Computational Physics

Physics MSci/BSc, Level 3

2018–19

Yoshi Uchida & Pat Scott

Contents

1 Course Description **3**

1.1 Overview . . . 3
1.2 Course components . . . 4
1.3 Assessment . . . 5
1.4 Course website . . . 6
1.5 Practical sessions . . . 7
1.6 Programming language . . . 7
1.7 Plagiarism . . . 7
1.8 Contact . . . 8
1.9 Literature . . . 8

2 Introduction to Numerical Calculations **11**

2.1 Numerical Accuracy and Errors . . . 11
2.2 Natural units . . . 14
2.3 Solving Non-Linear Algebraic Equations . . . 15

3 Matrix Algebra **17**

3.1 Introduction . . . 17
3.2 General Considerations . . . 18
3.3 Gaussian and Gauss-Jordan elimination . . . 18
3.4 LU Decomposition . . . 18
3.5 Iterative Solution – Jacobi Method . . . 20
3.6 Iterative Gauss-Seidel Method . . . 23
3.7 Iterative Successive Over-Relaxation Method . . . 24
3.8 Largest Eigenvalue of a Matrix . . . 24
3.9 Other Eigenvalues . . . 25

4 Interpolation **27**

4.1 Introduction . . . 27
4.2 Linear interpolation . . . 27
4.3 Bi-linear interpolation . . . 28
4.4 Lagrange polynomials . . . 28
4.5 Cubic splines . . . 28

5 Fourier Transforms **31**

5.1 Fourier Transforms—Recap . . . 32
5.2 Discrete FTs . . . 33
5.3 Sampling & Aliasing . . . 36
5.4 Fast Fourier Transforms—FFTs . . . 37
5.4.1 Pseudocode . . . 38
5.4.2 Fast Fourier Transforms in Pseudocode . . . 38

6 Random Numbers and Monte Carlo Methods 41
6.1 Random Numbers . . . 42
6.2 Non-uniform distributions – Transformation Method . . . 42
6.3 Non-uniform distributions – Rejection Method . . . 45
6.4 Monte Carlo Minimisation . . . 46
6.5 Monte Carlo Integration . . . 46

7 Minimisation and Maximisation of Functions 51
7.1 One–dimensional minimisation . . . 53
7.2 Multi-Dimensional methods . . . 54
7.3 Monte Carlo Minimisation – Simulated Annealing . . . 58
7.4 Using minimisation to solve matrix equations . . . 59

8 Finite Difference Methods 61
8.1 Numerical Differentiation . . . 61
8.2 Ordinary Differential Equations . . . 63
8.3 Consistency . . . 65
8.4 Accuracy . . . 66
8.5 Stability . . . 67
8.6 Stability Analysis of General (coupled) Linear Systems . . . 68
8.7 Convergence . . . 70
8.8 Efficiency . . . 70

9 ODEs: Initial Value Problems 71
9.1 Predictor-Corrector Method . . . 72
9.2 Multi-Step (Leapfrog) Method . . . 74
9.3 Runge-Kutta Methods . . . 74
9.4 Variable step size h . . . 76
9.5 Implicit Methods . . . 77

10 Numerical Integration 79
10.1 Introduction . . . 79
10.2 Quadrature Methods . . . 79
10.2.1 The Trapezoidal Rule . . . 80
10.2.2 Simpson's Rule . . . 81
10.2.3 Romberg integration . . . 82
10.3 Relationship to ODE methods . . . 82
10.4 Improper Integrals . . . 83

11 ODEs: Boundary Value Problems 87
11.1 Shooting Method . . . 87
11.2 Finite Difference Method . . . 88
11.3 Derivative Boundary Conditions . . . 89
11.4 Eigenvalue Problems . . . 90

12 Partial Differential Equations 91
12.1 Classification of PDEs . . . 91
12.2 Solving Elliptic PDEs . . . 93
12.2.1 Dirichlet boundary conditions . . . 93
12.2.2 Derivative boundary conditions . . . 97
12.2.3 Spectral methods – Solving the Poisson Equation by FFT . . . 98
12.2.4 Application to discrete system – 1-D . . . 99
12.3 Solving Hyperbolic PDEs . . . 100
12.3.1 Upwind method . . . 102
12.3.2 Stability analysis – von Neumann (Fourier) method . . . 104

12.3.3 Stability analysis – matrix method . . . 106
12.3.4 FTCS method – (one not to use!) . . . 106
12.3.5 Lax method . . . 106
12.3.6 Lax-Wendroff . . . 107
12.3.7 Coupled equations . . . 107
12.4 Solving Parabolic PDEs . . . 108

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 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 out 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 the field.

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, root-finding.
- Linear matrix algebra. Iterative methods for solution of linear equations.
- Interpolation. How to represent a functional form in a computer, given a finite number of data points.
- Fourier transform methods and their use in data processing.
- Random numbers. How random number generators work and how to use them to generate non-uniform random number distributions.
- Monte Carlo methods for integration, minimisation and simulating equilibria of ensembles of particles.
- Optimisation methods. Methods for finding the minimum and maxima of general multi-dimensional functions.

- Finite difference methods. Theory and analysis of the accuracy and stability of numerical methods for solving differential equations.
- Solution of initial-value and boundary-value ordinary differential equations using finite difference methods.
- Numerical Integration. Deterministic rules, evaluation of improper and bady-behaved integrals, ODE methods
- Solution of initial-value 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).
- Find the roots of an equation numerically.
- Invert matrices and solve matrix equations numerically.
- Interpolate regularly-spaced data in one dimension.
- Select suitable random number generators and use them for simulations and integration.
- Show 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 & efficiency) and implement finite difference methods to solve differential equations in physics.
- Formulate and solve initial value and boundary value problems.
- Evaluate definite integrals numerically, including improper examples.
- 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 series of $15 + 1$ lectures, covering the theory of numerical methods and their applications in physics. (See table 1.1 for the lecture schedule.)
- Two additional lectures on “Computational Physics in Action”, to put the course material into context.
- Eight weekly practical sessions, where you will practice applying the methods discussed in the lectures and these notes (mostly by working on problem sheets, the assignment and your project).

- A series of non-assessed problem sheets, featuring problems that will help you understand the lecture material.
- 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.

Practical sessions will be held in the undergraduate computing suite (Level 3 Blackett) on **Wednesday mornings** from **9–12pm** during weeks 4–11 of term. We will use both the main and the teaching room. See table 1.2 for the schedule of practical sessions, and submission deadlines for the assignment and project.

1.3 Assessment

The course will be assessed in three units:

- **Assignment** (assessed problems), to be submitted by **12 noon** on **Monday, 12th November 2018**. This will be worth **20%** of the marks for the course. There will be no choice of questions in the assignment.
- **Project**, to be submitted by the first Monday after the last day of term (**Monday, 17th December 2018** by **12 noon**). There will be a choice of several topics for this. The project will be worth **40%** of your final grade for the course.
- **Written Exam**, to be held in January 2019, 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 take place on the first day of Term 2 in the afternoon: **Monday, 7th January 2019, 2–4pm**. Make sure that you are back at Imperial for the first day of Term 2 and **allow for travel disruptions over the Christmas and New Year break. As with any other exam, apart from any officially approved Mitigating Circumstances, no exceptions will be allowed for delays or absences**.

The exam will cover all material from the lectures, and all examinable material from these notes.

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. There is a 2,500 word limit for the project. 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 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 a first & 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. Exam past papers from the last few years are available on the course website on Blackboard Learn and the central Departmental UG Examinations web pages.

Table 1.1: Approximate schedule of lectures held in LT3

Topic	Lectures
Introduction to the course and to Numerical Calculations	2
Matrix algebra	1
Interpolation	1
Numerical Methods for Fourier Transforms	1
Random Numbers	1
Monte Carlo Methods	2
Minimisation and Maximisation of Functions	1
Finite Difference Methods	2
Initial Value Problems	1
Numerical Integration	1
Boundary Value Problems	1
Partial Differential Equations	1
Computational Physics in Action	2
Revision Lecture	*1*

Table 1.2: Practical Sessions and Assessment Deadlines.

Lab	Date			Time	Topic
1	Wednesday	October	24th	9am–12pm	Problem sheets from lectures 1–8
2	Wednesday	October	31st	9am–12pm	and assignment
3	Wednesday	November	7th	9am–12pm	
*	**Monday**	**November**	**12th**	**12 noon**	**Assignment submission deadline**
4	Wednesday	November	14th	9am–12pm	
5	Wednesday	November	21st	9am–12pm	Problem sheets since lecture 8
6	Wednesday	November	28th	9am–12pm	and your project
7	Wednesday	December	5th	9am–12pm	
8	Wednesday	December	12th	9am–12pm	
*	**Monday**	**December**	**17th**	**12 noon**	**Project submission deadline**

1.4 Course website

The course website is **Computational Physics (2018-19)** on **Blackboard Learn**: https://bb.imperial.ac.uk.

Lecture notes, course handouts, problems sheets & solutions and reference material for coding will be made available there. 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 Practical sessions

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

The standard platform for practical work is Python on the PCs in the computing suite. Use of your own laptop and programming environment is allowed. However, you do so at your own risk; there is no guarantee that any demonstrator will be familiar enough with your set up to help you out of technical problems. It is important that you (re)acquaint yourselves with the program development-environment (e.g. Python on PCs in the computing suite, or the IDE on your laptop) before the first session. In particular, note that the PCs in the computing suite have been upgraded to Windows 10 this year, so you may need to relearn a few minor things.

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). 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 learnt in years 1 & 2. If you wish to work in another language, such as C, C++, FORTRAN or Matlab, this will normally be acceptable but you must check yourself that the software is available.[1] One important condition is that, unless indicated otherwise in the project script, you must implement numerical methods yourself rather than using a solver (e.g. DE 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 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.[2] (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

[1]Before you choose Matlab, however, please seriously consider expanding your horizons to a 'real' language like Python/C/FORTRAN. Matlab will not really prove that much use to you in the future, whether in academia or industry.

[2]The last two make pretty ugly plots though, so avoid them if you feel you have a choice.

have been posted online by graduated students, and are sometimes passed directly on by previous 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 will be the scheduled Practical Sessions (see Table 1.2). At these sessions, we will both be available to answer questions, as will the course demonstrators listed in Table 1.3. Prior to the start of the practical sessions on the 24th of October, we will be available for questions after the lectures. We (and any of the demonstrators) are also happy to arrange to meet with you in private or in small groups; just email any of us to arrange a time.

If you would like to raise a question with us or one of the demonstrators outside these times, please use the forum feature on Blackboard, and send us an email notifying us that you have posted the question there. 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.

Table 1.3: Computational Physics Demonstrators and Assessors.

Name	email
Timur Avni	t.avni17@imperial.ac.uk
Jonathan Beesley	jonathan.beesley17@imperial.ac.uk
Bingshen Chen	bingsheng.chen14@imperial.ac.uk
Ravindra Desai	ravindra.desai@imperial.ac.uk
Tom Hodson	t.hodson18@imperial.ac.uk
Sarunas Jurgilas	s.jurgilas16@imperial.ac.uk
Alexander Kuhn-Regnier	alexander.kuhn-regnier14@imperial.ac.uk
Tom Leyshon	t.leyshon17@imperial.ac.uk
Phil Litchfield	p.litchfield@imperial.ac.uk
Vito Palladino	v.palladino@imperial.ac.uk
Sam Palmer	samuel.palmer12@imperial.ac.uk
Justin Rumbutis	j.rumbutis18@imperial.ac.uk
Hugh Sparks	hugh.sparks10@imperial.ac.uk
Carl Thomas	c.thomas18@imperial.ac.uk

1.9 Literature

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

- C. Gerald and P. Wheatley, *Applied Numerical Analysis*, International Edition, 7th edition, (Pearson, 2004), ISBN 0-321-19019-X.
- W. H. Press, B. P. Flannery, S. A. Teukolsky, and W.T Vetterling *Numerical Recipes in C++: The art of scientific computing* (Cambridge: CUP 2007, 3rd ed.). The full text of the

second edition is available on the internet at http://www.nr.com/ but the access is rather convoluted.

- J. D. Hoffman, *Numerical Methods for Engineers and Scientists*, 2nd ed., (Marcel Dekker, Inc., 2001), ISBN 0-8247-0443-6.

The following resources are also useful:

- 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 and E. van Sebille.

Chapter 2

Introduction to Numerical Calculations

Outline of Section

- Nature of errors
- Natural units
- Solving non-linear algebraic equations

2.1 Numerical Accuracy and Errors

Review of Taylor series

In this course, we will be approximating derivatives as truncated series. To do this we review some basic concepts of Taylor series.

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

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

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

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

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

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

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

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

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

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

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

where the differential operator $(\ldots)^i$ is expanded by the binomial expansion, e.g., $\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^2 = h^2\frac{\partial^2}{\partial x^2} + 2hk\frac{\partial^2}{\partial x \partial y} + k^2\frac{\partial^2}{\partial y^2}$, operates on the function and is then evaluated at (x, y). The remainder term is similar to (2.2), in that it involves $(n+1)^{\text{th}}$ order partial derivatives of f evaluated at the point (ξ, η), where $x \le \xi \le x + h$ and $y \le \eta \le y + k$. Equation (2.4) 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.

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

Truncation errors

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

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

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

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

then

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

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

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

where ξ lies somewhere between x and a.

Round-off errors

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 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 Mac OSX) 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 For this reason, it is advisable to use integer variables for loop counters rather than floating point variables.

A computer actually stores numbers in binary representation

$$\begin{array}{rcl} \text{binary representation} & & \text{decimal representation} \\ 0 & = & 0 \\ 1 & = & 1 \\ 10 & = & 2 \\ 11 & = & 3 \\ 100 & = & 4 \\ 101 & = & 5 \\ 110 & = & 6 \\ 111 & = & 7 \\ 1000 & = & 8 \\ \text{etc...} & & \end{array} \tag{2.7}$$

32 and 64 bits are normally used to store numbers in single and double precision respectively with the following scheme, e.g. for the number 1.2345×10^{13}

	sign	mantissa	exponent
	±	.12345	13
single	1-bit	23-bits	8-bits
double	1-bit	52-bits	11-bits

(2.8)

The exponent is stored as an 8-bit binary value ranging between -127 and +128 for single precision and an 11-bit binary value ranging from -1023 to +1024 for double precision. Thus the range of numbers allowed are roughly $10^{\pm 38}$ and $10^{\pm 308}$ for single and double precision respectively, as numbers are stored in base 2 (i.e. $2^{256/2} \sim 10^{38}$). Numbers smaller or greater than these will cause an **underflow** or **overflow** respectively. Overflows are replaced by the symbol `Inf` in some languages. The IEEE standard for handling, e.g., 0/0, $\sqrt{-1}$, etc. is to replace with `NaN`, i.e., Not a Number.

It's important that you know about 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. The best place to read more about that is in Golberg, 1992 available on Blackboard.

Initial condition errors

Even in the absence of any truncation error or round-off error, an error in defining the starting point of the calculation can put the calculation onto a different solution of the equation being solved. This new solution (e.g. $x(t)$ curve) may diverge from the intended solution, perhaps more & more quickly with time (or whatever the independent variable is). Chaotic systems are particularly sensitive with respect to initial conditions, with the dynamics of Earth's atmosphere being a prime example. (Just think of the uncertainty of weather forecasting, particularly in the UK!)

Propagating errors

A propagation error is akin to an initial condition error. It is the error that would be seen in successive steps of a calculation given an error present at the current step, *if the rest of the calculation were to be done exactly (i.e. without truncation or round-off errors).* The **inherited error** at the

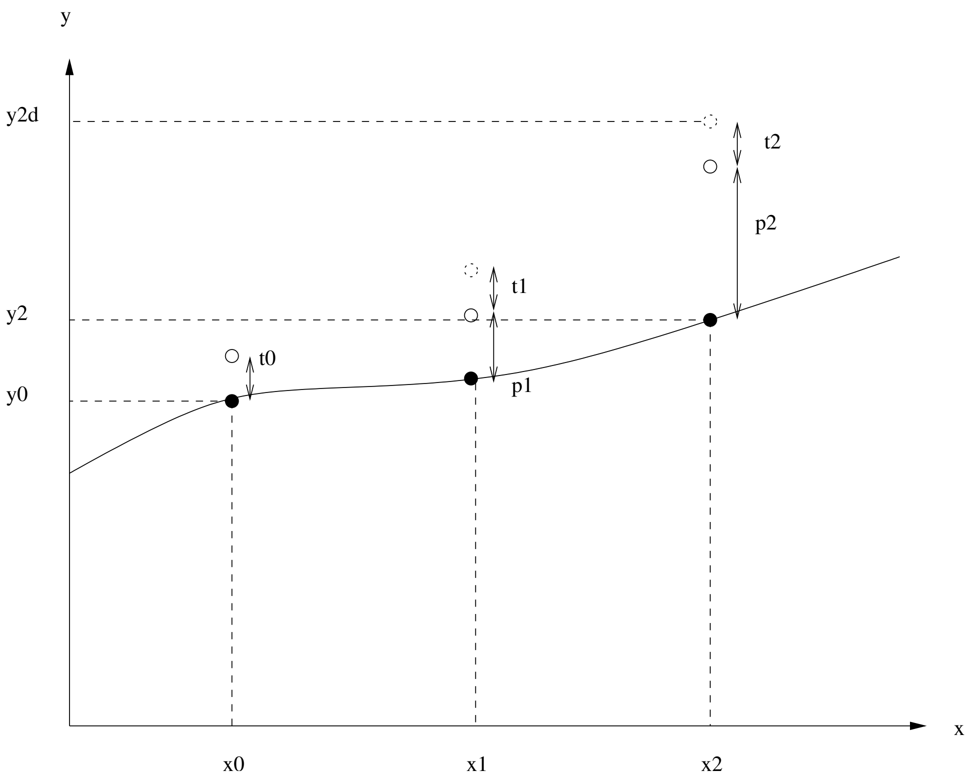

Figure 2.1: Illustration of truncation and propagating errors incurred in solving an ODE $dy/dx = f(y,x)$. The solid line depicts the exact solution. Dashed open circles depict the numerical solution. The calculation starts at (x_0, y_0) (off the plot) where there is no error. $\tilde{y}_3$ is the numerical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

current step is the accumulation of errors from all previous steps. Figure 2.1 illustrates propagating errors and truncation errors during each step of the numerical solution of an ordinary differential equation (ODE) $dy/dx = f(y,x)$. (Concrete examples of numerical schemes & ODEs will be given later.) The numerical scheme attempts to solve the ODE at discrete values of x; x_1, x_2, etc. You can see that the numerical solution (open circles) moves off the exact solution (solid line).

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

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

2.2 Natural units

It is very important to use the correct units when integrating systems numerically due to the limited range of numbers a computer can deal with. It also helps to understand the dynamics 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** are all interchangeable terms. 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

$$\frac{dy}{dt} + \alpha\, y \;=\; g \tag{2.9}$$

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

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

and the equation becomes

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

where $[G] = \mathrm{m}$. We could chose to go further and scale y too. Given that the solution settles down to $y = g/\alpha = G$, a good choice is $\hat{y} = y/G$ so that

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

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 don't have to worry about units in our integration;

- 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 $[g/\alpha]$.

2.3 Solving Non-Linear Algebraic Equations

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

$$f(x) = 0,$$

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

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

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 get the mid point

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

and determine whether the pair x_l and x_m or x_m and x_r now bracket the root. Update x_l or x_r to x_m accordingly and repeat until $\epsilon_i = x_r - x_l$ (where subscript i denotes the iteration number)

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

is sufficiently small or $\max[|f(x_l)|, |f(x_r)|]$ is sufficiently close to zero. (These are convergence criteria.) This method converges linearly, with the error ϵ_i (the uncertainty in where the root is) halving with each iteration;

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

Bisection will also find where discontinuous functions jump through zero.

Newton's method

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

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.} \tag{2.13}$$

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

$$\epsilon_{i+1} = \text{const} \times (\epsilon_i)^2. \tag{2.14}$$

Secant method

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

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}. \tag{2.15}$$

Inserting into Newton's method gives

$$\boxed{x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}.} \tag{2.16}$$

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 basically 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.

Chapter 3

Matrix Algebra

Outline of Section

- Linear Algebraic Equations
- Direct Methods: Gaussian and Gauss-Jordan elimination
- Direct Methods: LU Decomposition
- Iterative Methods
- Eigensystems

3.1 Introduction

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

$$\begin{aligned} a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 + \ldots + a_{1N}\,x_N &= b_1, \\ a_{21}\,x_1 + a_{22}\,x_2 + a_{23}\,x_3 + \ldots + a_{2N}\,x_N &= b_2\,, \\ \ldots &= \ldots, \\ a_{M1}\,x_1 + a_{M2}\,x_2 + a_{M3}\,x_3 + \ldots + a_{MN}\,x_N &= b_M, \end{aligned} \tag{3.1}$$

which can be written as the matrix operation

$$\mathbf{A} \cdot \vec{x} = \vec{b}, \tag{3.2}$$

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 (which can be zero) becomes the element of the matrix located at row i (i.e. the first subscript) and column j (the 2nd subscript). We want to solve for the unknown variables $\vec{x}$ for a given linear operator $\mathbf{A}$ and right-hand side $\vec{b}$. M is the number of equations in the system and N is the number of unknowns we have to solve for.

The need to solve a matrix equation such as (3.2) arises frequently in many numerical methods. In section 9.5 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 11) and minimisation of functions (section 7). Of course matrix equations arise directly in many physical problems too, such as quantum mechanics and classical mechanics.

The need to find eigenvalues of a matrix is also a common task both in numerical methods and in physics problems. We will encounter one such case in the stability analysis of finite difference methods for coupled ODEs in section 8.6.

3.2 General Considerations

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

Consider the inhomogeneous matrix equation $\mathbf{A} \cdot \vec{x} = \vec{b}$. If $M = N$ and all equations are **linearly independent** then there should exist a unique solution for $\vec{x}$. However if some equations are degenerate (i.e. can be written as linear combination of other equations) then effectively the system has fewer equations than unknowns (i.e., $M < N$) and there may not be a solution (or 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.

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

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B}, \tag{3.3}$$

where $\mathbf{X}$ and $\mathbf{B}$ are now $N \times N$ matrices too. Each column of $\mathbf{X}$ is one of the vectors $\vec{x}_i$ of the N equations (where $1 \leq i \leq N$). Similarly, each column of $\mathbf{B}$ is the right hand-side vector $\vec{b}_i$ of one of the equations.

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

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}, \tag{3.4}$$

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

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 only if $\det \mathbf{A} = 0$.

3.3 Gaussian and Gauss-Jordan elimination

Direct methods such as **Gaussian elimination** and **Gauss-Jordan elimination** involve manipulating the matrix to eliminate elements so that the system can be easily solved. These are basically what you would do if you needed to solve the matrix equation using pen and paper. The manipulation involves swapping rows and adding multiples of different rows to each other (or subtracting them from each other). The 'row-swapping' operation is known as 'pivoting', and is in general the only part of direct methods that can be tricky to code up.

For Gaussian elimination, one manipulates the augmented matrix formed by $\left(\mathbf{A}|\vec{b}\right)$, in an effort to end up with $\mathbf{A}$ in upper triangular form. For Gauss-Jordan elimination one manipulates the augmented matrix formed by $(\mathbf{A}|\mathbf{I})$, until one has $(\mathbf{I}|\mathbf{C})$, 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 / 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

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

$$\mathbf{A} \equiv \mathbf{L} \cdot \mathbf{U} \tag{3.5}$$

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

$$\begin{pmatrix} \alpha_{11} & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix} \text{ and } \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & \beta_{33} \end{pmatrix}, \tag{3.6}$$

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

If the above factorisation holds, we can write the system

$$\boxed{\mathbf{A}\cdot\vec{x} = \mathbf{L}\cdot\mathbf{U}\cdot\vec{x} = \mathbf{L}\cdot\vec{y} = \vec{b},} \tag{3.7}$$

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

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

$$y_1 = \frac{b_1}{\alpha_{11}} \tag{3.8}$$

$$y_i = \frac{1}{\alpha_{ii}}\left(b_i - \sum_{j=1}^{i-1}\alpha_{ij}y_j\right) \quad i = 2, 3, ..., N(\text{in this order}). \tag{3.9}$$

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

$$x_N = \frac{y_N}{\beta_{NN}} \tag{3.10}$$

$$x_i = \frac{1}{\beta_{ii}}\left(y_i - \sum_{j=i+1}^{N}\beta_{ij}x_j\right) \quad i = N-1, N-2, ..., 1, \tag{3.11}$$

where the calculation in (3.11) must be carried out in the order shown.

The following representation of the coupled equations may help in interpreting the forward and backward substitution expressions above:

$$\begin{aligned}
\alpha_{11}\,y_1 \quad &= b_1,\\
\alpha_{21}\,y_1 + \alpha_{22}\,y_2 \quad &= b_2,\\
\alpha_{31}\,y_1 + \alpha_{32}\,y_2 + \alpha_{33}\,y_3 \quad &= b_3,\\
\alpha_{41}\,y_1 + \alpha_{42}\,y_2 + \alpha_{43}\,y_3 + ... \quad &= b_4,\\
\alpha_{51}\,y_1 + \alpha_{52}\,y_2 + \alpha_{53}\,y_3 + ... \quad &= b_5,\\
... \quad &= ...,\\
\alpha_{N1}\,y_1 + \alpha_{N2}\,y_2 + \alpha_{N3}\,y_3 + ... + \alpha_{NN}\,y_N \quad &= b_N
\end{aligned}$$

$$\begin{aligned}
\beta_{11}\,x_1 + \beta_{12}\,x_2 + \beta_{13}\,x_3 + ... + \beta_{1N}\,x_N &= y_1,\\
\beta_{22}\,x_2 + \beta_{23}\,x_3 + ... + \beta_{2N}\,x_N &= y_2,\\
\beta_{33}\,x_3 + ... + \beta_{3N}\,x_N &= y_3,\\
... + \beta_{4N}\,x_N &= y_4,\\
... + \beta_{5N}\,x_N &= y_5,\\
... &= ...,\\
\beta_{NN}\,x_N &= y_N
\end{aligned}$$

Here we briefly outline how such a decomposition can be performed. Details can be found in the recommended reading material. We follow the treatment of *Numerical Recipes*, but some may prefer the highly pedagogical description in Gerald and Wheatley.

The matrix equation 3.5 represesents $N \times N$ individual equations, with $N^2 + N$ unknown variables (the α_{ij} and β_{ij} terms, where the diagonals are counted twice). This means that N of these variables can be specified arbitrarily—and Crout's algorithm for decomposition chooses to fix the diagonal α_{ii} values to 1. Following this, for each $j = 1, 2, 3, \ldots, N$, it is possible to solve for β_{ij}, $(i = 1, 2, \ldots, j)$:

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{3.12}$$

Then, one can solve for α_{ij} for $(i = j+1, j+2, \ldots, N)$:

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right). \tag{3.13}$$

The above procedure determines all the α and β 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 α and β values in-situ in the same matrix that A was held in.

As with many of these algorithms, stability is a critical requirement, and the division by β_{jj} is an operation that can introduce instability if the value of β_{jj} is small. Crout's altgorithm *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 β_{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 $\mathbf{A}$; 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 $\mathbf{A}$ once first.

If we want to find the inverse of $\mathbf{A}$ then we can decompose the matrix ($\mathcal{O}(N^3)$) once and solve equation (3.4) for each column of $\mathbf{A}^{-1}$ which requires $N \times \mathcal{O}(N^2)$ operations i.e. also $\mathcal{O}(N^3)$. Thus taking an inverse costs $\mathcal{O}(N^3)$ operations. This can become very slow even on the fastest computers when $N > \mathcal{O}(10^3)$.

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

$$\det \mathbf{A} = \prod_{j=1}^{N} \beta_{jj} \tag{3.14}$$

(quoted here without proof). However this will quickly lead to overflows so it is usually calculated as

$$\ln(\det \mathbf{A}) = \sum_{j=1}^{N} \ln \beta_{jj} \equiv tr(\mathbf{A}). \tag{3.15}$$

The rest of this chapter concentrates mainly on **iterative methods** for solving $\mathbf{A} \cdot \vec{x} = \vec{b}$ and finding eigenvalues. For large matrices, these turn out to be more suitable (= faster or with smaller memory footprints) than direct methods.

3.5 Iterative Solution – Jacobi Method

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

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \tag{3.16}$$

for all rows i 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 will be shown shortly. [a]

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

$$\begin{pmatrix} \cdot & \cdot & & & & & \\ \cdot & \cdot & \cdot & & & & \\ & \cdot & \cdot & \cdot & & & 0 \\ & & \cdot & \cdot & \cdot & & \\ & 0 & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \\ & & & & & \cdot & \cdot \quad \cdot \end{pmatrix} . \tag{3.17}$$

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

$$\boxed{\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}.} \tag{3.18}$$

Note that the $\mathbf{L}$ and $\mathbf{U}$ matrices here are nothing to do with those from LU decomposition! (For a start, we are adding rather than multiplying to compose $\mathbf{A}$.) We then have

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

giving

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

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

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

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

$$\boxed{\vec{x}_{n+1} = -\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U}) \cdot \vec{x}_n + \mathbf{D}^{-1} \cdot \vec{b}} \tag{3.19}$$

which we can use to find $\vec{x}$ starting from a guess $\vec{x}_0$. Equation (3.19) is the Jacobi method. In general this method will converge from any starting guess if all the eigenvalues of the update matrix $\mathbf{T} \equiv \mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U})$ satisfy $|\lambda^i| < 1$. [b]

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

We now need to introduce some new terminology: the **spectral radius** of $\mathbf{T}$. This must be less than unity. The spectral radius ρ of a matrix $\mathbf{M}$ is defined as the largest modulus of any of its eigenvalues, i.e., $\rho(\mathbf{M}) = \max\{|\lambda^i|\}$. The condition $\rho(\mathbf{T}) < 1$ is guaranteed for any matrix $\mathbf{A}$ that is strictly diagonally dominant. [c] Note that the inverse of $\mathbf{D}$ is easy to calculate since it is a diagonal matrix. For any diagonal matrix $\mathbf{D} = \text{diag}(d_1, d_2, \ldots d_N)$ the inverse is

$$\mathbf{D}^{-1} = \text{diag}\left(\frac{1}{d_1}, \frac{1}{d_2}, \ldots, \frac{1}{d_N}\right). \tag{3.20}$$

Proof of Jacobi convergence condition – The condition $|\lambda^i| < 1$ is simple to prove, using similar principles to those used previously in section 8.6 for the stability analysis of finite difference solution of a set of coupled linear ODEs. As before, we consider the errors at each iteration

$$\vec{\epsilon}_n = \vec{x}_n - \vec{x} \qquad \text{and} \qquad \vec{\epsilon}_{n+1} = \vec{x}_{n+1} - \vec{x}, \tag{3.21}$$

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

$$\begin{aligned} \vec{\epsilon}_{n+1} = \vec{x}_{n+1} - \vec{x} &= -\mathbf{T} \cdot \vec{x}_n + \mathbf{D}^{-1} \cdot \vec{b} - \left(-\mathbf{T} \cdot \vec{x} + \mathbf{D}^{-1} \cdot \vec{b}\right) \\ &= -\mathbf{T} \cdot \vec{x}_n + \mathbf{T} \cdot \vec{x} \\ &= -\mathbf{T} \cdot (\vec{x}_n - \vec{x}), \\ \therefore \qquad \vec{\epsilon}_{n+1} &= -\mathbf{T} \cdot \vec{\epsilon}_n. \end{aligned} \tag{3.22}$$

If $\mathbf{T}$ is a diagonalisable matrix so that it can be factorised using the eigen-decomposition

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

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

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

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

$$\left| \frac{\zeta^i_{n+1}}{\zeta^i_n} \right| \equiv \left|\lambda^i\right| < 1. \tag{3.25}$$

(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 $\mathbf{T}$ (or anything else) 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 a matrix–vector multiplication (i.e. $\mathbf{T} \cdot \vec{x}_n$) 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., $\mathbf{T} \cdot \vec{x}_n$ takes $\mathcal{O}(N)$ operations. For a full matrix, $k = 2$. As long as N_{iter} is substantially less than N, iterative solution scales much better than $\mathcal{O}(N^3)$ needed for direct inversion methods, even for a full matrix. (It should be noted that LU decomposition can be carried out faster then $\mathcal{O}(N^3)$ 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 in the norm of the solution vector

$$\varepsilon = \left| \frac{\|\vec{x}_{n+1}\| - \|\vec{x}_n\|}{\|\vec{x}_n\|} \right| \tag{3.26}$$

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

$$\|\vec{x}\|_\ell \equiv \left(\sum_{i=1}^{n} |x_i|^\ell \right)^{\frac{1}{\ell}} . \tag{3.27}$$

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.

3.6 Iterative Gauss-Seidel Method

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

$$\boxed{\vec{x}_{n+1} = -(\mathbf{L}+\mathbf{D})^{-1} \cdot \mathbf{U} \cdot \vec{x}_n + (\mathbf{L}+\mathbf{D})^{-1} \cdot \vec{b},} \tag{3.28}$$

where the update matrix is now $\mathbf{T} = (\mathbf{L}+\mathbf{D})^{-1} \cdot \mathbf{U}$. This algorithm is derived similarly to the Jacobi method, except that $\mathbf{U} \cdot \vec{x}$ (rather than $(\mathbf{L}+\mathbf{U}) \cdot \vec{x}$) 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., $\rho(\mathbf{T}_{GS}) < \rho(\mathbf{T}_J)$.

At first glance it looks like we have to calculate $(\mathbf{L}+\mathbf{D})^{-1}$, so that Gauss-Seidel (G-S) would seem to be more computationally costly than Jacobi. However it turns out that the G-S iteration can be carried out without this inverse if we use **forward substitution**, thanks to the shape of the matrix $\mathbf{L}$. We rewrite (3.28) as

$$\mathbf{D} \cdot \vec{x}_{n+1} = -\mathbf{L} \cdot \vec{x}_{n+1} - \mathbf{U} \cdot \vec{x}_n + \vec{b}.$$

We then solve for each component $x_k^{(n+1)}$ of $\vec{x}_{n+1}$ in turn, starting with $k = 1$ and sequentially increasing until $k = N$. This is the same approach used to solve $\mathbf{L} \cdot \vec{y} = \vec{b}$ when using LU decomposition. (See equation (3.9).) In this form G-S looks like

$$x_k^{(n+1)} = \frac{1}{D_{kk}} \left(-\sum_{j=1}^{k-1} L_{kj} x_j^{(n+1)} - \sum_{j=k+1}^{N} U_{kj} x_j^{(n)} + b_k \right), \quad k = 1, 2, \dots, N. \tag{3.29}$$

The components of $\vec{x}_{n+1}$ must be calculated in the order shown.

Note that the result for $x_{k-1}^{(n+1)}$ is used when calculating $x_k^{(n+1)}$; allowing for more up-to-date information to be used for each calculation, rather than only using the nth-step result for step $n+1$.

3.7 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. The Gauss-Seidel relationship (equation 3.29) can be rewritten as follows:

$$x_k^{(n+1)} = x_k^n + \frac{\omega}{D_{kk}} \left(-\sum_{j=1}^{k-1} L_{kj} x_j^{(n+1)} - \sum_{j=k+1}^{N} U_{kj} x_j^{(n)} + b_k - D_{kk} x_k^n \right), \quad k = 1, 2, \dots, N, \tag{3.30}$$

if the newly-introduced parameter ω is set to 1.

The above expression is equivalent to

$$\boxed{\vec{x}_{n+1} = (\omega \mathbf{L} + \mathbf{D})^{-1} \cdot \left(-[\omega \mathbf{U} + (\omega - 1)\mathbf{D}] \cdot \vec{x}_n + \omega \vec{b} \right).} \tag{3.31}$$

This 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. $\omega < 1$ corresponds to under-relaxation, which is not beneficial to speed of convergence.

The above SOR iteration can be implemented using forward substitution in a similar way to Gauss-Seidel, avoiding the need to explicitly compute $(\omega \mathbf{L} + \mathbf{D})^{-1}$.

There are even faster converging iterative matrix solvers than SOR, which are based on iterative optimisation methods—which we describe in the next section.

3.8 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

$$\mathbf{A} \cdot \vec{x} = \lambda \, \vec{x}, \tag{3.32}$$

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 [d] with associated linearly independent eigenvectors $(\vec{e}_i)$

$$\mathbf{A} \cdot \vec{e}_i = \lambda_i \, \vec{e}_i \,. \tag{3.33}$$

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

$$\vec{v} = \sum_{i=1}^{N} c_i \, \vec{e}_i \,. \tag{3.34}$$

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

$$\begin{aligned} \mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\mathbf{A}\cdot\vec{e}_i = \sum_{i=1}^{N} c_i\,\lambda_i\,\vec{e}_i, \\ \mathbf{A}\cdot\mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^2\,\vec{e}_i, \\ \mathbf{A}^n\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^n\,\vec{e}_i, \\ \therefore \quad \lim_{n\to\infty} \quad \mathbf{A}^n\cdot\vec{v} &\to c_j\,\lambda_j^n\,\vec{e}_j, \end{aligned} \tag{3.35}$$

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

$$\boxed{\vec{\omega}_j = \frac{\mathbf{A}^n\cdot\vec{v}}{|\mathbf{A}^n\cdot\vec{v}|} \equiv \frac{\vec{e}_j}{|\vec{e}_j|},} \tag{3.36}$$

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

$$\boxed{\vec{\omega}_j^{\,T}\cdot\mathbf{A}\cdot\vec{\omega}_j = \vec{\omega}_j^{\,T}\cdot(\lambda_j\vec{\omega}_j) = \lambda_j.} \tag{3.37}$$

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 (3.36) and (3.37). This works because, the inverse of a matrix has the same eigenvectors but with the inverse eigenvalues. This can be shown since

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

thus

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

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

3.9 Other Eigenvalues

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

$$\mathbf{A}' \equiv \mathbf{A} - \alpha\,\mathbf{I}, \tag{3.38}$$

such that

$$\mathbf{A}'\cdot\vec{e}_i = \mathbf{A}\cdot\vec{e}_i - \alpha\,\mathbf{I}\cdot\vec{e}_i = \lambda_i\,\vec{e}_i - \alpha\,\vec{e}_i = (\lambda_i - \alpha)\,\vec{e}_i\,. \tag{3.39}$$

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

A crucial point is then to chose a suitable value for α. 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

$$|\lambda_i - a_{ii}| \leq \sum_{j\neq i} |a_{ij}| \,, \tag{3.40}$$

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}. [e] 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[f]

$$\mathbf{A} \equiv \begin{pmatrix} 1 & 0.1 & 0 \\ 0.1 & 5 & 0.2 \\ 0.1 & 0.3 & 10 \end{pmatrix} . \tag{3.41}$$

We then have that $0.9 \leq \lambda_1 \leq 1.1$, $4.7 \leq \lambda_2 \leq 5.3$, or $9.6 \leq \lambda_3 \leq 10.4$. 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 3.5, 3.6, 3.7), and thus determining the suitability of a matrix $\mathbf{A}$ for solution by, e.g., Jacobi iteration.

[a] 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.

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

[c] If $\mathbf{A}$ 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 $\mathbf{A}$.

[d] Note the change in notation here for eigenvalues! λ_i is now being used in place of λ^i, to accommodate the need to raise eigenvalues to powers.

[e] The role of 'i' in equation (3.40) 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).

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

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 purposes.

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 needed, and 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 $0 \le i \le n$ 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. For example, for the $n = 2$ case

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

Note that 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 boring—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

$$C(x) \equiv \tfrac{1}{6}(A(x)^3 - A(x))(x_{i+1} - x_i)^2 \tag{4.8}$$

$$D(x) \equiv \tfrac{1}{6}(B(x)^3 - B(x))(x_{i+1} - x_i)^2 \tag{4.9}$$

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

Fourier Transforms

Outline of Section

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

In the previous chapter, 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.

5.1 Fourier Transforms—Recap

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

$$\begin{aligned} \tilde{f}(\omega) &= \int_{-\infty}^{+\infty} e^{i\omega t} f(t)\,\mathrm{d}t = \mathcal{F}(f(t)), && (5.1) \\ \text{and} \qquad f(t) &= \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-i\omega t}\tilde{f}(\omega)\,\mathrm{d}\omega = \mathcal{F}^{-1}(\tilde{f}(\omega)), && (5.2) \end{aligned}$$

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')

$$\tilde{f}(-\omega) = \tilde{f}^*(\omega), \tag{5.3}$$

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

$$f(t) \rightleftharpoons \tilde{f}(\omega). \tag{5.4}$$

$\mathcal{F}^{-1}(\tilde{f}(\omega))$ 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

$$\vec{x} \equiv (x, y, z) \qquad \leftrightarrow \qquad \vec{k} \equiv (k_x, k_y, k_z)\,, \tag{5.5}$$

and the FTs are modified accordingly:

$$\begin{aligned} \tilde{f}(\vec{k}) &= \int_{-\infty}^{+\infty} e^{i\vec{k}\cdot\vec{x}} f(\vec{x})d\vec{x} = \mathcal{F}(f(\vec{x})), && (5.6) \\ f(\vec{x}) &= \frac{1}{(2\pi)^3}\int_{-\infty}^{+\infty} e^{-i\vec{k}\cdot\vec{x}}\tilde{f}(\vec{k})d\vec{k} = \mathcal{F}^{-1}(\tilde{f}(\vec{k})). && (5.7) \end{aligned}$$

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

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

Taking the FT of this equation yields

$$-ik\tilde{u}(k) = \tilde{v}(k). \tag{5.8}$$

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

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-ikx}\tilde{u}(k)\,\mathrm{d}k\right) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-ikx}\tilde{v}(k)\,\mathrm{d}k\,. \tag{5.9}$$

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

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

Equating the integrands on both sides we have (5.8).

This can be generalised to higher derivatives:

$$\frac{d^n}{dx^n}f(x) \rightleftharpoons (-ik)^n\tilde{f}(k). \tag{5.10}$$

It can be generalised to 3D too:

$$\nabla f(\vec{x}) \rightleftharpoons -i\vec{k}\tilde{f}(\vec{k}) \quad , \qquad \nabla^2 f(\vec{x}) \rightleftharpoons -|\vec{k}|^2\tilde{f}(\vec{k}) \quad ,$$
$$\nabla\cdot\vec{E}(\vec{x}) \rightleftharpoons -i\vec{k}\cdot\tilde{\vec{E}}(\vec{k}) \quad , \qquad \nabla\times\vec{E}(\vec{x}) \rightleftharpoons -i\vec{k}\times\tilde{\vec{E}}(\vec{k}), \tag{5.11}$$

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

5.2 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 5.1 ;

$$f_n \equiv f(t_n) \qquad \text{with} \qquad n = 0, 1, 2, ..., N-1 \qquad \text{and} \qquad t_n = n\Delta t. \tag{5.12}$$

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.

Frequency domain— Figure 5.2 illustrates the frequency domain and discrete spectrum of the signal sampled in figure 5.1. 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**

$$\boxed{\omega_{max} = \frac{\pi}{\Delta t} = \Delta\omega\frac{N}{2}.} \tag{5.13}$$

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

$$\boxed{\tilde{f}_p \approx \frac{1}{\Delta t}\tilde{f}(\omega_p) \quad \text{with} \quad p = -\frac{N}{2}, \ldots, 0, \ldots, \frac{N}{2} \quad \text{and} \quad \omega_p = p\Delta\omega = p\frac{2\pi}{N\Delta t}.} \tag{5.14}$$

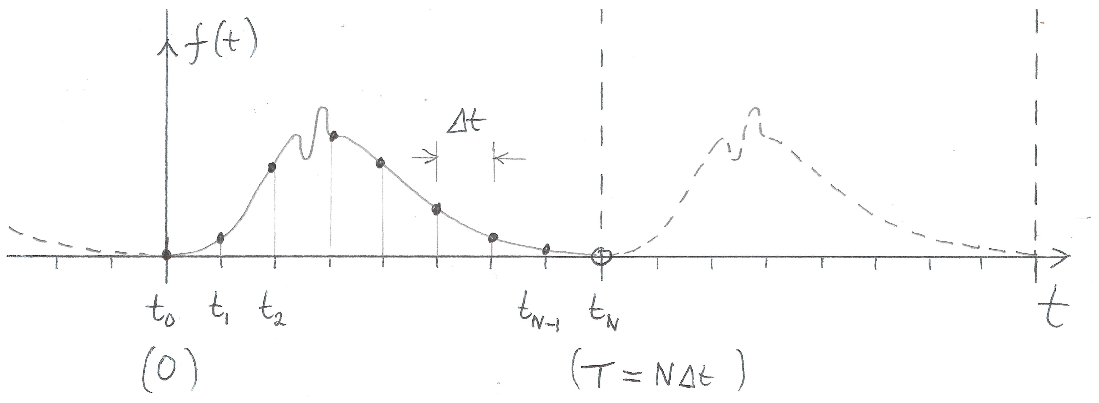

Figure 5.1: Illustration of a signal in the time domain. $N = 8$ here. The signal/function is assumed to be periodically extended beyond $0 \leq t \leq T$.

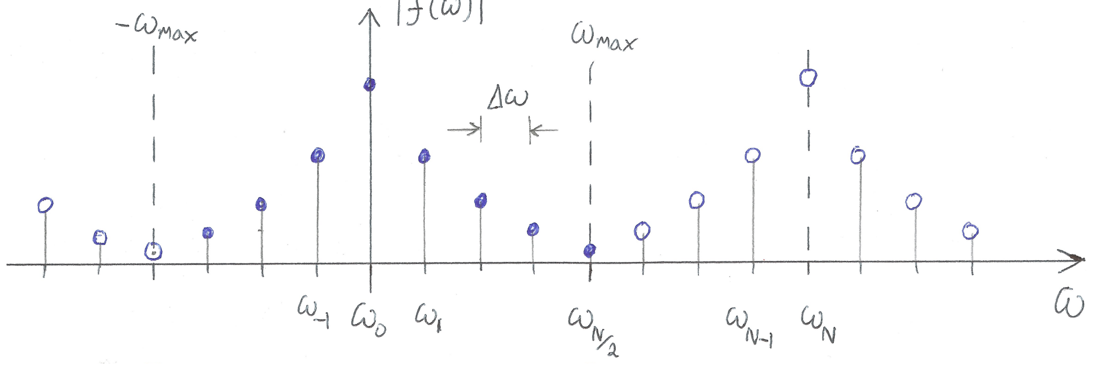

Figure 5.2: Frequency domain and discrete spectrum corresponding to figure 5.1. The discrete spectrum is also periodically extended (beyond $|\omega_{max}|$ in this case). The modulus of $\tilde{f}(\omega)$ is depicted.

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 5.3.)

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 (5.14) 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 5.3) 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 $\int \ldots dt$ replaced by a finite sum $\sum \ldots \Delta t$:

$$\boxed{\tilde{f}_p = \sum_{n=0}^{N-1} f_n e^{i\omega_p t_n} = \sum_{n=0}^{N-1} f_n e^{i2\pi pn/N},} \tag{5.15}$$

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

$$f_n = \frac{1}{N}\sum_{p=-N/2+1}^{N/2} \tilde{f}_p e^{-i2\pi pn/N}. \tag{5.16}$$

The different normalisation factor of $1/N$ accounts for the discrete sampling. This comes from $dt \to \Delta t$ and $d\omega \to \Delta\omega = 2\pi/(N\Delta t)$. (Note that in going from the FT (5.1) to the DFT (5.15) a factor of Δt has been absorbed into $\tilde{f}_p$, i.e., $\tilde{f}_p \approx \tilde{f}(\omega_p)/\Delta t$.) The backward DFT (5.16) is usually defined as

$$\boxed{f_n = \frac{1}{N}\sum_{p=0}^{N-1} \tilde{f}_p e^{-i2\pi pn/N},} \tag{5.17}$$

which is more symmetrical with the forward DFT. Why this is possible will be seen in section 5.3.

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

$$c_k = \frac{1}{T}\int_0^T f(t)e^{i\omega_k t}\,\mathrm{d}t\,, \tag{5.18}$$

$$f(t) = \sum_{k=-\infty}^{+\infty} c_k e^{-i\omega_k t}. \tag{5.19}$$

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 (5.15) and (5.16), are easily implemented on a computer, however the naive method of implementing it requires $\mathcal{O}(N^2)$ operations. With typical samples volumes of $N \sim 10^6$ in modern applications this becomes prohibitive even on the fastest computers. The DFT can actually be done in **$\mathcal{O}(N\log_2 N)$ 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

$$\tilde{f}_{p,q} = \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} f_{n,m}e^{i2\pi pn/N}e^{i2\pi qm/M}. \tag{5.20}$$

The backward DFT is

$$f_{n,m} = \frac{1}{NM}\sum_{p=0}^{N-1}\sum_{q=0}^{M-1} \tilde{f}_{p,q}e^{-i2\pi pn/N}e^{-i2\pi qm/M}. \tag{5.21}$$

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.

5.3 Sampling & Aliasing

The discrete sampling of the function to be FT'd, given by equation (5.12), introduces some sampling effects and care has to be taken in allowing for them. There is a minimum sampling rate that needs to be used so that we 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.

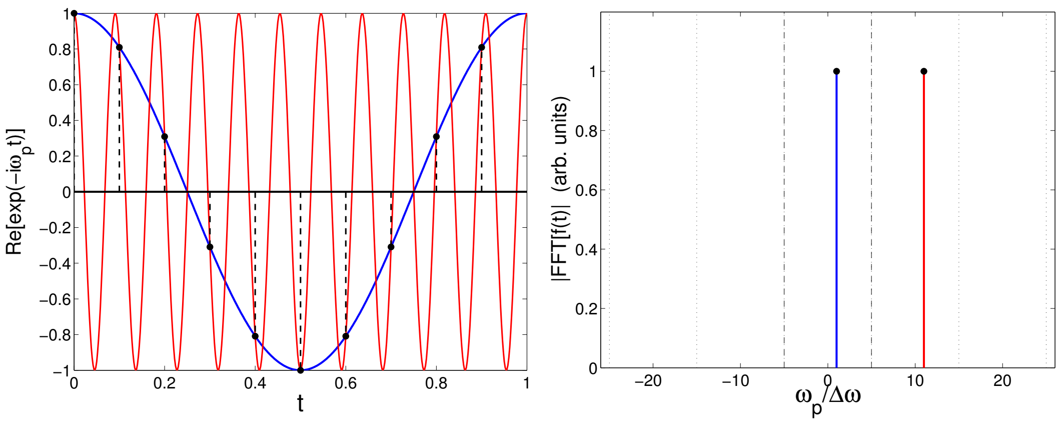

Figure 5.3: (Left) Indistinguishable harmonic waves below and above the Nyquist frequency, for $N = 10$. Blue line: $\omega = \omega_1 = \Delta\omega$. Red line: $\omega = \omega_{N+1} = \Delta\omega + 2\omega_{max}$. (Waves are periodically extended beyond range shown.) (Right) Corresponding (angular) frequency spectrum. The vertical dashed lines lie at $\pm$ the Nyquist frequency.

Aliasing & indistinguishable frequencies – For any wave with a frequency ω_p lying in the frequency domain captured by the DFT, there are higher frequency waves with $\omega_{p'} = \omega_p + m\Omega$ (where $m \in \mathbb{Z}$ and $\Omega = 2\omega_{max}$ is the width of the frequency domain) that look exactly the same to the discrete time-sampling grid. This is illustrated in figure 5.3. This can be understood by considering the kernel of the DFT, i.e., $\exp\big(i\omega_{p'}t_n\big)$.

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

This has the following implications;

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

(b) It also explains why equation (5.17) for the backwards DFT 'works'. The "negative" frequency part of the spectrum $p = \left\{-\frac{N}{2} + 1, \ldots, -1\right\}$ maps to the positive spectrum at $p' = p + N = \left\{\frac{N}{2} + 1, \frac{N}{2} + 2, \ldots, N - 1\right\}$ lying beyond the Nyquist frequency. In other words, although the $\sum_{p=N/2+1}^{N-1}$ parts of (5.17) are above the Nyquist frequency, they happen to capture the desired negative frequencies within the Nyquist range.

(c) Aliasing:

$$\tilde{f}_p = \sum_m \mathcal{F}(f)|_{\omega_p + m\Omega}, \tag{5.22}$$

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 $\omega'_p = p + m\Omega$.

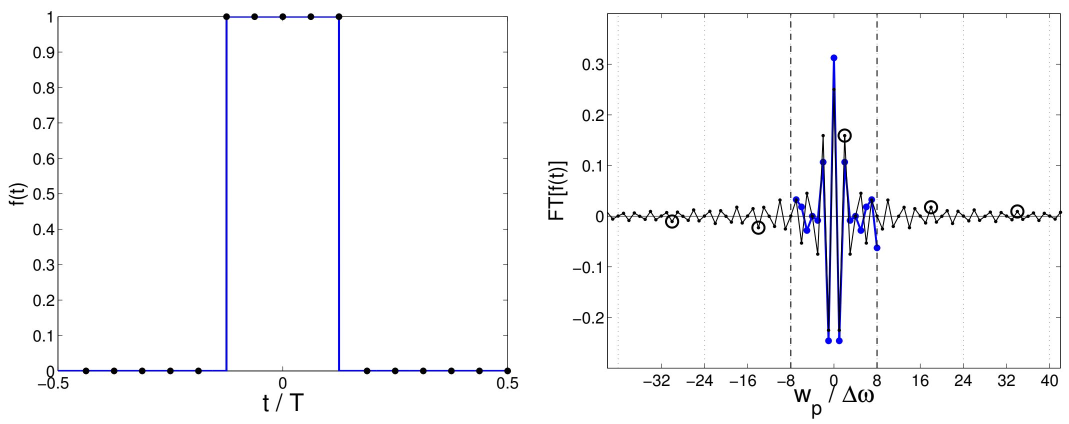

Figure 5.4: Aliasing: (Left) $f(t)$ in the time domain; centred "rect" function of width $T/4$. Sampled with $N = 16$. (Right) The DFT (blue, solid line + markers), and exact FT (black, thin solid line) in the frequency domain. The open circles denote Fourier components outside the range $-\omega_{max} \le \omega \le \omega_{max}$ that are aliased into that range during the DFT. The real part of the DFT & FT are shown. The DFT been divided by Δt.

Figure 5.4 shows the phenomenon of aliasing for a top-hat function

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

5.4 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 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 $0 \le p < N/2$:

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

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

5.4.1 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.

5.4.2 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:
        return f[0] # for a sample size of 1, the FT is the value itself
    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 array made up of farray_even[n]
                          + exp(i2pi n) * farray_odd[n] (for n = 0..N/2-1)
             and of farray_even[n]
                          - exp(i2pi n) * farray_odd[n] (for n = N/2..N)
```

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. When I was young, 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 5.23 to "e" and "o" yields:

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

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

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

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

$$\tilde{f}^{eooeeoeoeo...e}_p = f_n, \tag{5.23}$$

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:

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

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

$$N \times p = \frac{N \log N}{\log 2} \sim \mathcal{O}(N \log N), \tag{5.24}$$

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 6

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.

6.1 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 $0 \le x < 1$. 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**

$$\boxed{I_{n+1} = (a\, I_n + c)\%m,} \tag{6.1}$$

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

$$x_n = \frac{I_n}{m} = \frac{I_n}{\texttt{RAND\_MAX} + 1} \qquad \text{where} \qquad 0 \le x_n < 1. \tag{6.2}$$

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 $0 \le I < m$ 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.

6.2 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., $P(y) \ge 0$.

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

We are given x with PDF

$$U(x)\,dx = \begin{cases} dx & 0 \le x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

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

$$|P(y)\,dy| = |P(x)\,dx|, \tag{6.4}$$

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

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

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

$$\boxed{F(y) = \int_{-\infty}^{y} P(\tilde{y})\,d\tilde{y},} \tag{6.5}$$

and then using the fundamental transformation law of probabilities it follows that

$$F(y) = \int_{-\infty}^{y} \frac{d\tilde{x}}{d\tilde{y}}\,d\tilde{y} = \int_{0}^{x} d\tilde{x} = x. \tag{6.6}$$

(Note that $\sim$ is used to denote dummy integration variables here.) So if $F(y)$ is an invertible function, i.e.,

$$\boxed{y = F^{-1}(x)} \tag{6.7}$$

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 $0 \le F(y) < 1$. The transformation method is illustrated in fig. 6.1.

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 $P(y)$.

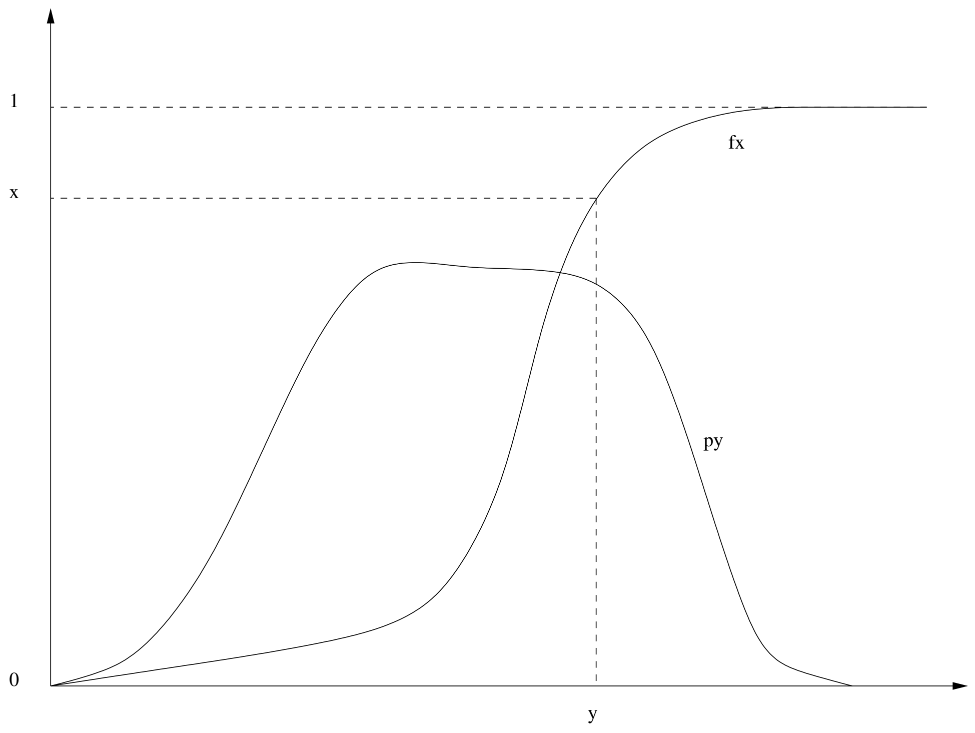

Figure 6.1: The transformation method. The Cumulative Distribution function $F(y) = \int_{-\infty}^{y} P(\tilde{y}) d\tilde{y}$ is inverted to find a deviate y for every uniform random deviate x.

6.3 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

$$f(y) = C, \tag{6.8}$$

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

$$\text{Prob} = \frac{dy}{y_{\text{max}} - y_{\text{min}}} \times \frac{P(y)}{C} = \text{const} \times P(y)\, dy\,. \tag{6.9}$$

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

$$\text{Eff} = \frac{1}{C}\frac{1}{y_{\text{max}} - y_{\text{min}}}. \tag{6.10}$$

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 (6.10) 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 $A_f = C \times (y_{\text{max}} - y_{\text{min}})$. 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.

6.4 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 7 where we also describe so-called *direct search* methods which do not involve random numbers.

6.5 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

$$I = \int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 \ldots f(\vec{x}) = \int_V f(\vec{x})\, d\vec{x}. \tag{6.11}$$

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

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

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

$$I \equiv V \langle f \rangle. \tag{6.12}$$

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

$$\hat{I} = \frac{V}{N} \sum_{i=1}^{N} f(\vec{x}_i). \tag{6.13}$$

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

$$\sigma_{f_i}^2 = \frac{1}{N-1} \sum_{i=1}^{N} \left(f(\vec{x}_i) - \langle f \rangle \right)^2. \tag{6.14}$$

Given this, the variance of the mean $\langle f \rangle$ is $\sigma_{\langle f \rangle}^2 = \sigma_{f_i}^2 / N$ and given (6.12), we can write the variance in the estimate of I as

$$\sigma_{\hat{I}}^2 = V^2 \sigma_{\langle f \rangle}^2 = \frac{V^2}{N} \sigma_{f_i}^2. \tag{6.15}$$

Therefore we can state the estimate of I (including its error) as

$$\boxed{\hat{I} \;=\; \frac{V}{N} \sum_{i=1}^{N} f(\vec{x}_i) \;\pm\; \frac{V}{\sqrt{N}} \sigma_{f_i}.} \tag{6.16}$$

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. 6.2, a uniform circle of radius 1/2;

$$f(x,y) = \begin{cases} 1 & \text{if } \; (x-\frac{1}{2})^2 + (y-\frac{1}{2})^2 < (\frac{1}{2})^2 \\ 0 & \text{otherwise} \end{cases}. \tag{6.17}$$

In this example, we have the luxury of knowing that

$$I = \int f(x,y)\, dx\, dy = \text{circle area} = \pi r^2 = \frac{1}{4}\pi. \tag{6.18}$$

To calculate I using MC integration, we sample N random, uniformly distributed points in the range $0 \le x < 1$ and $0 \le y < 1$. The value of the sampled integrand $f(x_i, y_i)$ will either be 1

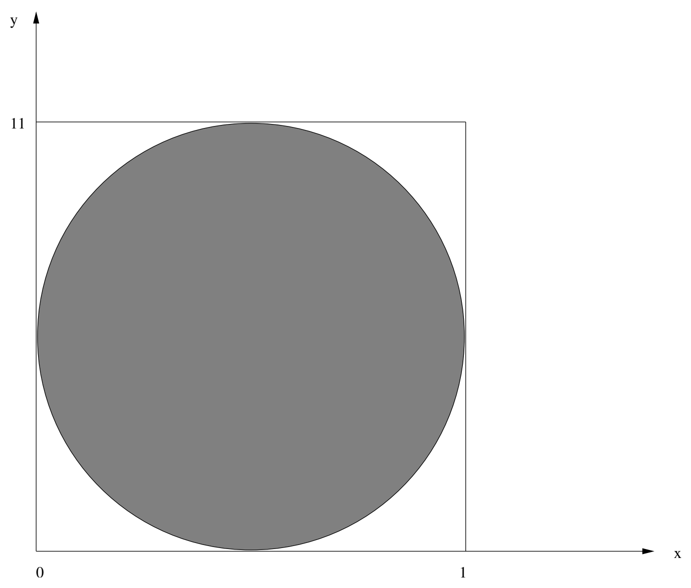

Figure 6.2: Circle of radius 1/2. The function $f(x, y) = 1$ inside the circle and $f(x, y) = 0$ outside

(*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

$$p \equiv \frac{I}{A}. \tag{6.19}$$

The successes should follow a binomial distribution. Let N_1 be the number of successes. We can define an estimator for p as

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

We know that for a binomial distribution the variance of N_1 is

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

such that

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

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

$$\boxed{\hat{I} = \hat{p}\, A = \hat{p} = \frac{N_1}{N} \pm \sqrt{\frac{p\,(1-p)}{N}}.} \tag{6.20}$$

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

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

$$h \sim N^{-1/d}, \tag{6.21}$$

such that the error in the trapezium method calculation of the integral is

$$\epsilon \sim \mathcal{O}(h^2) \sim \mathcal{O}(N^{-2/d}). \tag{6.22}$$

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

Error scalings $\longrightarrow$ (6.23)

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

The Metropolis Algorithm—For some integrals/summations the integrand is highly weighted to particular regions of $\vec{x}$. Then it is 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 an integral of the form

$$I = \int_V f(\vec{x})d\vec{x} = \int_V Q(\vec{x})P(\vec{x})d\vec{x} = \langle Q\rangle \tag{6.24}$$

where $P(\vec{x})$ behaves as (or is) a PDF (i.e. normalised and ≥ 0) the Metropolis algorithm can be used to pick samples that concentrate where the $P(\vec{x})$ 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

$$\langle Q\rangle = \frac{1}{Z}\int_{\vec{x}} Q(\vec{x}) \exp\left(-\frac{E(\vec{x})}{k_B T}\right) d\vec{x}, \tag{6.25}$$

where $Z = \int_{\vec{x}} \exp\left(-E(\vec{x})/k_B T\right) d\vec{x}$ is the partition function.

The Metropolis algorithm for calculating $\langle Q\rangle$ is

- Randomly pick a starting point $\vec{x}$.
- Repeatedly use the Metropolis algorithm to move to a new sample point.
  - Make a small trial step/change in the parameters to get $\vec{x}\,'$.
  - Calculate $P(\vec{x}\,')$. Accept or reject step using

$$p_{\text{acc}} = \begin{cases} 1 & \text{if} \quad P(\vec{x}\,') \geq P(\vec{x}) \\ P(\vec{x}\,')/P(\vec{x}) & \text{if} \quad P(\vec{x}\,') < P(\vec{x}) \end{cases} \tag{6.26}$$

    (Acceptance means $\vec{x}\,'$ becomes the next $\vec{x}$.)
  - 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 $\langle Q\rangle$, etc.

We are essentially doing

$$\langle \hat{Q} \rangle \;=\; \frac{1}{N}\sum_{i=1}^{N} Q(\vec{x}_i) \;\pm\; \frac{1}{\sqrt{N}}\sigma_{Q_i}, \tag{6.27}$$

as before (c.f. (6.16)), but with a different strategy for picking the samples. Instead of sampling even across the entire domain in which the integral is defined, as was done in equation 6.16, 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 their number 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).

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. 6.26 to:

$$p_{\text{acc}} = \begin{cases} 1 & \text{if} A(\vec{x}\,', \vec{x}) \geq 1 \\ A(\vec{x}\,', \vec{x}) & \text{if} \quad A(\vec{x}\,', \vec{x}) < 1 \end{cases} \tag{6.28}$$

where A is given by:

$$A(\vec{x}\,', \vec{x}) = \frac{P(\vec{x}\,')}{g(\vec{x}\,'|\vec{x})} \Big/ \frac{P(\vec{x})}{g(\vec{x}|\vec{x}\,')} \tag{6.29}$$

Chapter 7

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 2.3. 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 s 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 7.1. The model we would like to fit is a linear one

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

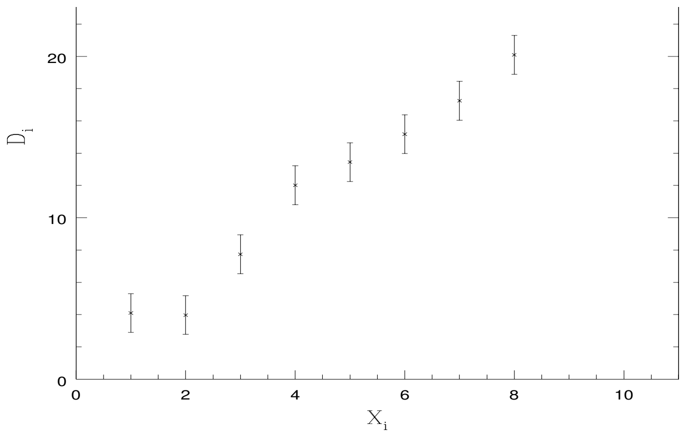

Figure 7.1: Some data with errors.

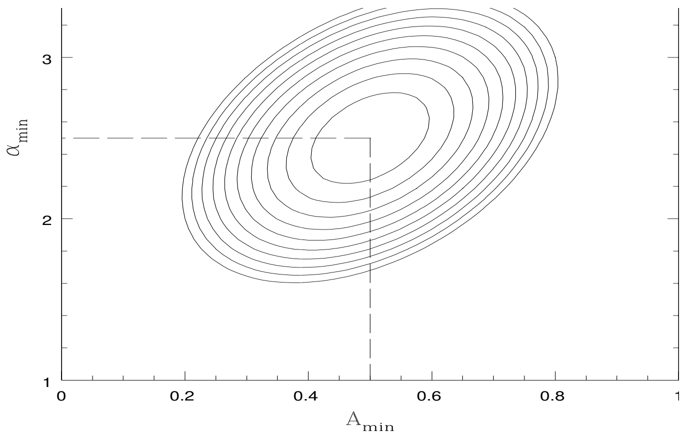

Figure 7.2: Contour plot of the χ^2 around the minimum. a_{min} and α_{min} are the maximum likelihood values for the parameters in the model. The curvature around the minimum is related to the error in the parameters.

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

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

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

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

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

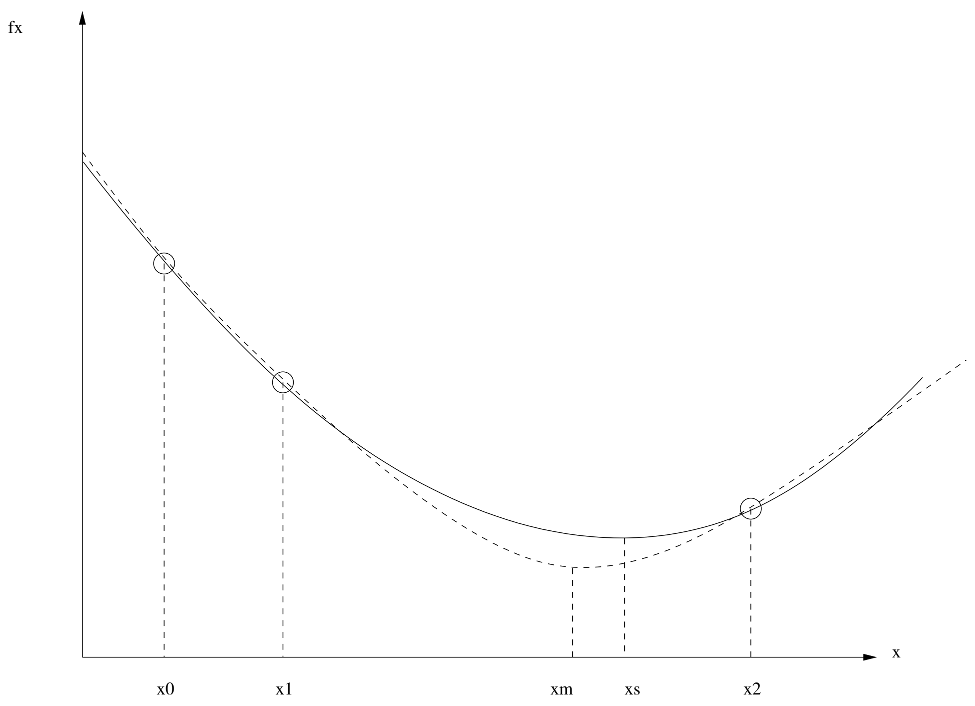

Figure 7.3: **The parabolic minimum search.** The function being minimised $f(x)$ is the dashed curve. A parabola (solid curve) is fit to the three points at x_0, x_1, x_2 and the minimum $x = x_3$ is found. Then the highest of the four points x_0, x_1, x_2, and x_3 is discarded and the method repeated.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

7.1 One–dimensional minimisation

Naively we might think that finding the minima of a function is trivial. All we need to do is differentiate the function and find the roots of the resulting equation. However there are many cases where the analytical method is not applicable, e.g., where the derivative in the function is discontinuous or the function has a regular (possibly infinite) set of minima, 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

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

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 (7.6) below; this value is x_3. This procedure is illustrated in fig. 7.3. We then keep the three lowest points out of $f(x_0)$, $f(x_1)$, $f(x_2)$, and $f(x_3)$ and repeat the procedure. At each successive iteration the point x_3 will converge towards x_* where the minimum of the function $f(x)$ is located. The iterations can be stopped once the change in x_3 is less than a desired value.

The quadratic interpolating the 3 points is given by

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

[c.f. equation (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

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

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

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

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.

7.2 Multi-Dimensional methods

Univariate method

For the case where the function is multi-dimensional, i.e., $f(\vec{x})$, we can extend the one-dimensional parabolic method. This can be done by searching for the minimum in each direction successively and iterating. However this is a very inefficient method which is not useful in most cases.

Univariate search is illustrated by the black arrows in figure 7.4. Contours are shown for a 2D function $f(x,y)$. The univariate search spirals into the minimum, converging slowly.

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

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

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.,

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

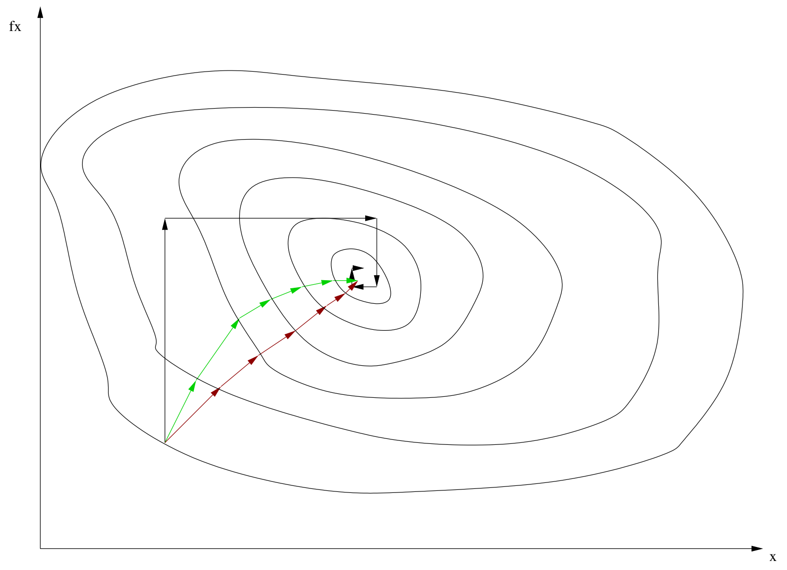

Figure 7.4: **Multi-dimensional minimisation** - for a function $f(x, y)$. Three methods are depicted: univariate method (black arrows), gradient method (green arrows) and Newton's method (red arrows). [Nb: this is not quite accurate; the green arrows are all supposed to be perpendicular to the contours!]

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

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

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

The gradient $\vec{\nabla} f$ can be found analytically or using a finite difference approximation; e.g. via a forward-difference scheme (to be discussed in detail in a later chapter) applied in each variable x_i for $i = 1, \dots, N$,

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

Newton's method

A more efficient method to find the minimum is achieved by including the local curvature at each step. Consider starting at a location $\vec{x}_0$. 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 $\vec{x}_0$.)

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

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

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

(c.f. equation 2.4 and the paragraph which precedes it in section 2.1.)

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:

$$f(x, y) = \frac{1}{2}\begin{pmatrix} x & y \end{pmatrix} \mathbf{H} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} a & b \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + C, \tag{7.12}$$

which is to say:

$$f(x,y) = \frac{1}{2}\left(\frac{\partial^2 f}{\partial x^2}x^2 + 2\frac{\partial f}{\partial x \partial y} + \frac{\partial^2 f}{\partial y^2}y^2\right) + ax + by + C.$$

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.,

$$\vec{\delta}^{\,T} \cdot \mathbf{H} \cdot \vec{\delta} > 0 \qquad \text{(for } \textit{all} \quad \vec{\delta} \neq \vec{0}\text{).} \tag{7.13}$$

To determine $\vec{\delta}$ we can use the Taylor expansion (7.10) 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 $\vec{\delta}$,

$$\boxed{\mathbf{H}(\vec{x}_0) \cdot \vec{\delta} = -\vec{\nabla} f(\vec{x}_0).} \tag{7.14}$$

The notation $\mathbf{H}(\vec{x}_0)$ means (7.11) evaluated at $\vec{x} = \vec{x}_0$. If $f(\vec{x})$ is exactly 'parabolic', i.e.,

$$f(\vec{x}) = \vec{x}^{\,T} \cdot \mathbf{A} \cdot \vec{x} + \vec{b}^{\,T} \cdot \vec{x} + c, \tag{7.15}$$

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 (7.14) will not take us directly to the minimum so we iterate this until we get arbitrarily close to it, i.e., $\vec{x}_{n+1} = \vec{x}_n + \vec{\delta}$ which can be written as

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - [\mathbf{H}(\vec{x}_n)]^{-1} \cdot \vec{\nabla} f(\vec{x}_n).} \tag{7.16}$$

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

Example (adapted from Gerald and Wheatley p. 424) – Consider the 2D parabolic function

$$f(x,y) = x^2 + 2y^2 + xy + 3x = (2x+y+3)(x+4y).$$

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

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} 2x+y+3 \\ x+4y \end{pmatrix} \quad , \qquad \mathbf{H} = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 4 \end{pmatrix} \quad .$$

It is easy to find the inverse of the Hessian matrix:

$$\mathbf{H}^{-1} = \begin{pmatrix} \frac{4}{7} & \frac{-1}{7} \\ \frac{-1}{7} & \frac{2}{7} \end{pmatrix} \tag{7.17}$$

.

Hence starting at $\vec{x} = \vec{x}_0 = (x_0, y_0)$, equation (7.14) 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.

<u>**Aside**</u> – What we have done in obtaining equation (7.14) 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 (7.14) 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

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - \alpha\, \mathbf{G}_n \cdot \vec{\nabla} f(\vec{x}_n),} \qquad (7.18)$$

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

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

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

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

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

we see that to linear order

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

(This is equivalent to equation (7.14) and the preceding paragraph when $\nabla f(\vec{x}_0 + \vec{\delta}) \neq 0$.) The trick is then to update $\mathbf{G}$ to satisfy

$$\mathbf{G}_n \cdot \vec{\gamma}_n = \vec{\delta}_n \, . \tag{7.19}$$

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

$$\mathbf{G}_{n+1} = \mathbf{G}_n + \frac{(\vec{\delta}_n \otimes \vec{\delta}_n)}{\vec{\gamma}_n \cdot \vec{\delta}_n} - \frac{\mathbf{G}_n \cdot (\vec{\delta}_n \otimes \vec{\delta}_n) \cdot \mathbf{G}_n}{\vec{\gamma}_n \cdot \mathbf{G}_n \cdot \vec{\gamma}_n}, \tag{7.20}$$

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.

7.3 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**

$$P(E)\, dE \sim \exp\left(-\frac{E}{k_B T}\right) dE, \tag{7.21}$$

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

$$\boxed{p_{\text{acc}} = \begin{cases} 1 & \text{if} \quad \Delta E \leq 0 \\ \exp(-\Delta E / k_B T) & \text{if} \quad \Delta E > 0 \end{cases}} \tag{7.22}$$

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.

7.4 Using minimisation to solve matrix equations

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

$$\boxed{f(\vec{x}) = \frac{1}{2}\vec{x}^T \cdot \mathbf{A} \cdot \vec{x} - \vec{b}^T \cdot \vec{x},} \tag{7.23}$$

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

$$\vec{\nabla} f(\vec{x}) = \mathbf{A} \cdot \vec{x} - \vec{b}. \tag{7.24}$$

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 (7.15) seen earlier.) So to solve the matrix equation, one of the multi-dimensional iterative methods seen in section 7.2 can be used with the quadratic form above.

The best iterative methods 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. 7.2, 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 (7.23);

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

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 8

Finite Difference Methods

Outline of Section

- Numerical differentiation
- Introduction to solving ODEs
- Consistency
- Accuracy
- Stability
- Convergence
- Efficiency

8.1 Numerical Differentiation

We are used to defining a derivative, e.g., dy/dx or $y'(x)$, as

$$\frac{dy(x)}{dx} \equiv \lim_{h\to 0} \frac{y(x+h)-y(x)}{h} \equiv \lim_{\Delta x\to 0} \frac{\Delta y}{\Delta x}\,. \tag{8.1}$$

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

$$\boxed{\tilde{y}'_f(x) \equiv \frac{y(x+h)-y(x)}{h}\,.} \tag{8.2}$$

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

Notation: in the remainder of this 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;

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

where we have defined the truncation error as

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

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

$$y'(x) = \frac{y(x+h)-y(x)}{h} - \frac{y''(\xi)}{2}\,h = \tilde{y}'_f(x) - \mathcal{O}(h)\,. \tag{8.3}$$

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)**

$$\boxed{\tilde{y}'_b(x) \equiv \frac{y(x) - y(x-h)}{h}\,,} \tag{8.4}$$

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,

$$\boxed{\tilde{y}'_c(x) \equiv \frac{\tilde{y}'_f(x) + \tilde{y}'_b(x)}{2} = \frac{y(x+h) - y(x-h)}{2h}\,.} \tag{8.5}$$

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

$$y(x+h) = y(x) + y'(x)\,h + \frac{1}{2}y''(x)\,h^2 + \frac{1}{3!}y'''(\xi)\,h^3 \tag{8.6}$$

and

$$y(x-h) = y(x) - y'(x)\,h + \frac{1}{2}y''(x)\,h^2 - \frac{1}{3!}y'''(\zeta)\,h^3 \tag{8.7}$$

then we can use this to define

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

which gives

$$y'(x) = \frac{y(x+h) - y(x-h)}{2h} - \mathcal{O}(h^2) \equiv \tilde{y}'_c(x) - \mathcal{O}(h^2), \tag{8.8}$$

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 8.1; 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

$$\boxed{\tilde{y}''(x) \equiv \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} = y''(x) + \mathcal{O}(h^2).} \tag{8.9}$$

This can be obtained by adding together the 4th order Taylor expansions for $y(x+h)$ and $y(x-h)$, i.e., equations (8.6) and (8.7) with one more term. Another way to get (8.9) 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.

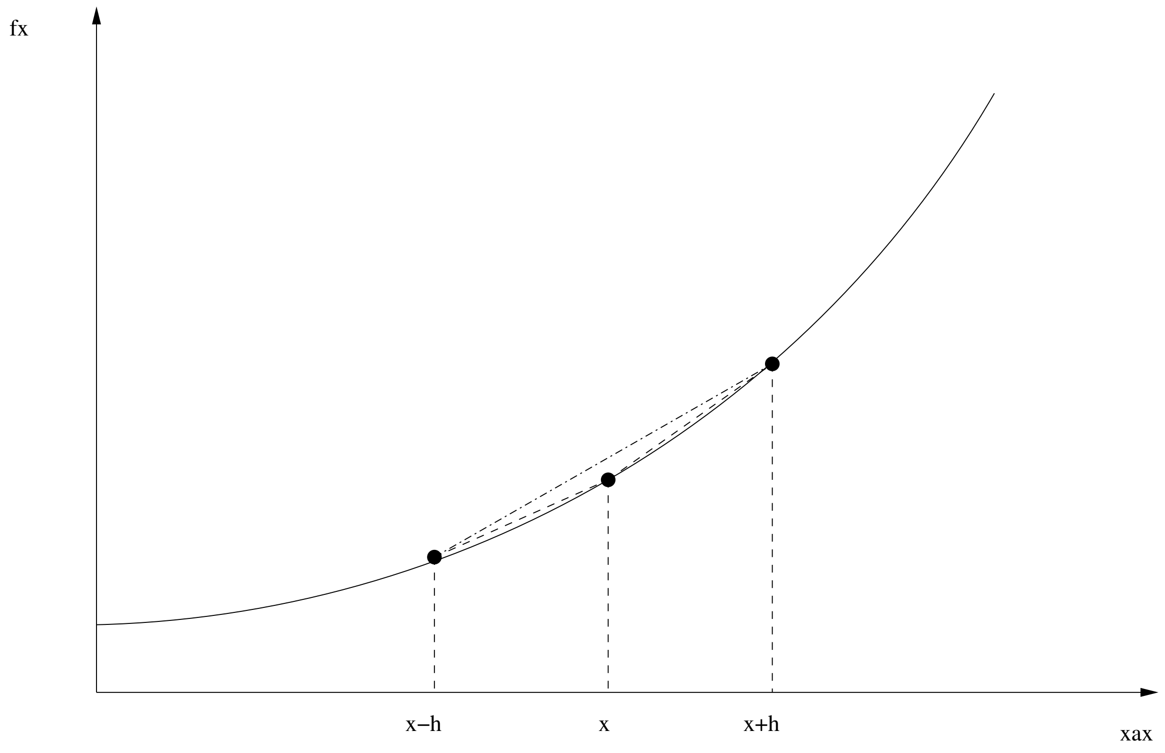

Figure 8.1: Forward, backward, and central difference schemes for approximating the derivative of the function $y(x)$ at the point x

8.2 Ordinary Differential Equations

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

$$\begin{aligned} \frac{du}{d\eta} &= f_u(\eta, \vec{y}) \\ \frac{dv}{d\eta} &= f_v(\eta, \vec{y}) \\ \frac{dw}{d\eta} &= f_w(\eta, \vec{y}) \\ &\vdots \end{aligned} \tag{8.10}$$

A compact notation for these coupled 1st order ODEs is

$$\frac{d\vec{y}}{d\eta} = \vec{f}(\eta, \vec{y}), \tag{8.11}$$

where $\vec{f}(\eta,\vec{y}) = \{\, f_u(\eta,\vec{y})\,,\; f_v(\eta,\vec{y})\,,\; f_w(\eta,\vec{y})\,,\; \ldots\}$, i.e., the 'components' of $\vec{f}$ are the functions in each ODE.

Often the independent variable η is time t, or denoted by the standard symbol for an independent variable: x. Often, we will talk about 'timesteps' and 'time-stepping' in solving systems of differential equations. This simply refers to choosing values of the independent variable (whether that independent variable actually physically corresponds to time or not).

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

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

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

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

then

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

giving

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

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

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

Euler method

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

$$\begin{aligned} y(t+h) &= y(t) + y'(t,y)\,h + \mathcal{O}(h^2) \\ &\approx y(t) + f(t,y)\,h\,. \end{aligned} \tag{8.12}$$

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

This Euler step can be iterated, stepping the solution forward h each time. In terms of the discretised limit of the function, and remembering that y is now an approximate solution (due to the truncation error incurred in dropping the $\mathcal{O}(h^2)$ terms), we can write this as

$$\tilde{y}_{n+1} = \tilde{y}_n + f(t_n, \tilde{y}_n)\,h, \tag{8.13}$$

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

$$t_n = t_0 + (n-1)\,h. \tag{8.14}$$

Notation: For brevity and convenience, the tilde ($\sim$) will normally be dropped when writing out such finite difference methods. (On occasion, when it is necessary to distinguish between actual/exact solutions and approximate solutions, $\sim$ symbols will be reinserted as appropriate.) Hence we chose to write the Euler method as

$$\boxed{y_{n+1} = y_n + f(t_n, y_n)\,h.} \tag{8.15}$$

Higher order systems

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

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

We can introduce three new variables

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

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

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

or in matrix notation

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

Generalising further, a pair of coupled m^{th}-order and n^{th}-order ODEs can be reduced to a set of $m+n$ 1st-order coupled ODEs. (This can be extended to sets of coupled arbitrary order ODEs.)

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

$$\frac{d\vec{y}}{d\eta} = \mathbf{L}\,\vec{y}, \tag{8.16}$$

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

$$\boxed{\vec{y}_{n+1} = (\mathbf{I} + h\mathbf{L})\,\vec{y}_n \equiv \mathbf{T}\,\vec{y}_n\,,} \tag{8.17}$$

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

A general, non-linear system is written as

$$\frac{d\vec{y}}{d\eta} = \vec{f}(\eta, \vec{y}) \qquad \text{or alternatively} \qquad \frac{d\vec{y}}{d\eta} = \mathcal{L}(\vec{y}), \tag{8.18}$$

where $\mathcal{L}$ is a **non-linear operator**. Non-linear systems cannot be expressed as linear matrix equations such as equation (8.16).

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

8.3 Consistency

This is a check that the finite difference method (e.g. Euler method) reduces to the correct differential equation (e.g. $dy/dt = f(t, y)$) in the limit of 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 x_n). The consistency analysis also tells us the order of the method. This is useful if one is given a finite difference equation, but is not told what its order of accuracy is.

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

$$y_{n+1} = y_n + f_n\,h, \tag{8.19}$$

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

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

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

$$y_n' = f_n - \frac{y_n''}{2} \, h - \frac{y_n'''}{3!} h^2 - \ldots . \tag{8.20}$$

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

$$\boxed{y' = f(x, y) - \frac{y''}{2} \, h - \frac{y'''}{3!} h^2 - \ldots} . \tag{8.21}$$

This is the **modified differential equation** (MDE). The discrete points solved by the Euler method (x_n, y_n) lie on the continuous curve that solves the MDE (given the same initial condition of course!). Letting $h \to 0$, equation (8.21) becomes

$$y' = f(x, y), \tag{8.22}$$

which is the intended ODE the Euler method set out to approximate. Thus we say that (8.19) ***is consistent*** with the ODE $y' = f(x, y)$. The ***lowest order in h of the correction terms*** in the MDE, e.g. $-y'' \, h/2 \sim \mathcal{O}(h)$ in this case, ***is the order of accuracy of the method***. This is the same as the order of the *global error*, discussed in the following section.

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

8.4 Accuracy

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

Using the Euler method as an example we know that the local truncation error is $\mathcal{O}(h^2)$. [See equation (8.12).] y_{n+1}, the **true** value of the function at x_{n+1}, is related to the numerical approximation of the function $\tilde{y}_{n+1}$ by

$$\tilde{y}_{n+1} \equiv y_n + f_n \, h = y_{n+1} + \mathcal{O}(h^2), \tag{8.23}$$

where we have assumed here that y_n is known exactly. Equation (8.23) shows us the error incurred in performing one step of the Euler method. However to integrate from x_0 to x_{end} would take $\mathcal{O}(1/h)$ steps. If we assume the local error at each of the steps simply adds up, in general, we will have

$$\boxed{\textbf{global error} \approx \text{local error} \times \text{number of steps}\,.} \tag{8.24}$$

In this case

$$\epsilon_{\text{end}} \approx \mathcal{O}(h^2) \times \mathcal{O}(1/h) \sim \mathcal{O}(h)\,. \tag{8.25}$$

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

You might think that all we need to do is reduce the step size h (increase the number of steps) arbitrarily to increase the accuracy arbitrarily. Unfortunately the round–off error places a limit on how accurate any method can be. To see this consider a round-off error $\mathcal{O}(\mu)$ added to every step in the integration, then the global error after $\mathcal{O}(1/h)$ steps will be

$$\epsilon_{\text{end}} \sim \frac{\mu}{h} + h\,. \tag{8.26}$$

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

$$h \sim \mu^{1/2} \,. \tag{8.27}$$

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

8.5 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

$$\tilde{y}_n = y_n + \epsilon_n, \tag{8.28}$$

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

$$\tilde{y}_{n+1} = y_{n+1} + \epsilon_{n+1}. \tag{8.29}$$

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

$$\boxed{g \equiv \left|\frac{\epsilon_{n+1}}{\epsilon_n}\right| \leq 1,} \tag{8.30}$$

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

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

$$\tilde{y}_{n+1} = \tilde{y}_n + f(x_n, \tilde{y}_n)\, h. \tag{8.31}$$

Substituting in the true values of the function we have

$$y_{n+1} + \epsilon_{n+1} = y_n + \epsilon_n + f(x_n, y_n + \epsilon_n)\, h. \tag{8.32}$$

Stability analysis can only be carried out for linear ODEs. Luckily for non-linear ODEs, i.e., when f is a non-linear function of y, we can still do a stability analysis by assuming that the errors are small $|\epsilon_n|$, $|\epsilon_{n+1}| \ll |y_n|$ and **linearising $\boldsymbol{f(x,y)}$**. To linearise, the function $f(x_n, y_n + \epsilon_n)$ is expanded around the point (x_n, y_n) in the variable y. To 1st-order in the expansion parameter ϵ_n the equation (8.32) becomes

$$\begin{aligned} y_{n+1} + \epsilon_{n+1} &= y_n + \epsilon_n + \left[f(x_n, y_n) + \left.\frac{\partial f}{\partial y}\right|_n \epsilon_n + \mathcal{O}(\epsilon_n^2) \right] h, \\ &= y_n + f(x_n, y_n)h + \epsilon_n \left[1 + \left.\frac{\partial f}{\partial y}\right|_n h \right] + \mathcal{O}(\epsilon_n^2) \,. \end{aligned} \tag{8.33}$$

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

$$y_{n+1} = y_n + f(x_n, y_n)h + \mathcal{O}(h^2), \tag{8.34}$$

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

$$\boxed{\epsilon_{n+1} \approx \epsilon_n \left(1 + \left.\frac{\partial f}{\partial y}\right|_n h \right) \,,} \tag{8.35}$$

for the Euler method. Inserting $\epsilon_{n+1}/\epsilon_n$ into equation (8.30), and dropping the $|_n$ notation, we can get a condition on the step size h for stability:

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

then assuming $h > 0$ the Euler method is only stable if

$$\boxed{\frac{\partial f}{\partial y} < 0} \quad \text{and} \quad \boxed{h \leq \frac{2}{|\partial f/\partial y|}.} \tag{8.36}$$

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

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

Note that this analysis does not tell us about the global error when the method is stable. Even when $g < 1$ there is still a global error that accumulates with each step. Global error (for a stable method) comes in through the $\mathcal{O}(h^2)$ term dropped in equation (8.34).

8.6 Stability Analysis of General (coupled) Linear Systems

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

$$\tilde{\vec{y}}_{n+1} = \mathbf{T}\, \tilde{\vec{y}}_n, \tag{8.37}$$

where $\mathbf{T}$ is the update matrix (e.g. $\mathbf{T} = (\mathbf{I} + \mathbf{L}\, h)$ for the Euler method). For a set of inhomogeneous ODEs $\tilde{\vec{y}}_{n+1} = \mathbf{T}\, \tilde{\vec{y}}_n + h\, \vec{q}(x_n)$ the term $h\, \vec{q}(x_n)$ does not influence numerical stability. Only the homogeneous part of the equation set, i.e., equation (8.37) need be analysed.

We can relate the true solution vector $\vec{y}_n = \{y_n^1, y_n^2, \ldots, y_n^m\}$ (where y^i are each of the dependent variables in the coupled ODE set) to the numerical approximation $\tilde{\vec{y}}_n$ in an analogous way to before;

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

where now $\vec{\epsilon}_n = \{\epsilon_n^1, \epsilon_n^2, \ldots, \epsilon_n^m\}$. Inserting this into the finite difference equation (8.37) yields

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

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

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

where $\vec{f}_n = \{f^1(\vec{y}_n), f^2(\vec{y}_n), \ldots, f^m(\vec{y}_n)\}$, i.e., its components are the functions in each ODE $dy^i/dx = f^i(x, \vec{y})$. This yields a matrix equation for the error propagation;

$$\vec{\epsilon}_{n+1} = \mathbf{T}\, \vec{\epsilon}_n. \tag{8.38}$$

The terms of $\mathcal{O}(h^2)$, discarded in obtaining at (8.38), 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 $\mathbf{T}$ must be less than or equal to unity

$$\boxed{\left|\lambda^i\right| \leq 1 \quad \text{for} \quad i = 1, \ldots, m} \tag{8.39}$$

For real eigenvalues the condition then translates to

$$\lambda^i \leq 1 \quad \text{and} \quad -\lambda^i \leq 1, \tag{8.40}$$

To show condition (8.39) 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

$$\boxed{\mathbf{T} = \mathbf{R}\,\mathbf{D}\,\mathbf{R}^{-1} \quad \text{with} \quad \mathbf{D} \equiv \text{diag}(\lambda^1, \lambda^2, ..., \lambda^m),} \tag{8.41}$$

where λ^i are the eigenvalues of $\mathbf{T}$ and $\mathbf{R}$ is the matrix whose columns are the eigenvectors of $\mathbf{T}$. Substituting (8.41) into (8.38) and operating on both sides with a further $\mathbf{R}^{-1}$ we obtain the diagonal (uncoupled) system

$$\vec{\zeta}_{n+1} = \mathbf{D}\,\vec{\zeta}_n, \tag{8.42}$$

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

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

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

$$g^i = \left|\frac{\zeta_{n+1}^i}{\zeta_n^i}\right| = |\lambda_n^i| \le 1, \tag{8.43}$$

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 (8.43) holds for the original equation (8.38) too.

Non-linear Systems

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

Considering the Euler method, we have a set of equations

$$\tilde{y}_{n+1}^i = \tilde{y}_n^i + f^i(x,\,\tilde{y}_n^1,\,\tilde{y}_n^2,\,\ldots\,,\,\tilde{y}_n^m)\,h. \tag{8.44}$$

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

$$f^i(x, y_n^1 + \epsilon_n^1,\, y_n^2 + \epsilon_n^2,\,\ldots\,,\, y_n^m + \epsilon_n^m) = f^i(x, \vec{y}_n) + \sum_{k=1}^m \left.\frac{\partial f^i}{\partial(y^k)}\right|_n \epsilon_n^k + \mathcal{O}((\epsilon_n)^2). \tag{8.45}$$

This is essentially what was done in section 8.5 with equation (8.33). Following the procedure used before we then get

$$\epsilon_{n+1}^i \approx \epsilon_n^i + h\sum_{k=1}^m \left.\frac{\partial f^i}{\partial(y^k)}\right|_n \epsilon_n^k \tag{8.46}$$

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

$$\boxed{T_{jk} = \delta_{jk} + h\,\frac{\partial f^j}{\partial(y^k)},} \tag{8.47}$$

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

8.7 Convergence

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

8.8 Efficiency

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

Chapter 9

ODEs: Initial Value Problems

Outline of Section

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

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

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

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

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

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

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

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

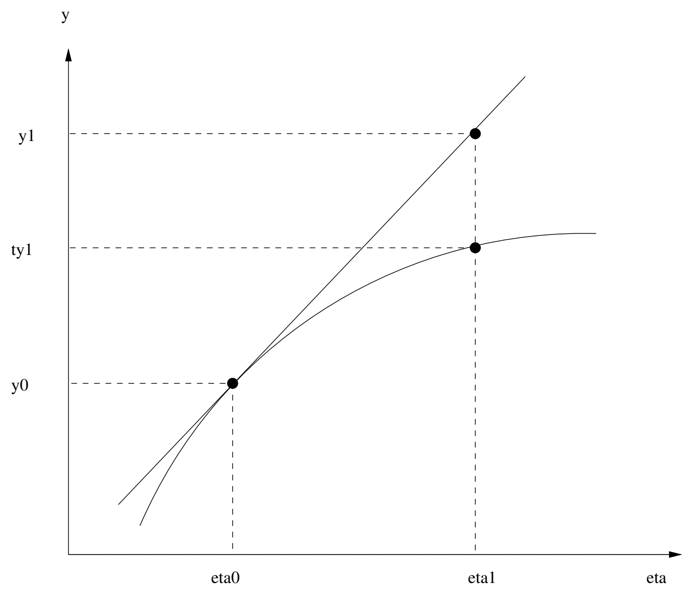

Figure 9.1: The Euler method applied to a non-linear function. The truncation error arises because the gradient is evaluated at the starting point and used to obtain the next point. Here, $\tilde{y}_1$ is the 'approximate' solution according to the Euler method and y_1 (on the curve) is the true solution to the ODE.

9.1 Predictor-Corrector Method

This is classified as an explicit, single-step method. It is a simple improvement to the Euler method. As shown in Fig. 9.1 the Euler method is inaccurate because it uses the gradient evaluated at the initial point to calculate the next point. This only gives a good estimate if the function is linear since the truncation error is quadratic in the step size.

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

$$y_{n+1} = y_n + \left(\frac{f_n + f_{n+1}}{2}\right) h, \tag{9.1}$$

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

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

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

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

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

to get

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

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

$$y_{n+1} = y_n + \frac{f_{n+1} + f_n}{2}\, h + \mathcal{O}(h^3), \tag{9.2}$$

So the local error (truncation) is 3rd order. Since the integration requires $\mathcal{O}(1/h)$ evaluations the **global error is 2nd order** hence Predictor-Corrector is a **2nd-order method**. Using (9.1) will give an approximate solution to the ODE since the remainder terms in the Taylor expansion of y_{n+1} and in using the forward difference approximation of f'_n have been discarded.

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

$$\boxed{\begin{aligned} &\text{step 1} \quad \text{(predict):} && y^\star_{n+1} = y_n + f_n\, h, \\ & && f^\star_{n+1} = f(x_{n+1}, y^\star_{n+1}), && (9.3a) \\ &\text{step 2} \quad \text{(correct):} && y^{\text{new}}_{n+1} = y_n + \frac{f^\star_{n+1} + f_n}{2}\, h. && (9.3b) \end{aligned}}$$

The above boxed equation is the algorithm for the predictor-corrector method, for a single 1st-order ODE.

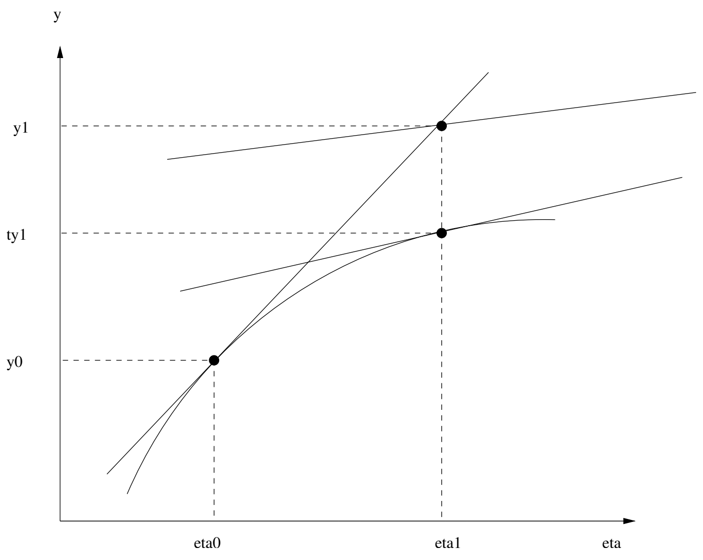

Figure 9.2: (Predictor-corrector) Ideally we would like to use the gradient at y_{n+1} to get the average gradient in the interval. However we do not have the true value y_{n+1} so we use the gradient calculated with the Euler estimate $\tilde{y}_{n+1}$. (Nb: $n = 0$ is used in the plot.)

The gradient calculated with the predicted point will not, in general, be exactly the correct value since it depends on both x and y but it will still be more accurate than the Euler method (see Fig. 9.2). The predictor-corrector method is $\mathcal{O}(h^2)$ accurate but involves two evaluations per step which is less efficient. It is conditionally stable for decaying solutions ($y' = -\alpha y$), but unstable for pure oscillating solutions ($y'' = -\omega^2 y$).

Coupled equations

For a set of m coupled 1st-order ODEs, step 1 should first be applied to each of the m ODEs to get the predicted value $(y^\star_{n+1})^i$ for each dependent variable y^i. Then $(f^\star_{n+1})^i = f^i(x_{n+1}, \vec{y}^\star_{n+1})$ can be evaluated for each ODE. Then step 2 (corrected Euler) can be applied to each ODE. In the compact vector notation, this is

$$\begin{aligned} &\text{step 1} \quad \text{(predict):} && \vec{y}^\star_{n+1} = \vec{y}_n + \vec{f}_n\, h, \\ & && \vec{f}^\star_{n+1} = \vec{f}(x_{n+1}, \vec{y}^\star_{n+1}), && (9.4a) \\ &\text{step 2} \quad \text{(correct):} && \vec{y}^{\,\text{new}}_{n+1} = \vec{y}_n + \frac{\vec{f}^\star_{n+1} + \vec{f}_n}{2}\, h. && (9.4b) \end{aligned}$$

9.2 Multi-Step (Leapfrog) Method

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

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

$$\boxed{y_{n+1} = y_{n-1} + 2 f_n\, h,} \tag{9.5}$$

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

General multi-step

This kind of multi-step method can use any number of previous values, say 'm'. The Leapfrog is an $m = 1$ order multi-step method. In general, an m^{th}-order multi-step method is an **$(m+1)^{\text{th}}$ order accurate method**.

One way to get an m^{th}-order multi-step method is to fit $P_{m+1}(x)$, an $(m+1)^{\text{th}}$-degree Lagrange polynomial, through the $m+2$ points $y_{n+1}, y_n, y_{n-1}, \ldots, y_{n-m}$ (i.e. including the point to be determined) and then approximate y' in the ODE by P'_{m+1} (for which an analytic expression can easily be obtained, once the coefficients of the interpolating polynomial have been found). Another viable way is to fit $P_m(x)$ to $f(x, y)$ through the $m+1$ points (x_n, f_n), $(x_{n-1}, f_{n-1}), \ldots, (x_{n-m}, f_{n-m})$ and then analytically integrate this interpolated $f(x, y)$ from x_n to x_{n+1} thus advancing the solution to y_{n+1}. This yields the 'Adam-Bashforth' family of methods.

However the drawback of multi-step methods is that $m + 1$ points are needed. So to apply a multi-step method from the outset, values before $x = x_0$ have to be guessed. If the guess is not a good one then the method can suffer from a initial value error.

Starting off

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

$$\begin{aligned}
y_1 &= y_0 + f_0\, h, \\
y_2 &= y_0 + 2 f_1\, h, \\
y_3 &= y_1 + 2 f_2\, h, \\
&\ldots \\
y_{n+1} &= y_{n-1} + 2 f_n\, h.
\end{aligned} \tag{9.6}$$

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

9.3 Runge-Kutta Methods

The Runge-Kutta methods are a family of explicit, single-step methods using more than one term in the Taylor series expansion of y_{n+1}. The method generalises the idea, employed by the Predictor-

Corrector scheme, of using a weighted average of gradients, to using m estimates of the gradient calculated at various points in the interval $x_n \le x \le x_{n+1}$ to determine the change in y.

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

RK2

The RK2 class of methods includes the Predictor-Corrector method discussed in section 9.1. The general RK2 scheme is;

$$\boxed{\begin{aligned} y_{n+1} &= y_n + a\,k_1 + b\,k_2, &(9.7a)\\ k_1 &= h\,f(x_n, y_n), &(9.7b)\\ k_2 &= h\,f(x_n + \alpha\,h, y_n + \beta\,k_1). &(9.7c)\end{aligned}}$$

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

$$\begin{aligned} &a = 0\,, \quad b = 1\,, \quad \alpha = \tfrac{1}{2}\,, \quad \beta = \tfrac{1}{2} && \text{:Single mid-point}\\ &a = \tfrac{1}{2}\,, \quad b = \tfrac{1}{2}\,, \quad \alpha = 1\,, \quad \beta = 1 && \text{:Predictor-Corrector} \qquad (9.8)\\ &a = \tfrac{1}{3}\,, \quad b = \tfrac{2}{3}\,, \quad \alpha = \tfrac{3}{4}\,, \quad \beta = \tfrac{3}{4} && \text{:Smallest error RK2}\end{aligned}$$

The RK2 method is illustrated in Fig. 9.3. It uses two stages of gradient estimation. At each stage, the gradient is used to estimate the change in y going from x_n to x_{n+1}. The first estimate is an Euler step yielding an estimated shift in y of $k_1 \equiv y^{(1)}_{n+1} - y_n$. 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 x and by an amount $\beta\,k_1$ in y. This yields a better estimate in the shift in y, i.e., $k_2 = y^{(2)}_{n+1} - y_n$. The two shifts are then averaged with weights a and b.

Coupled ODEs

For coupled ODEs, the first estimate stage must be carried out for all the dependent variables (e.g. u, v, w in $\vec{y} = \{u, v, w\}$) before moving onto the 2nd estimation stage. The RK2 scheme is then;

$$\begin{aligned} \vec{y}_{n+1} &= \vec{y}_n + a\,\vec{k}_1 + b\,\vec{k}_2, &(9.9a)\\ \vec{k}_1 &= h\,\vec{f}(x_n, \vec{y}_n), &(9.9b)\\ \vec{k}_2 &= h\,\vec{f}(x_n + \alpha\,h, \vec{y}_n + \beta\,\vec{k}_1). &(9.9c)\end{aligned}$$

RK4

This uses 4 stages of gradient estimation to calculate an average one and is 4th-order accurate.

$$\boxed{\begin{aligned} y_{n+1} &= y_n + \frac{1}{6}\left(k_1 + 2\,k_2 + 2\,k_3 + k_4\right), &(9.10a)\\ k_1 &= h\,f(x_n, y_n), &(9.10b)\\ k_2 &= h\,f(x_n + \frac{1}{2}\,h, y_n + \frac{1}{2}\,k_1), &(9.10c)\\ k_3 &= h\,f(x_n + \frac{1}{2}\,h, y_n + \frac{1}{2}\,k_2), &(9.10d)\\ k_4 &= h\,f(x_n + h, y_n + k_3). &(9.10e)\end{aligned}}$$

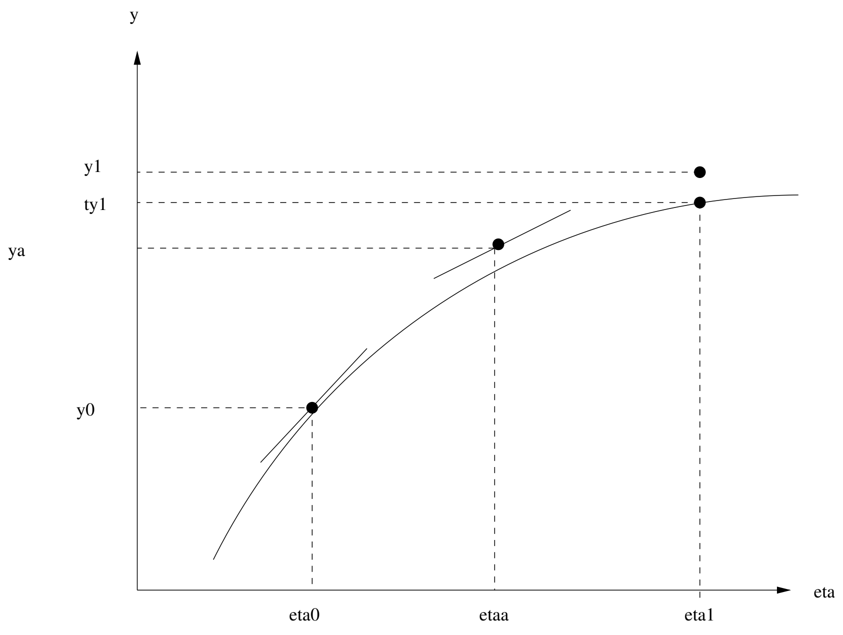

Figure 9.3: The RK2 method. A second gradient is calculated at a shifted position and averaged with the gradient at the starting position to obtain $\tilde{y}_1$, the next value of y. Comparing to fig. 9.1 you can see that $\tilde{y}_1$ is more accurate for RK2 compared to Euler.

Again the first stage is an Euler estimate, as in RK2. The 2nd and 3rd stages estimate the gradient using a fraction $\frac{1}{2}$ of the preceding shifts, k_1 and k_2, respectively. The 4th stage estimates the gradient using the gradient function f evaluated at the best estimate yet of the true value of y_{n+1}, i.e., $y^{(3)}_{n+1} = y_n + k_3$ 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 than the Euler and Predictor-Corrector methods. 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 don't need to store any previous steps. This makes coding it a little easier.

9.4 Variable step size h

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 the lecture slides for an outline of how to do this, and 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 y_{n-1}, etc., using a good single-step method.

9.5 Implicit Methods

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

$$\boxed{y_{n+1} = y_n + f(x_{n+1}, y_{n+1})\, h.} \tag{9.11}$$

Implicit Euler is still a 1st-order method, with global accuracy $\mathcal{O}(h)$, just like its explicit cousin. For a linear function $f(x, y) = p(x)y + q(x)$ equation (9.11) can easily be rearranged to give $y_{n+1} = g(x_{n+1}, y_n)$ (where g is a new function) which can easily be solved. For a non-linear function, (9.11) is a non-linear algebraic equation in y_{n+1}, that needs to be solved using something like the bisection, Newton or secant method (see section 2.3).

Stability

What is the advantage then? The advantage of implicit Euler over explicit Euler is that it is **unconditionally stable**. There is no critical step size h, above which numerical instability occurs. Of course, using a large h in implicit Euler will give a very inaccurate solution. The stability analysis carried out in section 8.5 (for explicit Euler) can be applied to implicit Euler, in much the same way, yielding

$$\frac{\epsilon_{n+1}}{\epsilon_n} \approx \left(1 - \frac{\partial f}{\partial y}\, h\right)^{-1} \tag{9.12}$$

so that for decaying problems ($\partial f/\partial y < 0$) we have $g \le 1$ for ***any*** (positive) h. What happens to y_{n+1} for large h? It simply tends to zero; the same behaviour as the exact solution. Inspection of equation (9.11) for the linear decay problem ($f = -\beta\, y$) shows that $y_{n+1} = y_n/(1 + \beta\, h) \to 0$ as $h \to \infty$.

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

Coupled ODEs

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

$$\begin{aligned} \vec{y}_{n+1} &= \vec{y}_n + h\mathbf{L} \cdot \vec{y}_{n+1} \\ \therefore \qquad (\mathbf{I} - h\mathbf{L}) \cdot \vec{y}_{n+1} &= \vec{y}_n \\ \therefore \qquad \vec{y}_{n+1} &= (\mathbf{I} - h\mathbf{L})^{-1} \cdot \vec{y}_n = \bar{\mathbf{T}} \cdot \vec{y}_n \end{aligned} \tag{9.13}$$

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

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

Chapter 10

Numerical Integration

Outline of Section

- The Trapezoidal Rule
- Simpson's Rule
- Romberg integration
- Relationship to ODE methods
- Improper Integrals

10.1 Introduction

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

$$I = \int_a^b f(x)\mathrm{d}x. \tag{10.1}$$

Numerical integration is an entire subfield of numerical analysis, with numerous sophisticated methods and codes available. These basically all fall into three main categories: Monte Carlo, quadrature and ODE methods. We have already seen Monte Carlo integration in Chapter 6. In this Chapter, we will deal predominantly with Newton-Coates quadrature methods, touching on ODE methods towards the end. We will also see how to transform badly-behaved integrals into a form that is more conducive to numerical evaluation.

10.2 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. 10.1 – 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:

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

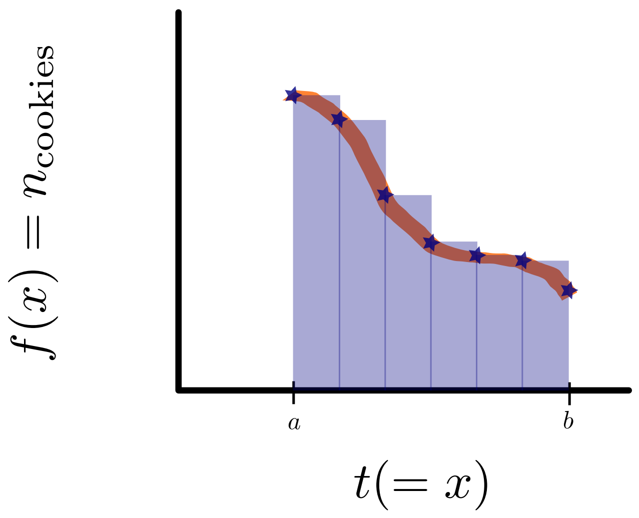

Figure 10.1: Basic anatomy of numerical integration by quadrature: the integral is divided up into rectangle-like shapes, the integrand is sampled at their edges, and the final integral is obtained by assuming some interpolating function passing between the sampled points. Cookies for dramatic effect only.

The simplest methods in this class are the *Newton-Coates rules*, which use a constant stepsize h; quadrature methods with a variable h across the integration domain are referred to as *Gaussian quadrature*. The example shown in Fig. 10.1 is therefore in fact a Newton-Coates method. Newton-Coates 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-Coates methods; the difference between them is essentially in the choice of interpolating function.

10.2.1 The Trapezoidal Rule

Apart from a basic Riemann sum, the simplest way to approximate the integrand between samples is interpolate between the 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:

$$\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x = h\left[\frac{1}{2}f(x_i) + \frac{1}{2}f(x_{i+1})\right] + O\left(h^3\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right). \tag{10.2}$$

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

$$\int_{x_0}^{x_{n-1}} f(x)\,\mathrm{d}x = h\left[\frac{1}{2}f(x_0) + f(x_1) + f(x_2) + \ldots + f(x_{n-2}) + \frac{1}{2}f(x_{n-1})\right] \tag{10.3}$$

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 $f(\frac{a+b}{2})$

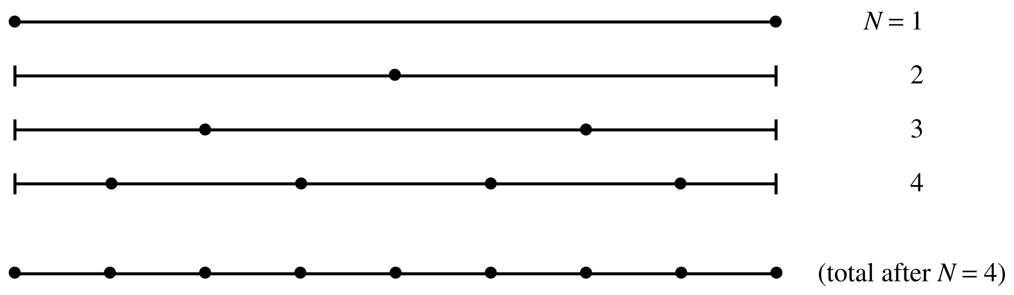

Figure 10.2: Sampling strategy for successive iterations of the Extended Trapezoidal Rule. Each new step fills in the midpoints between the previous points, never evaluating the integrand at the same value of x twice.

(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).

(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. 10.2), 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):

$$T_{j+1} = \frac{1}{2}T_j + h\sum_{i=1}^{(n-1)/2} f(x_{2i-1}). \tag{10.4}$$

10.2.2 Simpson's Rule

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

$$\int_{x_i}^{x_{i+2}} f(x)\,\mathrm{d}x = h\left[\frac{1}{3}f(x_i) + \frac{4}{3}f(x_{i+1}) + \frac{1}{3}f(x_{i+2})\right] + O\left(h^5\frac{\mathrm{d}^4 f}{\mathrm{d}x^4}\right). \tag{10.5}$$

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:

$$\begin{aligned}\int_{x_0}^{x_{n-1}} f(x)\,\mathrm{d}x =& h\left[\frac{1}{3}f(x_0) + \frac{4}{3}f(x_1) + \frac{2}{3}f(x_2) + \frac{4}{3}f(x_3)\ldots\right.\\ &\left.\ldots + \frac{4}{3}f(x_{n-4}) + \frac{2}{3}f(x_{n-3}) + \frac{4}{3}f(x_{n-2}) + \frac{1}{3}f(x_{n-1})\right]\end{aligned} \tag{10.6}$$

Note in particular 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:

$$S_j = \frac{4}{3}T_{j+1} - \frac{1}{3}T_j. \tag{10.7}$$

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.

10.2.3 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 tradeoff 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-Coates 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-Coates 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. These methods are covered in the following Section.

10.3 Relationship to ODE methods

Doing a definite integral is mathematically equivalent to solving the initial value problem (IVP)

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x); \qquad y(a) = 0 \tag{10.8}$$

for $x = b$, i.e. $I \equiv y(b)$. We can see this by

$$\begin{aligned} I \equiv \int_a^b f(x)\,\mathrm{d}x &= \int_a^b \frac{\mathrm{d}y}{\mathrm{d}x}\,\mathrm{d}x \\ &= \int_{y(a)}^{y(b)} \mathrm{d}y \\ &= y(b) - y(a) \end{aligned}$$

Here the choice of the initial value $y(a)$ is entirely arbitrary, as we care only about its derivative. We can therefore choose any constant $y(a) = C \implies I = y(b) - C$. For simplicity, we can therefore just choose $C = 0$, giving $I = y(b)$.

To solve the integral, we therefore need to solve the ODE that describes the evolution of $y(x)$, starting from from $x = a$ and finishing at $x = b$. We have seen in the previous Chapter that there are a number of reasonable methods for this. RK45 with an adaptive stepsize is one of the most robust and efficient.

So what happens if we just directly use RK45 for doing definite integrals by recasting them as ODEs? Well, we end up with a nice adaptive numerical integration method – but we can actually do a bit better than that. Recall the general form for the RK4 step:

$$k_1 = hf(x_n, y_n) \tag{10.9}$$
$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \tag{10.10}$$
$$k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \tag{10.11}$$
$$k_4 = hf(x_n + h, y_n + k_3) \tag{10.12}$$
$$y_{n+1} = y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4. \tag{10.13}$$

Now notice that in the case where the ODE comes from a transformed definite integral, f does not depend on y. Therefore, $k_2 = k_3$, and we only need to evaluate one of them. We can therefore collapse the equations into a reduced RK4 step:

$$k_1 = hf(x_n) \tag{10.14}$$
$$k_2 = hf(x_n + \frac{h}{2}) \tag{10.15}$$
$$k_3 = hf(x_n + h) \tag{10.16}$$
$$y_{n+1} = y_n + \frac{1}{6}k_1 + \frac{2}{3}k_2 + \frac{1}{6}k_3. \tag{10.17}$$

This is another 25% more efficient than the regular RK45 step when solving definite integrals.

Hang on though – this last expression looks eerily familiar. In fact, it is *exactly* Simpson's rule with $h \to \frac{h}{2}$:

$$\int_{x_n}^{x_{n+1}} f(x)\,\mathrm{d}x = \frac{h}{2}\left[\frac{1}{3}f(x_n) + \frac{4}{3}f(x_{n+\frac{1}{2}}) + \frac{1}{3}f(x_{n+1})\right] + O(h^5) \tag{10.18}$$

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 h^4. Indeed,

$$y(x_{n+1}) = y(x_n) + \int_{x_n}^{x_{n+1}} \frac{\mathrm{d}y}{\mathrm{d}x}(x)\,\mathrm{d}x = y(x_n) + \int_{x_n}^{x_{n+1}} f(x)\,\mathrm{d}x, \tag{10.19}$$

so RK4 is just Simpson's Rule generalised to a variable stepsize and an $f(x)$ that also depends on $y(x) \equiv \int f(x)\,\mathrm{d}x$.

10.4 Improper Integrals

Everything we have seen so far in this Chapter corresponds to a *closed* Newton-Coates rule (or its variable-stepsize generalisation, in the case of the RK45 implementation of the generalised Simpson's method). 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*,

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = h\left[f\left(\frac{x_0 + x_1}{2}\right)\right], \tag{10.20}$$

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 subdomain 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 subdomains 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 many copies of 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

$$I = \int_a^b f(x)\mathrm{d}x. \tag{10.21}$$

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

$$\int_a^b f(x)\,\mathrm{d}x = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right)\,\mathrm{d}t, \tag{10.22}$$

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

$$\alpha = t^{\frac{1}{1-\gamma}} + x_s \quad \text{for lower limit } x_s \tag{10.23}$$

$$\alpha = x_s - t^{\frac{1}{1-\gamma}} \quad \text{for upper limit } x_s. \tag{10.24}$$

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

$$\int_{a'}^{b'} f(x)\,\mathrm{d}x = \frac{1}{1-\gamma}\int_0^{(b'-a')^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(\alpha)\,\mathrm{d}t, \tag{10.25}$$

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, Gaussain 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 11

ODEs: Boundary Value Problems

Outline of Section

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

Consider the general, linear, second order equation

$$\alpha \frac{d^2y}{dx^2} + \beta \frac{dy}{dx} + \gamma\, y = k\,. \tag{11.1}$$

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

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

11.1 Shooting Method

A first method we can use is the **shooting method** where we start at a with $y(a) = A$, make a guess for $y'(a) = C_1$ and find a solution $y_1(x)$. This can be obtained via a finite difference method (from Chapter 9), or algebraically if we are lucky. In general the solution will not end at the desired point $y(b) = B$ say, but will end instead at $y_1(b) = B_1$.

We then make another guess starting from the same point $y(a) = A$ but with $y'(a) = C_2$ and find a new solution $y_2(x)$ which ends at $y_2(b) = B_2$. The system (11.1) is **linear**, so a sum of two solutions will itself be a solution of the system. In this case, we can introduce a weighted average of y_1 and y_2 as a new solution

$$\boxed{y_c(x) = c\, y_1(x) + (1-c)\, y_2(x),} \tag{11.2}$$

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

We can check that

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

as required and the condition that

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

gives us a solution for c

$$\boxed{c = \frac{B - B_2}{B_1 - B_2}.} \tag{11.3}$$

So finding the required solution to a linear system is relatively straightforward using the shooting method. However it still requires solving the system twice to get a single solution.

In order to solve the more general problem, where the ODEs may in general constitute *non*-linear systems, we need to iterate. Specifically, we need to keep choosing new values of $y'(a)$ until we hit on a value that results in $y_2(b) = B$. This then becomes a root-finding problem: if B_1 and B_2 bracket B, then by the intermediate value theorem, at least one of the values of $y'(a)$ that solves the problem is bracketed by C_1 and C_2. 'All' that remains to do is then to refine the brackets as quickly as possible using your preferred root-finding method. This is of course much easier said than done when the underlying ODE is more complex, and e.g. multiple higher-order derivatives need to be chosen for each 'shot' to be fired from $x = a$.

Next we will look at another method which solves the system explicitly via matrix methods.

11.2 Finite Difference Method

Consider the system

$$\frac{d^2y}{dx^2} = k\,. \tag{11.4}$$

We can re-write the second derivative as a central difference and discretise the solution with m intervals of size h as in Chapter 8.1 (see also Problem Sheet)

$$\frac{y_{i-1} - 2\,y_i + y_{i+1}}{h^2} = k\,. \tag{11.5}$$

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

$$\begin{array}{rcl} A - 2\,y_1 + y_2 &=& k\,h^2, \\ y_1 - 2\,y_2 + y_3 &=& k\,h^2, \\ &.& \\ &.& \\ &.& \\ y_{m-2} - 2\,y_{m-1} + B &=& k\,h^2, \end{array} \tag{11.6}$$

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

$$\begin{pmatrix} -2 & 1 & 0 & 0 & .... & 0 & 0 \\ 1 & -2 & 1 & 0 & .... & 0 & 0 \\ & & & . & & & \\ & & & . & & & \\ & & & . & & & \\ 0 & 0 & 0 & 0 & .... & 1 & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_{m-1} \end{pmatrix} = \begin{pmatrix} k\,h^2 - A \\ k\,h^2 \\ . \\ . \\ . \\ k\,h^2 - B \end{pmatrix}, \tag{11.7}$$

i.e. $\mathbf{M} \cdot \tilde{\vec{y}} = \vec{b}$. The '$\sim$' is used to denote that the solution is approximate (compared to the true solution of eqn (11.4) at the sample points), because of the use of finite difference approximation to derivatives. The matrix $\mathbf{M}$ is close to being diagonally dominated (since it has $|m_{ii}| \geq \sum_{j\neq i} |m_{ij}|$) and is in fact suitable for solving using the Jacobi, Gauss-Seidel and SOR methods introduced in Chapters 3.5, 3.6 and 3.7. This is because the spectral radius of the associated update matrix (for each method) is less than unity. Figure 11.1 illustrates the finite difference method applied to solving a boundary value problem.

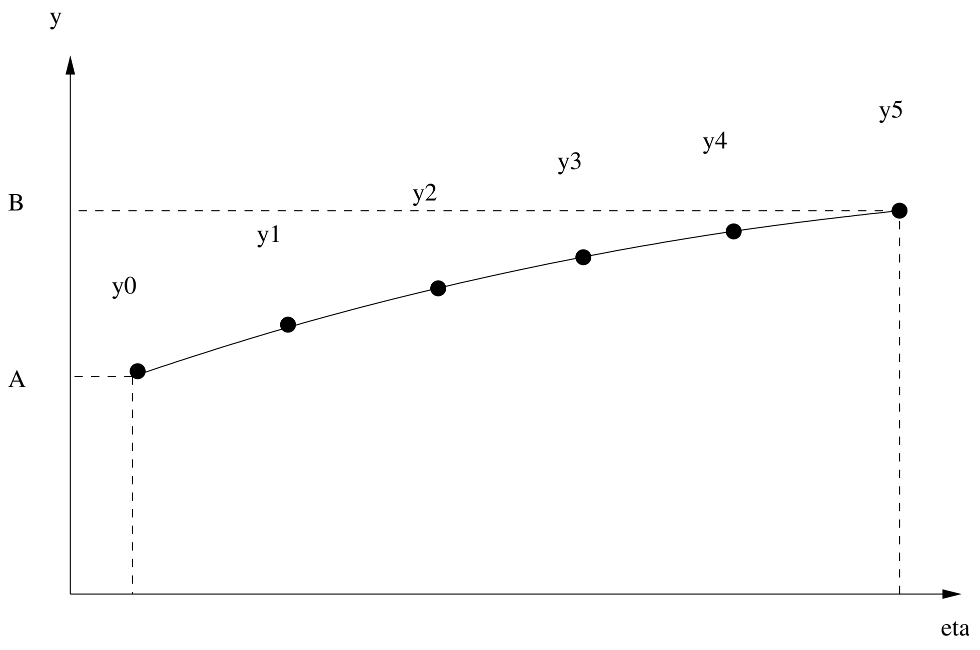

Figure 11.1: Finite difference method, for solution of boundary value problem. Example of discrete points with $m = 5$.

11.3 Derivative Boundary Conditions

On occasion we know the derivative at one of the boundaries but not the initial condition for the variable. Considering the same system as in (11.4) we could have boundary conditions

$$y'(a) = C \quad \text{and} \quad y(b) = B\,. \tag{11.8}$$

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

$$y_0'(a) \approx \frac{y_1 - y_{-1}}{2h} = C, \tag{11.9}$$

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

$$y_{-1} - 2\,y_0 + y_1 = k\,h^2, \tag{11.10}$$

which, given (11.9), is

$$-2\,y_0 + 2y_1 = k\,h^2 + 2\,h\,C\,. \tag{11.11}$$

The system can then be written as m linear equations

$$\begin{pmatrix} -2 & 2 & 0 & 0 & .... & 0 & 0 \\ 1 & -2 & 1 & 0 & .... & 0 & 0 \\ & & & . & & & \\ & & & . & & & \\ & & & . & & & \\ 0 & 0 & 0 & 0 & .... & 1 & -2 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ . \\ . \\ . \\ y_{m-1} \end{pmatrix} = \begin{pmatrix} k\,h^2 + 2\,h\,C \\ k\,h^2 \\ . \\ . \\ . \\ k\,h^2 - B \end{pmatrix}. \tag{11.12}$$

11.4 Eigenvalue Problems

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

$$\frac{d^2y}{dx^2} + k^2 y = 0, \tag{11.13}$$

with boundary conditions $y(0) = 0$ and $y(1) = 0$. This describes the vibrations on a string of length 1 and fixed at the endpoints. The general solution to this system is easily found to be

$$y(x) = A\,\sin(k\,x) + B\,\cos(k\,x)\,. \tag{11.14}$$

The boundary conditions imply that $B = 0$ and that $k = \pm n\,\pi$. The solution describes fundamental modes of vibrations on the string where $n = 1, 2, 3, \ldots$, etc.

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

$$\frac{y_{i-1} - 2\,y_i + y_{i+1}}{h^2} + k^2\,y_i = 0, \tag{11.15}$$

which gives the eigensystem

$$\begin{pmatrix} +2 & -1 & 0 & 0 & \ldots. & 0 & 0 \\ -1 & +2 & -1 & 0 & \ldots. & 0 & 0 \\ & & & . & & & \\ & & & . & & & \\ & & & . & & & \\ 0 & 0 & 0 & 0 & \ldots. & -1 & +2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_{m-1} \end{pmatrix} = h^2\,k^2 \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_{m-1} \end{pmatrix}, \tag{11.16}$$

i.e.

$$\mathbf{A} \cdot \tilde{\vec{y}} = \lambda \tilde{\vec{y}} \tag{11.17}$$

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 3.8) can be used to readily find these.

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 11.1 and mixed conditions in 11.3 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 11.2 by discretising the system onto a 2D lattice of points and using finite difference approximations to the partial derivatives. The lattice is also known as a **mesh** or a **grid**. As an example we start with Laplace’s equation

$$\nabla^2 u(x, y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \tag{12.7}$$

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 11.3 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 $0 \leq x \leq L_x$ and $0 \leq y \leq L_y$. 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]

[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!

To keep things simple the grid spacing will be taken as h in both directions. (The numerical equations below can readily be generalised for different spacings, e.g., $\Delta x \neq \Delta y$. It does not change the approach.) The location of the grid points are defined as

$$x_i = i\,h \qquad (i = 0,\,1,\,\ldots,n_x+1) \qquad L_x = (n_x+1)\,h \tag{12.8}$$
$$y_j = j\,h \qquad (j = 0,\,1,\,\ldots,n_y+1) \qquad L_y = (n_y+1)\,h. \tag{12.9}$$

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 8.1, i.e., equation (8.9), 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

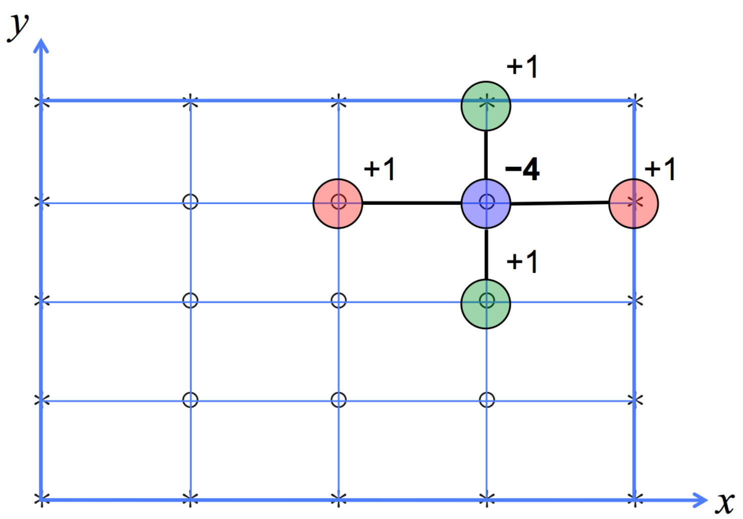

Figure 12.1: **FD stencil for the 2D Laplacian** $\nabla^2 u_{i,j}$. (Note that the $1/h^2$ factor is omitted in the indicated weights.)

(+1). The relative weight of the centre point is −4. This is also known as a **finite difference stencil**.

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 $m{\times}m = n_xn_y \times n_xn_y$ 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.

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

$$\vec{u}^{n+1} = -\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U}) \cdot \vec{u}^n + \mathbf{D}^{-1} \cdot \vec{b}. \tag{12.19}$$

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

$$\vec{u}^{n+1} = \frac{1}{4}\mathbf{I} \cdot \left[(\mathbf{L} + \mathbf{U}) \cdot \vec{u}^n - \vec{b}\right]. \tag{12.20}$$

However this can be represented using the pictorial operator as

$$u^{n+1}_{i,j} = \frac{1}{4}\begin{bmatrix} & 1 & \\ 1 & 0 & 1 \\ & 1 & \end{bmatrix} u^n_{i,j}, \tag{12.21}$$

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 (12.21). 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 3.5, page 23 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).

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

$$\nabla^2 u = -\frac{1}{12}\left(\frac{\partial^4 u}{\partial x^4} + \frac{\partial^4 u}{\partial y^4}\right) h^2 + \mathcal{O}(h^4) \tag{12.22}$$

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

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

$$\nabla^2 u = \rho(x, y), \tag{12.23}$$

when using the Jacobi method in pictorial-operator form requires the minor modification

$$u_{i,j}^{n+1} = \frac{1}{4}\begin{bmatrix} & 1 & \\ 1 & 0 & 1 \\ & 1 & \end{bmatrix} u_{i,j}^{n} - \frac{h^2}{4}\rho_{i,j}, \tag{12.24}$$

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 11.2 we will approximate the derivative as a central difference at the boundary involving a fictitious point. Below, we illustrate derivative boundary conditions in y, i.e., applied at $y = 0$ and $y = L_y$. Dirichlet conditions are still used in x. 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

$$\begin{array}{ccccc}
 & u_{1,5} & u_{2,5} & u_{3,5} & \\
 & * & * & * & \\
C_{0,4} & u_{1,4} & u_{2,4} & u_{3,4} & C_{4,4} \\
* & \circ & \circ & \circ & * \\
C_{0,3} & u_{1,3} & u_{2,3} & u_{3,3} & C_{4,3} \\
* & \circ & \circ & \circ & * \\
C_{0,2} & u_{1,2} & u_{2,2} & u_{3,2} & C_{4,2} \\
* & \circ & \circ & \circ & * \\
C_{0,1} & u_{1,1} & u_{2,1} & u_{3,1} & C_{4,1} \\
* & \circ & \circ & \circ & * \\
C_{0,0} & u_{1,0} & u_{2,0} & u_{3,0} & C_{4,0} \\
* & \circ & \circ & \circ & * \\
 & u_{1,-1} & u_{2,-1} & u_{3,-1} & \\
 & * & * & * &
\end{array} . \tag{12.27}$$

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 $0 \le x \le L$ and $t_i \le t \le t_f$, i.e., an initial value problem. The boundary conditions for this system have to be provided for all x at t_i (i.e. the initial condition) and at the edges ($x = 0$, $x = L$) for $t_i < t \le t_f$. We can discretise the solutions using h as the spatial step and Δt as the time step. The 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 $v \ge 0$ 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

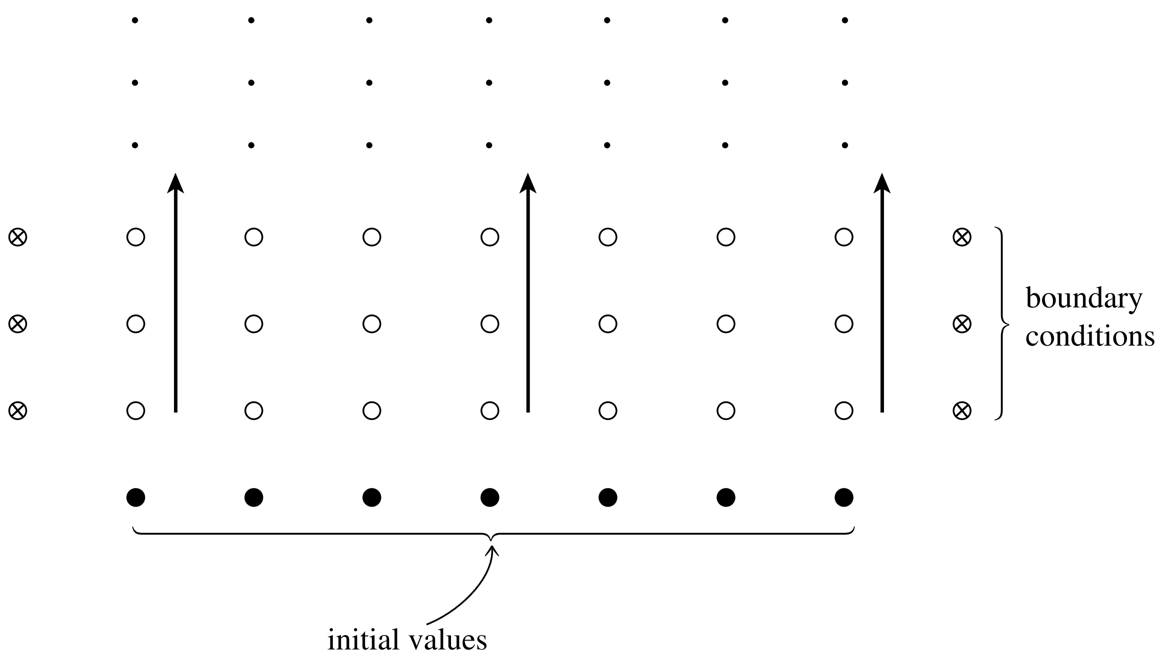

Figure 12.2: Initial value problem in (x, t) and the 1+1 discretisation scheme. The solution is integrated forward in time from an initial solution at a given "time slice", with boundary conditions given at the extremal values of x over all times. From Numerical Recipes.

general solution of the linear wave-equation is thus $u(x, t) = F(x - vt) + G(x + vt)$. In 3D the CVA equation looks like

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

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

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 8–9, which were used to solve

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

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

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u}) = 0, \qquad \text{(mass conservation law)} \tag{12.46}$$

$$\frac{\partial (\rho \vec{u})}{\partial t} + \vec{u} \cdot \nabla (\rho \vec{u}) = -\nabla P, \qquad \text{(eqn. of motion)} \tag{12.47}$$

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 $\mathbf{F}$ 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 $\mathbf{F}$ 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 $\mathbf{F}$ 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.

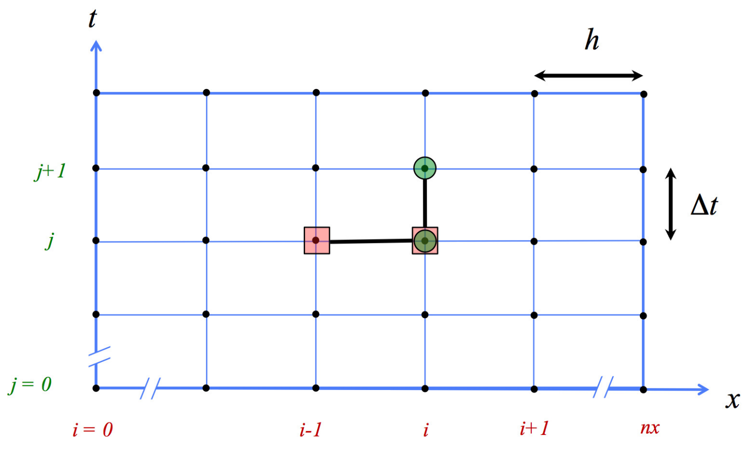

Figure 12.3: **FD stencil for upwind method.** Square and circle symbols denote contributions from $\partial u/\partial x$ and $\partial u/\partial t$ terms, respectively. The $v_x > 0$ case is shown.

For $v_x < 0$ we choose

$$\left.\frac{\partial u}{\partial x}\right|_i^j = \frac{u_{i+1}^j - u_i^j}{h} \qquad \{\,\mathcal{O}(h)\,\},$$

and thus the FDE is

$$u_i^{j+1} = u_i^j + a(u_{i+1}^j - u_i^j). \tag{12.55}$$

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.

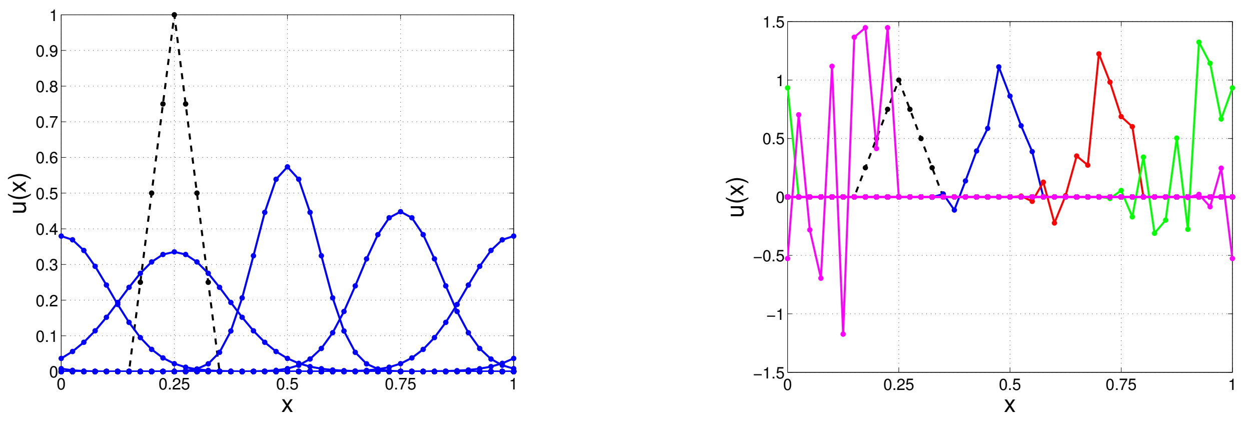

Figure 12.4: **Advection via the upwind method.** (Left) Advection number $a = 0.5$, and (Right) $a = 1.05$. The initial profile is a triangle (dotted black lines). Parameters: $v_x = +1$, $h = 1/40$, (left) $\Delta t = h/2$ and (right) $\Delta t = 1.05h$. The profile moves to the right and eventually 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

$$\frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} = -\frac{1}{2}\frac{\partial^2 u}{\partial t^2}\,\Delta t + \mathcal{O}(\Delta t^2) + \frac{1}{2}\frac{\partial^2 u}{\partial x^2}\,v_x h + \mathcal{O}(h^2). \tag{12.56}$$

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

$$\begin{aligned}\frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} &= \frac{|v_x| h}{2}(1-a)\,\frac{\partial^2 u}{\partial x^2} \;+\; \mathcal{O}(\Delta t^2) \;+\; \mathcal{O}(h^2) \\ &\approx D_{\text{num}}\,\frac{\partial^2 u}{\partial x^2},\end{aligned} \tag{12.57}$$

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

$$u^j_{n_x} = u^j_0, \tag{12.58}$$

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

$$\begin{aligned}\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n_x} \end{pmatrix}^{j+1} &= \begin{pmatrix} (1-a) & 0 & 0 & \dots & 0 & a \\ a & (1-a) & 0 & \dots & 0 & 0 \\ 0 & a & (1-a) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a & (1-a) \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n_x} \end{pmatrix}^{j}, \\ \vec{u}^{\,j+1} &= \mathbf{T}\cdot\vec{u}^{\,j},\end{aligned} \tag{12.59}$$

where $\mathbf{T}$ is the update matrix (analogous to those seen with ODE FD methods in sections 2–9).

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 $u(x,t)$

$$\boxed{u_i^j = \exp\left[i(kx_i - \omega t^j)\right] = (g)^j \exp(ikx_i),} \tag{12.60}$$

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

$$u_i^{j+1} = g\, u_i^j \qquad , \qquad u_{i\pm1}^j = u_i^j \exp(\pm i\, k\, h). \tag{12.61}$$

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

$$|g|^2 = 1 - 4a(1-a)\sin^2(kh/2). \tag{12.62}$$

The maximum wave-number allowable on the grid is

$$k_{max} = \frac{\pi}{h} \qquad (\text{i.e. } \lambda_{min} = 2h\,), \tag{12.63}$$

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

$$|g| \le 1,$$

i.e., $1 - 4a(1-a) \le 1$ which yields $a \ge 0$ and

$$\boxed{a \le 1 \qquad \text{or equivalently} \qquad \Delta t \le h/|v_x|.} \tag{12.64}$$

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

$$|v_x| \le v_{\text{grid}}, \tag{12.65}$$

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 (12.60), we will see in Section 5.2 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 8.6. 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. (12.59)).

For stability we require that the spectral radius of the update matrix does not exceed unity, i.e. , $\rho(\mathbf{T}) \le 1$. Thus all eigenvalues λ_i of the update matrix must satisfy

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

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

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

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

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

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 $a \le 1$ is required for $|\lambda| \le 1$ 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.

12.3.4 FTCS method – (one not to use!)

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

$$\left.\frac{\partial u}{\partial x}\right|_i^j = \frac{u_{i+1}^j - u_{i-1}^j}{2h} \qquad \left\{\, \mathcal{O}(h^2) \,\right\}.$$

This leads to the innocuous looking FDE (for any v_x, +ve and -ve)

$$u_i^{j+1} = u_i^j - \frac{v_x \Delta t}{2h}\left(u_{i+1}^j - u_{i-1}^j\right), \tag{12.66}$$

known as the 'Forward Time Centred Space' (FTCS) method. It is a consistent, $\mathcal{O}(\Delta t) + \mathcal{O}(\Delta h^2)$ FD scheme. It may surprise you that this is **unconditionally unstable!** A stability analysis (by either method) yields the following impossible condition for stability $|v_x|\Delta t/(2h) \le 0$. (Impossible for positive Δt and h.)

12.3.5 Lax method

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

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

giving the FDE

$$u_i^{j+1} = \frac{1}{2}\left(u_{i+1}^j + u_{i-1}^j\right) - \frac{v_x \Delta t}{2h}\left(u_{i+1}^j - u_{i-1}^j\right), \tag{12.67}$$

which turns out to have the usual CFL stability condition, $0 \le a \le 1$ (Remember $a = |v_x|\Delta t/h$.) However the MDE can be shown to be

$$\frac{\partial u}{\partial t} + v_x \frac{\partial u}{\partial x} = -\frac{1}{2}\frac{\partial^2 u}{\partial t^2}\Delta t - \frac{1}{6}\frac{\partial^3 u}{\partial x^3} v_x h^2 + \frac{1}{2}\frac{\partial^2 u}{\partial x^2}\frac{h^2}{\Delta t} + \mathcal{O}(\Delta t^2) + \mathcal{O}(h^2). \tag{12.68}$$

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

12.3.6 Lax-Wendroff

This is FTCS with a little extra diffusion added

$$\boxed{u_i^{j+1} = u_i^j - \frac{v_x \Delta t}{2h}\left(u_{i+1}^j - u_{i-1}^j\right) + \frac{(v_x \Delta t)^2}{2h^2}\left(u_{i+1}^j - 2u_i^j + u_{i-1}^j\right),} \tag{12.69}$$

which can be shown to be a consistent, $\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta h^2)$ scheme. It has the usual CFL condition $0 \le a \le 1$. Interestingly, adding in just the right amount of the FD diffusion operator makes this scheme higher order in Δt as well as stabilising the method. Also Lax-Wendroff doesn't have two different versions of the FDE, in contrast to the one-sided upwind method. The improved accuracy of the Lax-Wendroff method is demonstrated in figure 12.5.

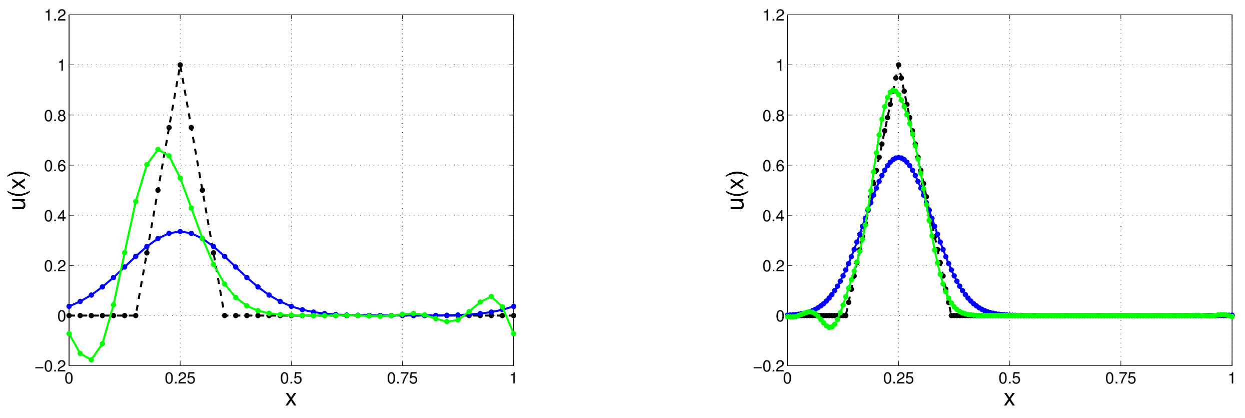

Figure 12.5: **Advection via the Lax-Wendroff method.** The same problem as in figure 12.4 is used. The Lax-Wendroff (green curves) and upwind method (blue curves) are compared after one transit time ($t = L/v_x = 1$) for advection number $a = 0.5$. (Left) $h = 1/40$. (Right) $h = 1/160$. Lax-Wendroff shows less numerical diffusion.

12.3.7 Coupled equations

The Lax-Wendroff and Lax methods can be applied to the coupled 1st-order PDEs seen earlier, e.g., equations (12.48) for ρ (mass density) and u (fluid velocity). Using the 'flux operator' approach and choosing the simpler Lax scheme to illustrate;

$$\vec{U}_i^{j+1} = \frac{1}{2}\left(\vec{U}_{i+1}^j + \vec{U}_{i-1}^j\right) - \mathbf{F} \cdot \left[\frac{\Delta t}{2h}\left(\vec{U}_{i+1}^j - \vec{U}_{i-1}^j\right)\right], \tag{12.70}$$

where $\vec{U} = [\rho, u]^T$, which splits into

$$\begin{aligned} \rho_i^{j+1} &= \tfrac{1}{2}\left(\rho_{i+1}^j + \rho_{i-1}^j\right) - \tfrac{\rho_o\,\Delta t}{2h}\left(u_{i+1}^j - u_{i-1}^j\right), \\ u_i^{j+1} &= \tfrac{1}{2}\left(u_{i+1}^j + u_{i-1}^j\right) - \tfrac{c^2\,\Delta t}{2\rho_o\,h}\left(\rho_{i+1}^j - \rho_{i-1}^j\right). \end{aligned} \qquad (12.71)$$

12.4 Solving Parabolic PDEs

Consider the diffusion equation in one spatial dimension

$$\frac{\partial u}{\partial t} = D\,\frac{\partial^2 u}{\partial x^2}. \qquad (12.72)$$

with D constant.

Explicit method

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

$$\left.\frac{\partial u}{\partial t}\right|_i^j = \frac{u_i^{j+1} - u_i^j}{\Delta t} \qquad \{\,\mathcal{O}(\Delta t)\,\},$$

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

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_i^j = \frac{u_{i-1}^j - 2\,u_i^j + u_{i+1}^j}{h^2} \qquad \{\,\mathcal{O}(h^2)\,\}. \qquad (12.73)$$

Inserting these approximations into the diffusion equation (12.72) we get

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} = D\left(\frac{u_{i-1}^j - 2\,u_i^j + u_{i+1}^j}{h^2}\right), \qquad (12.74)$$

giving the FDE

$$\boxed{u_i^{j+1} = (1 - 2\,d)\,u_i^j + d\,(u_{i-1}^j + u_{i+1}^j),} \qquad (12.75)$$

where $$d = \frac{D\,\Delta t}{h^2}, \qquad (12.76)$$

is the **diffusion number**, a dimensionless parameter. An example of the explicit method for diffusion is shown in figure 12.6.

The FDE (12.75) 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 (12.75) to be stable is

$$\boxed{d = \frac{D\,\Delta t}{h^2} < \frac{1}{2} \qquad \text{or equivalently} \qquad \Delta t < \frac{h^2}{2D}.} \qquad (12.77)$$

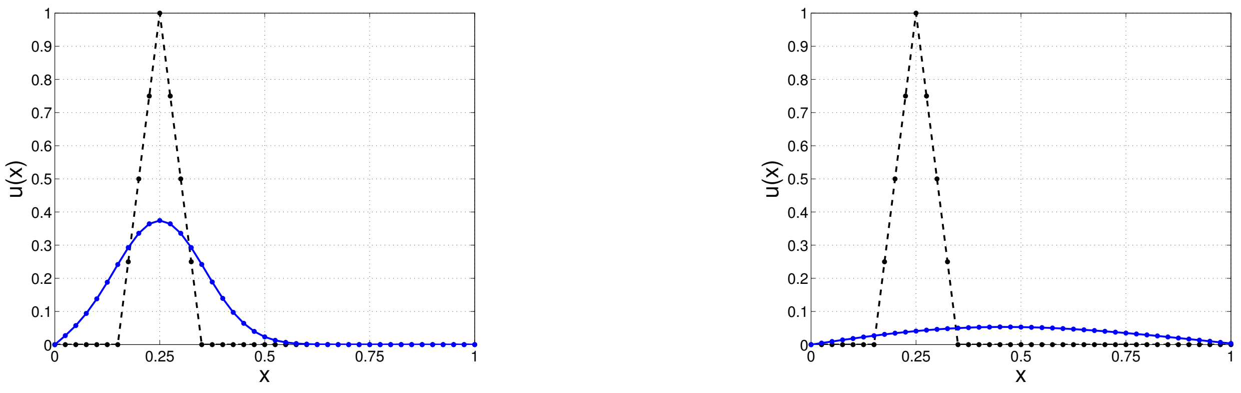

Figure 12.6: **Diffusion via the explicit method.** (Left) $t = 0.005$ and (Right) $t = 0.1$. The initial profile is a triangle (dotted black lines). Fixed (Dirichlet) boundary conditions are used; $u = 0$ on the boundaries. Parameters: $D = 1$, $h = 1/40$, $\Delta t = 0.25\tau_{\text{diff-grid}}$ (i.e. $d = 0.25$).

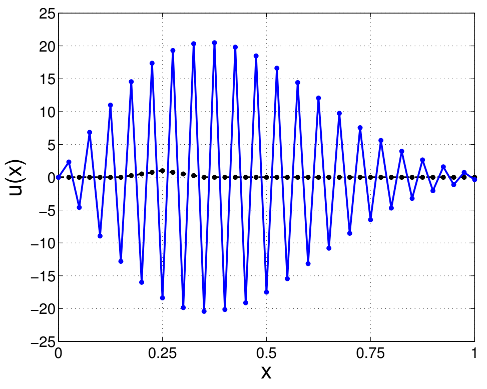

Figure 12.7: **Diffusion via the explicit method.** Numerical instability when the CFL condition is exceeded; shown at $t = 0.1$. Here $d = 0.51$. (The same system, and other parameters, are used as in figure 12.6.)

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**

$$\tau_{\mathrm{diff-grid}} \equiv \frac{h^2}{D}, \tag{12.78}$$

which characterises the time it takes for information to propagate between sites on the grid. Figure 12.7 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

$$\left.\frac{\partial u}{\partial t}\right|_i^{j+1/2} = \frac{u_i^{j+1} - u_i^j}{\Delta t} \qquad \{\,\mathcal{O}(\Delta t^2)\,\}. \tag{12.79}$$

For the spatial derivative we can take the average of those at times j and $j+1$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_i^{j+1/2} = \frac{1}{2}\left[\left.\frac{\partial^2 u}{\partial x^2}\right|_i^{j} + \left.\frac{\partial^2 u}{\partial x^2}\right|_i^{j+1}\right]. \tag{12.80}$$

Rearranging all $j+1$ terms onto the LHS we get the system

$$\boxed{(2+2\,d)\,u_i^{j+1} - d\,(u_{i-1}^{j+1} + u_{i+1}^{j+1}) = (2-2\,d)\,u_i^j + d\left(u_{i-1}^j + u_{i+1}^j\right),} \tag{12.81}$$

or in matrix form

$$\mathbf{M}\cdot\vec{u}^{\,j+1} = \vec{b}^{\,j}, \tag{12.82}$$

where

$$\mathbf{M} \equiv \begin{pmatrix} (2+2\,d) & -d & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ -d & (2+2\,d) & -d & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & -d & (2+2\,d) & -d & \cdot & \cdot & \cdot & 0 \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & -d & (2+2\,d) \end{pmatrix}, \tag{12.83}$$

and

$$\vec{b}^{\,j} \equiv \begin{pmatrix} (2-2\,d) & d & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ d & (2-2\,d) & d & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & d & (2-2\,d) & d & \cdot & \cdot & \cdot & 0 \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & d & (2-2\,d) \end{pmatrix} \cdot \vec{u}^{\,j} = \mathbf{A}\cdot\vec{u}^{\,j}. \qquad (12.84)$$

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. $1 \le i \le n_x - 1$).

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 (12.82) using an iterative matrix solver (Jacobi, Gauss-Seidel or SOR) at each time step.