contributing. However, for relatively simple analytic functions, this is a good choice of method.

In practise, the function $f(t,x)$ is not necessarily known analytically and even if it is, there can be difficulties in implementing the derivatives accurately. It might be thought that the higher derivatives could be accurately estimated using numerical calculus techniques, e.g. for x'' we need

$$\frac{\partial f}{\partial t} \approx \frac{f(t_n + \epsilon_t, x_n) - f(t_n, x_n)}{\epsilon_t} \quad \text{and} \quad \frac{\partial f}{\partial x} \approx \frac{f(t_n, x_n + \epsilon_x) - f(t_n, x_n)}{\epsilon_x} \tag{8.10}$$

for some very small ϵ_t and ϵ_x. While this is true, the number of function evaluations needed per step may be quite high so the method would not be very efficient. In a similar way to the secant method compared to the Newton-Raphson method in Section (3.10), calculating with $\epsilon_x \ll h$ does not bring any advantages as we would only be finding a very precise result for the truncated (i.e. approximate) equation anyway. Hence, it turns out there are clever ways to space out any extra function evaluations which are more efficient; these are the Runge-Kutta methods described later in this chapter.

8.3 Multi-step methods

As stated previously, multi-step methods use other values of t and/or x than just t_n and x_n in order to find x_{n+1}. Clearly, these other values bring in the extra information needed.

Explicit multi-step methods use the previous values already calculated in earlier time steps, e.g. x_{n-1} and/or $f(t_{n-1}, x_{n-1})$. Since these values are already known, no extra function evaluations are required and so these methods can be very efficient per step.

Second-order multi-step using x

How would we do an explicit multi-step method to one order better than Euler, i.e. retaining the h^2 term? We have one unknown x''_n so we need one extra value; let's use x_{n-1}. The Taylor expansion around x_n for x_{n-1} (which we know) and x_{n+1} (which we want to find) retaining the h^2 terms is

$$\begin{aligned} x_{n-1} &= x_n - x'_n h + x''_n \frac{h^2}{2!} \\ x_{n+1} &= x_n + x'_n h + x''_n \frac{h^2}{2!} \end{aligned} \tag{8.11}$$

In this case, it is trivial to eliminate x''_n by hand but let's do this a little more systematically as it is then obvious how to go to even higher orders. The above can be written as a matrix equation

$$\begin{pmatrix} x_{n-1} - x_n \\ x_{n+1} - x_n \end{pmatrix} = \begin{pmatrix} -1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x'_n h \\ x''_n h^2/2 \end{pmatrix} \tag{8.12}$$

This is easily inverted to give

$$\begin{pmatrix} x'_n h \\ x''_n h^2/2 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} -1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_{n-1} - x_n \\ x_{n+1} - x_n \end{pmatrix} \tag{8.13}$$

Writing these out in full gives

$$\begin{aligned} x'_n h &= \frac{1}{2}(x_{n+1} - x_{n-1}) \\ x''_n h^2/2 &= \frac{1}{2}(x_{n+1} - 2x_n + x_{n-1}) \end{aligned} \tag{8.14}$$

which you should recognise as the three-point "centre difference" gradient and second-order derivative equations presented in Section (5.1). Indeed, you should know from that section than the centre difference gradient estimate is more accurate than the forward difference used in the Euler method.

The second equation above shows we have evaluated x''_n, as required. However, it turns out we don't need that solution explicitly; the first equation we know is equal to $f(t_n, x_n)h$ so that directly gives

$$\boxed{x_{n+1} = x_{n-1} + 2f(t_n, x_n)\, h,} \tag{8.15}$$

This is called the **leapfrog** method as it jumps over x_n to move from x_{n-1} to x_{n+1}. In this method the point (t_n, x_n) – where the gradient is used – is the midpoint between x_{n-1} and x_{n+1}. The leapfrog algorithm therefore essentially uses a central difference scheme to approximate the average derivative $\overline{x}'_n$ in the ODE. The leapfrog (second-order, multi-step) method is **explicit, $\mathcal{O}(h^2)$ accurate** and only requires one evaluation per step. This makes it very efficient per step.

We can get a bit more insight into this method by rearranging the step equation into the following form

$$x_{n+1} = x_n + h \left[2f(t_n, x_n) - \left(\frac{x_n - x_{n-1}}{h} \right) \right] \tag{8.16}$$

It is now seen to be a weighted average of the gradient function evaluated at t_n and x_n and the numerical calculus backward difference gradient estimate; see section (5.1). The weights are 2 and -1 respectively which do sum to 1 and so this is a consistent method. This looks sensible but in fact, having large weights with positive and negative signs which require cancellation can make the solution unstable to the small errors in the x_n and x_{n-1} as they are not known exactly. The error amplification factor for a test function $x' = ax$ is

$$g = ah \pm \sqrt{1 + a^2 h^2} \tag{8.17}$$

The leapfrog method is just stable for oscillating solutions but is unstable for exponential solutions.

Higher-order multi-step using x

We might hope to improve by going to even higher orders, which requires more points. We can introduce another point, specifically x_{n-2}, to allow us to solve for another unknown, specifically x_n'''. The calculation involves an extension of the leapfrog derivation using a 3×3 matrix equation. This leads to

$$x_{n+1} = x_n + \frac{h}{2}\left[6f(t_n, x_n) - 5\left(\frac{x_n - x_{n-1}}{h}\right) + \left(\frac{x_{n-1} - x_{n-2}}{h}\right)\right] \tag{8.18}$$

The weights are now 3, $-5/2$ and $1/2$, so while they sum to one and so are consistent, if anything they have got larger magnitudes and hence would be expected to be even less stable! It turns out this is not a good approach; it is similar to including too many higher-order terms in Lagrange polynomial interpolation, where the resulting function can have large fluctuations.

Second-order multi-step using f

The solution is to dampen down the fluctuations by using information on the gradient instead. Having well controlled gradients can limit the ability of the solution to fluctuate wildly. The method is very similar but we use $f(t_{n-1}, x_{n-1})$ rather than x_{n-1} as the extra information. To get a second-order step method, doing the Taylor expansion for x' (not x) around x_n' gives

$$x_{n-1}' = x_n' - h\,x_n'' \tag{8.19}$$

(You can alternatively think of this as the derivative of the x_n Taylor series with respect to h.) Hence the matrix equation to use is

$$\begin{pmatrix} h\,x_{n-1}' \\ x_{n+1} - x_n \end{pmatrix} = \begin{pmatrix} 1 & -2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} h\,x_n' \\ h^2\,x_n''/2 \end{pmatrix} \tag{8.20}$$

This is easily inverted to give

$$\begin{pmatrix} h\,x_n' \\ h^2\,x_n''/2 \end{pmatrix} = \frac{1}{3}\begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} h\,x_{n-1}' \\ x_{n+1} - x_n \end{pmatrix} \tag{8.21}$$

The first equation then yields

$$x_{n+1} = x_n + \frac{h}{2}\left[3f(t_n, x_n) - f(t_{n-1}, x_{n-1})\right] \tag{8.22}$$

This is the **second-order Adams-Bashforth** method, known as "AB2". The weights again sum to 1 and in the $h \to 0$ limit, the two function evaluations are at the same point so the method is consistent. The weights of $3/2$ and $-1/2$ should be contrasted with the leapfrog 2 and -1; these are somewhat reduced in magnitude and indeed its stability is better. It has the same amplification factor as for the second-order Taylor method, i.e. $g = 1 + ah + a^2h^2/2$. This method can be written as

$$f_a = f(t_n, x_n), \tag{8.23a}$$

$$f_b = f(t_{n-1}, x_{n-1}), \tag{8.23b}$$

$$x_{n+1} = x_n + \frac{h}{2}\,(3f_a - f_b) \tag{8.23c}$$

Higher-order multi-step using f

The same method can be enlarged to include more $f(t, x)$ terms in the same way as we extended the leapfrog method. The **fourth-order Adams-Bashforth** method can be calculated by inverting a 4×4 matrix and yields

$$\boxed{x_{n+1} = x_n + \frac{h}{24}\left[55f(t_n, x_n) - 59f(t_{n-1}, x_{n-1}) + 37f(t_{n-2}, x_{n-2}) - 9f(t_{n-3}, x_{n-3})\right]} \tag{8.24}$$

Note the matrix inversions only need to be done once to calculate the weights, not for each step. They are tabulated in textbooks. This method can be written as

$$f_a = f(t_n, x_n), \tag{8.25a}$$
$$f_b = f(t_{n-1}, x_{n-1}), \tag{8.25b}$$
$$f_c = f(t_{n-2}, x_{n-2}), \tag{8.25c}$$
$$f_d = f(t_{n-3}, x_{n-3}), \tag{8.25d}$$
$$x_{n+1} = x_n + \frac{h}{24}\left(55 f_a - 59 f_b + 37 f_c - 9 f_d\right) \tag{8.25e}$$

Starting off

A significant drawback of multi-step methods is that the previous values are needed; the number of points required depends on the order of the method. Hence, we cannot apply a multi-step method immediately from t_0 as we don't have any previous values. 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 x_1 from the initial condition x_0. Then there are enough points to continue with the leapfrog scheme:

$$\begin{aligned} x_1 &= x_0 + f_0\, h, \\ x_2 &= x_0 + 2 f_1\, h, \\ x_3 &= x_1 + 2 f_2\, h\,, \\ &\cdots \\ x_{n+1} &= x_{n-1} + 2 f_n\, h\,. \end{aligned} \tag{8.26}$$

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 x_1 for starting leapfrog.)

It can be the case that a few low accuracy steps at the beginning dominate the global error of a solution. This is effectively equivalent to an initial condition error. Hence, it is important to be careful with the accuracy of the starting steps. It is better to use a single step method with at least the same order accuracy even if less efficient; the extra computational time should be small if it is only for 3 or 4 steps.

8.4 Implicit methods

Implicit methods for solving an ODE require evaluation of $f(t, x)$ using the unknown value x_{n+1}. The simplest is the **implicit Euler method**:

$$\boxed{x_{n+1} = x_n + h\, f(t_{n+1}, x_{n+1}).} \tag{8.27}$$

This is basically the Euler method but using the derivative at the *end* of the time step rather than at the beginning. Implicit Euler is still a first-order method and so has global accuracy $\mathcal{O}(h)$, just like its explicit cousin.

The problem is how to solve for the unknown x_{n+1}. For a linear system, where $f(t, x) = a(t)x + b(x)$, equation (8.27) is easily solved as

$$x_{n+1} = x_n + h a(t_{n+1}) x_{n+1} + h b(t_{n+1}) \qquad \text{so} \qquad x_{n+1} = \frac{x_n + h b(t_{n+1})}{1 - h a(t_{n+1})} \tag{8.28}$$

For a non-linear function, equation (8.27) is a non-linear algebraic equation in x_{n+1}, that in principle needs to be solved using something like the bisection, Newton-Raphson or secant method, as discussed in Section (3.10). This can be very computationally intensive. The original equation is only accurate to $\mathcal{O}(h)$ so an approximate solution to the step equation to the same order is usually sufficient. This can be done iteratively using two steps. The first step is the explicit Euler to get the first iteration of x_{n+1}

$$x_{n+1}^{(0)} = x_n + h\, f(t_n, x_n) \tag{8.29}$$

and this is then used at t_{n+1} to get a second iteration

$$x_{n+1}^{(1)} = x_n + h\, f(t_{n+1}, x_{n+1}^{(0)}) \tag{8.30}$$

For comparison later, we can write the iterative procedure as

$$\boxed{\begin{aligned} f_a &= f(t_n, x_n), && (8.31a)\\ f_b &= f(t_{n+1}, x_n + h\, f_a), && (8.31b)\\ x_{n+1} &= x_n + h\, f_b && (8.31c) \end{aligned}}$$

which requires one more evaluation than explicit Euler. Note, all the terms on the RH sides of these equations are known and all x values come from t_n.

The pattern of using a explicit method to get an approximate result and an implicit method to then update is widely used for all implicit methods. It is called "predictor-corrector" (PC) as the explicit method **predicts** the value of x_{n+1} and the implicit method **corrects** it to the final value.

Stability of implicit Euler

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 (7.7) for explicit Euler can be applied to implicit Euler, in much the same way. The usual test function yields

$$\frac{\epsilon_{n+1}}{\epsilon_n} \approx (1 - ah)^{-1} \tag{8.32}$$

so that for decaying problems ($a < 0$) we have $g \leq 1$ for ***any*** (positive) h. What happens to x_{n+1} for large h? It simply tends to zero; the same behaviour as the exact solution.

Implicit Euler for multiple variables

Implicit Euler also works for coupled first-order ODEs. For the linear case

$$\begin{aligned} \vec{x}_{n+1} &= \vec{x}_n + h\mathbf{A} \cdot \vec{x}_{n+1} \\ \therefore \qquad (\mathbf{I} - h\mathbf{A}) \cdot \vec{x}_{n+1} &= \vec{x}_n \\ \therefore \qquad \vec{x}_{n+1} &= (\mathbf{I} - h\mathbf{A})^{-1} \cdot \vec{x}_n = \overline{\overline{\mathbf{H}}} \cdot \vec{x}_n \end{aligned} \tag{8.33}$$

where $\mathbf{A}$ is the same matrix operator as for explicit Euler. $\overline{\overline{\mathbf{H}}}$ is the update matrix for implicit Euler and is the inverse of the matrix $(\mathbf{I} - h\mathbf{A})$. Hence implicit Euler works if $(\mathbf{I} - h\mathbf{A})$ is a non-singular matrix so that $\overline{\overline{\mathbf{H}}}$ 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 linearise the functions $\vec{f}$ in each ODE about the known 'point' $(t_n, \vec{x}_n)$. But then the update matrix $\overline{\overline{\mathbf{H}}}$ has to be calculated at each step, since the matrix $\mathbf{A}$ obtained by linearisation depends on $\vec{x}_n$. Thus implicit methods are generally less efficient than explicit methods.

Predictor-corrector methods

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

To get second-order accuracy, we have to include the x''_n term using extra information. We will use both $f(t_n, x_n)$ (as for explicit Euler) *and* $f(t_{n+1}, x_{n+1})$ (as for implicit Euler). Using both gives one extra constraint and allows us to include x''_n. This is therefore a multi-step method. The Taylor expansions at x_n for x_{n+1} and x'_{n+1} to order h^2 are

$$x_{n+1} = x_n + h\,x'_n + \frac{h^2}{2}\,x''_n \qquad \text{and} \qquad x'_{n+1} = x'_n + h\,x''_n \tag{8.34}$$

Multiplying the second by $h/2$ and subtracting from the first eliminates x''_n and results in

$$x_{n+1} - \frac{h}{2}\,x'_{n+1} = x_n + \frac{h}{2}\,x'_n \qquad \text{i.e.} \qquad x_{n+1} - \frac{h}{2}\,f(t_{n+1}, x_{n+1}) = x_n + \frac{h}{2}\,f(t_n, x_n) \tag{8.35}$$

This is the equivalent of equation (8.27) which now has to be solved for x_{n+1}. Note, it can be rearranged to

$$x_{n+1} = x_n + \frac{h}{2}[f(t_{n+1}, x_{n+1}) + f(t_n, x_n)] \tag{8.36}$$

and so is seen to be using an average of the two derivatives at each end of the time step, with equal weights of 1/2. This can be solved iteratively using an explicit method as a predictor in the same way as for implicit Euler. If we again use explicit Euler as the predictor, the procedure becomes

$$\boxed{\begin{aligned} f_a &= f(t_n, x_n), \\ f_b &= f(t_{n+1}, x_n + h\,f_a), \\ x_{n+1} &= x_n + \frac{h}{2}\,(f_a + f_b) \end{aligned}} \tag{8.37a}$$

This iteration method is called the second-order **predictor-corrector method** ("PC2"). We could use other explicit methods such as AB2 as a predictor if Euler does not give an accurate enough starting point.

To compare explicit Euler, implicit iterated Euler and PC2, we could write all three as calculating f_a and f_b as above, with the solution being written as

$$x_{n+1} = x_n + h(w_a f_a + w_b f_b) \tag{8.38}$$

where explicit Euler has weights of $w_a = 1$, $w_b = 0$, implicit iterated Euler has $w_a = 0$, $w_b = 1$ and PC2 has $w_a = 1/2$, $w_b = 1/2$.

The PC2 local step error (due to truncation) is third-order and hence this is a **second-order method**. Using equation (8.37) will give an approximate solution to the ODE since the remainder terms in the Taylor expansion of x_{n+1} and in using the forward difference approximation of f'_n have been discarded.

The gradient calculated with the predicted point will not, in general, be exactly the correct value since it depends on both t and x but it will still be more accurate than the Euler method (see Fig. 8.2). The predictor-corrector method is $\mathcal{O}(h^2)$ accurate but involves two evaluations per step which is less efficient. It is stable for decaying solutions, but just stable for pure oscillating solutions.

Higher-order implicit methods use values calculated in earlier steps. As previously, using the f_{n-1}, etc., values gives better stability than the x_{n-1}, etc., values. These are known as Adams-Moulton methods.

Predictor-corrector for multiple variables

The predictor-corrector method can be applied to a set of N coupled first-order ODEs. However, it is important that the first iteration is found for all the variables to get $\vec{x}^{(0)}_{n+1}$. Only then should $\vec{f}_{n+1} = \vec{f}(t_{n+1}, \vec{x}^{(0)}_{n+1})$ be evaluated for each ODE and the corrected values then found. This is

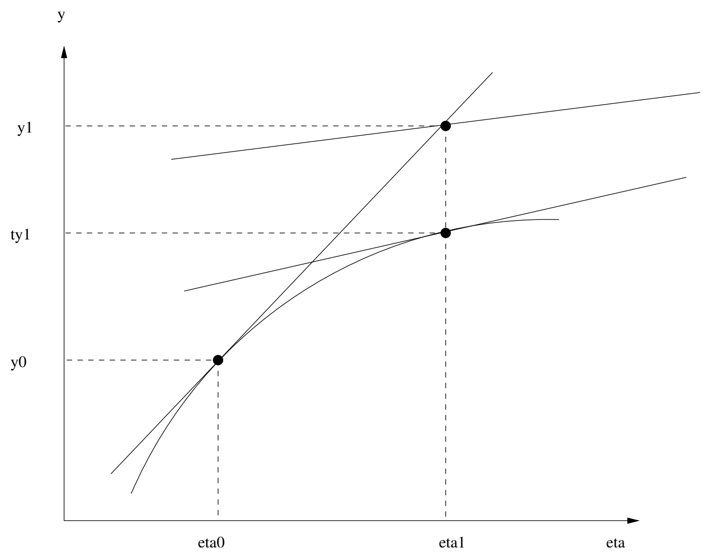

Figure 8.2: (Predictor-corrector) Ideally we would like to use the gradient at x_{n+1} to get the average gradient in the interval. However we do not have the true value x_{n+1} so we use the gradient calculated with the Euler estimate $x_{n+1}^{(0)}$. (Nb: $n = 0$ is used in the plot.)

because in general all of the $\vec{f}_{n+1}$ functions depend on all the $\vec{x}_{n+1}$ values. The procedure is:

$$\boxed{\begin{aligned} \vec{f}_a &= \vec{f}(t_n, \vec{x}_n), && (8.39a)\\ \vec{f}_b &= \vec{f}(t_{n+1}, \vec{x}_n + h\,\vec{f}_a), && (8.39b)\\ \vec{x}_{n+1} &= \vec{x}_n + \frac{h}{2}\,(\vec{f}_a + \vec{f}_b) && (8.39c) \end{aligned}}$$

where all $\vec{f}_a$ must be evaluated before starting on $\vec{f}_b$.

8.5 Runge-Kutta Methods

When considered as a "black box", the Runge-Kutta methods are a family of explicit, single-step methods, i.e. they only require x_n as an input to find x_{n+1}.

To get to higher-order, the extra information needed is obtained by doing function evaluations *within* the overall time step. Hence, internally, the method generalises the idea of using m estimates of the gradient calculated at various points in the interval $t_n \leq t \leq t_{n+1}$ to find a weighted average of gradients.

In general the **RK*m*** 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 method of order m will require m gradient function evaluations. A number of weighting schemes for the averaging of these gradients can be used for each RKm.

Second-order Runge-Kutta

The Taylor series to second-order means the general step equation can be written as

$$x_{n+1} = x_n + h\left(f_n + \frac{h}{2}f'_n\right) \tag{8.40}$$

To the same approximation, the two terms in brackets can be written as

$$f_n + \frac{h}{2}f'_n \approx f(t_n + h/2, x_n + h\,f_n/2) \tag{8.41}$$

Therefore this corresponds to an Euler predictor and a "midstep" corrector

$$f_a = f(t_n, x_n), \tag{8.42a}$$
$$f_b = f(t_n + h/2, x_n + h\, f_a/2), \tag{8.42b}$$
$$x_{n+1} = x_n + h\, f_b \tag{8.42c}$$

However, this could also have been written as

$$f_n + \frac{h}{2} f'_n = \frac{1}{2} f_n + \frac{1}{2}(f_n + h\, f'_n) \approx \frac{1}{2} f_n + \frac{1}{2} f(t_n + h, x_n + h\, f_n) \tag{8.43}$$

which is the original predictor-corrector PC2.

There is a free parameter in these equations. The second-order Runge-Kutta ("RK2") class of methods are predictor-corrector methods, where the explicit Euler method is used as the predictor and the corrector evaluation is made at αh. The general RK2 scheme is;

$$\boxed{\begin{aligned} f_a &= f(t_n, x_n), && (8.44a)\\ f_b &= f(t_n + \alpha\, h, x_n + \alpha\, h\, f_a), && (8.44b)\\ x_{n+1} &= x_n + \frac{h}{2\alpha}\left[(2\alpha - 1)\, f_a + f_b\right], && (8.44c)\end{aligned}}$$

Here α is a coefficient which differs for the different RK2 methods. The factors depending on α multiplying the functions are the weights for averaging the gradients and these depend on the chosen value of α. Different choices for the coefficients give different methods; some common ones are

$$\begin{array}{llll} w_a = 0\,, & w_b = 1\,, & \alpha = \frac{1}{2}\,, & \text{Single mid-point} \\ w_a = \frac{1}{2}\,, & w_b = \frac{1}{2}\,, & \alpha = 1\,, & \text{Second-order predictor-corrector} \\ w_a = \frac{1}{3}\,, & w_b = \frac{2}{3}\,, & \alpha = \frac{3}{4}\,, & \text{Smallest error RK2} \end{array} \tag{8.45}$$

The RK2 method is illustrated in Fig. 8.3. It uses two stages of gradient estimation. At each stage, the gradient is used to estimate the change in x going from t_n to t_{n+1}. The first estimate is an Euler step yielding an estimated shift in x of $h\, f_a = x^{(0)}_{n+1} - x_n$. In the second stage, a more accurate value of the gradient is estimated using the function f at a point shifted by an amount $\alpha\, h$ in t and by an amount $\alpha\, h f_a$ in x. This yields a better estimate in the shift in x, i.e., $f_b h = x^{(1)}_{n+1} - x_n$. The two shifts are then averaged with weights w_a and w_b.

Coupled ODEs

For coupled ODEs, the first (predictor) stage must be carried out for all the dependent variables $\vec{x}$ before moving onto the second (corrector) stage. The RK2 scheme is then;

$$\vec{f}_a = \vec{f}(t_n, \vec{y}_n), \tag{8.46a}$$
$$\vec{f}_b = \vec{f}(t_n + \alpha\, h, \vec{x}_n + \alpha\, \vec{f}_a h) \tag{8.46b}$$
$$\vec{x}_{n+1} = \vec{x}_n + h\, (w_a\, \vec{f}_a + w_b\, \vec{f}_b), \tag{8.46c}$$

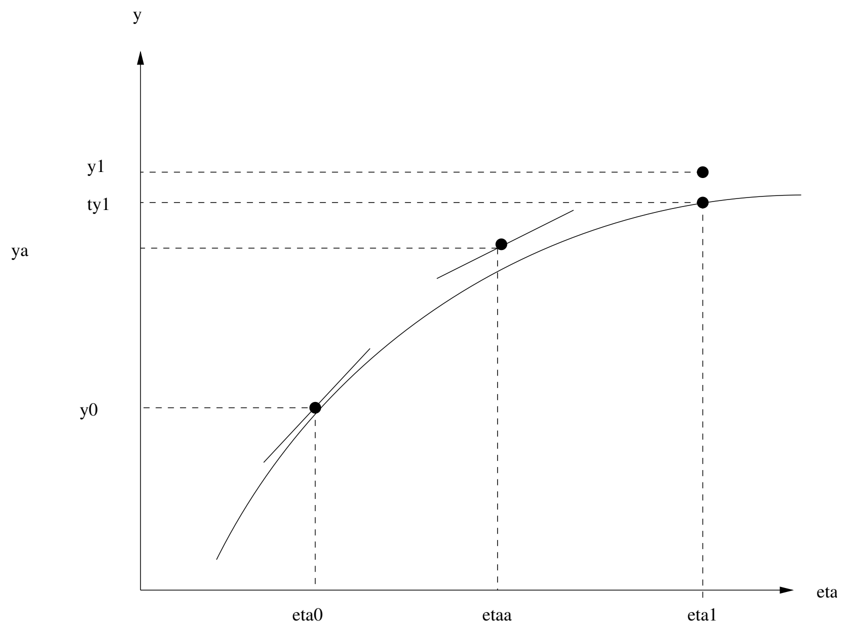

Figure 8.3: The RK2 method. A second gradient is calculated at a shifted position and averaged with the gradient at the starting position to obtain $\tilde{y}_1$, the next value of y. Comparing to fig. 8.1 you can see that $\tilde{y}_1$ is more accurate for RK2 compared to Euler.

Fourth-order Runge-Kutta

The "RK4" methods uses four stages of gradient estimation to calculate an average and hence is fourth-order accurate.

$$\boxed{\begin{aligned} f_a &= f(t_n, x_n), && \text{(8.47a)}\\ f_b &= f(t_n + h/2, x_n + h\, f_a/2), && \text{(8.47b)}\\ f_c &= f(t_n + h/2, x_n + h\, f_b/2), && \text{(8.47c)}\\ f_d &= f(t_n + h, x_n + h\, f_c) && \text{(8.47d)}\\ x_{n+1} &= x_n + \frac{h}{6}\,(f_a + 2\, f_b + 2\, f_c + f_d) && \text{(8.47e)} \end{aligned}}$$

Again the first stage is an Euler estimate, as in RK2. The second and third stages estimate the gradient using a shift of half of the preceding shifts, $f_a h$ and $h\, f_b$, respectively. The fourth stage estimates the gradient using the gradient function f evaluated at the best estimate yet of the true value of x_{n+1}, i.e., $x^{(3)}_{n+1} = y_n + h\, f_c$ and x_{n+1}. The values of the weights and shift coefficients are careful chosen to minimise the error.

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 overall than the Euler and implicit methods. It therefore is a standard 'workhorse' for real numerical ODE solving. Even 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 do not need to store any previous steps. This makes coding it a little easier.

8.6 Variable step size *h*

For real applications, varying the step size is a crucial tool to controlling errors when the function is changing rapidly. This is a complex area and a detailed discussion is beyond this course. With single-step methods, it is pretty easy to vary h during the calculation to, e.g., keep the error within a specified bound. Classic examples are the so-called 'RK45' methods, which produce a fourth-order global error estimate and a fifth-order function estimate. The adaptive stepping in this case is done either via step doubling, or by embedding lower-order Runge-Kutta formulae in higher-order ones. See Numerical Recipes for a detailed discussion.

Note that whilst implementing a dynamic step size is pretty easy in single-step algorithms, it is more difficult in multi-step methods, as this requires back-calculating missing intermediate values using a good single-step method.

8.7 Relationship to interpolation

For finite difference methods, we only have the solution at particular times. A smooth function will require interpolation. Interpolation between the discrete values was covered in Section (4). These techniques can be applied to the initial value solution to give a function with the required continuity properties.

However, these interpolation methods are based on only knowing the function values x_n at the discrete times. However, for the solution of an ODE, we (at least in principle) know the derivatives as well. This is either because they are stored as part of the solution or because we can evaluate $f(t_n, x_n)$ for each point specifically for the interpolation. In general the standard interpolation techniques will give a smooth function for which the derivatives at each point are not the same as $f(t_n, x_n)$ and hence are inconsistent with the ODE to some level. We can include the extra derivative information in the interpolation to avoid this issue.

Interpolation within a step

If we just look within a single step, we have one point at either end. For standard interpolation the only possibility is linear interpolation, which joins the two points by a straight line. If used over more than one step, this gives a continuous function but a discontinuous first derivative at the step times (and so all higher derivatives are not defined).

Including the two derivative values at either end, we have four pieces of information which allows us to solve for a cubic interpolation function

$$x(t) = x_n + (t - t_n)\, f_n + (t - t_n)^2\, \frac{x''_n}{2!} + (t - t_n)^3\, \frac{x'''_n}{6!} \tag{8.48}$$

which means

$$x'(t) = f_n + (t - t_n)\, x''_n + (t - t_n)^2\, \frac{x'''_n}{2!} \tag{8.49}$$

These have to match to x_{n+1} and f_{n+1} for $t = t_n + h$ so we have

$$x_{n+1} = x_n + h\, f_n + h^2\, \frac{x''_n}{2!} + h^3\, \frac{x'''_n}{6!} \tag{8.50}$$

$$f_{n+1} = f_n + h\, x''_n + h^2\, \frac{x'''_n}{2!} \tag{8.51}$$

The solution is

$$h^2\, \frac{x''_n}{2!} = 3(x_{n+1} - x_n) - h\,(f_{n+1} + 2f_n) \tag{8.52}$$

$$h^3\, \frac{x'''_n}{6!} = -2(x_{n+1} - x_n) + h\,(f_{n+1} + f_n) \tag{8.53}$$

This cubic polynomial will now be consistent with the ODE. If used over more than one step, this gives a continuous function and first derivative at the step times but a discontinuous second derivative.

Overall interpolation

A good technique for standard interpolation over all the points is to use cubic splines which give a smooth curve. At the step edges, it in continuous in the function as well as the first and second derivatives, and only discontinuous in the third derivative. However, as stated above, the first derivative will not match the value from the ODE.

We already have the internal step interpolation above, which gives a first derivative which is continuous *and* matched to the ODE; this is often sufficient. However, if second derivative matching is required, then the same idea as the cubic splines can be applied. Matching the second derivatives at every point will require another coefficient in each cell, so this results in a quartic spline. This is a straightforward generalisation of the standard technique.

8.8 Relationship to integration

Section (5.2) described various methods for numerical integration of varying accuracy and complexity. These were of the form

$$I = \int_a^b f(x)\,dx = \sum_n \int_{x_n}^{x_n+h} f(x)\,dx \tag{8.54}$$

where the integral was split into multiple small steps, each of width h. We can change the variable names to be consistent with our use in the ODE section, and write one term of the sum as

$$\int_{t_n}^{t_{n+1}} f(t)\,dt \tag{8.55}$$

If we now define a variable $x(t)$ such that $x' = f(t)$, we have

$$\int_{t_n}^{t_{n+1}} f(t)\,dt = \int_{t_n}^{t_{n+1}} \frac{dx}{dt}\,dt = \int_{x(t_n)}^{x(t_{n+1})} dx = x_{n+1} - x_n \tag{8.56}$$

which also means

$$x_{n+1} = x_n + \int_{t_n}^{t_{n+1}} f(t)\,dt \tag{8.57}$$

Hence, this is seen to be equivalent to solving the ODE $x' = f(t)$. For a given $f(t)$, the ODE solutions and integrals should be the consistent, within the limitations of the numerical methods used.

Importantly, the integral methods are a special case for the ODEs, namely that the function $f(t)$ does not depend on x. For this special case, many of the ODE methods simplify and in fact several nominally different methods become identical.

Consider the explicit Euler method. This approximates the average gradient by the initial value, i.e. $\overline{x}'_n = f_n$. This means

$$\int_{t_n}^{t_{n+1}} f(t)\,dt \approx h\,f_n \tag{8.58}$$

which is equivalent to putting $f(t) = f_n$, i.e. the gradient is constant across the step. This is the basic integral technique called a left Riemann sum. Similarly, implicit Euler is equivalent to setting $f(t) = f_{n+1}$ which is a right Riemann sum.

Moving to second-order methods, since $f(t)$ only, the second-order predictor-corrector method simplifies to

$$x_{n+1} = x_n + \frac{h}{2}\,(f_n + f_{n+1}) \tag{8.59}$$

This is seen to correspond to the trapezoidal integration method. In addition, applying the leapfrog with half the step size gives

$$x_{n+1} = x_n + h\,f(t_n + h/2) = x_n + h\,f\left(\frac{t_n + t_{n+1}}{2}\right) \tag{8.60}$$

which is the midpoint Riemann (open rule) sum method.

Finally, considering fourth-order ODE methods, the RK4 simplifies to

$$f_a = f(t_n), \tag{8.61}$$
$$f_{bc} = f(t_n + h/2), \tag{8.62}$$
$$f_d = f(t_n + h), \tag{8.63}$$
$$x_{n+1} = x_n + \frac{h}{2}\left(\frac{1}{3}f_a + \frac{4}{3}f_{bc} + \frac{1}{3}f_d\right) \tag{8.64}$$

Note, this is another 25% more efficient than the regular RK4 step when solving definite integrals due to the middle two function evaluations being the same.

In this form, RK4 is identical to Simpson's rule as discussed in Section (5.2.1) but with $h \to h/2$:

$$\int_{x_n}^{x_{n+1}} f(x)\,\mathrm{d}x = \frac{h}{2}\left[\frac{1}{3}f(x_n) + \frac{4}{3}f(x_{n+\frac{1}{2}}) + \frac{1}{3}f(x_{n+1})\right] + O(h^5) \tag{8.65}$$

In the end this is not so surprising when we think about the fact that RK4 and Simpson's rule are both designed from the start to cancel local errors of order $\mathcal{O}(h^4)$.

Chapter 9

ODEs: Boundary Value Methods

Outline of Section

- Boundary value methods
- Shooting methods
- Matrix methods
- Second-order matrix method
- Eigensystems

9.1 Introduction

We have so far only looked at initial value problems, meaning that the constants of integration define values for the variables all at the same time t_0. However, for more than one variable, the variables can be defined at different times, which means we do not have any initial time where all variables are known which we can work from. We have to solve the region between the first and last specified variable times in a coherent way; these are called **boundary value problems**. There are in fact several ways of specifying the constants of integration. The above assumes there is a known value of each variable at some time (which can be different for each variable). However, in some cases, we can instead define two known values for one variable and none for one of the others. For multiple variables there are many combinations in general.

In particular, we have seen that a second-order equation can be represented by two first-order equations for x and the velocity v. Consider the general, linear, second-order equation

$$\alpha\frac{\mathrm{d}^2x}{\mathrm{d}t^2}+\beta\frac{\mathrm{d}x}{\mathrm{d}t}+\gamma\,x=k\,. \tag{9.1}$$

For an initial value problem, we would require initial conditions specifying the value of the solution at some initial point $t=t_0$, i.e. $x(t_0)$ and its derivative $x'(t_0)$, to integrate the system to a final point. The second value clearly is equivalent to setting $v(t_0)$ when working with two first-order equations. However, you will know that we can instead specify the start and end position, i.e. $x(t_0)$ and $x(t_N)$, where t_N is the end of the boundary time range. This means when we break the equation into two first-order equations, we have specified x twice and not specified v at all.

We should be able to find a solution whenever there are enough independent boundary conditions specified. The ways we do this are called **boundary value methods**. We will discuss two general way to do this; shooting methods and matrix methods.

9.2 Shooting methods

The first method we will look is the **shooting method**. This reuses a lot of the machinery we developed for initial value methods. In fact, the label 'shooting method' is an overall term for the general concept, while internally it uses whichever of the initial value methods we want.

To keep things simple, let's work with two variables $x(t)$ and $y(t)$ and assume we know x at the start time $x_0=x(t_0)$ and y at the end time $y_N=y(t_N)$. Compared with initial value problems, the issue is that we don't know $y_0 = y(t_0)$ and so cannot in general calculate $f_x(t_0, x_0, y_0)$ or $f_y(t_0, x_0, y_0)$ to take a first step. We therefore **guess** a value for y_0 which allows us to solve the problem as an initial value problem. We can use any of the initial value methods we like.

Of course, unless we got very lucky, we won't have guessed the correct value for y_0. Hence, when we get to the end of the time period, the $y(t_N)$ value we find will not be equal to the required value y_N. We could put in a lot of y_0 guesses and try to map out the dependence of the resulting $y(t_N)$ values to see which is close to y_N. This is why it is called a 'shooting' method; we keep shooting forward with an initial guess and then adjusting our aim through y_0 to get closer to the target.

However, rerunning the initial value method many times would usually be slow. We can be more systematic; we can consider the whole initial value method conceptually as a function which takes a value of y_0 and returns a value of $y(t_N)$, i.e. $y(t_N) = r(y_0)$. What we want is for this to be equal to the known value y_N, which we can write as

$$r(y_0) - y_N = 0 \tag{9.2}$$

We now see this is an algebraic root-finding problem; these are covered in section (3.10). We would not normally be able to calculate the derivatives of $r(y_0)$ analytically so we would use the bisection or secant method. The bisection method requires two values of $y(t_N)$ on either side of y_N which are not always easy to find, so the secant method is normally used. As always with root-finding, the closer the initial guess is to the true value, the quicker and more reliable the convergence. Hence some physics intuition is useful to pick an initial y_0.

There is one issue with the shooting method which is that it treats the boundary region asymmetrically, i.e. we start at t_0 and perform an initial value method which matches to y_N. We could have started at t_N taking a guess for x_N and doing our time step by $-h$ each time; this is called a "final value method" but these are essentially identical to the initial value methods. We will find that the two solutions will not be identical in general, although this is not normally a problem in practical use.

Linear systems

Applying the shooting method to a linear system such as the one in equation (9.1) is easier than the general case. Any two guesses for y_0 will allow us to find the solution without any iteration. In fact, the solution is just what would be found from setting up the general secant method and taking one iteration; it would converge immediately.

We take two guesses y_{0a} and y_{0b} and solve for each using our favourite initial value method to get two end values y_{Na} and y_{Nb}. Since it is linear, y_{Na} will depend linearly on y_{0a} and the same holds for the other solution. Also, because it is linear, a sum of two solutions is also a solution. We can sum the two solutions using a parameter c so the final values give the known value y_N by doing

$$c y_{Na} + (1-c) y_{Nb} = y_N \tag{9.3}$$

so if the first guess was correct, we would have $c = 1$ and if the second guess was correct, we would have $c = 0$. The general case solution for c of the above equation is easily found to be

$$c = \frac{y_N - y_{Nb}}{y_{Na} - y_{Nb}} \tag{9.4}$$

We can now sum each step value in the two initial value solutions for y_{0a} and y_{0b} using c in the same way as above to find the boundary value solution. Alternatively, given we know c, we can find the correct y_0 using

$$y_0 = c y_{0a} + (1-c) y_{0b} \tag{9.5}$$

and rerun the initial value method; we should now get $y(t_N) = y_N$ within computational errors.

Other boundary conditions

The case where x is known at x_0 and x_N, but y is not specified at all, can be handled in an effectively identical way. We again guess values of y_0 and the only difference is that we check on the value of x_N at the end of the initial value method, rather than y_N. Hence our algebraic root-finding equation is

$$s(y_0) - x_N = 0 \tag{9.6}$$

where $s(y_0)$ is now (at least conceptually) a function that returns $x(t_N)$ from the initial value method.

> The principle of the shooting method can be adapted to be used to solve more complicated problems; these can be boundary conditions that are defined at more than two different locations (which requires at least three variables) or where instabilities in the problem mean that you cannot shoot *into* a boundary, only *out* from it. For example, the three variable problem (e.g. for x, y and z) requires two guesses at the initial boundary (e.g. for y_0 and z_0) and we will want to match the y and z values where they are known (e.g. to $y(t_N)$ and $z(t_z)$ for an intermediate value of time t_z within the boundary time range). This is then a 2D root-finding problem but we will also need to interpolate in z near t_z as this will not in general sit exactly on a step edge.

9.3 Matrix methods

The shooting method, adapted to the specific problem through the use of insights on its nature, is usually the first choice for solving boundary value problems. Another method is to consider all the points of the solution at the same time, by converting the differential equation into a **finite difference equation** which involves neighbouring points, and solve the resulting matrix equation. These are also called relaxation methods.

The concept of solving ODEs using matrices is applicable to both initial value and boundary value problems. As for the shooting method, the use of matrices is a general system and internally we can use any method per step that we like. For initial value problems, the result will be identical to that obtained using the same method in the previous way; it is just the same equations written in a different form. However, for boundary value problems, matrix methods can solve the whole boundary region in a coherent way. We will show how the matrix methods correspond to the initial value methods and then look at boundary value problems.

Initial value matrix methods

Consider the one variable case. We have seen that all initial value methods come down to approximating the average gradient $\overline{x}'$ in the relation

$$x_{n+1} = x_n + h\,\overline{x}'_n \tag{9.7}$$

The first few steps therefore can be written as

$$x_1 = x_0 + h\,\overline{x}'_0, \qquad x_2 - x_1 = h\,\overline{x}'_1, \qquad x_3 - x_2 = h\,\overline{x}'_2, \qquad \ldots \tag{9.8}$$

where the LH sides have the values we want to solve for. Similarly the last few steps are

$$\ldots \qquad x_{N-2} - x_{N-3} = h\,\overline{x}'_{N-3}, \qquad x_{N-1} - x_{N-2} = h\,\overline{x}'_{N-2}, \qquad x_N - x_{N-1} = h\,\overline{x}'_{N-1} \tag{9.9}$$

These equations can be written in an $N \times N$ matrix form as

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-2} \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} x_0 + h\,\overline{x}'_0 \\ h\,\overline{x}'_1 \\ h\,\overline{x}'_2 \\ \vdots \\ h\,\overline{x}'_{N-3} \\ h\,\overline{x}'_{N-2} \\ h\,\overline{x}'_{N-1} \end{pmatrix}, \tag{9.10}$$

The matrix is seen to be a lower diagonal matrix $\mathbf{L}$ so these equations are of the type $\mathbf{L} \cdot \vec{x} = \vec{b}$. Section (3.4) shows that such matrix equations can be solved by forward substitution. However, there is an issue in this case; the RH vector $\vec{b}$ depends on the values of x_i, as these are needed to calculate the $\overline{x}'_i$. This seems to make the set of equations much more difficult to solve.

A closer look at the RH vector shows that each element is the average of the gradient at an **earlier step** than the x_i values which are being solved for in the corresponding row equation. This means whatever method we use, we will have solved for all the x_i needed to calculate the next $\overline{x}'_i$ before it is used. Therefore, the above matrix equation can be solved by forward substitution, if we include updating the RH vector $\vec{b}$ by substituting the x_i values into each element in turn. If you look at the literal equations for forward substitution, you would find this exactly reproduces the calculations of the initial value method.

This can be done for two or more variables; for the two variable case we have two matrix equations $\mathbf{L} \cdot \vec{x} = \vec{b}$ and $\mathbf{L} \cdot \vec{y} = \vec{c}$, where the lower diagonal matrix is identical in both cases. The RH vectors $\vec{b}$ and $\vec{c}$ contain the average gradients $\overline{x}'_i$ and $\overline{y}'_i$ respectively but in general these will both depend on x_i and y_i for coupled systems. Hence, we cannot solve for $\vec{x}$ first and $\vec{y}$ afterwards. A solution is possible but we must solve the first equation of both matrices first, use the resulting x_1 and y_1 values to calculate $\overline{x}'_1$ and $\overline{y}'_1$ so we can then solve the next row, and so on. Hence, the two forward substitution solutions must be calculated in parallel and again the resulting equations are identical to the equivalent initial value method.

Boundary value matrix methods

It is usually a lot more convenient to apply a standard initial value method to solve an initial value problem so the matrix method is not used in practice for these problems.

However, they can be very useful for boundary value problems. Assume again that we know $x(t_0) = x_0$ and $y(t_N) = y_N$. The matrix form for the $\vec{x}$ equations will be identical to above; we want to solve for x_1 to x_N. However the ones for $\vec{y}$ will now be different; in this case we want to solve for the unknowns y_0 to y_{N-1}. The first few steps will be

$$y_1 - y_0 = h\,\overline{y}'_0, \qquad y_2 - y_1 = h\,\overline{y}'_1, \qquad y_3 - y_2 = h\,\overline{y}'_2, \qquad \dots \tag{9.11}$$

and the last few steps are

$$\dots \qquad y_{N-2} - y_{N-3} = h\,\overline{y}'_{N-3}, \qquad y_{N-1} - y_{N-2} = h\,\overline{y}'_{N-2}, \qquad -y_{N-1} = -y_N + h\,\overline{y}'_{N-1} \tag{9.12}$$

This corresponds to a matrix equation of the form

$$\begin{pmatrix} -1 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{N-3} \\ y_{N-2} \\ y_{N-1} \end{pmatrix} = \begin{pmatrix} h\,\overline{y}'_0 \\ h\,\overline{y}'_1 \\ h\,\overline{y}'_2 \\ \vdots \\ h\,\overline{y}'_{N-3} \\ h\,\overline{y}'_{N-2} \\ -y_N + h\,\overline{y}'_{N-1} \end{pmatrix}, \tag{9.13}$$

This is seen to be an upper diagonal matrix $\mathbf{U}$ so these equations are now of the type $\mathbf{U}\cdot\vec{y}=\vec{c}$. In principle we would like to solve this using backward substitution but we have hit the same problem as before; we do not know the average gradient values. In this case, we cannot calculate them as we go, as we need to solve $\vec{x}$ from the top but $\vec{y}$ from the bottom.

We therefore have to guess a solution and iterate. We need to guess values for the whole vector $\vec{x}^{(0)}$ and $\vec{y}^{(0)}$. This could be a rough solution from a shooting method, for example. We then pre-calculate all the average gradients $\vec{x}_i'^{(0)}$ and $\vec{y}_i'^{(0)}$ using these values. The matrix equations are then solved for $\vec{x}^{(1)}$ and $\vec{y}^{(1)}$ and the whole process iterated until it converges. (This can formally be considered a Newton-Raphson iteration with the approximation of dropping terms of $\mathcal{O}(h)$ in the derivative.) The solution assumes the RH vectors are constant when finding the next iteration of $\vec{x}$ and $\vec{y}$ so we can solve using standard forward and backward substitution. Alternatively, the solution is $\vec{x}^{(i+1)}=\mathbf{L}^{-1}\cdot\vec{b}^{(i)}$ and $\vec{y}^{(i+1)}=\mathbf{U}^{-1}\cdot\vec{c}^{(i)}$. One trick is that these matrices have simple and known inverses; $\mathbf{L}^{-1}$ has all elements on and below the diagonal $=1$ and all above the diagonal $=0$, while $\mathbf{U}^{-1}$ has all elements on and above the diagonal $=-1$ and all below the diagonal $=0$.

There are obviously special cases which greatly simplify the above. For example, if the average gradients only depend on time and not x or y then the equations can be solved in one go. If the equations are linear, the parts proportional to x and y in the average gradients can be moved to the LH side and incorporated into the matrix. This leaves only constant (or possibly time-dependent) parts in the RH side vector, so again the system can be solved in one step.

There is also one special case which is very important for physics applications, which we will look at in the next section.

9.4 Second-order matrix method

Consider the case of a classical particle moving in a simple potential, meaning one which does not depend on the particle velocity so $V(t,x)$. The particle acceleration is given by

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2}=-\frac{1}{m}\frac{\partial V}{\partial x}=f(t,x) \tag{9.14}$$

We want to solve this second-order ODE as a boundary value problem where x is known at x_0 and x_N. We know we can separate this into two equations by using the velocity v

$$\frac{\mathrm{d}x}{\mathrm{d}t}=v \qquad \text{and} \qquad \frac{\mathrm{d}v}{\mathrm{d}t}=f(t,x) \tag{9.15}$$

We can then construct matrix equations for $\vec{x}$ and $\vec{v}$ as above. There is a fiddly difference in this case; we know x_0 and x_N so there are only $N-1$ unknowns for x_i, which are x_1 to x_{N-1}. In contrast, we have no information on y so all of y_0 to y_N are unknown, which is $N+1$ unknowns. However, it turns out the value of y_N is never used in the equations (as it would be used to calculate x_N) so we can just consider y_0 to y_{N-1}; this is N unknowns.

The equations can be reduced to a very convenient form if we use the Euler method for (at least) the $x'=v$ equation. This means we use $x_{n+1}=x_n+h\,v_n$ and the corresponding matrix equation becomes $\mathbf{X}\cdot\vec{x}=h\vec{v}+\vec{x}_{0N}$, where the vector $\vec{x}_{0N}$ contains the boundary condition values. (As there are different numbers of $\vec{x}$ and $\vec{v}$ unknowns, the matrix $\mathbf{X}$ is not square, but looks very similar to the previous $\mathbf{L}$). If we write the $\vec{v}$ equation as $\mathbf{V}\cdot\vec{v}=\vec{c}$ (where $\mathbf{V}$ is also not square) then we can do

$$\mathbf{V}\cdot\mathbf{X}\cdot\vec{x}=h\mathbf{V}\vec{v}+\mathbf{V}\vec{x}_{0N}=h\vec{c}+\mathbf{V}\vec{x}_{0N} \tag{9.16}$$

The product $\mathbf{A}=\mathbf{V}\cdot\mathbf{X}$ is a square matrix and has the form

$$\mathbf{A}=\begin{pmatrix} -2 & 1 & 0 & \dots & 0 & 0 & 0 \\ 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ 0 & 1 & -2 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -2 & 1 & 0 \\ 0 & 0 & 0 & \dots & 1 & -2 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 & -2 \end{pmatrix} \tag{9.17}$$

For simplicity, if we use the Euler method for the v' equation also, then the average gradient $\overline{v}'_n = h\, f(t_n, x_n) = h\, f_n$. This means the combination $\vec{b} = h\vec{c} + \mathbf{V}\vec{x}_{0N}$ reduces to the simple form

$$\vec{b} = \begin{pmatrix} h^2\, f_0 - x_0 \\ h^2\, f_1 \\ h^2\, f_2 \\ \vdots \\ h^2\, f_{N-3} \\ h^2\, f_{N-2} - x_N \end{pmatrix}, \tag{9.18}$$

We now have the matrix equations in the form $\mathbf{A} \cdot \vec{x} = \vec{b}$. Again, $\vec{b}$ depends on the values of x so we have to iterate. However note that as the potential does not depend on v, neither does $\vec{b}$ so we do not have to also solve for $\vec{v}$. Importantly, the inverted matrix $\mathbf{A}^{-1}$ is analytically known; $\det(\mathbf{A})\mathbf{A}$ can be written in terms of integers and the determinant is also an integer. Given that we may want to solve for thousands of steps, this is avoids having to do the inversion numerically and so introduce computation error.

As before, there are special cases which mean the equations can be solved immediately. The linear case again allows the matrix to be modified by taking the linear terms from the RH to the LH side of the equation.

Derivative Boundary Conditions

On occasion, we know the derivative at one of the boundaries but not the initial condition for the variable. For example, considering the same system as in (9.14) we could have boundary conditions given by x'_0 and x_N. We can adapt the finite difference method by taking a central difference for the derivative at the boundary

$$x'_0 \approx \frac{x_1 - x_{-1}}{2h}, \tag{9.19}$$

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

$$x_{-1} - 2\, x_0 + x_1 = h^2\, f_0, \tag{9.20}$$

which, given (9.19), is

$$-2\, x_0 + 2x_1 = h^2 + 2\, h\, x'_0\,. \tag{9.21}$$

The system can then be written as N linear equations

$$\begin{pmatrix} -2 & 2 & 0 & 0 & .... & 0 & 0 \\ 1 & -2 & 1 & 0 & .... & 0 & 0 \\ & & & . & & & \\ & & & . & & & \\ & & & . & & & \\ 0 & 0 & 0 & 0 & .... & 1 & -2 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ . \\ . \\ . \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} h^2\, f_0 + 2\, h\, x'_0 \\ h^2\, f_1 \\ . \\ . \\ . \\ h^2\, f_{N-1} - x_N \end{pmatrix}. \tag{9.22}$$

Note, specifying the boundary conditions using just the derivatives at both boundaries, and neither function value, will not generally work as the overall integration constant may not be determined and so means the absolute value of x can be undefined.

9.5 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

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + k^2 x = 0, \tag{9.23}$$

with boundary conditions $x(0) = 0$ and $x(1) = 0$. This describes the vibrations on a string of length 1 which is fixed at the endpoints. The analytic general solution to this system is easily found to be

$$x(t) = A\,\sin(k\,t) + B\,\cos(k\,t)\,. \tag{9.24}$$

The boundary conditions imply that $B = 0$ and that $k = \pm n\,\pi$. The solution describes fundamental modes of vibrations on the string where $n = 1, 2, 3, \ldots$, etc.

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

$$\frac{x_{i-1} - 2\,x_i + x_{i+1}}{h^2} + k^2\,x_i = 0, \tag{9.25}$$

which gives the eigensystem

$$\begin{pmatrix} +2 & -1 & 0 & 0 & .... & 0 & 0 \\ -1 & +2 & -1 & 0 & .... & 0 & 0 \\ & & & . & & & \\ & & & . & & & \\ & & & . & & & \\ 0 & 0 & 0 & 0 & .... & -1 & +2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_{N-1} \end{pmatrix} = h^2\,k^2 \begin{pmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_{N-1} \end{pmatrix}, \tag{9.26}$$

i.e.

$$\mathbf{A} \cdot \tilde{\vec{x}} = \lambda \tilde{\vec{x}} \tag{9.27}$$

where $\lambda = h^2\,k^2$ are the eigenvalues of the system. One then finds $\mathbf{A}$'s eigenvalues λ_i and eigenvectors $\vec{e}_i$ using a suitable numerical eigen-solver. These should be close numerical approximations to the eigenvalues and eigenfunctions of the original ODE eigen-problem. It is often useful to find just the eigenvector corresponding to the largest (or smallest) eigenvalues, as these define the fundamental modes of the system. The power method (section 3.9) can be used to readily find these.

Chapter 10

Random Numbers and Monte Carlo Methods

Outline of Section

- Pseudo-Random Numbers
- Transformation Method
- Rejection Method
- Monte Carlo Minimisation
- Monte Carlo Integration

The generation of (pseudo) random numbers according to pre-defined distributions is a tool that is used in many places in Computational Physics. Their use can, perhaps paradoxically given their "unpredictable" nature, add stability to otherwise deterministic but unstable algorithms, allow for calculations such as integrations to be made in high-dimensions that would not be feasible using more direct methods, and also mimic physical process that are by their nature fundamentally random—such as radioactive decay and other quantum mechanical processes where it is the probability distribution of outcomes that is known.

We first look at how to generate a uniform distribution of pseudo-random numbers in the range 0 to 1. We then see how to use this to generate arbitrary non-uniform distributions (e.g. triangular, Gaussian, etc.). Such distributions can directly be used in physical systems where randomness occurs, e.g., to determine the random direction of a particle emitted by a decay process. Another example is a ***stochastic force*** $\vec{F}_{\text{coll}}$ such as that producing Brownian motion of a small grain in a fluid, due to collisions with the fluid molecules.

Monte Carlo Methods are a broad class of methods where random numbers are used to statistically solve a problem. The 'problem' might be:

(a) A deterministic system where an enormous number of things are coupled together, and exact solution is unfeasible. Classical Brownian motion system is an example. Here it is impractical to actually follow the classical motion of the all the individual molecules. However, the statistics of all the collisions in a small time yield a distribution of net force on the grain. This can be randomly sampled.

(b) A statistical physics problem, such as calculating macroscopic thermodynamic quantities given the energy levels of the system.

(c) Minimising a function of many variables.

(d) Performing an integral over many variables/dimensions.

Generally, Monte Carlo Methods are useful for systems of many variables or dimensions.

10.1 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** (or variate) is a random number lying within a range where any number has the same probability as any other. The range is usually $0 \leq x < 1$. Here x is a *random variable* and the deviates are the values that x can randomly take.

Most high-level languages 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(s) to initiate the sequence. The seed is the number (or set of numbers) that the algorithm needs to start the sequence—once it has started, it will carry on indefinitely. This ability to regenerate the same sequence is an essential feature for debugging codes and reproducing results—compared to only being able to generate truly random sequences that are different every time.

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

$$\boxed{I_{n+1} = (a\, I_n + c)\%m,} \tag{10.1}$$

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 iteratively generate a sequence of 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

$$x_n = \frac{I_n}{m} = \frac{I_n}{\texttt{RAND\_MAX}+1} \qquad \text{where} \qquad 0 \leq x_n < 1. \tag{10.2}$$

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 two bytes on some systems (i.e. `RAND_MAX`=32767). This is is not a very long period for Monte Carlo applications! These typically require many 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. In `Python` the basic `random()` function uses the Mersenne Twister generator, with a period of $2^{19937} - 1$, and is written in C.

10.2 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

$$U(x)\, dx = \begin{cases} dx & 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{10.3}$$

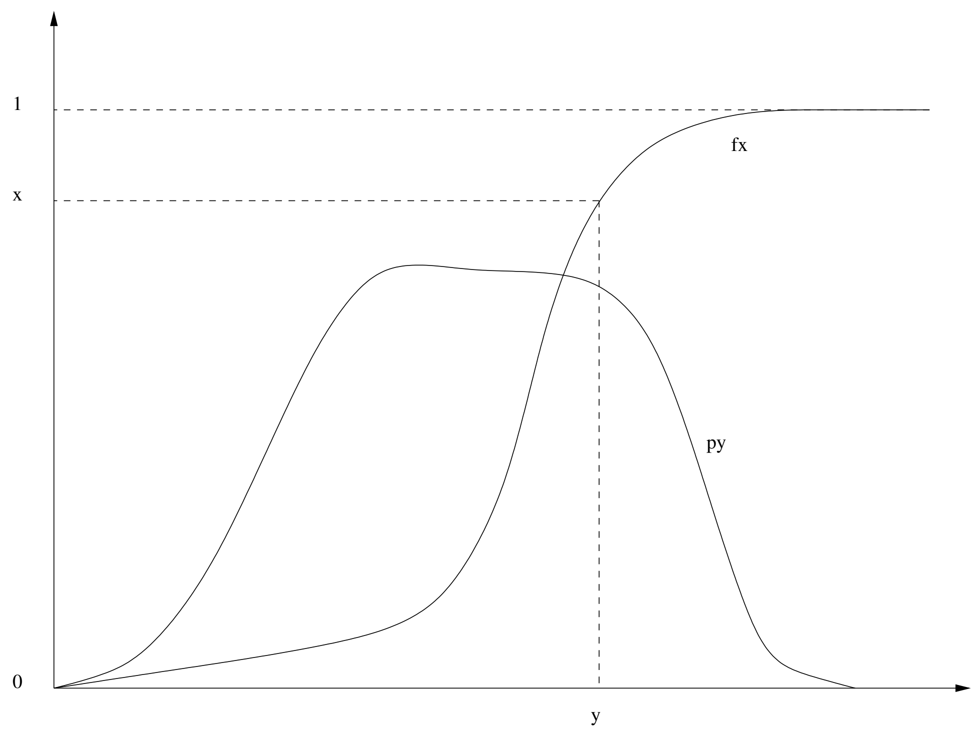

Figure 10.1: The transformation method. The Cumulative Distribution function $F(y) = \int_{-\infty}^{y} P(\tilde{y}) d\tilde{y}$ is inverted to find a deviate y for every uniform random deviate x.

which is also normalised; $\int_{-\infty}^{\infty} U(x)\,\mathrm{d}x = 1$. 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

$$|P(y)\,\mathrm{d}y\,| = |P(x)\,\mathrm{d}x\,|, \tag{10.4}$$

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 $|\dots|$ can be discarded) we have

$$\frac{\mathrm{d}x}{\mathrm{d}y} = P(y),$$

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

$$\boxed{F(y) = \int_{-\infty}^{y} P(\tilde{y})\,\mathrm{d}\tilde{y}\,,} \tag{10.5}$$

and then using the fundamental transformation law of probabilities it follows that

$$F(y) = \int_{-\infty}^{y} \frac{\mathrm{d}\tilde{x}}{\mathrm{d}\tilde{y}}\,\mathrm{d}\tilde{y} = \int_{0}^{x} d\tilde{x} = x. \tag{10.6}$$

(Note that $\sim$ is used to denote dummy integration variables here.) So if $F(y)$ is an invertible function, i.e.,

$$\boxed{y = F^{-1}(x)} \tag{10.7}$$

can be found, 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 $0 \leq F(y) < 1$. The transformation method is illustrated in fig. 10.1.

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 $P(y)$.

10.3 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

$$f(y) = C, \tag{10.8}$$

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.

The accepted $\boldsymbol{y_i}$ will have the desired distribution $\boldsymbol{P(y)}$. To prove this, consider the probability that the algorithm above creates an acceptable y_i in the range between y and $y + dy$; this is

$$\text{Prob} = \frac{\mathrm{d}y}{y_{\max} - y_{\min}} \times \frac{P(y)}{C} = \text{const } \times P(y)\,\mathrm{d}y\,. \tag{10.9}$$

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

$$\text{Eff} = \frac{1}{C}\frac{1}{y_{\max} - y_{\min}}. \tag{10.10}$$

1/Eff is the average number of times the steps (a)–(c) have to be carried out to get a y_i value. A way to interpret (10.10) is that it is the ratio of areas under the desired PDF $P(y)$ (perhaps a peaked curve) and the cover function $f(y)$ (a rectangle that fits over the PDF). By definition the PDF's area is $A_P = 1$. For the cover function $A_f = C \times (y_{\max} - y_{\min})$. Thus

$$\text{Eff} = \frac{A_P}{A_f}.$$

The rejection algorithm above effectively randomly picks a pair of coordinates (y_i, p_i) uniformly over the area A_f. A fraction of these 'throws' fall outside the area A_P under the PDF, in which case the coordinates are rejected and another throw is taken.

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;

(a) 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 $f(y)$.

Note that 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.

10.4 Monte Carlo Minimisation

An application of the Monte Carlo approach is to the problem of minimisation (optimisation) of functions. This is discussed in detail in section 11 where we also describe so-called *direct search* methods which do not involve random numbers.

10.5 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

$$I = \int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 \ldots f(\vec{x}) = \int_V f(\vec{x})\, d\vec{x}. \tag{10.11}$$

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

$$\langle f \rangle = \frac{1}{V} \int_V f(\vec{x})\, d\vec{x}. \tag{10.12}$$

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

$$I \equiv V \langle f \rangle. \tag{10.13}$$

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

$$\hat{I} = \frac{V}{N} \sum_{i=1}^{N} f(\vec{x}_i). \tag{10.14}$$

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

$$\sigma_{f_i}^2 = \frac{1}{N-1} \sum_{i=1}^{N} \left(f(\vec{x}_i) - \langle f \rangle\right)^2. \tag{10.15}$$

Given this, the variance of the mean $\langle f \rangle$ is $\sigma_{\langle f \rangle}^2 = \sigma_{f_i}^2/N$ and given (10.13), we can write the variance in the estimate of I as

$$\sigma_{\hat{I}}^2 = V^2 \sigma_{\langle f \rangle}^2 = \frac{V^2}{N} \sigma_{f_i}^2. \tag{10.16}$$

Therefore we can state the estimate of I (including its error) as

$$\boxed{\hat{I} \;=\; \frac{V}{N} \sum_{i=1}^{N} f(\vec{x}_i) \;\pm\; \frac{V}{\sqrt{N}} \sigma_{f_i}.} \tag{10.17}$$

Sampling random points within the integration volume in this way with an equal probability within the volume is called "uniform sampling". A key point is that the error in MC integration scales as $\mathcal{O}(h^{1/2})$.

Example – Consider the integral of the function shown in Fig. 10.2, a uniform circle of radius 1/2;

$$f(x,y) = \begin{cases} 1 & \text{if } \ (x-\frac{1}{2})^2 + (y-\frac{1}{2})^2 < (\frac{1}{2})^2 \\ 0 & \text{otherwise} \end{cases}. \tag{10.18}$$

In this example, we have the luxury of knowing that

$$I = \int f(x,y)\, dx\, dy = \text{circle area} = \pi r^2 = \frac{1}{4}\pi. \tag{10.19}$$

To calculate I using MC integration, we sample N random, uniformly distributed points in the range $0 \leq x < 1$ and $0 \leq y < 1$. The value of the sampled integrand $f(x_i, y_i)$ will either be 1

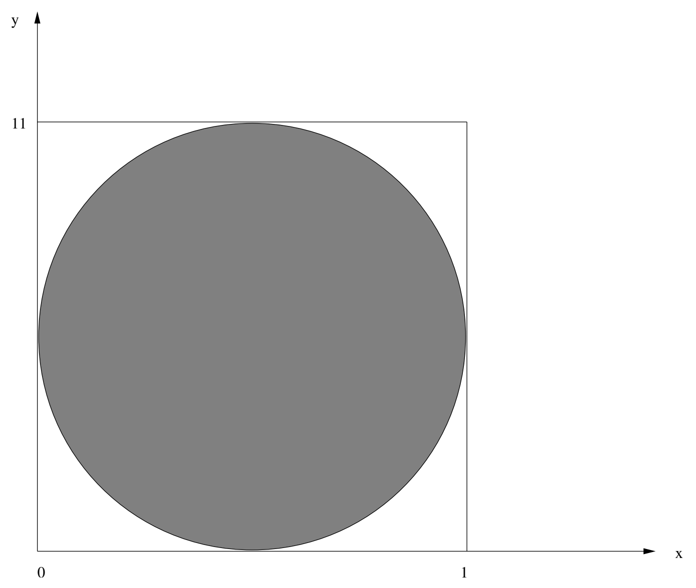

Figure 10.2: Circle of radius 1/2. The function $f(x, y) = 1$ inside the circle and $f(x, y) = 0$ outside

(*success*) or 0 (*fail*) to hit the circle. 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

$$p \equiv \frac{I}{A}. \tag{10.20}$$

The successes should follow a binomial distribution. Let N_1 be the number of successes. We can define an estimator for p as

$$\hat{p} = \frac{N_1}{N}. \tag{10.21}$$

We know that for a binomial distribution the variance of N_1 is

$$\sigma^2_{N_1} = N\, p\, (1 - p), \tag{10.22}$$

such that

$$\sigma^2_{\hat{p}} = \frac{\sigma^2_{N_1}}{N^2} = \frac{p\,(1-p)}{N}. \tag{10.23}$$

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

$$\boxed{\hat{I} = \hat{p}\, A = \hat{p} = \frac{N_1}{N} \pm \sqrt{\frac{p\,(1-p)}{N}}.} \tag{10.24}$$

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

Comparison with the trapezium method—Earlier in the course we covered other methods for integration. The extended 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

$$h \sim N^{-1/d}, \tag{10.25}$$

such that the error in the extended trapezium method calculation of the integral is

$$\epsilon \sim \mathcal{O}(h^2) \sim \mathcal{O}(N^{-2/d}). \tag{10.26}$$

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

$$\text{Error scalings} \quad \longrightarrow \quad \begin{array}{c|c|c} d & \text{Trapezium} & \text{MC} \\ \hline 1 & N^{-2} & N^{-1/2} \\ 2 & N^{-1} & N^{-1/2} \\ 3 & N^{-2/3} & N^{-1/2} \\ 4 & N^{-1/2} & N^{-1/2} \\ 5 & N^{-2/5} & N^{-1/2} \\ 6 & N^{-1/3} & N^{-1/2} \end{array} \tag{10.27}$$

Importance sampling—For some integrals the integrand is highly weighted to particular regions of $\vec{x}$. It is then more efficient to bias random samples to where the integrand has more 'weight' and less to the larger volume of parameter space where the integrand is vanishingly small. For such an integral the integrand can be split as follows:

$$I = \int_V f(\vec{x}) d\vec{x} = \int_V Q(\vec{x}) P(\vec{x}) d\vec{x} \tag{10.28}$$

where $P(\vec{x})$ behaves as (or is) a PDF (i.e. normalised and ≥ 0) and $Q = f(\vec{x})/P(\vec{x})$. Given that $\int_V P(\vec{x}) d\vec{x} = 1$ we can write

$$I = \int_V Q(\vec{x}) P(\vec{x}) d\vec{x} = \langle Q \rangle \tag{10.29}$$

where

$$\langle Q \rangle \; = \; \frac{1}{N} \sum_{i=1}^{N} Q(\vec{x}_i) \; \pm \; \frac{1}{\sqrt{N}} \sigma_{Q_i} \tag{10.30}$$

and

$$\sigma_{Q_i}^2 = \frac{1}{N-1} \sum_{i=1}^{N} \left(Q(\vec{x}_i) - \langle Q \rangle \right)^2 \tag{10.31}$$

From this we can see that Equ. 10.17 is a special case of Equ. 10.30, where $P(\vec{x})$ is uniform and equal to $1/V$ (therefore $Q = V f(\vec{x})$ and $\langle Q \rangle = V \langle f \rangle$). We are essentially doing the same thing as we did earlier, but with a different strategy for picking the samples. Instead of sampling evenly across the entire domain in which the integral is defined the samples are chosen to be proportional to $P(\vec{x})$. Because P is a PDF, the sum of the values of the samples divided by the number of samples gives the weighted average of Q with P as the weighting function.

The variation in $Q(\vec{x})$ is relatively flat, compared to if we had sampled/averaged $f = QP$ directly. This reduces σ_{Q_i} compared to σ_{f_i} and therefore *makes the error smaller* (for a given number of samples N).

To estimate I we now need to generate our random samples according the probability distribution $P(\vec{x})$. Sometimes this is simple - for some $P(\vec{x})$ distributions you can use the transformation method to convert a uniform random deviate to obtain $P(\vec{x})$, or you could use the rejection method. For more complex PDFs there is a better method.

The Metropolis Algorithm—The Metropolis algorithm can be used to pick samples that concentrate where $P(\vec{x})$ is largest. Essentially, it guides the 'trajectory' of $\vec{x}$ (taken to pick the samples) to seek the maximum of the $P(\vec{x})$ and wander about there. If the density of the samples is proportional to the size of $P(\vec{x})$, then the average of the values of $Q(\vec{x})$ at these sample points corresponds to the weighted integral of Q according to the PDF P.

Statistical Physics is a one example of an area where integrals can often be factored into a quantity to measure and a PDF. One wants to calculate the average of a quantity Q over all states, with a particular state being specified by $\vec{x}$. For example, for a classical system (or a hot quantum one) where the probability is governed by the Boltzmann energy distribution $P(\vec{x}) \propto \exp\left(-E(\vec{x})/k_B T\right)$, the average over states is

$$\langle Q \rangle = \frac{1}{Z} \int_{\vec{x}} Q(\vec{x}) \exp\left(-\frac{E(\vec{x})}{k_B T}\right) d\vec{x}, \tag{10.32}$$

where $Z = \int_{\vec{x}} \exp\left(-E(\vec{x})/k_B T\right) d\vec{x}$ is the partition function.

We can define

$$P(\vec{x}) = \frac{\exp\left(-\frac{E(\vec{x})}{k_B T}\right)}{Z} \tag{10.33}$$

and so the Metropolis algorithm for calculating $\langle Q \rangle$ is:

- Randomly pick a starting point $\vec{x}$.
- Repeatedly use the Metropolis algorithm to select a sample point.
  - Make a small trial step/change in the parameters to get $\vec{x}'$.
  - Calculate $P(\vec{x}')$ and $P(\vec{x})$. Accept or reject step using

$$p_{\text{acc}} = \begin{cases} 1 & \text{if} \quad P(\vec{x}') \geq P(\vec{x}) \\ P(\vec{x}')/P(\vec{x}) & \text{if} \quad P(\vec{x}') < P(\vec{x}) \end{cases} \tag{10.34}$$

    If accepted $\vec{x} = \vec{x}'$, else $\vec{x} = \vec{x}$.
  - Evaluate $Q(\vec{x})$ and add to running total $\sum Q$ (and $\sum Q^2$ if necessary).
- (Repeat whole process for a series of different starting points.)
- Divide through by the total number of samples to get $\langle Q \rangle$.

In the context of statistical physics the maximum of P corresponds to *equilibrium states* and the wandering corresponds to *thermal fluctuations*. Note that if started far from equilibrium, the samples obtained during the process of finding the equilibrium skew the result somewhat. They should be omitted unless they are a negligibly small proportion of the total samples. Repeating the process for different random starting points is optional but ensures good, even coverage of the integrand. Otherwise, with a single starting point, more samples will need to be taken.

In the above, we have not said anything about how the next point should be chosen. This is given by the proposal density $g(\vec{x}', \vec{x})$, the probability that we choose $\vec{x}'$ as a candidate (proposal) for the next point when we are currently at $\vec{x}$. A typical example is to choose a point near the current point by sampling from a Gaussian probability distribution that is centred on the current point: $g(\vec{x}', \vec{x}) = N(\vec{x}'|\vec{\mu} = \vec{x}, \vec{\sigma})$. The widths of the Gaussian $\vec{\sigma}$ (since we are in multiple dimensions) are problem-specific and need to be adjusted so that the jump to the next point is not too long (compared to the region sizes over which $P(\vec{x})$ remains large or small) and not too short (so that it is difficult to leave any local minima and span the entire space).

The Metropolis algorithm given here is a special case of the "Metropolis-Hastings" algorithm, which allows the proposal density function to be asymmetric—which is to say that the probability to jump from point x_b when you are currently at point x_a does not have to be the same as the probability to jump to point x_a if you are currently at point x_b. So for Metropolis-Hastings, $g(\vec{x}', \vec{x}) = g(\vec{x}, \vec{x}')$ does *not* need to hold. In the Gaussian case above, these probabilities are, of course, identical.

This generalisation to Metropolis-Hastings is allowed if we modify the acceptance criteria (Eq. 10.34 to:

$$p_{\text{acc}} = \begin{cases} 1 & \text{if} A(\vec{x}', \vec{x}) \geq 1 \\ A(\vec{x}', \vec{x}) & \text{if} \quad A(\vec{x}', \vec{x}) < 1 \end{cases} \tag{10.35}$$

where A is given by:

$$A(\vec{x}', \vec{x}) = \frac{P(\vec{x}')}{g(\vec{x}'|\vec{x})} \bigg/ \frac{P(\vec{x})}{g(\vec{x}|\vec{x}')} \tag{10.36}$$

An important point to note is that calculating $\langle Q \rangle$ requires $P(\vec{x})$ to be normalisable. In the example above we must be able to calculate the partition function. This is necessary to use the Metropolis and Metropolis-Hastings algorithms for integration, but is not required when using them to minimise functions, as discussed in the next chapter.

Chapter 11

Minimisation and Maximisation of Functions

Outline of Section

- One–dimensional minimisation
- Multi-dimensional minimisation
- Monte-Carlo minimisation
- Using minimisation to solve matrix equations

We first focus on **iterative methods** for 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. We also discuss Monte Carlo methods for function minimisation, borrowing the principles introduced in the previous chapter.

The function to be minimised is often called the "cost function" (and not just in economics), in which case the problem can be considered as minimising the "cost" of the system. It can also be called the "loss function". Both the position of a minimum $\vec{x}_*$ and the value of the function there $f(\vec{x}_*)$ 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 3.10. In principle roots of a non-linear function involving many variables $f(\vec{x})$ can be found by finding minima $\vec{x}_*$ of $|f(\vec{x})|^2$ where $f(\vec{x}_*) = 0$. However, root-finding is a significant area of study in its own right and this is not a generally-appropriate method.

Interestingly, it turns out that solving a matrix equation can be formulated as a minimisation problem, as we shall see later. Some of the most efficient iterative matrix solvers are based on this approach and converge much faster than Jacobi, Gauss-Seidel and SOR.

We will deal with **unconstrained** minimisation here. ***Constrained*** optimisation—where further constraints are placed on the independent variables or the values of the cost function—are not addressed here. Many of the concepts introduced here are, however, applicable in extended form when performing constrained minimisation.

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 $\vec{x} \in \mathbb{R}^N$. It is usually not difficult to generalise the methods here to these cases.

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^{obs} with errors σ_i in figure 11.1. The model we would like to fit is a linear one

$$d_i^{\text{model}} = a + \alpha\, X_i\,. \tag{11.1}$$

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

$$\chi^2(a, \alpha) = \sum_{i=1}^{N^{\text{data}}} \frac{\left(d_i^{\text{obs}} - d_i^{\text{model}}\right)^2}{\sigma_i^2}\,. \tag{11.2}$$

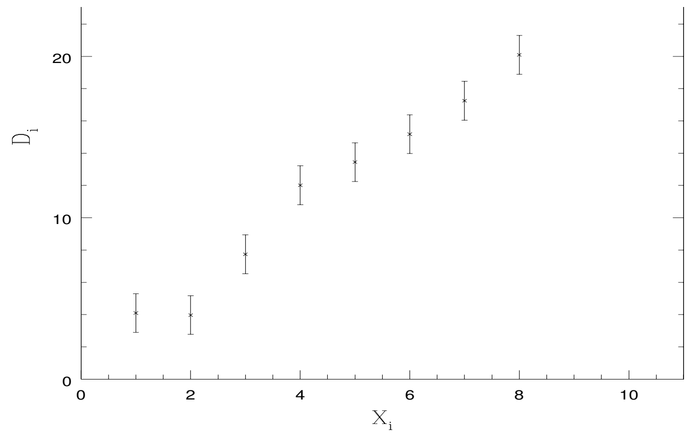

Figure 11.1: Some data with errors.

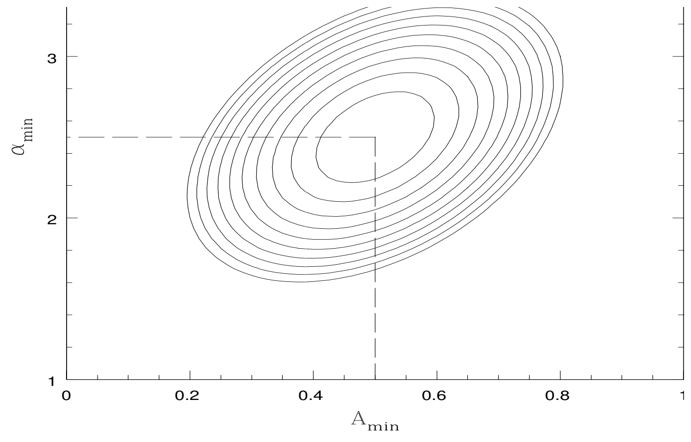

Figure 11.2: Contour plot of the χ^2 around the minimum. a_{min} and α_{min} are the maximum likelihood values for the parameters in the model. The curvature around the minimum is related to the error in the parameters.

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

$$L(a, \alpha) \propto e^{-\frac{1}{2}\chi^2(a,\alpha)}, \tag{11.3}$$

which is equivalent to finding the minimum in the χ^2. The result of the minimisation may look something like figure 11.2. In practice, the data that need to be fitted to and the models that are used are significantly more compliated than this example; cases with hundreds of free parameters (equivalently, degrees of freedom or dimensions) and numerous types of experimental data are common.

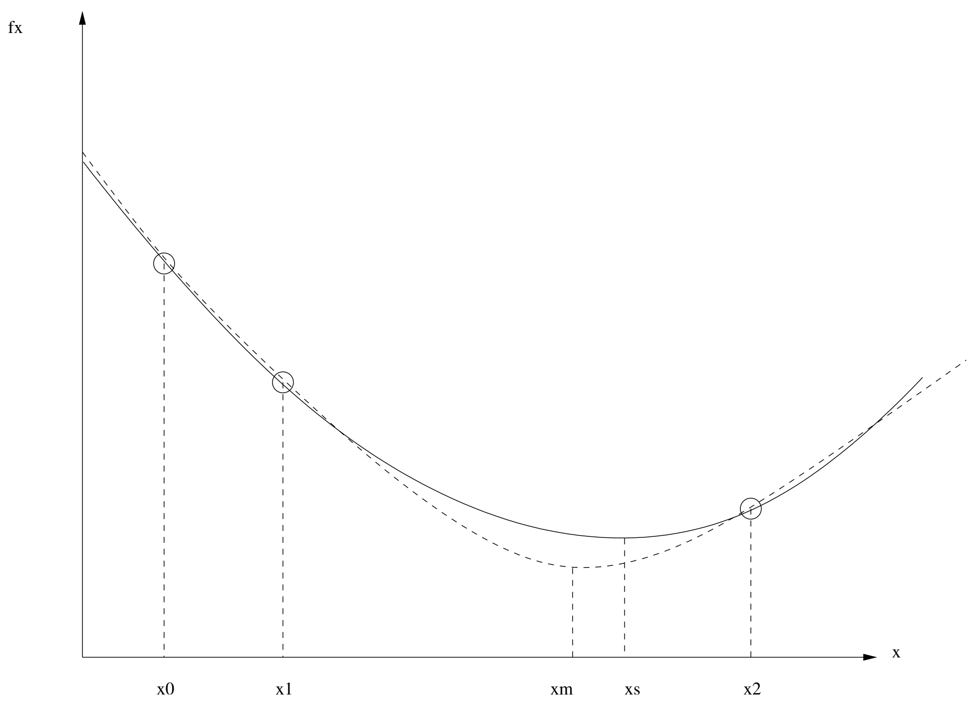

Figure 11.3: **The parabolic minimum search.** The function being minimised $f(x)$ is the dashed curve. A parabola (solid curve) is fit to the three points at x_0, x_1, x_2 and the minimum $x = x_3$ is found. Then the highest of the four points x_0, x_1, x_2, and x_3 is discarded and the method repeated. The minima of successive parabola approximations converge to the minimum of the function $(x_*, f(x_*))$.

11.1 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, when all we are interested in is the single global minimum—which we cannot find easily using this method. 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

$$f(x_0) = y_0, \quad f(x_1) = y_1, \quad \text{and} \quad f(x_2) = y_2 \tag{11.4}$$

and then fitting $P_2(x)$, the 2nd-order Lagrange polynomial, through the points (see section 4.1). The location of the minimum of the interpolating parabola $P_2(x)$ is found using equation (11.6) below; this value is x_3. This procedure is illustrated in fig. 11.3. 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_* 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

$$P_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2, \tag{11.5}$$

[c.f. equation (4.3)]. 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

$$\begin{aligned}\frac{\mathrm{d}P_2}{\mathrm{d}x} &= \frac{[(x-x_1)+(x-x_2)]\,(x_2-x_1)}{d}y_0 + \frac{[(x-x_0)+(x-x_2)]\,(x_0-x_2)}{d}y_1 + \\ &\quad \frac{[(x-x_0)+(x-x_1)]\,(x_1-x_0)}{d}y_2,\end{aligned}$$

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

$$\boxed{x_3 = \frac{1}{2}\frac{(x_2^2-x_1^2)y_0 + (x_0^2-x_2^2)y_1 + (x_1^2-x_0^2)y_2}{(x_2-x_1)y_0 + (x_0-x_2)y_1 + (x_1-x_0)y_2}.} \tag{11.6}$$

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.

11.2 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 11.4. Contours are shown for a 2D function $f(x,y)$. The univariate search spirals into the minimum, converging slowly.

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

$$\vec{d}(\vec{x}_n) = \vec{\nabla} f(\vec{x}_n) \qquad \text{where} \qquad \vec{\nabla} f = \begin{pmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \\ \vdots \\ \partial f/\partial x_N \end{pmatrix} \tag{11.7}$$

and is perpendicular to the local contour line. Note that the notation $\vec{\nabla} f(\vec{x}_n)$ is equivalent to $\vec{\nabla} f|_{\vec{x}_0}$, i.e., $\vec{\nabla} f$ evaluated at $\vec{x}=\vec{x}_0$. We can take a small step in the direction **opposite** to the gradient, i.e.,

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - \alpha\,\vec{d}(\vec{x}_n),} \tag{11.8}$$

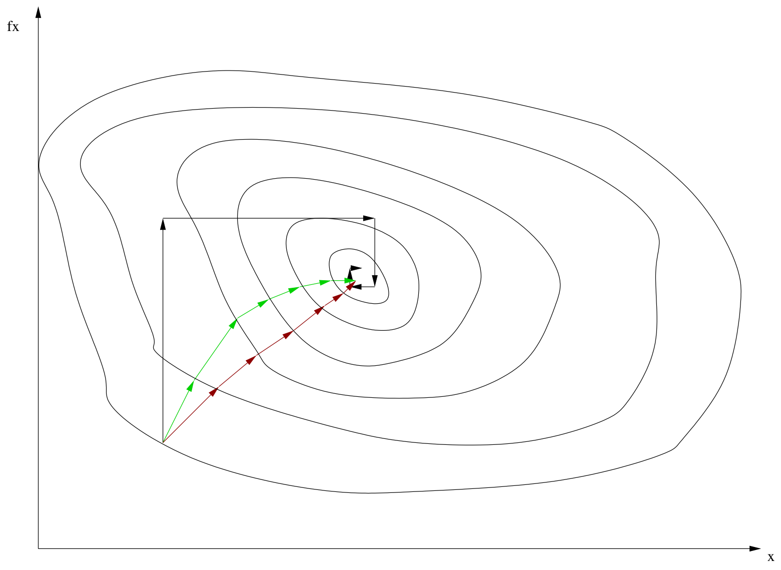

Figure 11.4: **Multi-dimensional minimisation** - for a function $f(x,y)$. Three methods are depicted: univariate method (black arrows), gradient method (green arrows) and Newton's method (red arrows). [Nb: this is not quite accurate; the green arrows are all supposed to be perpendicular to the contours!]

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

$$f(\vec{x}_{n+1}) < f(\vec{x}_n). \tag{11.9}$$

We can iterate this to find the minimum. (See green arrows in fig. 11.4.)

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

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, x_2, \dots, x_i + \Delta, \dots, x_N) - f(x_1, x_2, \dots, x_i, \dots, x_N)}{\Delta}. \tag{11.10}$$

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 $\vec{x}_0$. The minimum is displaced $\vec{\delta}$ away, at position $\vec{x}_0 + \vec{\delta}$. How can we estimate $\vec{\delta}$?

We use the Taylor series for the function about $\vec{x}$. (Later, $\vec{x}$ will be set to $\vec{x}_0$.)

$$f(\vec{x} + \vec{\delta}) = f(\vec{x}) + [\vec{\nabla} f(\vec{x})]^T \cdot \vec{\delta} + \frac{1}{2} \vec{\delta}^T \cdot \mathbf{H}(\vec{x}) \cdot \vec{\delta} + \mathcal{O}(|\vec{\delta}|^3), \tag{11.11}$$

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

$$H_{ij}(\vec{x}) = \frac{\partial^2 f(\vec{x})}{\partial x_i \partial x_j}. \tag{11.12}$$

(c.f. equation 2.6 and the paragraph which precedes it in section 2.2.)

To express this in an alternative format; any function that is the sum of the terms x^2, y^2, $x \times y$, x, y with coefficients and a constant can be written:

$$f(x, y) = \frac{1}{2} (x \;\; y) \, \mathbf{H} \begin{pmatrix} x \\ y \end{pmatrix} + (a \;\; b) \begin{pmatrix} x \\ y \end{pmatrix} + C, \tag{11.13}$$

which is to say:

$$f(x,y) = \frac{1}{2}\left(\frac{\partial^2 f}{\partial x^2}x^2 + \frac{\partial^2 f}{\partial x \partial y}2xy + \frac{\partial^2 f}{\partial y^2}y^2\right) + ax + by + C. \tag{11.14}$$

Since by definition $\vec{x}+\vec{\delta}$ is the location of the minimum, we must have $\vec{\nabla}f(\vec{x}+\vec{\delta}) = \vec{0}$. To actually be a minimum (rather than a maximum or a saddle point) the Hessian needs to be **positive definite**, i.e.,

$$\vec{\delta}^{\,T} \cdot \mathbf{H} \cdot \vec{\delta} > 0 \qquad \text{(for } \textit{all} \ \ \vec{\delta} \neq \vec{0}\ \text{).} \tag{11.15}$$

To determine $\vec{\delta}$ we can use the Taylor expansion (11.11) to first order in $\vec{\delta}$ and take its gradient yielding

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

Since we start at $\vec{x} = \vec{x}_0$ we can solve the following matrix-equation for $\vec{\delta}$,

$$\boxed{\mathbf{H}(\vec{x}_0)\cdot\vec{\delta} = -\vec{\nabla}f(\vec{x}_0).} \tag{11.16}$$

The notation $\mathbf{H}(\vec{x}_0)$ means (11.12) evaluated at $\vec{x} = \vec{x}_0$. If $f(\vec{x})$ is exactly 'parabolic', i.e.,

$$f(\vec{x}) = \vec{x}^{\,T}\cdot\mathbf{A}\cdot\vec{x} + \vec{b}^{\,T}\cdot\vec{x} + c, \tag{11.17}$$

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 estimate of $\vec{\delta}$ obtained from solving (11.16) 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

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - [\mathbf{H}(\vec{x}_n)]^{-1}\cdot\vec{\nabla}f(\vec{x}_n).} \tag{11.18}$$

This is illustrated by the red arrows in fig. 11.4. 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.

Example (adapted from Gerald and Wheatley p. 424) – Consider the 2D parabolic function

$$f(x,y) = x^2 + 2y^2 + xy + 3x$$

with the minimum at $\vec{x_*} = (x_*, y_*) = (-12/7, 3/7)$ from equating the derivatives to zero. If we start at the origin, the gradient and Hessian are given by

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} 2x+y+3 \\ x+4y \end{pmatrix} \quad , \quad \mathbf{H} = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 4 \end{pmatrix} \quad .$$

It is easy to find the inverse of the Hessian matrix:

$$\mathbf{H}^{-1} = \begin{pmatrix} \frac{4}{7} & \frac{-1}{7} \\ \frac{-1}{7} & \frac{2}{7} \end{pmatrix} \tag{11.19}$$

.

Hence starting at $\vec{x} = \vec{x}_0 = (x_0, y_0)$, equation (11.16) yields

$$\begin{aligned}
\vec{\delta} &= -\mathbf{H}^{-1}\nabla f \\
&= -\begin{pmatrix} \frac{4}{7} & \frac{-1}{7} \\ \frac{-1}{7} & \frac{2}{7} \end{pmatrix} \cdot \begin{pmatrix} 2x_0 + y_0 + 3 \\ x_0 + 4y_0 \end{pmatrix} \\
&= -\frac{1}{7}\left(\begin{pmatrix} 8x_0 + 4y_0 + 12 - x_0 - 4y_0 \\ -2x_0 - y_0 - 3 + 2x_0 + 8y_0 \end{pmatrix}\right) \\
&= \begin{pmatrix} -x_0 - \frac{12}{7} \\ -y_0 + \frac{3}{7} \end{pmatrix} \\
&= \vec{x_*} - \vec{x}_0
\end{aligned}$$

so that $\vec{\delta}$ takes us directly to the minimum (i.e. $\vec{x}_0 + \vec{\delta} = \vec{x_*}$) in this case of a parabolic function.

Aside – What we have done in obtaining equation (11.16) 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 $x = -b/2a \equiv x_*$. 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 $h = x_* - x_0$, demonstrating that getting h from the gradient of the Taylor expansion does indeed work for a parabola. For the multi-dimensional case equation (11.16) 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 derivatives 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

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - \alpha\, \mathbf{G}_n \cdot \vec{\nabla} f(\vec{x}_n),} \tag{11.20}$$

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 $\vec{\nabla} f$;

$$\vec{\delta}_n = \vec{x}_{n+1} - \vec{x}_n \qquad , \qquad \vec{\gamma}_n = \vec{\nabla} f(\vec{x}_{n+1}) - \vec{\nabla} f(\vec{x}_n). \tag{11.21}$$

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 (11.16) and the preceding paragraph when $\nabla f(\vec{x}_0 + \vec{\delta}) \neq 0$.) The trick is then to update $\mathbf{G}$ to satisfy

$$\mathbf{G}_n \cdot \vec{\gamma}_n = \vec{\delta}_n . \tag{11.22}$$

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–Powell) algorithm where the update is

$$\mathbf{G}_{n+1} = \mathbf{G}_n + \frac{(\vec{\delta}_n \otimes \vec{\delta}_n)}{\vec{\gamma}_n \cdot \vec{\delta}_n} - \frac{\mathbf{G}_n \cdot (\vec{\gamma}_n \otimes \vec{\gamma}_n) \cdot \mathbf{G}_n}{\vec{\gamma}_n \cdot \mathbf{G}_n \cdot \vec{\gamma}_n}, \tag{11.23}$$

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 (at worst) matrix multiplications, i.e., $\mathcal{O}(N^k)$ operations at each iteration and the resulting (approximated) Hessian is positive definite by construction. For full matrices (such as $\mathbf{G}_n$ and $(\vec{\delta}_n \otimes \vec{\delta}_n)$ here), naively one would think that the index k is 3. However it can be shown that matrix multiplication can be done with $k < 3$, in particular the Coppersmith & Winograd (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!

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.

11.3 Monte Carlo Minimisation – Simulated Annealing

Simulated annealing and related methods for optimisation introduce some randomness in the search for the minimum. They 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 local minima exist in which direct search methods can get stuck instead of finding the global minimum.

The simulated annealing approach is motivated by thermodynamics where random fluctuations can temporarily move a system to a less optimal (i.e. higher energy) configuration. The analogy is quantified in the use of the **Boltzmann Probability Distribution**

$$P(E)\, dE \sim \exp\left(-\frac{E}{k_B T}\right) dE, \tag{11.24}$$

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 obtain 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

$$\boxed{p_{\text{acc}} = \begin{cases} 1 & \text{if} \quad \Delta E \leq 0 \\ \exp(-\Delta E / k_B T) & \text{if} \quad \Delta E > 0 \end{cases}} \tag{11.25}$$

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. At high T the steps explore a large region of the the parameter space $\vec{x}$. As T is reduced the accepted steps move closer to the minimum of the function, but always have the possibility of stepping to a different region to escape local minima. After running the annealing for sufficient time one can examine the accepted steps in $\vec{x}$ to identify $\vec{x}_*$

11.4 Using minimisation to solve matrix equations

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

$$\boxed{f(\vec{x}) = \frac{1}{2}\vec{x}^T \cdot \mathbf{A} \cdot \vec{x} - \vec{b}^T \cdot \vec{x},} \tag{11.26}$$

happens to be the solution to

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

for a **symmetric, positive-definite** $N \times N$ matrix $\mathbf{A}$. This is because the gradient of this quadratic form is

$$\vec{\nabla} f(\vec{x}) = \mathbf{A} \cdot \vec{x} - \vec{b}. \tag{11.27}$$

At the minimum $\vec{x} = \vec{x}_*$,

$$\vec{\nabla} f(\vec{x_*}) = 0 \qquad \text{hence} \qquad \mathbf{A} \cdot \vec{x}_* = \vec{b}.$$

(Note the similarity of this quadratic form with equation (11.17) seen earlier.) So to solve the matrix equation, one of the multi-dimensional iterative methods seen in section 11.2 can be used with the quadratic form above.

The best iterative methods will (in the absence of round-off error) arrive at the exact solution in $N_{\text{iter}} = N$ iterations or less. This is better than Jacobi where convergence is as slow as $N_{iter} \sim \mathcal{O}(N^2)$ for some problems, and better than G-S & SOR. 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. 11.2, but more elongated), so that a descent along the local gradient comes close to the global minimum.

To show that $\vec{\nabla} f = \mathbf{A} \cdot \vec{x} - \vec{b}$, consider the k-th component of the gradient of (11.26);

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

where the last line makes use of the fact that $\mathbf{A}$ is symmetric. We have also used $\partial x_i / \partial x_k = \delta_{ik}$ where δ_{ik} is the Kronecker delta function.

Chapter 12

Partial Differential Equations

Outline of Section

- Classification of PDEs
- Solving Elliptic PDEs
- Solving Hyperbolic PDEs
- Solving Parabolic PDEs

12.1 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 dynamic (t) characteristics of a system, or when we are modelling 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.

Most of the PDEs of interest in physics are 2nd order linear PDEs. The general form of a 2nd order linear PDE involving two independent variables x and y with solution $u(x, y)$ takes the form

$$\boxed{A\,\frac{\partial^2 u}{\partial x^2} + B\,\frac{\partial^2 u}{\partial x \partial y} + C\,\frac{\partial^2 u}{\partial y^2} + f\left(u, x, y, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right) = G(x, y),} \tag{12.1}$$

where $f(\ldots)$ is an arbitrary function involving anything other than 2nd derivatives (but involving u somehow). Note that the coefficients A, B and C of the 2nd derivatives can depend on x, y, u, $\partial u/\partial x$ and/or $\partial u/\partial y$. $G(x, y)$ is independent of u and, if present, turns (12.1) into an inhomogeneous PDE.

In analogy with conic sections (in geometry)

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

(where A, ..., F are constants) we define the **discriminant**

$$\boxed{Q = B^2 - 4\,A\,C,} \tag{12.2}$$

and mathematically classify PDEs as

$$\boxed{\begin{aligned} Q < 0 \quad &: \text{Elliptic,} \\ Q = 0 \quad &: \text{Parabolic,} \\ Q > 0 \quad &: \text{Hyperbolic.} \end{aligned}} \tag{12.3}$$

For conic sections, $Q < 0$ yields an elliptical curve, $Q = 0$ a parabola and $Q > 0$ a hyperbola. The presence or absence of the inhomogeneous term $G(x, y)$ does not impact the elliptical/parabolic/hyperbolic classification of the PDE.

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. Luckily, for the common PDEs occurring in physics, the classification based on 2 variables (either 1 time + 1 spatial, or 2 spatial) happens to be the same when you expand the problem into more spatial dimensions. We will not show this, but merely state it for a few examples.

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

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

$$\nabla^2 u \equiv \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x,y), \tag{12.4}$$

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'. With reference to the general equation (12.1), for Poisson's equation $A = 1$, $B = 0$, $C = 1$ hence $Q = -4 < 0$. 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)**:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D\,\frac{\partial u}{\partial x}\right), \tag{12.5}$$

where $D \equiv D(x,t)$ is the diffusion coefficient. Here, $A = D$, $B = C = 0$ hence $Q = 0$. (Note that B is the coefficient of $\partial^2 u/\partial t \partial x$ and C pertains to $\partial^2 u/\partial t^2$.) 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)**:

$$\frac{\partial^2 u}{\partial t^2} = v^2\,\frac{\partial^2 u}{\partial x^2}, \tag{12.6}$$

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 by 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 method**. We may also impose conditions on the spatial boundaries in this case.

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).
$u(-x) = u(x)$	: Reflective BC (e.g. about $x_l = 0$).

where, e.g., x_l and x_r denote the left and right edges of the domain. 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 9.2 and mixed conditions in 9.4 when solving boundary value problems for ODEs.

12.2 Solving Elliptic PDEs

12.2.1 Dirichlet boundary conditions

To solve an elliptic equation we take a similar approach to the finite difference method of section 9.4 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

$$\nabla^2 u(x, y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \tag{12.7}$$

with Dirichlet boundary conditions, i.e., u specified on the boundary. We will see that we obtain a matrix equation $\mathbf{A} \cdot \vec{u} = \vec{b}$ (as we did in section 9.4 for 1D boundary value problems). 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 will see that there is a way to implement the Jacobi method without resorting to actual matrix algebra, but rather *directly* working on the 2D array of gridded values for u.

We consider a rectangular domain extending $0 \le x \le L_x$ and $0 \le y \le L_y$. There are $n_x + 2$ regularly-spaced points in the x-direction including the edges and similarly n_y+2 in the y-direction.[1] 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., $\Delta x \ne \Delta y$. It does not

[1]Notational note: "n_x", "n_y", etc. are what we use here for clarity, but you will often see the (horrible) alternative notation "nx" and "ny" in numerical analysis texts. Beware!

change the approach.) The location of the grid points are defined as

$$x_i = i\,h \qquad (i = 0,\,1,\,\dots,n_x+1) \qquad L_x = (n_x+1)\,h \tag{12.8}$$

$$y_j = j\,h \qquad (j = 0,\,1,\,\dots,n_y+1) \qquad L_y = (n_y+1)\,h. \tag{12.9}$$

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

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

$$\begin{array}{ccccc}
C_{0,4} & C_{1,4} & C_{2,4} & C_{3,4} & C_{4,4} \\
* & * & * & * & * \\
C_{0,3} & u_{1,3} & u_{2,3} & u_{3,3} & C_{4,3} \\
* & \circ & \circ & \circ & * \\
C_{0,2} & u_{1,2} & u_{2,2} & u_{3,2} & C_{4,2} \\
* & \circ & \circ & \circ & * \\
C_{0,1} & u_{1,1} & u_{2,1} & u_{3,1} & C_{4,1} \\
* & \circ & \circ & \circ & * \\
C_{0,0} & C_{1,0} & C_{2,0} & C_{3,0} & C_{4,0} \\
* & * & * & * & *
\end{array} \tag{12.10}$$

where

$$u_{i,j} = u(x_i, y_j). \tag{12.11}$$

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 5.1, i.e., equation (5.13), which yields

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{i,j} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} \qquad \{\,\mathcal{O}(h^2)\,\}\,, \tag{12.12}$$

and

$$\left.\frac{\partial^2 u}{\partial y^2}\right|_{i,j} = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} \qquad \{\,\mathcal{O}(h^2)\,\}\,, \tag{12.13}$$

giving the finite difference approximation to the Laplacian

$$\boxed{\nabla^2 u_{i,j} = \frac{u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1} - 4u_{i,j}}{h^2} \qquad \{\,\mathcal{O}(h^2)\,\}\,.} \tag{12.14}$$

(The $u_{i,j}$ in (12.14) are FD approximations to the true solution, but we dispense with the $\tilde{u}$ notation here.) This Laplacian can be represented as a **pictorial operator** acting on each unknown grid point

$$\nabla^2 u_{i,j} = \frac{1}{h^2}\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} u_{i,j}. \tag{12.15}$$

As depicted in figure 12.1, this visually shows how the 4 points to the left, right, above and below the centre point (where the Laplacian is being determined) are involved and gives their relative weights (+1). The relative weight of the centre point is −4. This is also known as a **finite difference stencil**.

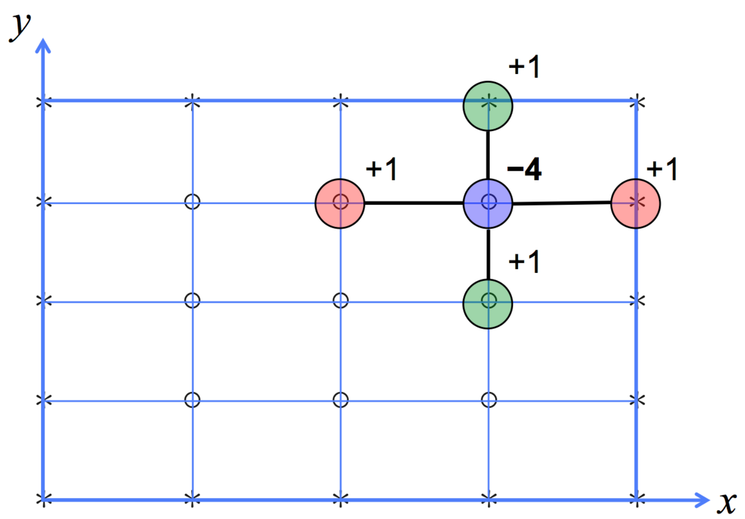

Figure 12.1: **FD stencil for the 2D Laplacian** $\nabla^2 u_{i,j}$. (Note that the $1/h^2$ factor is omitted in the indicated weights.)

Equation (12.14) applied at each internal grid point forms one of $m = n_x \times n_y$ simultaneous equations. We can represent the system of simultaneous equations as a matrix equation involving an $m \times m = n_x n_y \times n_x n_y$ matrix (i.e. 9×9 in our example)

$$\left(\begin{array}{ccc|ccc|ccc} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{array}\right) \left(\begin{array}{c} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ \hline u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ \hline u_{3,1} \\ u_{3,2} \\ u_{3,3} \end{array}\right) = - \left(\begin{array}{c} C_{0,1} + C_{1,0} \\ C_{0,2} \\ C_{0,3} + C_{1,4} \\ \hline C_{2,0} \\ 0 \\ C_{2,4} \\ \hline C_{3,0} + C_{4,1} \\ C_{4,2} \\ C_{4,3} + C_{3,4} \end{array}\right), \tag{12.16}$$

$$\text{i.e.} \qquad \mathbf{A} \cdot \vec{u} \; = \; \vec{b}. \tag{12.17}$$

Each unknown is one element of the solution vector $\vec{u}$. We use a 2D-array to 1D-array indexing scheme to package the unknowns into the vector. Our chosen mapping from the spatial indices i, j into the row index p of the solution vector $\vec{u}$ is

$$p = j + n_y \times (i - 1), \tag{12.18}$$

e.g. for our case with $n_x = n_y = 3$, unknown $u_{2,3}$ with $i = 2$ and $j = 3$ is located at row $p = 3 + 3 \times (2 - 1) = 6$. This mapping is not unique: We could equally as well have cycled over the x grid points first, and then the y points. (Nb: for 3 spatial dimensions, a 3D-array to 1D-array scheme – an extension of (12.18) – needs to be used.) The horizontal lines in (12.16) serve to visually vertically-partition the matrix and vectors into n_x blocks. Each block has n_y rows.

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 $u_{i,j\pm1}$ and $u_{i\pm1,j}$ are substituted into (12.14), the FD equivalent of Laplace's equation, then the resulting MDE can be shown to be

$$\nabla^2 u = -\frac{1}{12}\left(\frac{\partial^4 u}{\partial x^4} + \frac{\partial^4 u}{\partial y^4}\right) h^2 + \mathcal{O}(h^4) \tag{12.19}$$

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.

Solving the FD matrix equation – We can use the Jacobi method to solve this system since the spectral radius of the Jacobi update matrix $\mathbf{T}$ corresponding to $\mathbf{A}$ satisfies $\rho(\mathbf{T}) < 1$. [a]

Starting with some initial guess $\vec{u}^0$ we can iterate with

$$\vec{u}^{n+1} = -\mathbf{D}^{-1}\cdot(\mathbf{L}+\mathbf{U})\cdot\vec{u}^n + \mathbf{D}^{-1}\cdot\vec{b}. \tag{12.20}$$

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

$$\vec{u}^{n+1} = \frac{1}{4}\mathbf{I}\cdot\left[(\mathbf{L}+\mathbf{U})\cdot\vec{u}^n - \vec{b}\right]. \tag{12.21}$$

However this can be represented using the pictorial operator as

$$u^{n+1}_{i,j} = \frac{1}{4}\begin{bmatrix} & 1 & \\ 1 & 0 & 1 \\ & 1 & \end{bmatrix} u^n_{i,j}, \tag{12.22}$$

This pictorial operator averages over the nearest four neighbours for each of the internal points. At first sight, it seems that the boundary conditions have been forgotten since $\vec{b}$ is missing in (12.22). However, when updating an unknown on the edge of the internal domain (e.g. $u_{2,1}$) the pictorial operator automatically accesses the necessary boundary value(s) (i.e. $u_{2,0} = C_{2,0}$ in this case).

Thus the algorithm to solve the system using the Jacobi method in 'pictorial operator' form is

- Initialise all grid points including boundary values.
- Operate on all internal points with the pictorial operator, placing all $u^{n+1}_{i,j}$ values in a new array, i.e., leave $u^n_{i,j}$ values alone whilst scanning the pictorial operator over each grid point.
- Iterate until the solution has converged.

(See section 3.6, page 22 for details on checking for convergence). Note that Gauss-Seidel and SOR can also be implemented via a (different) pictorial operator scanning over the gridded values $u_{i,j}$ but there are extra considerations (not covered here).

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

$$\nabla^2 u = \rho(x,y), \tag{12.23}$$

when using the Jacobi method in pictorial-operator form requires the minor modification

$$u^{n+1}_{i,j} = \frac{1}{4}\begin{bmatrix} & 1 & \\ 1 & 0 & 1 \\ & 1 & \end{bmatrix} u^n_{i,j} - \frac{h^2}{4}\rho_{i,j}, \tag{12.24}$$

where the source term is discretised on an identical grid.

Extension to 3D – The methods above can readily be extended to 3D. The unknowns then become $u_{i,j,k}$ where 'k' indexes the third dimension (e.g. z) of the grid. One needs the FD form of the Laplacian in 3D. The (Jacobi) pictorial operator would then be a 3D stencil, accessing the 6 nearest neighbouring grid points.

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, so there is no need to worry about causality.

[a]The FD matrix of the 2D (and 3D) Laplacian is an important example of a matrix that is not *strictly* diagonally dominant (since $|A_{ii}|$ equals $\sum_{j\neq i}|A_{ij}|$ for some rows) but does actually work with the Jacobi method. This is because the Jacobi update matrix (formed from $\mathbf{A}$) turns out to have a spectral radius less then unity. Similarly, the FD form of the Laplacian is suitable for Gauss-Seidel and SOR iterative matrix solvers.

12.2.2 Derivative boundary conditions

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

$$u'_{i,j} = Q_{i,j} \tag{12.25}$$

i.e. Neumann boundary conditions. As we did in section 9.4 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. The setup is as follows for $n_x = 3$, $n_y = 3$ interior points to be solved for:

$$\begin{array}{ccccc}
 & u'_{1,4} = Q_{1,4} & u'_{2,4} = Q_{2,4} & u'_{3,4} = Q_{3,4} & \\
 & * & * & * & \\
C_{0,3} & u_{1,3} & u_{2,3} & u_{3,3} & C_{4,3} \\
* & \circ & \circ & \circ & * \\
C_{0,2} & u_{1,2} & u_{2,2} & u_{3,2} & C_{4,2} \\
* & \circ & \circ & \circ & * \\
C_{0,1} & u_{1,1} & u_{2,1} & u_{3,1} & C_{4,1} \\
* & \circ & \circ & \circ & * \\
 & u'_{1,0} = Q_{1,0} & u'_{2,0} = Q_{2,0} & u'_{3,0} = Q_{3,0} & \\
 & * & * & * &
\end{array}, \tag{12.26}$$

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

$$\begin{array}{ccccc}
 & u_{1,5} & u_{2,5} & u_{3,5} & \\
 & \ast & \ast & \ast & \\
C_{0,4} & u_{1,4} & u_{2,4} & u_{3,4} & C_{4,4} \\
\ast & \circ & \circ & \circ & \ast \\
C_{0,3} & u_{1,3} & u_{2,3} & u_{3,3} & C_{4,3} \\
\ast & \circ & \circ & \circ & \ast \\
C_{0,2} & u_{1,2} & u_{2,2} & u_{3,2} & C_{4,2} \\
\ast & \circ & \circ & \circ & \ast \\
C_{0,1} & u_{1,1} & u_{2,1} & u_{3,1} & C_{4,1} \\
\ast & \circ & \circ & \circ & \ast \\
C_{0,0} & u_{1,0} & u_{2,0} & u_{3,0} & C_{4,0} \\
\ast & \circ & \circ & \circ & \ast \\
 & u_{1,-1} & u_{2,-1} & u_{3,-1} & \\
 & \ast & \ast & \ast &
\end{array} . \tag{12.27}$$

This system has $m = n_x \times (n_y + 2)$ unknowns. The fictitious points then become the boundaries and can be initialised using the definition of the central difference scheme. For the lower boundary at $y = 0$ this yields

$$\left.\frac{\partial u}{\partial y}\right|_{i,0} = \frac{u_{i,1} - u_{i,-1}}{2h} = Q_{i,0} \qquad \text{giving} \qquad u_{i,-1} = u_{i,1} - 2h\,Q_{i,0}. \tag{12.28}$$

A similar condition needs to be applied at $y = L_y$ to give u_{i,n_y+2} in terms of Q_{i,n_y+1}. This system of simultaneous equations forms an $m \times m = n_x(n_y + 2) \times n_x(n_y + 2)$ matrix equation, which can then be solved iteratively as before, taking care to update the fictitious points along with the 'real' ones at each iteration.

12.2.3 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

$$\nabla^2 u(\vec{x}) = \rho(\vec{x}), \tag{12.29}$$

where ρ is some source density and u is some potential that we are seeking to solve for. For electrostatic problems this specialises to

$$\nabla^2 \phi(\vec{x}) = -\rho_c(\vec{x})/\epsilon_o,$$

where ϕ and ρ_c are the electrostatic 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.

The iterative method works from before works for any boundary conditions; periodic, fixed, etc. However, **for the case of periodic BCs**, using FFTs 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 in

Fourier space is

$$\tilde{u}(\vec{k}) = -\frac{\tilde{\rho}(\vec{k})}{|\vec{k}|^2}. \tag{12.30}$$

To obtain the exact solution in real (coordinate) space we just need to inverse transform the above

$$u(\vec{x}) = \mathcal{F}^{-1}\left[-\frac{\tilde{\rho}(\vec{k})}{|\vec{k}|^2}\right]. \tag{12.31}$$

In practice for general $\rho(\vec{x})$, tractable, analytic expressions for the Fourier transform integrals will be difficult (even impossible) to find.

Equipped with a discrete Fourier transform, the obvious way to solve (12.29) numerically (in, e.g., 2D) would seem to be to take an 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 12.2 (i.e. equation (12.24)). The reason is that we have not yet taken into account the finite difference approximation of the Laplacian.

12.2.4 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

$$\frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} = \rho_j. \tag{12.32}$$

Expanding each term through the backwards DFT we have

$$u_j = \frac{1}{N}\sum_p \tilde{u}_p e^{-i2\pi pj/N}, \tag{12.33}$$

and

$$u_{j\pm1} = \frac{1}{N}\sum_p \tilde{u}_p e^{-i2\pi p(j\pm1)/N} = \frac{1}{N}\sum_p \tilde{u}_p e^{\mp i2\pi p/N} e^{-i2\pi pj/N}. \tag{12.34}$$

Putting these into (12.32) gives

$$\frac{1}{N}\sum_p \tilde{u}_p e^{-i2\pi pj/N}\left[e^{+i2\pi p/N} + e^{-i2\pi p/N} - 2\right] = \frac{h^2}{N}\sum_p \tilde{\rho}_p e^{-i2\pi pj/N}. \tag{12.35}$$

Equating the bits inside the sum we have

$$\tilde{u}_p = \frac{h^2\tilde{\rho}_p}{\left[e^{+i2\pi p/N} + e^{-i2\pi p/N} - 2\right]}, \tag{12.36}$$

and since

$$e^{i\theta} = \cos\theta + i\sin\theta, \tag{12.37}$$

the denominator simplifies to

$$\left[e^{+i2\pi p/N} + e^{-i2\pi p/N} - 2\right] = \left[e^{+i\pi p/N} - e^{-i\pi p/N}\right]^2 = (2i)^2\sin^2(\pi p/N), \tag{12.38}$$

giving

$$\tilde{u}_p = -\frac{h^2\tilde{\rho}_p}{4\sin^2(\pi p/N)} \tag{12.39}$$

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, the 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 (12.39)

$$u_j = -\frac{h^2}{4N} \sum_p \frac{\tilde{\rho}_p}{\sin^2(\pi p/N)} e^{-i2\pi pkj/N}. \tag{12.40}$$

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.

12.3 Solving Hyperbolic PDEs

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

$$\frac{\partial^2 u}{\partial t^2} - v^2 \frac{\partial^2 u}{\partial x^2} = 0, \tag{12.41}$$

in the 1D, linear case where v is the phase speed. We are looking to solve for $u(x,t)$ with $0 \leq x \leq L$ and $t_i \leq t \leq t_f$, 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 $t_i < t \leq t_f$. We can discretise the solutions using h as the spatial step and Δt as the time step. The locations of the grid points in (x,t) are defined as

$$x_i = i\,h \qquad (i = 0, 1, \ldots, n_x) \qquad h = L/n_x \tag{12.42}$$

$$t^j = t_i + j\,\Delta t \qquad (j = 0, 1, \ldots, n_t) \qquad \Delta t = (t_f - t_i)/n_t. \tag{12.43}$$

This approach is depicted in figure 12.2. 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 to solve (12.41) 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 the next (unknown) time t^{j+1}. However, this is not the best way in this particular case. 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,

$$\frac{\partial f}{\partial t} + v\frac{\partial f}{\partial x} = 0 \quad , \quad \frac{\partial g}{\partial t} - v\frac{\partial g}{\partial x} = 0, \tag{12.44}$$

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

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

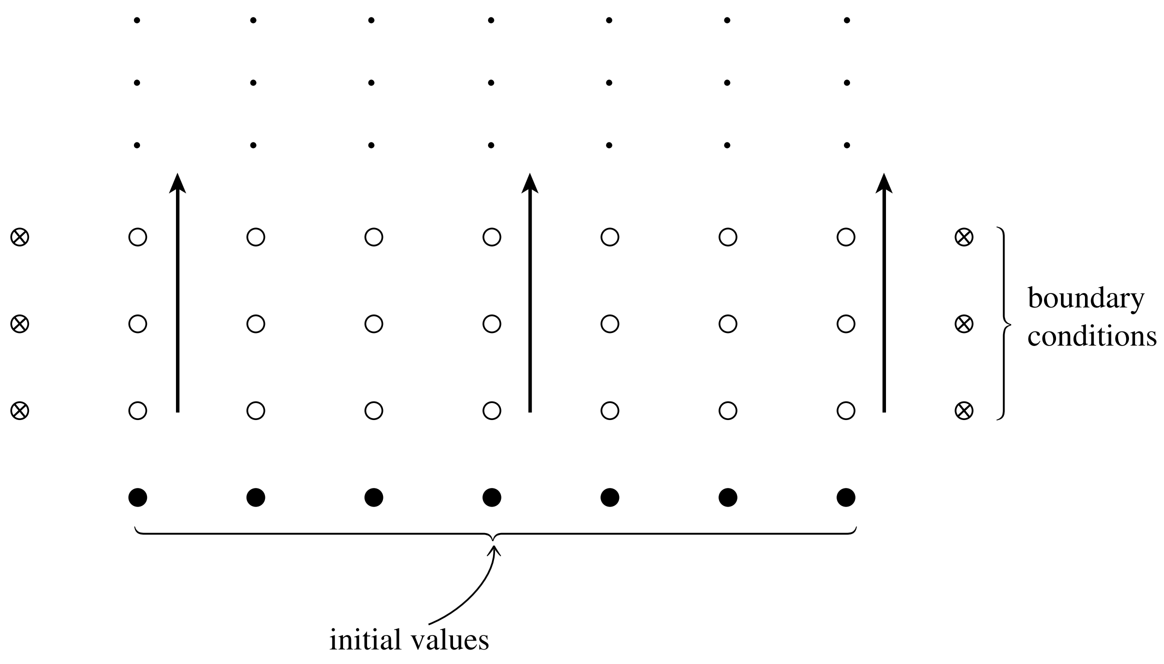

Figure 12.2: Initial value problem in (x,t) and the 1+1 discretisation scheme. The solution is integrated forward in time from an initial solution at a given "time slice", with boundary conditions given at the extremal values of x over all times. From Numerical Recipes.

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

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

$$\frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} = 0, \tag{12.45}$$

where now v_x is a velocity, so that its sign determines the direction of propagation. In practical terms, what we will be doing can be considered as an extension of the ODE methods (for initial value problems), seen in sections 7–8, which were used to solve

$$\frac{\mathrm{d}u}{\mathrm{d}t} = f(t,u),$$

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

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.* For example, in the case of an acoustic wave in gas

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\vec{u}) &= 0, \qquad \text{(mass conservation law)} \qquad (12.46)\\ \frac{\partial(\rho\vec{u})}{\partial t} + \vec{u}\cdot\nabla(\rho\vec{u}) &= -\nabla P, \qquad \text{(eqn. of motion)} \qquad (12.47)\end{aligned}$$

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 ($PV^\gamma = \text{const.}$) which yields $P\rho^{-\gamma} = P_o\rho_o^{-\gamma}$ (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

$$\begin{aligned}\frac{\partial\rho}{\partial t} + \rho_o\frac{\partial u}{\partial x} &= 0,\\ \frac{\partial u}{\partial t} + \frac{c^2}{\rho_o}\frac{\partial\rho}{\partial x} &= 0,\end{aligned} \qquad (12.48)$$

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

$$\frac{\partial\vec{U}}{\partial t} = -\frac{\partial(\mathbf{F}\cdot\vec{U})}{\partial x}, \qquad (12.49)$$

where **F** is known as a **flux operator**. In our example

$$\vec{U} \equiv \begin{bmatrix}\rho(x,t)\\ u(x,t)\end{bmatrix} \qquad \text{and} \qquad \mathbf{F} \equiv \frac{1}{\rho_o}\begin{bmatrix}0 & \rho_o^2\\ c^2 & 0\end{bmatrix}. \qquad (12.50)$$

Equation (12.49) looks like an advection equation for a vector of functions, when we bear in mind that **F** is a matrix 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).

12.3.1 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.* For $v_x > 0$ we have

$$\left.\frac{\partial u}{\partial x}\right|_i^j = \frac{u_i^j - u_{i-1}^j}{h} \qquad \{\mathcal{O}(h)\}. \qquad (12.51)$$

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

$$\left.\frac{\partial u}{\partial t}\right|_i^j = \frac{u_i^{j+1} - u_i^j}{\Delta t} \qquad \{\mathcal{O}(\Delta t)\}. \qquad (12.52)$$

The resulting FDE equation for advection (for +ve v_x) is thus

$$\boxed{u_i^{j+1} = u_i^j - \frac{|v_x|\Delta t}{h}\left(u_i^j - u_{i-1}^j\right)}, \qquad (12.53)$$

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

$$a = |v_x|\Delta t/h, \qquad (12.54)$$

is known as the **advection number**, an important dimensionless parameter with respect to the numerical properties of FD methods for hyperbolic PDEs (as we shall see shortly).

Figure 12.3 shows the **finite difference stencil** corresponding to equation (12.53). Such stencils are used to visualise the points involved in the algorithm.

For $v_x < 0$ we choose

$$\left.\frac{\partial u}{\partial x}\right|_i^j = \frac{u_{i+1}^j - u_i^j}{h} \qquad \{\mathcal{O}(h)\}, \qquad (12.55)$$

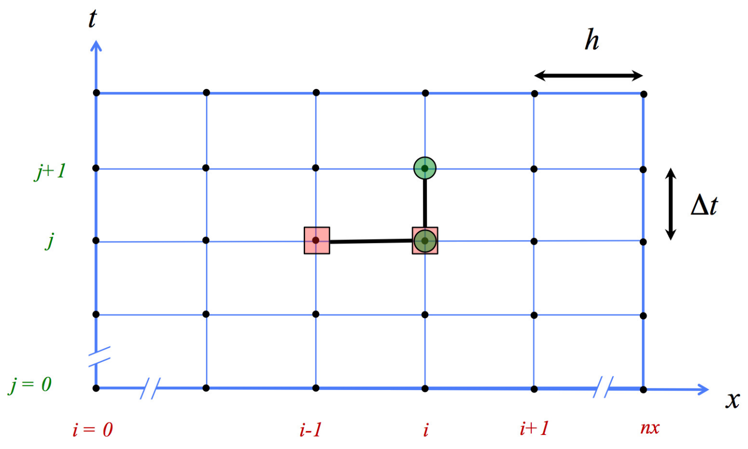

Figure 12.3: **FD stencil for upwind method.** Square and circle symbols denote contributions from $\partial u/\partial x$ and $\partial u/\partial t$ terms, respectively. The $v_x > 0$ case is shown.

and thus the FDE is

$$u_i^{j+1} = u_i^j + a(u_{i+1}^j - u_i^j). \tag{12.56}$$

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 physical speed of information propagation is finite for hyperbolic PDEs, explicit schemes are generally best. Implicit schemes are more computationally intensive for the same accuracy. They generally offer no benefits for hyperbolic PDEs.

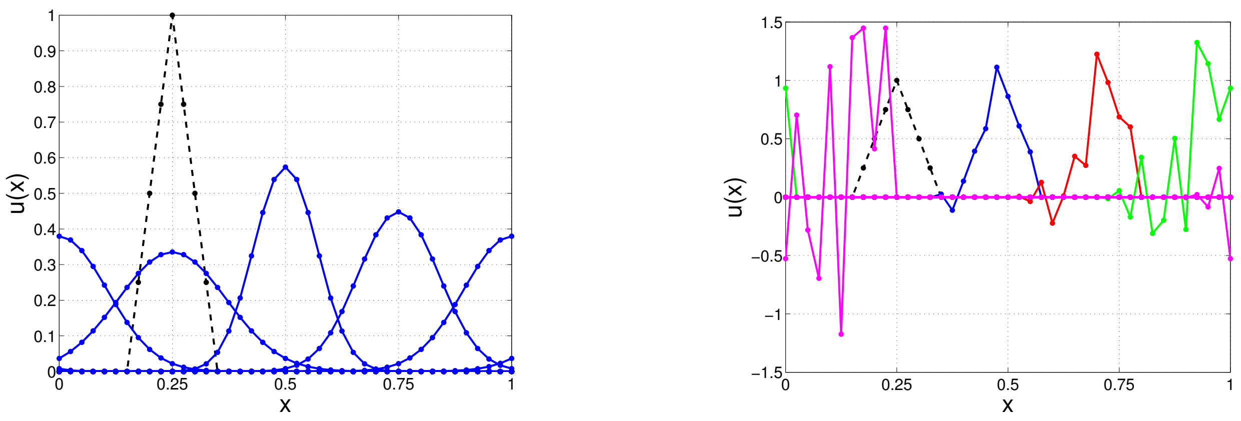

Figure 12.4: **Advection via the upwind method.** (Left) Advection number $a = 0.5$, and (Right) $a = 1.05$. The initial profile is a triangle (dotted black lines). Parameters: $v_x = +1$, $h = 1/40$, (left) $\Delta t = h/2$ and (right) $\Delta t = 1.05h$. The profile moves to the right and eventually returns to its starting location because of the periodic BCs. The 4 snapshots are at $t = 0.25$, $0.5.\,0.75$, 1.0. For $a = 0.5$, severe numerical diffusion can be seen. For $a = 1.05$ the upwind method is unstable.

Figure 12.4 shows the upwind method applied to an initially triangular profile, in the case of positive v_x. The spatial resolution is quite coarse ($n_x = 40$). For an advection number of $a = 0.5$ the profile unphysically spreads and blurs as it moves, due to inaccuracies of the upwind method. For $a = 1.05$ the profile breaks up due to numerical instability. We will look more closely at these two issues (numerical diffusion and instability) below.

Consistency & accuracy – The modified differential equation obtained from (12.53) is

$$\frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} \;=\; -\frac{1}{2}\frac{\partial^2 u}{\partial t^2}\,\Delta t + \mathcal{O}(\Delta t^2) + \frac{1}{2}\frac{\partial^2 u}{\partial x^2}\,v_x h + \mathcal{O}(h^2). \qquad (12.57)$$

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**'.

Numerical diffusion – The above MDE can be rewritten as

$$\begin{aligned}\frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} \;&=\; \frac{|v_x|h}{2}(1-a)\,\frac{\partial^2 u}{\partial x^2} \;+\; \mathcal{O}(\Delta t^2) \;+\; \mathcal{O}(h^2) \\ &\approx\; D_{\text{num}}\,\frac{\partial^2 u}{\partial x^2}, \end{aligned} \qquad (12.58)$$

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. Numerical diffusion causes a pulse with sharp features (e.g. top hat function) to artificially blur out as it propagates along. The higher order error terms further artificially modify the dynamics but are weaker than numerical diffusion.

Matrix equation – Practical implementation of (12.53) 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

$$u^j_{n_x} = u^j_0, \qquad (12.59)$$

on our choice of spatial grid. Although there are $n_x + 1$ spatial points, periodic BCs reduces the number of unknowns to $m = n_x$. The matrix equation (for the $v_x > 0$ case) is then

$$\begin{aligned}\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n_x} \end{pmatrix}^{j+1} &= \begin{pmatrix} (1-a) & 0 & 0 & \dots & 0 & a \\ a & (1-a) & 0 & \dots & 0 & 0 \\ 0 & a & (1-a) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a & (1-a) \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n_x} \end{pmatrix}^{j}, \\ \vec{u}^{\,j+1} &= \mathbf{T}\cdot\vec{u}^{\,j}, \end{aligned} \qquad (12.60)$$

where $\mathbf{T}$ is the update matrix (analogous to those seen with ODE FD methods in sections 2–8).

12.3.2 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**. However, it only works for periodic boundary conditions.

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

$$\boxed{u^j_i = \exp\!\big[i(kx_i - \omega t^j)\big] = (g)^j\,\exp(ikx_i),} \qquad (12.61)$$

into the FDE. This mode is sampled at discrete spatial and temporal points. Here the amplification factor g (a complex quantity) 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

$$u_i^{j+1} = g\, u_i^j \qquad , \qquad u_{i\pm1}^j = u_i^j \, \exp(\pm i\, k\, h). \tag{12.62}$$

Substituting into the upwind method FDE (12.53) 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 $1-\cos(2\theta) = 2\sin^2\theta$ yields

$$|g|^2 = 1 - 4a(1-a)\sin^2(kh/2). \tag{12.63}$$

The maximum wave-number allowable on the grid is

$$k_{max} = \frac{\pi}{h} \qquad (\text{i.e. } \lambda_{min} = 2h\,), \tag{12.64}$$

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

$$|g| \leq 1,$$

i.e., $1 - 4a(1-a) \leq 1$ which yields $a \geq 0$ and

$$\boxed{a \leq 1 \qquad \text{or equivalently} \qquad \Delta t \leq h/|v_x|.} \tag{12.65}$$

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

$$|v_x| \leq v_{\text{grid}}, \tag{12.66}$$

i.e. the physical, speed of propagation of information 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 physically 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 CFL condition to get good accuracy.

Referring back to the form of the Fourier mode (12.61), we saw in Section 6.2 that a discrete (and finite) set of k values are allowed on a spatial grid. For a given, allowed wavelength the numerical scheme determines the effective numerical wave frequency ω_{num} (since $\omega_{\text{num}} = i \ln g/\Delta t$) that can propagate. ω_{num} can differ from the true value of kv_x, especially at short wavelengths, leading to incorrect phase speed, a phenomenon known as **numerical dispersion**. If ω_{num} has an imaginary part, then there will be unphysical **numerical damping** (or growth!) of the Fourier mode.

12.3.3 Stability analysis – matrix method

This is the same as the approach used for coupled ODEs in section 7.7. In this context the coupled variables are the u_i's at each spatial grid point.

We can 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. (12.60)).

For stability we require that the spectral radius of the update matrix does not exceed unity, i.e. , $\rho(\mathbf{T}) \leq 1$. Thus all eigenvalues λ_i of the update matrix must satisfy

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

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

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

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

$$|\lambda - (1-a)| \leq a,$$

where a is a positive real value. So λ 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.

12.4 Solving Parabolic PDEs

Consider the diffusion equation in one spatial dimension with D constant:

$$\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}. \tag{12.67}$$

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. Let us choose the first-order, forward difference scheme for the time derivative

$$\left.\frac{\partial u}{\partial t}\right|_i^j = \frac{u_i^{j+1} - u_i^j}{\Delta t} \qquad \{\, \mathcal{O}(\Delta t) \,\}, \tag{12.68}$$

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

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_i^j = \frac{u_{i-1}^j - 2\,u_i^j + u_{i+1}^j}{h^2} \qquad \{\, \mathcal{O}(h^2) \,\}. \tag{12.69}$$

Inserting these approximations into the diffusion equation (12.67) we get

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} = D \left(\frac{u_{i-1}^j - 2\,u_i^j + u_{i+1}^j}{h^2} \right), \tag{12.70}$$

giving the FDE

$$\boxed{u_i^{j+1} = (1 - 2\,d)\,u_i^j + d\,(u_{i-1}^j + u_{i+1}^j),} \tag{12.71}$$

where

$$d = \frac{D\,\Delta t}{h^2}, \tag{12.72}$$

is the **diffusion number**, a dimensionless parameter. An example of the explicit method for diffusion is shown in figure 12.5.

The FDE (12.71) is a consistent, $\mathcal{O}(\Delta t) + \mathcal{O}(h^2)$ scheme. As usual, these properties can be shown by the MDE method.

Either the Von-Neumann method (section 12.3.2) or the matrix method (section 12.3.3) can be used to show that the condition for the FDE above (12.71) to be stable is

$$\boxed{d = \frac{D\,\Delta t}{h^2} < \frac{1}{2} \qquad \text{or equivalently} \qquad \Delta t < \frac{h^2}{2D}.} \tag{12.73}$$

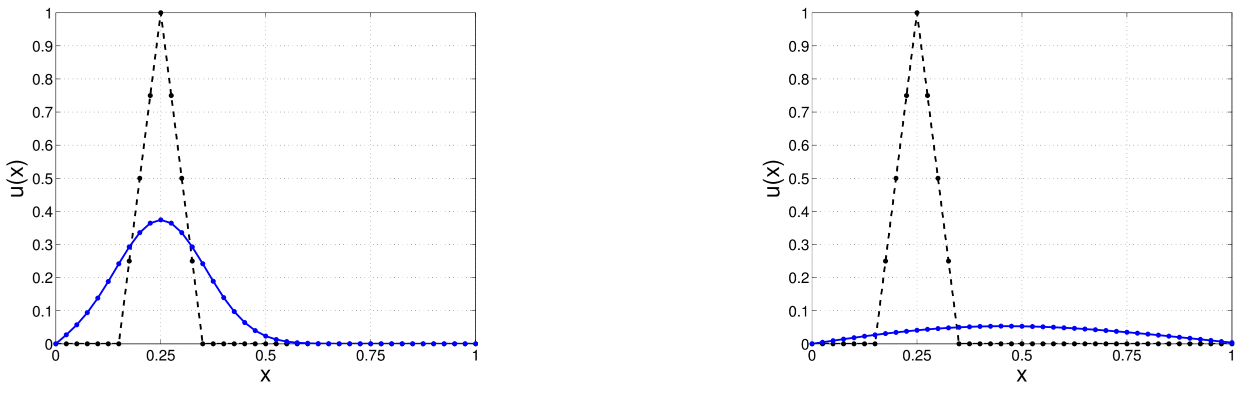

Figure 12.5: **Diffusion via the explicit method.** (Left) $t = 0.005$ and (Right) $t = 0.1$. The initial profile is a triangle (dotted black lines). Fixed (Dirichlet) boundary conditions are used; $u = 0$ on the boundaries. Parameters: $D = 1$, $h = 1/40$, $\Delta t = 0.25\tau_{\mathrm{diff-grid}}$ (i.e. $d = 0.25$).

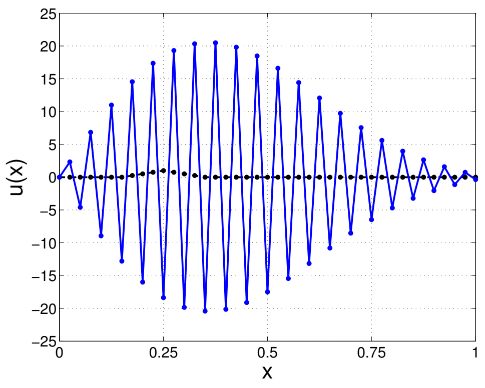

Figure 12.6: **Diffusion via the explicit method.** Numerical instability when the CFL condition is exceeded; shown at $t = 0.1$. Here $d = 0.51$. (The same system, and other parameters, are used as in figure 12.5.)

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

$$\tau_{\text{diff-grid}} \equiv \frac{h^2}{D}, \tag{12.74}$$

which characterises the time it takes for information to propagate between sites on the grid. Figure 12.6 shows an example of the numerical instability that occurs if the CFL condition is violated.

This is actually quite restrictive. If the spatial scale of the problem is l (e.g. pulse width) then the physical diffusion time scale is $\tau_D \sim l^2/D$. Given the CFL condition above, we have

$$\Delta t \leq \frac{\tau_D}{2}\left(\frac{h}{l}\right)^2 .$$

Since $h \ll l$ is required for good spatial resolution, many time steps are needed to observe the dynamics of the system (e.g. diffusive spreading of the pulse).

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

$$\left.\frac{\partial u}{\partial t}\right|_i^{j+1/2} = \frac{u_i^{j+1} - u_i^j}{\Delta t} \qquad \{\, \mathcal{O}(\Delta t^2)\,\}. \tag{12.75}$$

For the spatial derivative we can take the average of those at times j and $j+1$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_i^{j+1/2} = \frac{1}{2}\left[\left.\frac{\partial^2 u}{\partial x^2}\right|_i^{j} + \left.\frac{\partial^2 u}{\partial x^2}\right|_i^{j+1}\right]. \tag{12.76}$$

Rearranging all $j+1$ terms onto the LHS we get the system

$$\boxed{(2+2\,d)\,u_i^{j+1} - d\,(u_{i-1}^{j+1} + u_{i+1}^{j+1}) = (2-2\,d)\,u_i^j + d\left(u_{i-1}^j + u_{i+1}^j\right),} \tag{12.77}$$

or in matrix form

$$\mathbf{M}\cdot\vec{u}^{\,j+1} = \vec{b}^{\,j}, \tag{12.78}$$

where

$$\mathbf{M} \equiv \begin{pmatrix} (2+2\,d) & -d & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ -d & (2+2\,d) & -d & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & -d & (2+2\,d) & -d & \cdot & \cdot & \cdot & 0 \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & -d & (2+2\,d) \end{pmatrix}, \tag{12.79}$$

and

$$\vec{b}^j \equiv \begin{pmatrix} (2-2\,d) & d & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ d & (2-2\,d) & d & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & d & (2-2\,d) & d & \cdot & \cdot & \cdot & 0 \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & d & (2-2\,d) \end{pmatrix} \cdot \vec{u}^j = \mathbf{A} \cdot \vec{u}^j. \qquad (12.80)$$

where Dirichlet BCs of $u_0^j = u_{n_x}^j = 0$ have been chosen in getting the above forms of $\mathbf{M}$ and $\vec{b}$. Hence the number of unknowns is $m = n_x - 1$ (i.e. $1 \leq i \leq n_x - 1$).

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).

To follow the dynamics of the system with acceptable accuracy without using an exceedingly small time step, we can solve equation (12.78) using an iterative matrix solver (Jacobi, Gauss-Seidel or SOR) at each time step.