Computational Mathematics II (MATH2731)

Dr Andrew Krause & Dr Denis Patterson, Durham University

2025-06-01

Table of contents

Introduction

Welcome to Computational Mathematics II!

This course aims to help you build skills and knowledge in using modern computational methods to do and apply mathematics. It will involve a blend of hands-on computing work and mathematical theory—this theory will include aspects of numerical analysis, computational algebra, and other topics within scientific computing. These areas consist of studying the mathematical properties of the computational representations of mathematical objects (numerical values as well as symbolic manipulations). The computing skills developed in this module will be valuable in all subsequent courses in your degree at Durham and well beyond. We will also introduce you to the use (and abuse) of various computational tools invaluable for doing mathematics, such as AI and searchable websites. While we will encourage you throughout to use all the tools at your disposal, it is **imperative that you understand the details and scope of what you are doing!** You will also develop your communication, presentation, and group-work skills through the various assessments involved in the course – more on that below!

This module has **no final exam**. In fact, there are no exams of any kind. Instead, the summative assessment and associated final grade are entirely based on coursework undertaken during the term. This means that you should expect to spend more time on this course during the term relative to your other modules. We believe this workload distribution is a better way to train the skills we are trying to develop, and as a bonus, you will not need to worry about this course any further once the term ends!

Content

The module's content is divided into six chapters of roughly equal length; some will focus slightly more on theory, while others have a more practical and hands-on nature.

- **Chapter 1: Introduction to Computational Mathematics**
  - Programming basics (including GitHub, and numerical versus symbolic computation)
  - LaTeX, Overleaf, and presenting lab reports
  - Finite-precision arithmetic, rounding error, symbolic representations

- **Chapter 2: Continuous Functions**
  - Interpolation using polynomials – fitting curves to data (Lagrange polynomials, error estimates, convergence, and Chebyshev nodes)
  - Solving nonlinear equations (bisection, fixed-point iteration, Newton's method)
- **Chapter 3: Linear Algebra**
  - Solving linear systems numerically (LU decomposition, Gaussian elimination, conditioning) and symbolically
  - Applications: PageRank, computer graphics
- **Chapter 4: Calculus**
  - Numerical differentiation (finite differences)
  - Numerical integration (quadrature rules, Newton-Cotes formulae)
- **Chapter 5: Ordinary Differential Equations (ODEs)**
  - Numerically approximating solutions of ODEs
  - Timestepping: explicit and implicit methods
  - Stability and convergence order
- **Chapter 6: Selected Further Topics**
  - Intro. to random numbers and stochastic processes
  - Intro. to partial differential equations

Weekly workflow and summative assessment

The final grade for this module is determined as follows:

- **Weekly lab reports (weeks 1-6)** – 20%
- **Weekly e-assessments (weeks 1-6)** – 30%
- **Project (weeks 7-10)** – 50%

Lab reports

Each week for the first six weeks of the course, we will release a short set of exercises based on the lectures from the previous week. Students will be expected to submit a brief report (1-2 pages A4, including figures) with their solutions to the set of exercises – the report will consist of written answers and figures/plots. The reports will be evaluated for correctness and quality of the presentation and communication (quality of figures, clarity of argumentation, etc.).

The lab report for a given week will be due at noon on Monday of the following week (e.g., week one's lab report is due on Monday of week two and so on). Solutions and generalised feedback will be provided to the class on common mistakes and issues arising in each report. Students can also seek detailed feedback on their submission from the lecturers during drop-in sessions and office hours. There will be six lab reports in total, and **your mark is based on your four highest-scoring submissions.**

E-assessments

Each week for the first six weeks of the course, we will release a set of e-assessments based on the lectures from the previous week. These exercises are designed to complement the lab reports by focusing exclusively on coding skills. The e-assessments will involve submitting code auto-marked by an online grading tool, and hence give immediate feedback. As with the lab reports, **the set of e-assessments for a given week will be due at noon on Monday of the following week**. There will be six sets of e-assessments in total, and **your mark is based on your four highest-scoring submissions.**

Project

The single largest component of the assessment for this module is the project. **Weeks 7-10 of this course focus exclusively on project work with lectures ending in Week 6.** We will be releasing more detailed instructions on the project submission format and assessment criteria separately, but briefly, the main aspects of the project are as follows:

- There will be approximately eight different project options to choose from across different areas of mathematics (e.g., pure, applied, probability, mathematical physics, etc.); each project has a distinct member of the Maths Department as supervisor.
- Students will submit their preferred project options (ranked choice preferences) in Week 4 of the term and be allocated to projects by the end of Week 6 (there are maximum subscription numbers for each option to ensure equity of supervision).
- Each project consists of two parts: a **guided component** that is completed as part of a small group and an **extension component** that is open-ended and completed as an individual. Group allocations will be done by the lecturers.
- Each group will jointly submit a five-page report for the guided component of the project, and this is worth 60% of the project grade.
- Each student will also submit a three-page report and a six-minute video presentation on their extension component. This submission is worth 40% of the project grade.

In Weeks 7-10 of the term, lectures will be replaced by project workshop sessions during which students can discuss their project with the designated supervisor. This will be an opportunity to discuss progress, ask questions, and seek clarification. Each student only needs to attend the one project drop-in weekly session relevant to their project. Computing drop-in sessions will

continue as scheduled in the first six weeks to provide additional support for coding pertinent tasks for the projects – there will be two timetabled computing drop-ins per week and students are encouraged to attend at least one of them.

Lectures, computing drop-ins & project workshops

Lectures will primarily present, explain, and discuss new material (especially theory), but will also feature computer demonstrations of the algorithms and numerical methods. As such, students are encouraged to bring their laptops to lectures to run the examples themselves. Students must bring a laptop or device capable of running code to the computer drop-ins to work on the e-assessments and lab reports.

	Activities	Content
Week 1	Introductory lecture, 2 lectures	Chapter 1
Week 2	3 lectures, 1 computing drop-in	Chapter 2
Week 3	3 lectures, 1 computing drop-in	Chapter 3
Week 4	3 lectures, 1 computing drop-in	Chapter 4
Week 5	3 lectures, 1 computing drop-in	Chapter 5
Week 6	3 lectures, 1 computing drop-in	Chapter 5/6
Week 7	0 lectures, 1 project workshop	Project
Week 8	0 lectures, 1 project workshop	Project
Week 9	0 lectures, 1 project workshop	Project
Week 10	0 lectures, 1 project workshop	Project

Contact details and Reading Materials

If you have questions or need clarification on any of the above, please speak to us during lectures and drop-in sessions. Alternatively, email one or both of us at denis.d.patterson@durham.ac.uk or andrew.krause@durham.ac.uk.

The lecture notes are designed to be sufficient and self-contained. Hence, students do not need to purchase a textbook to complete the course successfully. References for additional reading will also be given at the end of each chapter.

The following texts may be useful supplementary references for students wishing to read further into topics from the course:

- Burden, R. L., & Faires, J. D. (1997). *Numerical Analysis* (6th ed.). Pacific Grove, CA: Brooks/Cole Publishing Company.
- Süli, E., & Mayers, D. F. (2003). *An Introduction to Numerical Analysis*. Cambridge: Cambridge University Press.

Acknowledgements

We are indebted to Prof. Anthony Yeates (Durham) whose numerical analysis notes formed the basis of several chapters of the coures notes.

1 Numerical and Symbolic Computing

The goal of this chapter is to explore and begin to answer the following question:

> *How do we represent numbers and symbolic expressions on a computer?*

Integers and arithmetic operations on computers can be represented exactly, up to some maximum size.

If 1 bit (binary digit) is used to store the sign $\pm$, the largest possible number is

$$1 \times 2^{62} + 1 \times 2^{61} + \ldots + 1 \times 2^1 + 1 \times 2^0 = 2^{63} - 1.$$

In contrast to the integers, only a subset of real numbers within any given interval can be represented exactly.

> **Note**
>
> Some modern languages (such as Python) automatically promote large integers to arbitrary precision ("long"), but most statically-typed languages (C, Java, Matlab, etc.) do not; an **overflow** will occur and the type remains fixed.

> **Note**
>
> A statically typed language is one in which the type of every variable is determined before the program runs.

Symbolic expressions representing a range of mathematical objects and operations can also be manipulated exactly using Computer Algebra Systems (CAS), although such operations are almost always much slower than numerical computations using integer or real-valued numbers. These dualities between numerical and symbolic computation will be a key theme throughout the course.

1.1 Fixed-point numbers

In everyday life, we tend to use a **fixed point** representation of real numbers:

$$x = \pm(d_1 d_2 \cdots d_{k-1}.d_k \cdots d_n)_\beta, \quad \text{where} \quad d_1, \ldots, d_n \in \{0, 1, \ldots, \beta - 1\}.$$

Here β is the base (e.g. 10 for decimal arithmetic or 2 for binary).

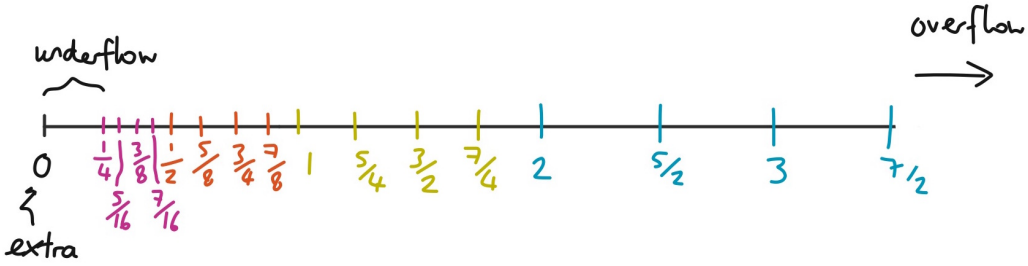
If we require that $d_1 \neq 0$ unless $k = 2$, then every number has a unique representation of this form, except for infinite trailing sequences of digits $\beta - 1$.

1.2 Floating-point numbers

Computers use a **floating-point** representation. Only numbers in a **floating-point number system** $F \subset \mathbb{R}$ can be represented exactly, where

$$F = \{ \pm (0.d_1 d_2 \cdots d_m)_\beta \beta^e \mid \beta, d_i, e \in \mathbb{Z},\ 0 \leq d_i \leq \beta - 1,\ e_{\min} \leq e \leq e_{\max} \}.$$

Here $(0.d_1 d_2 \cdots d_m)_\beta$ is called the **fraction** (or **significand** or **mantissa**), β is the base, and e is the **exponent**. This can represent a much larger range of numbers than a fixed-point system of the same size, although at the cost that the numbers are not equally spaced. If $d_1 \neq 0$ then each number in F has a unique representation and F is called **normalised**.



> **Note**
>
> Notice that the spacing between numbers jumps by a factor β at each power of β. The largest possible number is $(0.111)_2 2^2 = (\frac{1}{2} + \frac{1}{4} + \frac{1}{8})(4) = \frac{7}{2}$. The smallest non-zero number is $(0.100)_2 2^{-1} = \frac{1}{2}(\frac{1}{2}) = \frac{1}{4}$.

Here $\beta = 2$, and there are 52 bits for the fraction, 11 for the exponent, and 1 for the sign. The actual format used is

$$\pm(1.d_1 \cdots d_{52})_2 2^{e-1023} = \pm(0.1 d_1 \cdots d_{52})_2 2^{e-1022}, \quad e = (e_1 e_2 \cdots e_{11})_2.$$

When $\beta = 2$, the first digit of a normalized number is always 1, so doesn't need to be stored in memory. The **exponent bias** of 1022 means that the actual exponents are in the range -1022 to 1025, since $e \in [0, 2047]$. Actually the exponents -1022 and 1025 are used to store ± 0 and $\pm\infty$ respectively.

The smallest non-zero number in this system is $(0.1)_2 2^{-1021} \approx 2.225 \times 10^{-308}$, and the largest number is $(0.1 \cdots 1)_2 2^{1024} \approx 1.798 \times 10^{308}$.

> **Note**
>
> IEEE stands for Institute of Electrical and Electronics Engineers. Matlab uses the IEEE 754 standard for floating point arithmetic. The automatic 1 is sometimes called the "hidden bit". The exponent bias avoids the need to store the sign of the exponent.

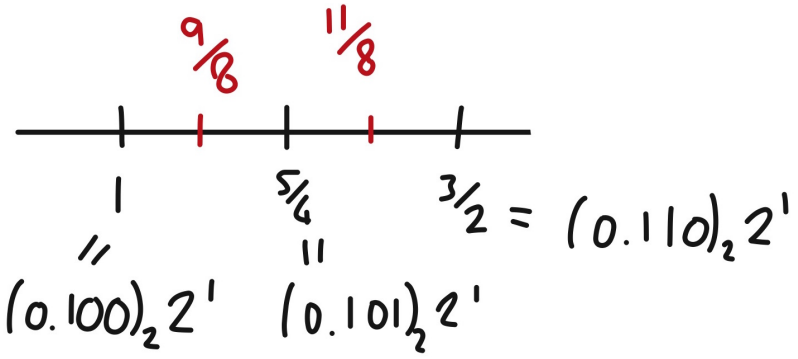
Numbers outside the finite set F cannot be represented exactly. If a calculation falls below the lower non-zero limit (in absolute value), it is called **underflow**, and usually set to 0. If it falls above the upper limit, it is called **overflow**, and usually results in a floating-point exception.

> **Note**
>
> **Ariane 5 rocket failure (1996):** The maiden flight ended in failure. Only 40 seconds after initiation, at altitude 3700m, the launcher veered off course and exploded. The cause was a software exception during data conversion from a 64-bit float to a 16-bit integer. The converted number was too large to be represented, causing an exception.

> **Note**
>
> In IEEE arithmetic, some numbers in the "zero gap" can be represented using $e = 0$, since only two possible fraction values are needed for ± 0. The other fraction values may be used with first (hidden) bit 0 to store a set of so-called **subnormal** numbers.

The mapping from $\mathbb{R}$ to F is called **rounding** and denoted $\mathrm{fl}(x)$. Usually it is simply the nearest number in F to x. If x lies exactly midway between two numbers in F, a method of breaking ties is required. The IEEE standard specifies *round to nearest even*—i.e., take the neighbour with last digit 0 in the fraction.

> **Note**
>
> This avoids statistical bias or prolonged drift.



$\frac{9}{8} = (1.001)_2$ has neighbours $1 = (0.100)_2 2^1$ and $\frac{5}{4} = (0.101)_2 2^1$, so is rounded down to 1.
$\frac{11}{8} = (1.011)_2$ has neighbours $\frac{5}{4} = (0.101)_2 2^1$ and $\frac{3}{2} = (0.110)_2 2^1$, so is rounded up to $\frac{3}{2}$.

i Note

Vancouver stock exchange index: In 1982, the index was established at 1000. By November 1983, it had fallen to 520, even though the exchange seemed to be doing well. Explanation: the index was rounded *down* to 3 digits at every recomputation. Since the errors were always in the same direction, they added up to a large error over time. Upon recalculation, the index doubled!

1.3 Significant figures

When doing calculations without a computer, we often use the terminology of **significant figures**. To count the number of significant figures in a number x, start with the first non-zero digit from the left, and count all the digits thereafter, including final zeros if they are after the decimal point.

To round x to n s.f., replace x by the nearest number with n s.f. An approximation $\hat{x}$ of x is "correct to n s.f." if both $\hat{x}$ and x round to the same number to n s.f.

1.4 Rounding error

There are two common ways of measuring the error of an approximation in numerical analysis that we will use throughout the course:

Definition 1.1: Absolute and relative errors

If x is the true value of a number and $\hat{x}$ is an approximation, we define the **absolute error** of this approximation as the quantity

$$|\hat{x} - x|,$$

while the **relative error** of the approximation is given by

$$\frac{|\hat{x} - x|}{|x|},$$

assuming $x \neq 0$.

If $|x|$ lies between the smallest non-zero number in F and the largest number in F, then

$$\mathrm{fl}(x) = x(1+\delta),$$

where the relative error incurred by rounding is

$$|\delta| = \frac{|\mathrm{fl}(x) - x|}{|x|}.$$

> **Note**
>
> Relative errors are often more useful because they are scale invariant. E.g., an error of 1 hour is irrelevant in estimating the age of a lecture theatre, but catastrophic in timing your arrival at the lecture.

Now x may be written as $x = (0.d_1 d_2 \cdots)_\beta \beta^e$ for some $e \in [e_{\min}, e_{\max}]$, but the fraction will not terminate after m digits if $x \notin F$. However, this fraction will differ from that of $\mathrm{fl}(x)$ by at most $\frac{1}{2}\beta^{-m}$, so

$$|\mathrm{fl}(x) - x| \leq \tfrac{1}{2}\beta^{-m}\beta^e \quad \Longrightarrow \quad |\delta| \leq \tfrac{1}{2}\beta^{1-m}.$$

Here we used that the fractional part of $|x|$ is at least $(0.1)_\beta \equiv \beta^{-1}$. The number $\epsilon_{\mathrm{M}} = \frac{1}{2}\beta^{1-m}$ is called the **machine epsilon** (or **unit roundoff**), and is independent of x. So the relative rounding error satisfies

$$|\delta| \leq \epsilon_{\mathrm{M}}.$$

> **Note**
>
> To check the machine epsilon value in Matlab you can just type 'eps' in the command line, which will return the value 2.2204e-16.

> **Note**
>
> The name "unit roundoff" arises because β^{1-m} is the distance between 1 and the next number in the system.

When adding/subtracting/multiplying/dividing two numbers in F, the result will not be in F in general, so must be rounded.

Let us multiply $x = \frac{5}{8}$ and $y = \frac{7}{8}$. We have

$$xy = \tfrac{35}{64} = \tfrac{1}{2} + \tfrac{1}{32} + \tfrac{1}{64} = (0.100011)_2.$$

This has too many significant digits to represent in our system, so the best we can do is round the result to $\mathrm{fl}(xy) = (0.100)_2 = \frac{1}{2}$.

> **Note**
>
> Typically additional digits are used during the computation itself, as in our example.

For $\circ = +, -, \times, \div$, IEEE standard arithmetic requires rounded exact operations, so that

$$\mathrm{fl}(x \circ y) = (x \circ y)(1+\delta), \quad |\delta| \leq \epsilon_{\mathrm{M}}.$$

1.5 Loss of significance

You might think that the above guarantees the accuracy of calculations to within ϵ_{M}, but this is true only if x and y are themselves exact. In reality, we are probably starting from $\bar{x} = x(1+\delta_1)$ and $\bar{y} = y(1+\delta_2)$, with $|\delta_1|, |\delta_2| \leq \epsilon_{\mathrm{M}}$. In that case, there is an error even before we round the result, since

$$\begin{aligned} \bar{x} \pm \bar{y} &= x(1+\delta_1) \pm y(1+\delta_2) \\ &= (x \pm y)\left(1 + \frac{x\delta_1 \pm y\delta_2}{x \pm y}\right). \end{aligned}$$

If the correct answer $x \pm y$ is very small, then there can be an arbitrarily large relative error in the result, compared to the errors in the initial $\bar{x}$ and $\bar{y}$. In particular, this relative error can be much larger than ϵ_{M}. This is called **loss of significance**, and is a major cause of errors in floating-point calculations.

To 4 s.f., the roots are

$$x_1 = 28 + \sqrt{783} = 55.98, \quad x_2 = 28 - \sqrt{783} = 0.01786.$$

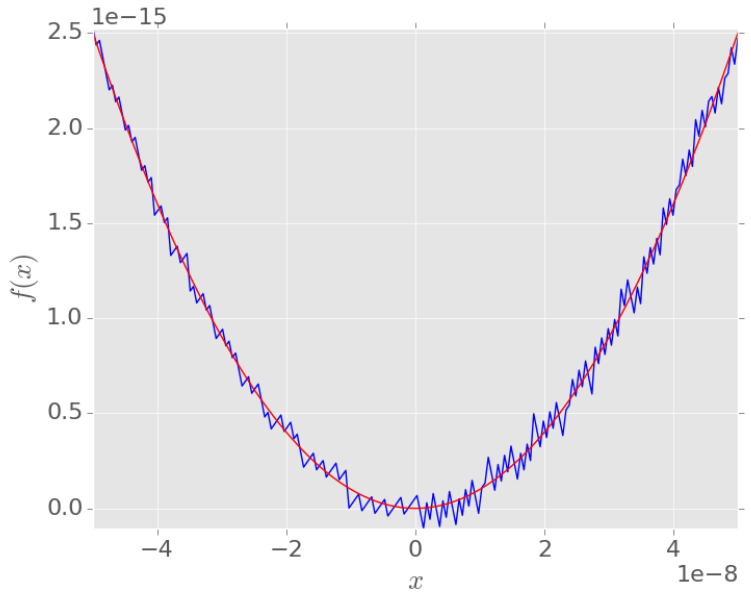
However, working to 4 s.f. we would compute $\sqrt{783} = 27.98$, which would lead to the results

$$\bar{x}_1 = 55.98, \quad \bar{x}_2 = 0.02000.$$

The smaller root is not correct to 4 s.f., because of cancellation error. One way around this is to note that $x^2 - 56x + 1 = (x - x_1)(x - x_2)$, and compute x_2 from $x_2 = 1/x_1$, which gives the correct answer.

> **Note**
>
> Note that the error crept in when we rounded $\sqrt{783}$ to 27.98, because this removed digits that would otherwise have been significant after the subtraction.

Let us plot this function in the range $-5 \times 10^{-8} \leq x \leq 5 \times 10^{-8}$ – even in IEEE double precision arithmetic we find significant errors, as shown by the blue curve:



The red curve shows the correct result approximated using the Taylor series

$$\begin{aligned} f(x) &= \left(1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots\right) - \left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots\right) - x \\ &\approx x^2 + \frac{x^3}{6}. \end{aligned}$$

This avoids subtraction of nearly equal numbers.

> **Note**
>
> We will look in more detail at polynomial approximations in the next section.

Note that floating-point arithmetic violates many of the usual rules of real arithmetic, such as $(a + b) + c = a + (b + c)$.

$$\begin{aligned} \text{fl}\big[(5.9 + 5.5) + 0.4\big] &= \text{fl}\big[\text{fl}(11.4) + 0.4\big] = \text{fl}(11.0 + 0.4) = 11.0, \\ \text{fl}\big[5.9 + (5.5 + 0.4)\big] &= \text{fl}\big[5.9 + 5.9\big] = \text{fl}(11.8) = 12.0. \end{aligned}$$

In $\mathbb{R}$, the average of two numbers always lies between the numbers. But if we work to 3 decimal digits,

$$\text{fl}\left(\frac{5.01 + 5.02}{2}\right) = \frac{\text{fl}(10.03)}{2} = \frac{10.0}{2} = 5.0.$$

The moral of the story is that sometimes care is needed to ensure that we carry out a calculation accurately and as intended!

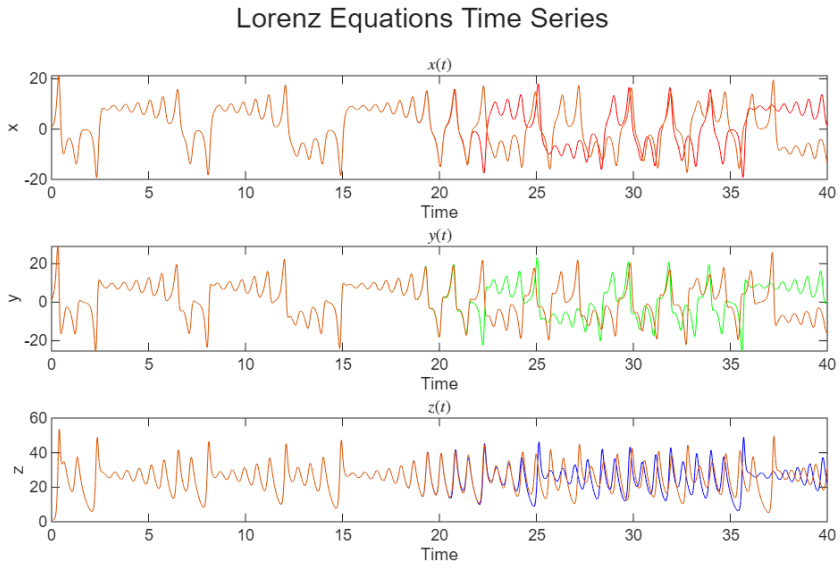
1.6 Example: Weather Modelling

A very simplistic description of an atmospheric fluid is given by the Lorenz system of differential equations:

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x),\\ \frac{dy}{dt} &= x(\rho - z) - y,\\ \frac{dz}{dt} &= xy - \beta z,\end{aligned}$$

where σ, ρ, and β are positive constants

We will describe how to solve such equations numerically in Chapter 5. For now, let's look at a simulation of these equations computed by just iterating a bunch of addition and multiplication steps:



Here we plot all three variables over a particular window of time (loosely corresponding to something between minutes to hours, depending on several details of the physics we are neglecting). For each variable, we have ran two simulations with two different values of β. One simulation has $\beta = 8/3 = 2.66666666666\ldots$ and the other has $\beta = 8/3 + 10^{-10} = 2.66666666676\ldots$. Nevertheless, you can clearly see that this tiny difference in parameters (well below what is experimentally measurable) has a huge impact on the solution. The reason for this, and a key reason why weather is fundamentally hard to predict, is the existence of chaos in many models of physical phenomena. However truncation errors and floating point errors also contribute to the discrepancies in this simulation.

The overall lesson of this Chapter so far is that small errors can exist due to analytically-understood reasons such as the *truncation error* in a Taylor series approximation, or the *floating point error* in the numerical methods used. Such errors can not only impact individual calculations, but they can accumulate and entirely change outcomes of simulations, as we will see throughout the course. Nevertheless, computational methods still underly an enormous range of scientific fields, so understanding these sources of error (and how they can become amplified) is a central theme of this course.

1.7 Exact and symbolic computing

Symbolic computations require different data types from numerical ones; in MATLAB we can use the Symbolic Math Toolbox. In the following chapters, we will compare symbolic and numerical approaches to solving mathematical problems. One key difference is that symbolic computations are exact, but much more expensive to scale up to solve larger problems; for example, we will tackle numerical problems involving matrices of size $10^4 \times 10^4$ or larger, which would not be possible to successfully manipulate symbolically on a modern computer.

Symbolic computations are used to check or do laborious analytical work, but also to rigorously prove mathematical Theorems which would be too onerous to carry out by hand. FOr instance, while Lorenz equations above were first noted to have strange dependencies on initial conditions, it was only in 2002 that this was mathematically proved to be a chaotic system. Among other tools, the proof relied on interval arithmetic, which is a method to exactly operate on intervals defined in terms of two rational numbers.

Knowledge checklist

Key topics:

1. Integer and floating point representations of real numbers on computers.
2. Overflow, underflow and loss of significance.
3. Symbolic and numerical representations.

Key skills:

- Understanding and distinguishing integer, fixed-point, and floating-point representations.
- Analyzing the effects of rounding and machine epsilon in calculations.
- Diagnosing and managing rounding errors, overflow, and underflow.

2 Continuous Functions

The goal of this chapter is to explore and begin to answer the following question:

How do we represent and manipulate continuous functions on a computer?

2.1 Interpolation

2.1.1 Polynomial Interpolation: Motivation

The main idea of this section is to find a polynomial that approximates a general function f. But why polynomials? Polynomials have many nice mathematical properties but from the perspective of function approximation, the key one is the following: Any continuous function on a compact interval can be approximated to arbitrary accuracy using a polynomial (provided you are willing to go high enough degree).

Theorem 2.1: Weierstrass Approximation Theorem (1885)

For any $f \in C([0,1])$ and any $\epsilon > 0$, there exists a polynomial $p(x)$ such that

$$\max_{0 \leq x \leq 1} |f(x) - p(x)| \leq \epsilon.$$

Note

This may be proved using an explicit sequence of polynomials, called Bernstein polynomials.

If f is a polynomial of degree n,

$$f(x) = p_n(x) = a_0 + a_1 x + \ldots + a_n x^n,$$

then we only need to store the $n+1$ coefficients $a_0, \ldots, a_n$. Operations such as taking the derivative or integrating f are also convenient. If f is not continuous, then something other than a polynomial is required, since polynomials can't handle asymptotic behaviour.

> **Note**
>
> To approximate functions like $1/x$, there is a well-developed theory of rational function interpolation, which is beyond the scope of this course.

In this chapter, we look for a suitable polynomial p_n by **interpolation**—that is, requiring $p_n(x_i) = f(x_i)$ at a finite set of points x_i, usually called **nodes**. Sometimes we will also require the derivative(s) of p_n to match those of f. This type of function approximation where we want to match values of the function that we know at particular points is very natural in many applications. For example, weather forecasts involve numerically solving huge systems of partial differential equations (PDEs), which means actually solving them on a discrete grid of points. If we want weather predictions between grid points, we must **interpolate**. Figure 2.1 shows the spatial resolutions of a range of current and past weather models produced by the UK Met Office.

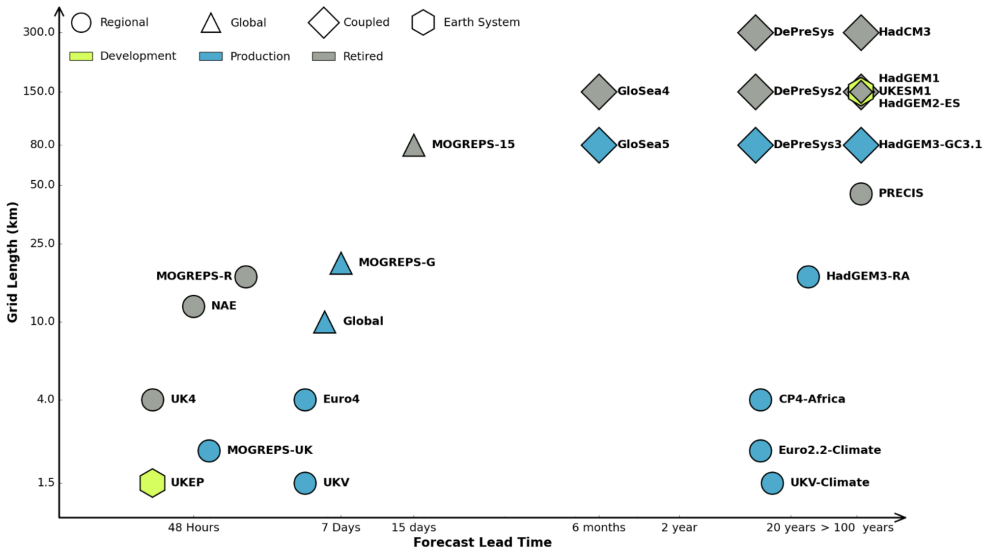


Figure 2.1: Chart showing a range of weather models produce by the UK Met Office. Even the highest spatial resolution models have more than 1.5km between grid point due to computational constraints.

2.1.2 Taylor series

A truncated Taylor series is (in some sense) the simplest interpolating polynomial since it uses only a single node x_0, although it does require p_n to match both f and some of its derivatives.

We can approximate this using a Taylor series about the point $x_0 = 0$, which is

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots.$$

This comes from writing

$$f(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \dots,$$

then differentiating term-by-term and matching values at x_0:

$$\begin{aligned}
f(x_0) &= a_0, \\
f'(x_0) &= a_1, \\
f''(x_0) &= 2a_2, \\
f'''(x_0) &= 3(2)a_3, \\
&\vdots \\
\implies f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \dots.
\end{aligned}$$

So

$$\begin{aligned}
\text{1 term} &\implies f(0.1) \approx 0.1, \\
\text{2 terms} &\implies f(0.1) \approx 0.1 - \frac{0.1^3}{6} = 0.099833\dots, \\
\text{3 terms} &\implies f(0.1) \approx 0.1 - \frac{0.1^3}{6} + \frac{0.1^5}{120} = 0.09983341\dots.
\end{aligned}$$

The next term will be $-0.1^7/7! \approx -10^{-7}/10^3 = -10^{-10}$, which won't change the answer to 6 s.f.

> **Note**
>
> The exact answer is $\sin(0.1) = 0.09983341\dots$.

Mathematically, we can write the remainder as follows.

Theorem 2.2: Taylor's Theorem

Let f be $n+1$ times differentiable on (a, b), and let $f^{(n)}$ be continuous on $[a, b]$. If $x, x_0 \in [a, b]$ then there exists $\xi \in (a, b)$ such that

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}.$$

The sum is called the **Taylor polynomial** of degree n, and the last term is called the **Lagrange form** of the remainder. Note that the unknown number ξ depends on x.

For $f(x) = \sin(x)$, we found the Taylor polynomial $p_6(x) = x - x^3/3! + x^5/5!$, and $f^{(7)}(x) = -\sin(x)$. So we have

$$|f(x) - p_6(x)| = \left| \frac{f^{(7)}(\xi)}{7!}(x - x_0)^7 \right|$$

for some ξ between x_0 and x. For $x = 0.1$, we have

$$|f(0.1) - p_6(0.1)| = \frac{1}{5040}(0.1)^7 |f^{(7)}(\xi)| \quad \text{for some } \xi \in [0, 0.1].$$

Since $|f^{(7)}(\xi)| = |\sin(\xi)| \leq 1$, we can say, before calculating, that the error satisfies

$$|f(0.1) - p_6(0.1)| \leq 1.984 \times 10^{-11}.$$

> **Note**
>
> The actual error is 1.983×10^{-11}, so this is a tight estimate.

Since this error arises from approximating f with a truncated series, rather than due to rounding, it is known as **truncation error**. Note that it tends to be lower if you use more terms (larger n), or if the function oscillates less (smaller $f^{(n+1)}$ on the interval (x_0, x)).

Error estimates like the Lagrange remainder play an important role in numerical analysis and computation, so it is important to understand where it comes from. The number ξ will ultimately come from Rolle's theorem, which is a special case of the mean value theorem from first-year calculus:

Theorem 2.3: Rolle's Theorem

If f is continuous on $[a, b]$ and differentiable on (a, b), with $f(a) = f(b) = 0$, then there exists $\xi \in (a, b)$ with $f'(\xi) = 0$.

> **Note**
>
> Note that Rolle's Theorem does not tell us what the value of ξ might actually be, so in practice we must take some kind of worst case estimate to get an error bound, e.g. calculate the max value of $f'(\xi)$ over the range of possible ξ values.

2.1.3 Polynomial Interpolation

The classical problem of **polynomial interpolation** is to find a polynomial

$$p_n(x) = a_0 + a_1 x + \ldots + a_n x^n = \sum_{k=0}^{n} a_k x^k$$

that interpolates our function f at a finite set of nodes $\{x_0, x_1, \ldots, x_m\}$. In other words, $p_n(x_i) = f(x_i)$ at each of the nodes x_i. Since the polynomial has $n+1$ unknown coefficients, we expect to need $n+1$ distinct nodes, so let us assume that $m = n$.

Here we have two nodes x_0, x_1, and seek a polynomial $p_1(x) = a_0 + a_1 x$. Then the interpolation conditions require that

$$\begin{cases} p_1(x_0) = a_0 + a_1 x_0 = f(x_0) \\ p_1(x_1) = a_0 + a_1 x_1 = f(x_1) \end{cases} \quad \Longrightarrow \quad p_1(x) = \frac{x_1 f(x_0) - x_0 f(x_1)}{x_1 - x_0} + \frac{f(x_1) - f(x_0)}{x_1 - x_0} x.$$

For general n, the interpolation conditions require

$$\begin{array}{ccccccc} a_0 & +a_1 x_0 & +a_2 x_0^2 & +\ldots & +a_n x_0^n & = f(x_0), \\ a_0 & +a_1 x_1 & +a_2 x_1^2 & +\ldots & +a_n x_1^n & = f(x_1), \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ a_0 & +a_1 x_n & +a_2 x_n^2 & +\ldots & +a_n x_n^n & = f(x_n), \end{array}$$

so we have to solve

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}.$$

This is called a **Vandermonde matrix**. The determinant of this matrix is

$$\det(A) = \prod_{0 \le i < j \le n} (x_j - x_i),$$

which is non-zero provided the nodes are all distinct. This establishes an important result, where $\mathcal{P}_n$ denotes the space of all real polynomials of degree $\le n$.

Theorem 2.4: Existence/uniqueness

Given $n+1$ distinct nodes $x_0, x_1, \ldots, x_n$, there is a unique polynomial $p_n \in \mathcal{P}_n$ that interpolates $f(x)$ at these nodes.

We may also prove uniqueness by the following elegant argument.

Proof (Uniqueness part of Existence/Uniqueness Theorem):
Suppose that in addition to p_n there is another interpolating polynomial $q_n \in \mathcal{P}_n$. Then the difference $r_n := p_n - q_n$ is also a polynomial with degree $\leq n$. But we have

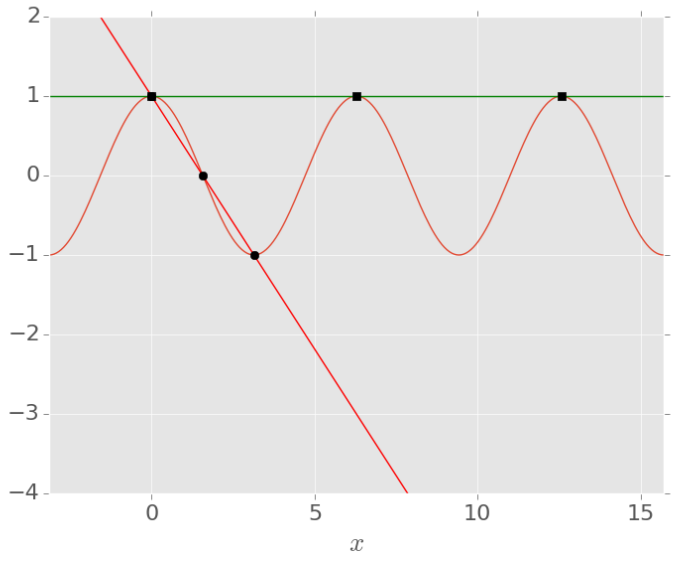
$$r_n(x_i) = p_n(x_i) - q_n(x_i) = f(x_i) - f(x_i) = 0 \quad \text{for } i = 0, \ldots, n,$$

so $r_n(x)$ has $n+1$ roots. From the Fundamental Theorem of Algebra, this is possible only if $r_n(x) \equiv 0$, which implies that $q_n = p_n$.

> **Note**
>
> Note that the unique polynomial through $n+1$ points may have degree $< n$. This happens when $a_0 = 0$ in the solution to the Vandermonde system above.

Example 2.1: Interpolate $f(x) = \cos(x)$ with $p_2 \in \mathcal{P}_2$ at the nodes $\{0, \frac{\pi}{2}, \pi\}$.

We have $x_0 = 0$, $x_1 = \frac{\pi}{2}$, $x_2 = \pi$, so $f(x_0) = 1$, $f(x_1) = 0$, $f(x_2) = -1$. Clearly the unique interpolant is a straight line $p_2(x) = 1 - \frac{2}{\pi}x$.
If we took the nodes $\{0, 2\pi, 4\pi\}$, we would get a constant function $p_2(x) = 1$.



One way to compute the interpolating polynomial would be to solve the Vandermonde system above, e.g. by Gaussian elimination. However, this is not recommended. In practice, we choose a different basis for p_n; there are two common and effective choices due to Lagrange and Newton.

> **Note**
>
> The Vandermonde matrix arises when we write p_n in the **natural basis** $\{1, x, x^2, \ldots\}$, but we could also choose to work in some other basis...

2.1.4 Lagrange Polynomials

This uses a special basis of polynomials $\{\ell_k\}$ in which the interpolation equations reduce to the identity matrix. In other words, the coefficients in this basis are just the function values,

$$p_n(x) = \sum_{k=0}^{n} f(x_k)\ell_k(x).$$

Example 2.2: Linear interpolation again.

We can re-write our linear interpolant to separate out the function values:

$$p_1(x) = \underbrace{\frac{x - x_1}{x_0 - x_1}}_{\ell_0(x)} f(x_0) + \underbrace{\frac{x - x_0}{x_1 - x_0}}_{\ell_1(x)} f(x_1).$$

Then ℓ_0 and ℓ_1 form the necessary basis. In particular, they have the property that

$$\ell_0(x_i) = \begin{cases} 1 & \text{if } i = 0, \\ 0 & \text{if } i = 1, \end{cases} \qquad \ell_1(x_i) = \begin{cases} 0 & \text{if } i = 0, \\ 1 & \text{if } i = 1, \end{cases}$$

For general n, the $n + 1$ **Lagrange polynomials** are defined as a product

$$\ell_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j}.$$

By construction, they have the property that

$$\ell_k(x_i) = \begin{cases} 1 & \text{if } i = k, \\ 0 & \text{otherwise.} \end{cases}$$

From this, it follows that the interpolating polynomial may be written as above.

Note

By the Existence/Uniqueness Theorem, the Lagrange polynomials are the *unique* polynomials with this property.

Example 2.3: Compute the quadratic interpolating polynomial to $f(x) = \cos(x)$ with nodes $\{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}$ using Lagrange polynomials.

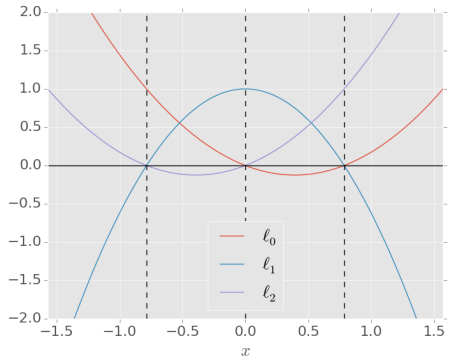
The Lagrange polynomials of degree 2 for these nodes are

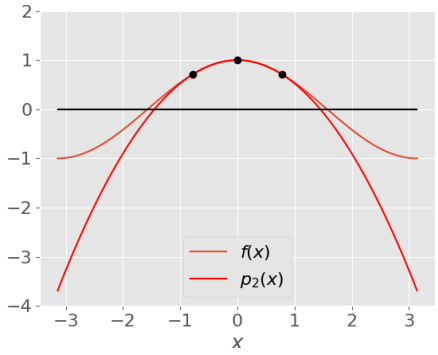
$$\begin{aligned}
\ell_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{x(x-\frac{\pi}{4})}{\frac{\pi}{4}\cdot\frac{\pi}{2}}, \\
\ell_1(x) &= \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} = \frac{(x+\frac{\pi}{4})(x-\frac{\pi}{4})}{-\frac{\pi}{4}\cdot\frac{\pi}{4}}, \\
\ell_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \\
&= \frac{x(x+\frac{\pi}{4})}{\frac{\pi}{2}\cdot\frac{\pi}{4}}.
\end{aligned}$$

So the interpolating polynomial is

$$\begin{aligned}
p_2(x) &= f(x_0)\ell_0(x) + f(x_1)\ell_1(x) + f(x_2)\ell_2(x) \\
&= \tfrac{1}{\sqrt{2}}\tfrac{8}{\pi^2}x(x-\tfrac{\pi}{4}) - \tfrac{16}{\pi^2}(x+\tfrac{\pi}{4})(x-\tfrac{\pi}{4}) + \tfrac{1}{\sqrt{2}}\tfrac{8}{\pi^2}x(x+\tfrac{\pi}{4}) = \tfrac{16}{\pi^2}(\tfrac{1}{\sqrt{2}}-1)x^2 + 1.
\end{aligned}$$

The Lagrange polynomials and the resulting interpolant are shown below:





Note

Lagrange polynomials were actually discovered by Edward Waring in 1776 and rediscovered by Euler in 1783, before they were published by Lagrange himself in 1795; a classic example

of Stigler's law of eponymy!

The Lagrange form of the interpolating polynomial is easy to write down, but expensive to evaluate since all of the ℓ_k must be computed. Moreover, changing any of the nodes means that the ℓ_k must all be recomputed from scratch, and similarly for adding a new node (moving to higher degree).

2.1.5 Newton/Divided-Difference Polynomials

It would be easy to increase the degree of p_n if

$$p_{n+1}(x) = p_n(x) + g_{n+1}(x), \quad \text{where } g_{n+1} \in \mathcal{P}_{n+1}.$$

From the interpolation conditions, we know that

$$g_{n+1}(x_i) = p_{n+1}(x_i) - p_n(x_i) = f(x_i) - f(x_i) = 0 \quad \text{for } i = 0, \ldots, n,$$

so

$$g_{n+1}(x) = a_{n+1}(x - x_0) \cdots (x - x_n).$$

The coefficient a_{n+1} is determined by the remaining interpolation condition at x_{n+1}, so

$$p_n(x_{n+1}) + g_{n+1}(x_{n+1}) = f(x_{n+1}) \quad \Longrightarrow \quad a_{n+1} = \frac{f(x_{n+1}) - p_n(x_{n+1})}{(x_{n+1} - x_0) \cdots (x_{n+1} - x_n)}.$$

The polynomial $(x - x_0)(x - x_1) \cdots (x - x_n)$ is called a **Newton polynomial**. These form a new basis

$$n_0(x) = 1, \qquad n_k(x) = \prod_{j=0}^{k-1} (x - x_j) \quad \text{for } k > 0.$$

The **Newton form** of the interpolating polynomial is then

$$p_n(x) = \sum_{k=0}^{n} a_k n_k(x), \qquad a_0 = f(x_0), \qquad a_k = \frac{f(x_k) - p_{k-1}(x_k)}{(x_k - x_0) \cdots (x_k - x_{k-1})} \text{ for } k > 0.$$

Notice that a_k depends only on $x_0, \ldots x_k$, so we can construct first a_0, then a_1, etc.

It turns out that the a_k are easy to compute, but it will take a little work to derive the method. We define the **divided difference** $f[x_0, x_1, \ldots, x_k]$ to be the coefficient of x^k in the polynomial interpolating f at nodes $x_0, \ldots, x_k$. It follows that

$$f[x_0, x_1, \ldots, x_k] = a_k,$$

where a_k is the coefficient in the Newton form above.

Example 2.4: Compute the Newton interpolating polynomial at two nodes.

$$f[x_0] = a_0 = f(x_0),$$
$$f[x_0, x_1] = a_1 = \frac{f(x_1) - p_0(x_1)}{x_1 - x_0} = \frac{f(x_1) - a_0}{x_1 - x_0} = \frac{f[x_1] - f[x_0]}{x_1 - x_0}.$$

So the **first-order** divided difference $f[x_0, x_1]$ is obtained from the **zeroth-order** differences $f[x_0]$, $f[x_1]$ by subtracting and dividing, hence the name "divided difference".

Example 2.5: Compute the Newton interpolating polynomial at three nodes.

Continuing from the previous example, we find

$$\begin{aligned} f[x_0, x_1, x_2] = a_2 &= \frac{f(x_2) - p_1(x_2)}{(x_2 - x_0)(x_2 - x_1)} = \frac{f(x_2) - a_0 - a_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)} \\ &= \ldots = \frac{1}{x_2 - x_0}\left(\frac{f[x_2] - f[x_1]}{x_2 - x_1} - \frac{f[x_1] - f[x_0]}{x_1 - x_0}\right) \\ &= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}. \end{aligned}$$

So again, we subtract and divide.

In general, we have the following.

Theorem 2.5

For $k > 0$, the divided differences satisfy

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}.$$

Proof:
Without loss of generality, we relabel the nodes so that $i = 0$. So we want to prove that

$$f[x_0, x_1, \ldots, x_k] = \frac{f[x_1, \ldots, x_k] - f[x_0, \ldots, x_{k-1}]}{x_k - x_0}.$$

The trick is to write the interpolant with nodes $x_0, \ldots, x_k$ in the form

$$p_k(x) = \frac{(x_k - x)q_{k-1}(x) + (x - x_0)\tilde{q}_{k-1}(x)}{x_k - x_0},$$

where $q_{k-1} \in \mathcal{P}_{k-1}$ interpolates f at the subset of nodes $x_0, x_1, \ldots, x_{k-1}$ and $\tilde{q}_{k-1} \in \mathcal{P}_{k-1}$ interpolates f at the subset $x_1, x_2, \ldots, x_k$. If this holds, then matching the coefficient of x^k on each side will give the divided difference formula, since, e.g., the leading coefficient of q_{k-1} is

$f[x_0, \ldots, x_{k-1}]$. To see that p_k may really be written this way, note that

$$\begin{aligned} p_k(x_0) &= q_{k-1}(x_0) = f(x_0), \\ p_k(x_k) &= \tilde{q}_{k-1}(x_k) = f(x_k), \\ p_k(x_i) &= \frac{(x_k - x_i)q_{k-1}(x_i) + (x_i - x_0)\tilde{q}_{k-1}(x_i)}{x_k - x_0} = f(x_i) \quad \text{for } i = 1, \ldots, k-1. \end{aligned}$$

Since p_k agrees with f at the $k+1$ nodes, it is the unique interpolant in $\mathcal{P}_k$.

Theorem above gives us our convenient method, which is to construct a **divided-difference table**.

Example 2.6: Construct the Newton polynomial at the nodes $\{-1, 0, 1, 2\}$ and with corresponding function values $\{5, 1, 1, 11\}$

We construct a divided-difference table as follows.

$$\begin{array}{llllll} x_0 = -1 & f[x_0] = 5 & & & & \\ & & f[x_0, x_1] = -4 & & & \\ x_1 = 0 & f[x_1] = 1 & & f[x_0, x_1, x_2] = 2 & & \\ & & f[x_1, x_2] = 0 & & f[x_0, x_1, x_2, x_3] = 1 & \\ x_2 = 1 & f[x_2] = 1 & & f[x_1, x_2, x_3] = 5 & & \\ & & f[x_2, x_3] = 10 & & & \\ x_3 = 2 & f[x_3] = 11 & & & & \end{array}$$

The coefficients of the p_3 lie at the top of each column, so

$$\begin{aligned} p_3(x) &= f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) \\ &\quad + f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2) \\ &= 5 - 4(x + 1) + 2x(x + 1) + x(x + 1)(x - 1). \end{aligned}$$

Now suppose we add the extra nodes $\{-2, 3\}$ with data $\{5, 35\}$. All we need to do to compute p_5 is add two rows to the bottom of the table — there is no need to recalculate the rest. This gives

$$\begin{array}{llllll} -1 & 5 & & & & \\ & & -4 & & & \\ 0 & 1 & & 2 & & \\ & & 0 & & 1 & \\ 1 & 1 & & 5 & & -\frac{1}{12} \\ & & 10 & & \frac{13}{12} & \\ 2 & 11 & & \frac{17}{6} & & -\frac{1}{12} \\ & & \frac{3}{2} & & \frac{5}{6} & \\ -2 & 5 & & \frac{9}{2} & & \\ & & 6 & & & \\ 3 & 35 & & & & \end{array} \qquad 0$$

The new interpolating polynomial is

$$p_5(x) = p_3(x) - \tfrac{1}{12}x(x+1)(x-1)(x-2).$$

Note

Notice that the x^5 coefficient vanishes for these particular data, meaning that they are consistent with $f \in \mathcal{P}_4$.

Note

Note that the value of $f[x_0, x_1, \ldots, x_k]$ is independent of the order of the nodes in the table. This follows from the uniqueness of p_k.

Divided differences are actually approximations for *derivatives* of f. In the limit that the nodes all coincide, the Newton form of $p_n(x)$ becomes the Taylor polynomial.

2.1.6 Interpolation Error

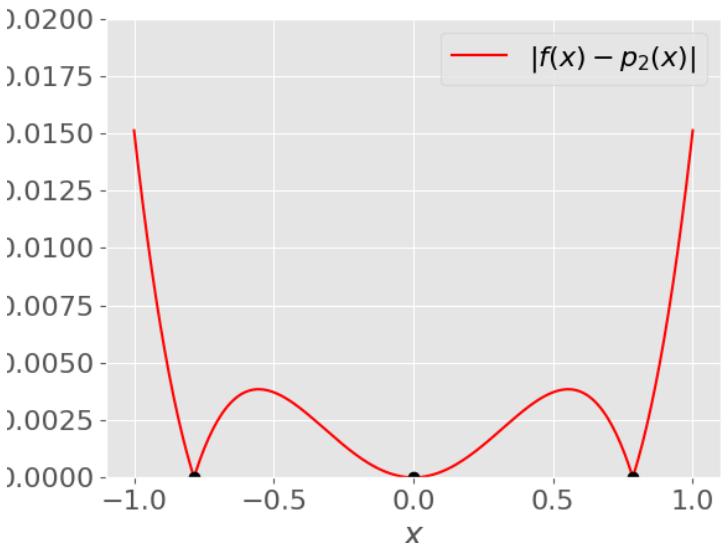
The goal here is to estimate the error $|f(x) - p_n(x)|$ when we approximate a function f by a polynomial interpolant p_n. Clearly this will depend on x.

Example 2.7: Quadratic interpolant for $f(x) = \cos(x)$ with $\{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}$.

From Section 2.1.4, we have $p_2(x) = \frac{16}{\pi^2}\left(\frac{1}{\sqrt{2}} - 1\right)x^2 + 1$, so the error is

$$|f(x) - p_2(x)| = \left|\cos(x) - \tfrac{16}{\pi^2}\left(\tfrac{1}{\sqrt{2}} - 1\right)x^2 - 1\right|.$$

This is shown below:



Clearly the error vanishes at the nodes themselves, but note that it generally does better near the middle of the set of nodes — this is quite typical behaviour.

We can adapt the proof of Taylor's theorem to get a quantitative error estimate.

Theorem 2.6: Cauchy's Interpolation Error Theorem

Let $p_n \in \mathcal{P}_n$ be the unique polynomial interpolating $f(x)$ at the $n+1$ distinct nodes $x_0, x_1, \ldots, x_n \in [a,b]$, and let f be continuous on $[a,b]$ with $n+1$ continuous derivatives on (a,b). Then for each $x \in [a,b]$ there exists $\xi \in (a,b)$ such that

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n).$$

This looks similar to the error formula for Taylor polynomials (see Taylor's Theorem). But now the error vanishes at multiple nodes rather than just at x_0.

From the formula, you can see that the error will be larger for a more "wiggly" function, where the derivative $f^{(n+1)}$ is larger. It might also appear that the error will go down as the number of nodes n increases; we will see in Section 2.1.7 that this is not always true.

Note

As in Taylor's theorem, note the appearance of an undetermined point ξ. This will prevent us knowing the error exactly, but we can make an estimate as before.

Example 2.8: Quadratic interpolant for $f(x) = \cos(x)$ with $\{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}$.

For $n = 2$, Cauchy's Interpolation Error Theorem says that

$$f(x) - p_2(x) = \frac{f^{(3)}(\xi)}{6} x(x + \tfrac{\pi}{4})(x - \tfrac{\pi}{4}) = \tfrac{1}{6} \sin(\xi) x(x + \tfrac{\pi}{4})(x - \tfrac{\pi}{4}),$$

for some $\xi \in [-\frac{\pi}{4}, \frac{\pi}{4}]$.
For an upper bound on the error at a particular x, we can just use $|\sin(\xi)| \leq 1$ and plug in x.
To bound the maximum error within the interval $[-1, 1]$, let us maximise the polynomial $w(x) = x(x + \frac{\pi}{4})(x - \frac{\pi}{4})$. We have $w'(x) = 3x^2 - \frac{\pi^2}{16}$ so turning points are at $x = \pm\frac{\pi}{4\sqrt{3}}$. We have

$$w(-\tfrac{\pi}{4\sqrt{3}}) = 0.186\ldots, \quad w(\tfrac{\pi}{4\sqrt{3}}) = -0.186\ldots, \quad w(-1) = -0.383\ldots, \quad w(1) = 0.383\ldots.$$

So our error estimate for $x \in [-1, 1]$ is

$$|f(x) - p_2(x)| \leq \tfrac{1}{6}(0.383) = 0.0638\ldots$$

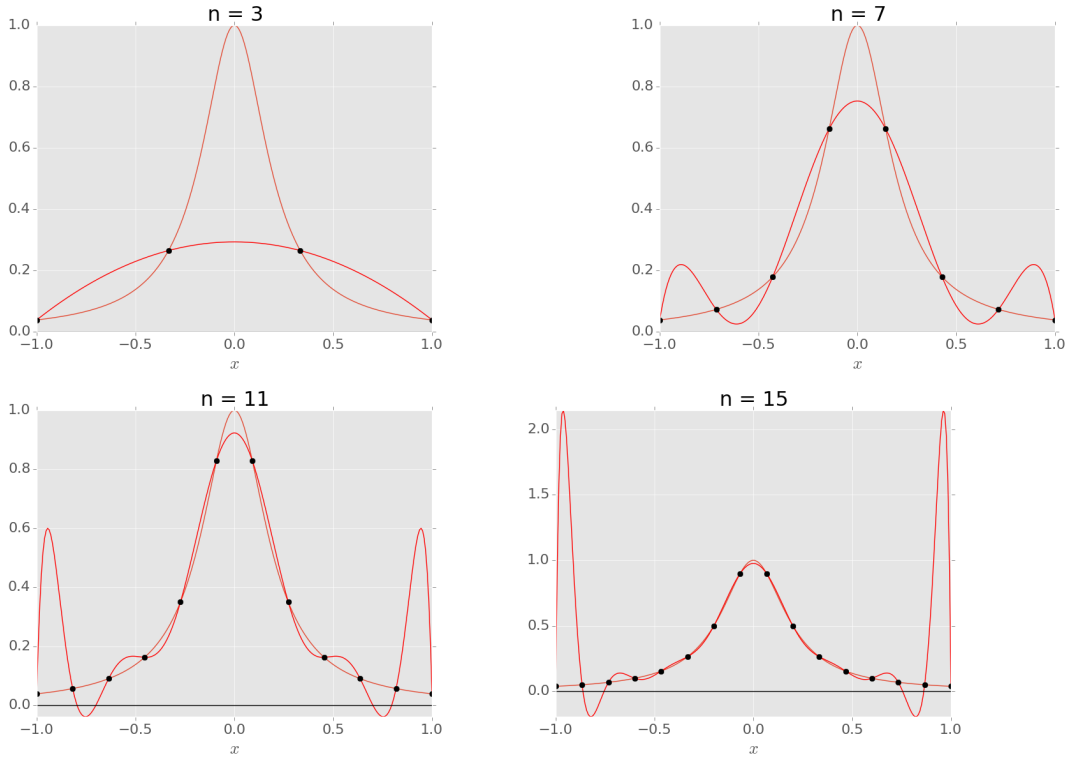
From the plot earlier, we see that this bound is satisfied (as it has to be), although not tight.

2.1.7 Node Placement: Chebyshev nodes

You might expect polynomial interpolation to *converge* as $n \to \infty$. Surprisingly, this is not the case if you take **equally-spaced** nodes x_i. This was shown by Runge in a famous 1901 paper.

Example 2.9: The Runge function $f(x) = 1/(1 + 25x^2)$ on $[-1, 1]$.

Here are illustrations of p_n for increasing n:



Notice that p_n is converging to f in the middle, but diverging more and more near the ends, even within the interval $[x_0, x_n]$. This is called the **Runge phenomenon**.

Note

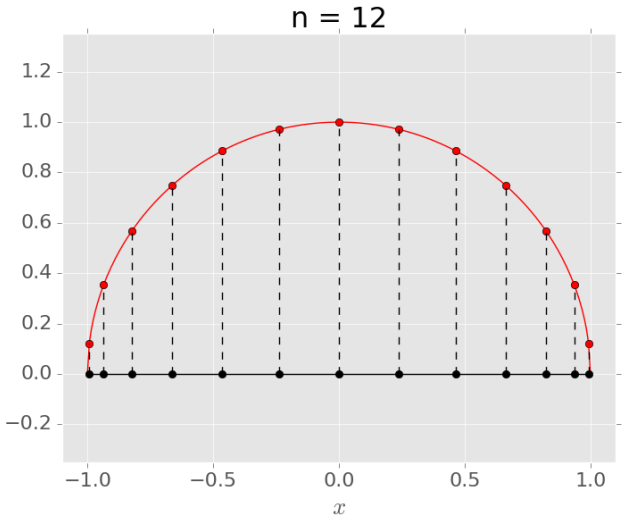
A full mathematical explanation for this divergence usually uses complex analysis — see Chapter 13 of *Approximation Theory and Approximation Practice* by L.N. Trefethen (SIAM, 2013). For a more elementary proof, see this StackExchange post.

The problem is (largely) coming from the interpolating polynomial

$$w(x) = \prod_{i=0}^{n} (x - x_i).$$

We can avoid the Runge phenomenon by choosing different nodes x_i that are **not uniformly spaced**.

Since the problems are occurring near the ends of the interval, it would be logical to put more nodes there. A good choice is given by taking equally-spaced points on the unit circle $|z| = 1$, and projecting to the real line:



The points around the circle are

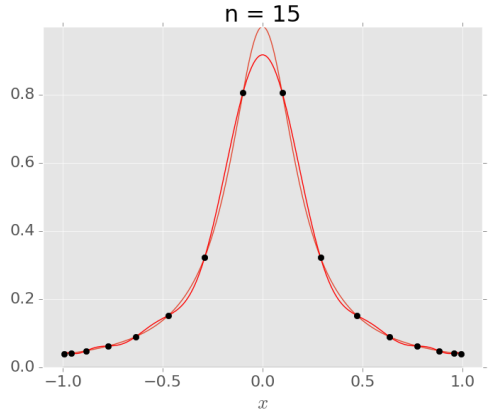
$$\phi_j = \frac{(2j+1)\pi}{2(n+1)}, \quad j = 0, \ldots, n,$$

so the corresponding **Chebyshev nodes** are

$$x_j = \cos\left[\frac{(2j+1)\pi}{2(n+1)}\right], \quad j = 0, \ldots, n.$$

Example 2.10: The Runge function $f(x) = 1/(1+25x^2)$ on $[-1, 1]$ using the Chebyshev nodes.

For $n = 3$, the nodes are $x_0 = \cos(\frac{\pi}{8})$, $x_1 = \cos(\frac{3\pi}{8})$, $x_2 = \cos(\frac{5\pi}{8})$, $x_3 = \cos(\frac{7\pi}{8})$. Below we illustrate the resulting interpolant for $n = 15$:



Compare this to the example with equally spaced nodes.

In fact, the Chebyshev nodes are, in one sense, an optimal choice. To see this, we first note that they are zeroes of a particular polynomial.

The Chebyshev points $x_j = \cos\left[\frac{(2j+1)\pi}{2(n+1)}\right]$ for $j = 0, \ldots, n$ are zeroes of the Chebyshev polynomial

$$T_{n+1}(t) := \cos\left[(n+1)\arccos(t)\right]$$

> **Note**
>
> The Chebyshev polynomials are denoted T_n rather than C_n because the name is transliterated from Russian as "Tchebychef" in French, for example.

In choosing the Chebyshev nodes, we are choosing the error polynomial $w(x) := \prod_{i=0}^{n}(x - x_i)$ to be $T_{n+1}(x)/2^n$. (This normalisation makes the leading coefficient 1) This is a good choice because of the following result.

Theorem 2.7: Chebyshev interpolation

Let $x_0, x_1, \ldots, x_n \in [-1, 1]$ be distinct. Then $\max_{[-1,1]} |w(x)|$ is minimized if

$$w(x) = \frac{1}{2^n} T_{n+1}(x),$$

where $T_{n+1}(x)$ is the **Chebyshev polynomial** $T_{n+1}(x) = \cos\Big((n+1)\arccos(x)\Big)$.

Having established that the Chebyshev polynomial minimises the maximum error, we can see convergence as $n \to \infty$ from the fact that

$$|f(x) - p_n(x)| = \frac{|f^{(n+1)}(\xi)|}{(n+1)!}|w(x)| = \frac{|f^{(n+1)}(\xi)|}{2^n(n+1)!}|T_{n+1}(x)| \leq \frac{|f^{(n+1)}(\xi)|}{2^n(n+1)!}.$$

If the function is well-behaved enough that $|f^{(n+1)}(x)| < M$ for some constant whenever $x \in [-1, 1]$, then the error will tend to zero as $n \to \infty$.

2.2 Nonlinear Equations

How do we find roots of nonlinear equations?

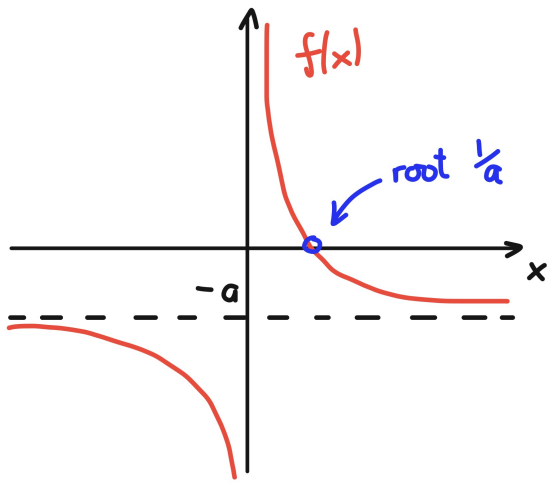
Given a general equation

$$f(x) = 0,$$

there will usually be no explicit formula for the root(s) x_*, so we must use an iterative method.

Rootfinding is a delicate business, and it is essential to begin by plotting a graph of $f(x)$, so that you can tell whether the answer you get from your numerical method is correct.

Example 2.11: $f(x) = \frac{1}{x} - a$, for $a > 0$.



Clearly we know the root is exactly $x_* = \frac{1}{a}$, but this will serve as a running example to test some of our methods

2.2.1 Interval Bisection

If f is continuous and we can find an interval where it changes sign, then it must have a root in this interval. Formally, this is based on:

Theorem 2.8: Intermediate Value Theorem

If f is continuous on $[a, b]$ and c lies between $f(a)$ and $f(b)$, then there is at least one point $x \in [a, b]$ such that $f(x) = c$.

If $f(a)f(b) < 0$, then f changes sign at least once in $[a, b]$, so by the Intermediate Value Theorem there must be a point $x_* \in [a, b]$ where $f(x_*) = 0$.

We can turn this into the following iterative algorithm:

Algorithm 2.1: Interval bisection

Let f be continuous on $[a_0, b_0]$, with $f(a_0)f(b_0) < 0$.

- At each step, set $m_k = (a_k + b_k)/2$.
- If $f(a_k)f(m_k) \geq 0$ then set $a_{k+1} = m_k$, $b_{k+1} = b_k$, otherwise set $a_{k+1} = a_k$, $b_{k+1} = m_k$.

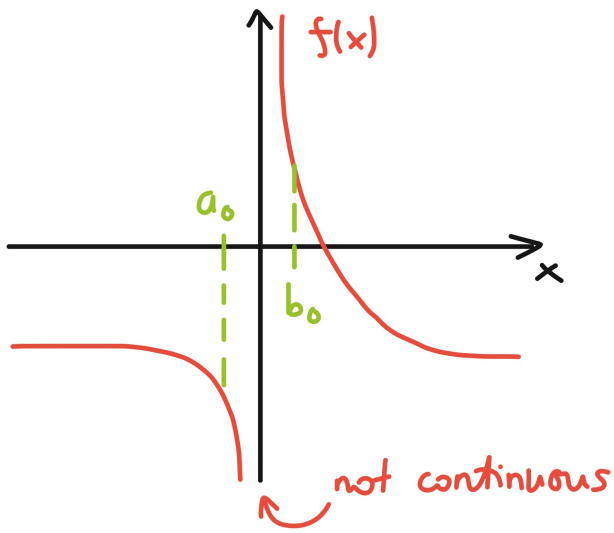
Example 2.12: $f(x) = \frac{1}{x} - 0.5$.

1. Try $a_0 = 1$, $b_0 = 3$ so that $f(a_0)f(b_0) = 0.5(-0.1666) < 0$.
   Now the midpoint is $m_0 = (1 + 3)/2 = 2$, with $f(m_0) = 0$.
   We are lucky and have already stumbled on the root $x_* = m_0 = 2$!

2. Suppose we had tried $a_0 = 1.5$, $b_0 = 3$, so $f(a_0) = 0.1666$ and $f(b_0) = -0.1666$, and again $f(a_0)f(b_0) < 0$.
   Now $m_0 = 2.25$, $f(m_0) = -0.0555$. We have $f(a_0)f(m_0) < 0$, so we set $a_1 = a_0 = 1.5$ and $b_1 = m_0 = 2.25$. The root must lie in $[1.5, 2.25]$.
   Now $m_1 = 1.875$, $f(m_1) = 0.0333$, and $f(a_1)f(m_1) > 0$, so we take $a_2 = m_1 = 1.875$, $b_2 = b_1 = 2.25$. The root must lie in $[1.875, 2.25]$.
   We can continue this algorithm, halving the length of the interval each time.

Since the interval halves in size at each iteration, and always contains a root, we are guaranteed to converge to a root provided that f is continuous. Stopping at step k, we get the minimum possible error by choosing m_k as our approximation.

Example 2.13: Same example with initial interval $[-0.5, 0.5]$.



In this case $f(a_0)f(b_0) < 0$, but there is no root in the interval.

The rate of convergence is steady, so we can pre-determine how many iterations will be needed to converge to a given accuracy. After k iterations, the interval has length

$$|b_k - a_k| = \frac{|b_0 - a_0|}{2^k},$$

so the error in the mid-point satisfies

$$|m_k - x_*| \leq \frac{|b_0 - a_0|}{2^{k+1}}.$$

In order for $|m_k - x_*| \leq \delta$, we need n iterations, where

$$\frac{|b_0 - a_0|}{2^{n+1}} \leq \delta \quad \Longrightarrow \log|b_0 - a_0| - (n+1)\log(2) \leq \log(\delta) \quad \Longrightarrow \quad n \geq \frac{\log|b_0 - a_0| - \log(\delta)}{\log(2)} - 1.$$

Example 2.14: Previous example continued

With $a_0 = 1.5$, $b_0 = 3$, as in the above example, then for $\delta = \epsilon_{\mathrm{M}} = 1.1 \times 10^{-16}$ we would need

$$n \geq \frac{\log(1.5) - \log(1.1 \times 10^{-16})}{\log(2)} - 1 \quad \Longrightarrow \quad n \geq 53 \text{ iterations.}$$

Note

This convergence is pretty slow, but the method has the advantage of being very robust (i.e., use it if all else fails...). It has the more serious disadvantage of *only working in one dimension.*

2.2.2 Fixed point iteration

This is a very common type of rootfinding method. The idea is to transform $f(x) = 0$ into the form $g(x) = x$, so that a root x_* of f is a **fixed point** of g, meaning $g(x_*) = x_*$. To find x_*, we start from some initial guess x_0 and iterate

$$x_{k+1} = g(x_k)$$

until $|x_{k+1} - x_k|$ is sufficiently small. For a given equation $f(x) = 0$, there are many ways to transform it into the form $x = g(x)$. Only some will result in a convergent iteration.

Example 2.15: $f(x) = x^2 - 2x - 3$.

Note that the roots are -1 and 3. Consider some different rearrangements, with $x_0 = 0$.

1. $g(x) = \sqrt{2x + 3}$, gives $x_k \to 3$ [to machine accuracy after 33 iterations].
2. $g(x) = 3/(x - 2)$, gives $x_k \to -1$ [to machine accuracy after 33 iterations].
3. $g(x) = (x^2 - 3)/2$, gives $x_k \to -1$ [but very slowly!].
4. $g(x) = x^2 - x - 3$, gives $x_k \to \infty$.
5. $g(x) = (x^2 + 3)/(2x - 2)$, gives $x_k \to -1$ [to machine accuracy after 5 iterations].

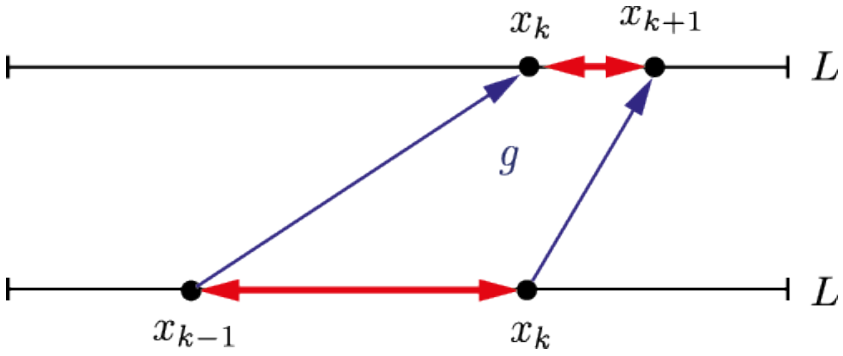
If instead we take $x_0 = 42$, then (1) and (2) still converge to the same roots, (3) now diverges, (4) still diverges, and (5) now converges to the other root $x_k \to 3$.

In this section, we will consider which iterations will converge, before addressing the *rate* of convergence in Section 2.2.3.

One way to ensure that the iteration will work is to find a **contraction mapping** g, which is a map $L \to L$ (for some closed interval L) satisfying

$$|g(x) - g(y)| \leq \lambda |x - y|$$

for some $\lambda < 1$ and for all $x, y \in L$. The sketch below shows the idea:



Theorem 2.9: Contraction Mapping Theorem

If g is a contraction mapping on $L = [a, b]$, then 1. There exists a unique fixed point $x_* \in L$ with $g(x_*) = x_*$. 2. For any $x_0 \in L$, the iteration $x_{k+1} = g(x_k)$ will converge to x_* as $k \to \infty$.

Proof:
To prove *existence*, consider $h(x) = g(x) - x$. Since $g : L \to L$ we have $h(a) = g(a) - a \geq 0$ and $h(b) = g(b) - b \leq 0$. Moreover, it follows from the contraction property above that g is continuous (think of "$\epsilon\delta$"), therefore so is h. So the Intermediate Value Theorem guarantees the existence of at least one point $x_* \in L$ such that $h(x_*) = 0$, i.e. $g(x_*) = x_*$.

For *uniqueness*, suppose x_* and y_* are both fixed points of g in L. Then

$$|x_* - y_*| = |g(x_*) - g(y_*)| \leq \lambda |x_* - y_*| < |x_* - y_*|,$$

which is a contradiction.

Finally, to show *convergence*, consider

$$|x_* - x_{k+1}| = |g(x_*) - g(x_k)| \leq \lambda |x_* - x_k| \leq \ldots \leq \lambda^{k+1} |x_* - x_0|.$$

Since $\lambda < 1$, we see that $x_k \to x_*$ as $k \to \infty$.

i Note

The Contraction Mapping Theorem is also known as the **Banach fixed point theorem**, and was proved by Stefan Banach in his 1920 PhD thesis.

To apply this result in practice, we need to know whether a given function g is a contraction mapping on some interval.

If g is differentiable, then Taylor's theorem says that there exists $\xi \in (x, y)$ with

$$g(x) = g(y) + g'(\xi)(x - y) \implies |g(x) - g(y)| \leq \left(\max_{\xi \in L} |g'(\xi)| \right) |x - y|.$$

So if (a) $g : L \to L$ and (b) $|g'(x)| \leq M$ for all $x \in L$ with $M < 1$, then g is a contraction mapping on L.

Example 2.16: Iteration (a) from previous example, $g(x) = \sqrt{2x+3}$.

Here $g' = (2x+3)^{-1/2}$, so we see that $|g'(x)| < 1$ for all $x > -1$.
For g to be a contraction mapping on an interval L, we also need that g maps L into itself. Since our particular g is continuous and monotonic increasing (for $x > -\frac{3}{2}$), it will map an interval $[a, b]$ to another interval whose end-points are $g(a)$ and $g(b)$.
For example, $g(-\frac{1}{2}) = \sqrt{2}$ and $g(4) = \sqrt{11}$, so the interval $L = [-\frac{1}{2}, 4]$ is mapped into itself. It follows by the Contraction Mapping Theorem that (1) there is a unique fixed point $x_* \in [-\frac{1}{2}, 4]$ (which we know is $x_* = 3$), and (2) the iteration will converge to x_* for any x_0 in this interval (as we saw for $x_0 = 0$).

In practice, it is not always easy to find a suitable interval L. But knowing that $|g'(x_*)| < 1$ is enough to guarantee that the iteration will converge if x_0 is close enough to x_*.

Theorem 2.10: Local Convergence Theorem

Let g and g' be continuous in the neighbourhood of an isolated fixed point $x_* = g(x_*)$. If $|g'(x_*)| < 1$ then there is an interval $L = [x_* - \delta, x_* + \delta]$ such that $x_{k+1} = g(x_k)$ converges to x_* whenever $x_0 \in L$.

Proof:
By continuity of g', there exists some interval $L = [x_* - \delta, x_* + \delta]$ with $\delta > 0$ such that $|g'(x)| \leq M$ for some $M < 1$, for all $x \in L$. Now let $x \in L$. It follows that

$$|x_* - g(x)| = |g(x_*) - g(x)| \leq M|x_* - x| < |x_* - x| \leq \delta,$$

so $g(x) \in L$. Hence g is a contraction mapping on L and the Contraction Mapping Theorem shows that $x_k \to x_*$.

Example 2.17: Iteration (a) again, $g(x) = \sqrt{2x+3}$.

Here we know that $x_* = 3$, and $|g'(3)| = \frac{1}{3} < 1$, so the Local Convergence Theorem tells us that the iteration will converge to 3 if x_0 is close enough to 3.

Example 2.18: Iteration (e) again, $g(x) = (x^2+3)/(2x-2)$.

Here we have

$$g'(x) = \frac{x^2 - 2x - 3}{2(x-1)^2},$$

so we see that $g'(-1) = g'(3) = 0 < 1$. So the Local Convergence Theorem tells us that

the iteration will converge to either root if we start close enough.

Note

As we will see, the fact that $g'(x_*) = 0$ is related to the fast convergence of iteration (e).

2.2.3 Orders of convergence

To measure the speed of convergence, we compare the error $|x_* - x_{k+1}|$ to the error at the previous step, $|x_* - x_k|$.

Example 2.19: Interval bisection.

Here we had $|x_* - m_{k+1}| \leq \frac{1}{2}|x_* - m_k|$. This is called **linear convergence**, meaning that we have $|x_* - x_{k+1}| \leq \lambda |x_* - x_k|$ for some constant $\lambda < 1$.

We can compare a few different iteration schemes that should converge to the same answer to get a sense for how our choice of scheme can impact the convergence order.

Example 2.20: Iteration (a) again, $g(x) = \sqrt{2x+3}$.

Look at the sequence of errors in this case:

x_k	$\lvert 3 - x_k \rvert$	$\lvert 3 - x_k \rvert / \lvert 3 - x_{k-1} \rvert$
0.0000000000	3.0000000000	-
1.7320508076	1.2679491924	0.4226497308
2.5424597568	0.4575402432	0.3608506129
2.8433992885	0.1566007115	0.3422665304
2.9473375404	0.0526624596	0.3362849319
2.9823941860	0.0176058140	0.3343143126
2.9941256440	0.0058743560	0.3336600063

We see that the ratio $|x_* - x_k|/|x_* - x_{k-1}|$ is indeed less than 1, and seems to be converging to $\lambda \approx \frac{1}{3}$. So this is a linearly convergent iteration.

Example 2.21: Iteration (e) again, $g(x) = (x^2 + 3)/(2x - 2)$.

Now the sequence is:

x_k	$\lvert(-1) - x_k\rvert$	$\lvert(-1) - x_k\rvert / \lvert(-1) - x_{k-1}\rvert$
0.0000000000	1.0000000000	-
-1.5000000000	0.5000000000	0.5000000000
-1.0500000000	0.0500000000	0.1000000000
-1.0006097561	0.0006097561	0.0121951220
-1.0000000929	0.0000000929	0.0001523926

Again the ratio $|x_* - x_k|/|x_* - x_{k-1}|$ is certainly less than 1, but this time we seem to have $\lambda \to 0$ as $k \to \infty$. This is called **superlinear convergence**, meaning that the convergence is in some sense "accelerating".

In general, if $x_k \to x_*$ then we say that the sequence $\{x_k\}$ **converges linearly** if

$$\lim_{k\to\infty} \frac{|x_* - x_{k+1}|}{|x_* - x_k|} = \lambda \quad \text{with} \quad 0 < \lambda < 1.$$

If $\lambda = 0$ then the convergence is **superlinear**.

> **Note**
>
> The constant λ is called the **rate** or **ratio**.

The following result establishes conditions for linear and superlinear convergence.

Theorem 2.11

Let g' be continuous in the neighbourhood of a fixed point $x_* = g(x_*)$, and suppose that $x_{k+1} = g(x_k)$ converges to x_* as $k \to \infty$.

1. If $|g'(x_*)| \neq 0$ then the convergence will be linear with rate $\lambda = |g'(x_*)|$.
2. If $|g'(x_*)| = 0$ then the convergence will be superlinear.

Proof:
By Taylor's theorem, note that

$$x_* - x_{k+1} = g(x_*) - g(x_k) = g(x_*) - \Big[g(x_*) + g'(\xi_k)(x_k - x_*)\Big] = g'(\xi_k)(x_* - x_k)$$

for some ξ_k between x_* and x_k. Since $x_k \to x_*$, we have $\xi_k \to x_*$ as $k \to \infty$, so

$$\lim_{k\to\infty} \frac{|x_* - x_{k+1}|}{|x_* - x_k|} = \lim_{k\to\infty} |g'(\xi_k)| = |g'(x_*)|.$$

This proves the result.

Example 2.22: Iteration (a) again, $g(x) = \sqrt{2x+3}$.

We saw before that $g'(3) = \frac{1}{3}$, so the theorem above shows that convergence will be linear with $\lambda = |g'(3)| = \frac{1}{3}$ as we found numerically.

Example 2.23: Iteration (e) again, $g(x) = (x^2+3)/(2x-2)$.

We saw that $g'(-1) = 0$, so the theorem above shows that convergence will be superlinear, again consistent with our numerical findings.

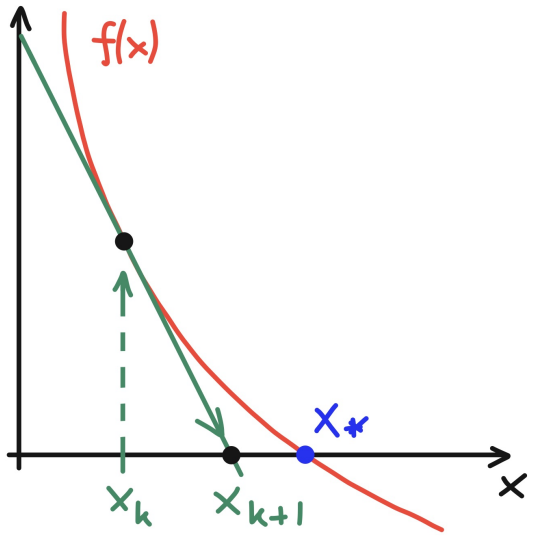
We can further classify superlinear convergence by the **order of convergence**, defined as

$$\alpha = \sup\left\{\beta \,:\, \lim_{k\to\infty} \frac{|x_* - x_{k+1}|}{|x_* - x_k|^\beta} < \infty\right\}.$$

For example, $\alpha = 2$ is called **quadratic** convergence and $\alpha = 3$ is called **cubic** convergence, although for a general sequence α need not be an integer (e.g. the secant method below).

2.2.4 Newton's method

This is a particular fixed point iteration that is very widely used because (as we will see) it usually converges superlinearly.



Graphically, the idea of **Newton's method** is simple: given x_k, draw the tangent line to f at $x = x_k$, and let x_{k+1} be the x-intercept of this tangent. So

$$\frac{0 - f(x_k)}{x_{k+1} - x_k} = f'(x_k) \qquad \Longrightarrow \qquad x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

> **Note**
>
> In fact, Newton only applied the method to polynomial equations, and without using calculus. The general form using derivatives ("fluxions") was first published by Thomas Simpson in 1740. [See "Historical Development of the Newton-Raphson Method" by T.J. Ypma, *SIAM Review* **37**, 531 (1995).]

Another way to derive this iteration is to approximate $f(x)$ by the linear part of its Taylor series centred at x_k:

$$0 \approx f(x_{k+1}) \approx f(x_k) + f'(x_k)(x_{k+1} - x_k).$$

The iteration function for Newton's method is

$$g(x) = x - \frac{f(x)}{f'(x)},$$

so using $f(x_*) = 0$ we see that $g(x_*) = x_*$. To assess the convergence, note that

$$g'(x) = 1 - \frac{f'(x)f'(x) - f(x)f''(x)}{[f'(x)]^2} = \frac{f(x)f''(x)}{[f'(x)]^2} \qquad \Longrightarrow\ g'(x_*) = 0 \quad \text{if } f'(x_*) \neq 0.$$

So if $f'(x_*) \neq 0$, the Local Convergence Theorem shows that the iteration will converge for x_0 close enough to x_*. Moreover, since $g'(x_*) = 0$, the order theorem shows that this convergence will be superlinear.

Example 2.24: Calculate a^{-1} using $f(x) = \frac{1}{x} - a$ for $a > 0$.

Newton's method gives the iterative formula

$$x_{k+1} = x_k - \frac{\frac{1}{x_k} - a}{-\frac{1}{x_k^2}} = 2x_k - ax_k^2.$$

From the graph of f, it is clear that the iteration will converge for any $x_0 \in (0, a^{-1})$, but will diverge if x_0 is too large. With $a = 0.5$ and $x_0 = 1$, Python gives

x_k	$\lvert 2 - x_k \rvert$	$\lvert 2 - x_k \rvert / \lvert 2 - x_{k-1} \rvert$	$\lvert 2 - x_k \rvert / \lvert 2 - x_{k-1} \rvert^2$
1.0	1.0	-	-

1.5	0.5	0.5	0.5
1.875	0.125	0.25	0.5
1.9921875	0.0078125	0.0625	0.5
1.999969482	3.05×10^{-5}	0.00390625	0.5
2.0	4.65×10^{-10}	1.53×10^{-5}	0.5
2.0	1.08×10^{-19}	2.33×10^{-10}	0.5

In 6 steps, the error is below ϵ_{M}: pretty rapid convergence! The third column shows that the convergence is superlinear. The fourth column shows that $|x_* - x_{k+1}|/|x_* - x_k|^2$ is constant, indicating that the convergence is quadratic (order $\alpha = 2$).

i Note

Although the solution $\frac{1}{a}$ is known exactly, this method is so efficient that it is sometimes used in computer hardware to do division!

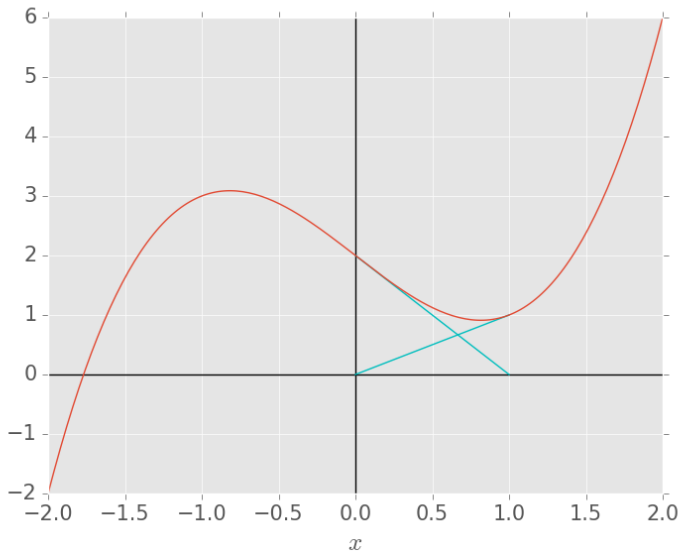
In practice, it is not usually possible to determine ahead of time whether a given starting value x_0 will converge.

A robust computer implementation should catch any attempt to take too large a step, and switch to a less sensitive (but slower) algorithm (e.g. bisection).

However, it always makes sense to avoid any points where $f'(x) = 0$.

Example 2.25: $f(x) = x^3 - 2x + 2$.

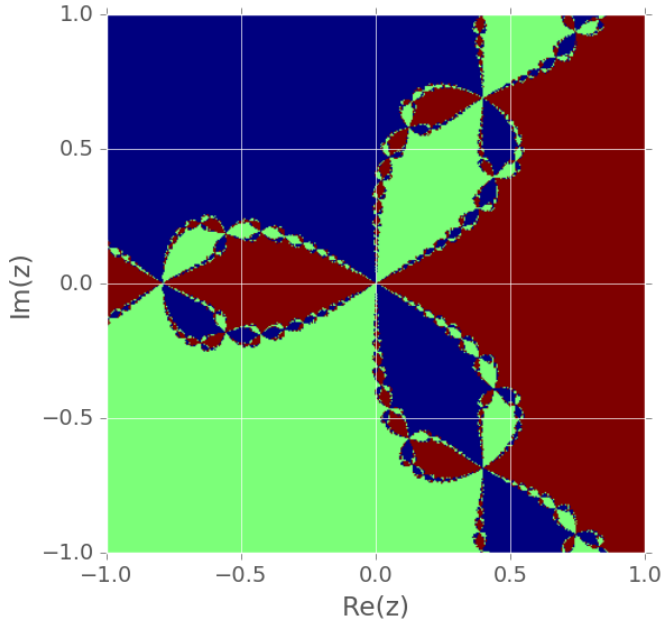
Here $f'(x) = 3x^2 - 2$ so there are turning points at $x = \pm\sqrt{\frac{2}{3}}$ where $f'(x) = 0$, as well as a single real root at $x_* \approx -1.769$. The presence of points where $f'(x) = 0$ means that care is needed in choosing a starting value x_0.
If we take $x_0 = 0$, then $x_1 = 0 - f(0)/f'(0) = 1$, but then $x_2 = 1 - f(1)/f'(1) = 0$, so the iteration gets stuck in an infinite loop:



Other starting values, e.g. $x_0 = -0.5$ can also be sucked into this infinite loop! The correct answer is obtained for $x_0 = -1.0$.

Note

The sensitivity of Newton's method to the choice of x_0 is beautifully illustrated by applying it to a **complex** function such as $f(z) = z^3 - 1$. The following plot colours points z_0 in the complex plane according to which root they converge to (1, $e^{2\pi i/3}$, or $e^{-2\pi i/3}$):



The boundaries of these **basins of attraction** are fractal.

2.2.5 Newton's method for systems

Newton's method generalizes to higher-dimensional problems where we want to find $\mathbf{x} \in \mathbb{R}^m$ that satisfies $\mathbf{f}(\mathbf{x}) = 0$ for some function $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^m$.

To see how it works, take $m = 2$ so that $\mathbf{x} = (x_1, x_2)^\top$ and $\mathbf{f} = [f_1(\mathbf{x}), f_2(\mathbf{x})]^\top$. Taking the linear terms in Taylor's theorem for two variables gives

$$0 \approx f_1(\mathbf{x}_{k+1}) \approx f_1(\mathbf{x}_k) + \left.\frac{\partial f_1}{\partial x_1}\right|_{\mathbf{x}_k} (x_{1,k+1} - x_{1,k}) + \left.\frac{\partial f_1}{\partial x_2}\right|_{\mathbf{x}_k} (x_{2,k+1} - x_{2,k}),$$
$$0 \approx f_2(\mathbf{x}_{k+1}) \approx f_2(\mathbf{x}_k) + \left.\frac{\partial f_2}{\partial x_1}\right|_{\mathbf{x}_k} (x_{1,k+1} - x_{1,k}) + \left.\frac{\partial f_2}{\partial x_2}\right|_{\mathbf{x}_k} (x_{2,k+1} - x_{2,k}).$$

In matrix form, we can write

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} f_1(\mathbf{x}_k) \\ f_2(\mathbf{x}_k) \end{pmatrix} + \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}_k) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}_k) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}_k) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}_k) \end{pmatrix} \begin{pmatrix} x_{1,k+1} - x_{1,k} \\ x_{2,k+1} - x_{2,k} \end{pmatrix}.$$

The matrix of partial derivatives is called the **Jacobian matrix** $J(\mathbf{x}_k)$, so (for any m) we have

$$\mathbf{0} = \mathbf{f}(\mathbf{x}_k) + J(\mathbf{x}_k)(\mathbf{x}_{k+1} - \mathbf{x}_k).$$

To derive Newton's method, we rearrange this equation for $\mathbf{x}_{k+1}$,

$$J(\mathbf{x}_k)(\mathbf{x}_{k+1} - \mathbf{x}_k) = -\mathbf{f}(\mathbf{x}_k) \quad \Longrightarrow \quad \mathbf{x}_{k+1} = \mathbf{x}_k - J^{-1}(\mathbf{x}_k)\mathbf{f}(\mathbf{x}_k).$$

So to apply the method, we need the inverse of J.

> **Note**
>
> If $m = 1$, then $J(x_k) = \frac{\partial f}{\partial x}(x_k)$, and $J^{-1} = 1/J$, so this reduces to the scalar Newton's method.

Example 2.26: Apply Newton's method to the simultaneous equations $xy - y^3 - 1 = 0$ and $x^2y + y - 5 = 0$, with starting values $x_0 = 2$, $y_0 = 3$.

The Jacobian matrix is

$$J(x, y) = \begin{pmatrix} y & x - 3y^2 \\ 2xy & x^2 + 1 \end{pmatrix},$$

and hence its inverse is given by

$$J^{-1}(x, y) = \frac{1}{y(x^2 + 1) - 2xy(x - 3y^2)} \begin{pmatrix} x^2 + 1 & 3y^2 - x \\ -2xy & y \end{pmatrix}.$$

The first iteration of Newton's method gives

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} - \frac{1}{3(5) - 12(2 - 27)} \begin{pmatrix} 5 & 25 \\ -12 & 3 \end{pmatrix} \begin{pmatrix} -22 \\ 10 \end{pmatrix} = \begin{pmatrix} 1.55555556 \\ 2.06666667 \end{pmatrix}.$$

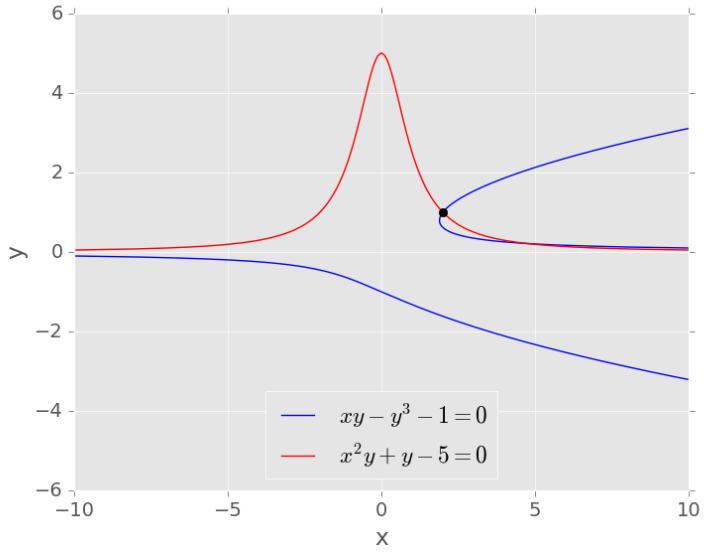
Subsequent iterations give

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1.54720541 \\ 1.47779333 \end{pmatrix}, \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1.78053503 \\ 1.15886481 \end{pmatrix},$$

and

$$\begin{pmatrix} x_4 \\ y_4 \end{pmatrix} = \begin{pmatrix} 1.952843 \\ 1.02844269 \end{pmatrix}, \begin{pmatrix} x_5 \\ y_5 \end{pmatrix} = \begin{pmatrix} 1.99776297 \\ 1.00124041 \end{pmatrix},$$

so the method is converging accurately to the root $x_* = 2$, $y_* = 1$, shown in the following plot:



By generalising the scalar analysis (beyond the scope of this course), it can be shown that the convergence is quadratic for $\mathbf{x}_0$ sufficiently close to $\mathbf{x}_*$, provided that $J(\mathbf{x}_*)$ is non-singular (i.e., $\det[J(\mathbf{x}_*)] \neq 0$).

> **Note**
>
> In general, finding a good starting point in more than one dimension is difficult, particularly because interval bisection is not available. Algorithms that try to mimic bisection in higher dimensions are available, proceeding by a 'grid search' approach.

2.2.6 Quasi-Newton methods

A drawback of Newton's method is that the derivative $f'(x_k)$ must be computed at each iteration. This may be expensive to compute, or may not be available as a formula. For example, the function f might be the right-hand side of some complex partial differential equation, and hence both difficult to differentiate and very high dimensional!

Instead we can use a **quasi-Newton method**

$$x_{k+1} = x_k - \frac{f(x_k)}{g_k},$$

where g_k is some easily-computed approximation to $f'(x_k)$.

Example 2.27: Steffensen's method

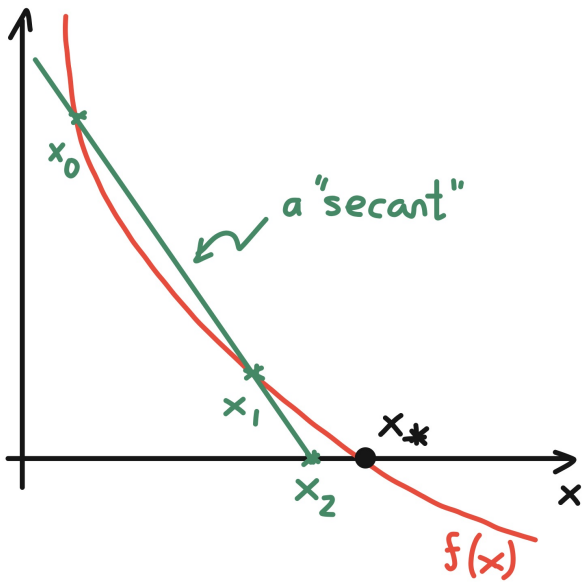
$$g_k = \frac{f(f(x_k) + x_k) - f(x_k)}{f(x_k)}.$$

This has the form $\frac{1}{h}(f(x_k + h) - f(x_k))$ with $h = f(x_k)$.

Steffensen's method requires two function evaluations per iteration. But once the iteration has started, we already have two nearby points x_{k-1}, x_k, so we could approximate $f'(x_k)$ by a backward difference

$$g_k = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \quad \Longrightarrow \quad x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}.$$

This is called the **secant method**, and requires only one function evaluation per iteration (once underway). The name comes from its graphical interpretation:



Note

The secant method was introduced by Newton.

Example 2.28: $f(x) = \frac{1}{x} - 0.5$.

Now we need two starting values, so take $x_0 = 0.25$, $x_1 = 0.5$. The secant method gives:

k	x_k	$\lvert x_* - x_k\rvert / \lvert x_* - x_{k-1}\rvert$
2	0.6875	0.75
3	1.01562	0.75
4	1.354	0.65625
5	1.68205	0.492188
6	1.8973	0.322998
7	1.98367	0.158976
8	1.99916	0.0513488

Convergence to ϵ_{M} is achieved in 12 iterations. Notice that the error ratio is decreasing, so the convergence is superlinear.

The secant method is a **two-point method** since $x_{k+1} = g(x_{k-1}, x_k)$. So theorems about single-point fixed-point iterations do not apply.

In general, one can have **multipoint methods** based on higher-order interpolation.

Theorem 2.12

If $f'(x_*) \neq 0$ then the secant method converges for x_0, x_1 sufficiently close to x_*, and the order of convergence is $(1 + \sqrt{5})/2 = 1.618\ldots$.

i Note

This illustrates that orders of convergence need not be integers, and is also an appearance of the **golden ratio**.

Proof:
To simplify the notation, denote the truncation error by

$$\varepsilon_k := x_* - x_k.$$

Expanding in Taylor series around x_*, and using $f(x_*) = 0$, gives

$$f(x_{k-1}) = -f'(x_*)\varepsilon_{k-1} + \frac{f''(x_*)}{2}\varepsilon_{k-1}^2 + \mathcal{O}(\varepsilon_{k-1}^3),$$
$$f(x_k) = -f'(x_*)\varepsilon_k + \frac{f''(x_*)}{2}\varepsilon_k^2 + \mathcal{O}(\varepsilon_k^3).$$

So using the secant formula above we get

$$\begin{aligned}
\varepsilon_{k+1} &= \varepsilon_k - (\varepsilon_k - \varepsilon_{k-1}) \frac{-f'(x_*)\varepsilon_k + \frac{f''(x_*)}{2}\varepsilon_k^2 + \mathcal{O}(\varepsilon_k^3)}{-f'(x_*)(\varepsilon_k - \varepsilon_{k-1}) + \frac{f''(x_*)}{2}(\varepsilon_k^2 - \varepsilon_{k-1}^2) + \mathcal{O}(\varepsilon_{k-1}^3)} \\
&= \varepsilon_k - \frac{-f'(x_*)\varepsilon_k + \frac{f''(x_*)}{2}\varepsilon_k^2 + \mathcal{O}(\varepsilon_k^3)}{-f'(x_*) + \frac{f''(x_*)}{2}(\varepsilon_k + \varepsilon_{k-1}) + \mathcal{O}(\varepsilon_{k-1}^2)} \\
&= \varepsilon_k + \frac{-\varepsilon_k + \frac{1}{2}\varepsilon_k^2 f''(x_*)/f'(x_*) + \mathcal{O}(\varepsilon_k^3)}{1 - \frac{1}{2}(\varepsilon_k + \varepsilon_{k-1})f''(x_*)/f'(x_*) + \mathcal{O}(\varepsilon_{k-1}^2)} \\
&= \varepsilon_k + \left(-\varepsilon_k + \frac{f''(x_*)}{2f'(x_*)}\varepsilon_k^2 + \mathcal{O}(\varepsilon_k^3)\right)\left(1 + (\varepsilon_k + \varepsilon_{k-1})\frac{f''(x_*)}{2f'(x_*)} + \mathcal{O}(\varepsilon_{k-1}^2)\right) \\
&= \varepsilon_k - \varepsilon_k + \frac{f''(x_*)}{2f'(x_*)}\varepsilon_k^2 - \frac{f''(x_*)}{2f'(x_*)}\varepsilon_k(\varepsilon_k + \varepsilon_{k-1}) + \mathcal{O}(\varepsilon_{k-1}^3) \\
&= -\frac{f''(x_*)}{2f'(x_*)}\varepsilon_k\varepsilon_{k-1} + \mathcal{O}(\varepsilon_{k-1}^3).
\end{aligned}$$

This is similar to the corresponding formula for Newton's method, where we have

$$\varepsilon_{k+1} = -\frac{f''(x_*)}{2f'(x_*)}\varepsilon_k^2 + \mathcal{O}(\varepsilon_k^3).$$

The above tells us that the error for the secant method tends to zero faster than linearly, but not quadratically (because $\varepsilon_{k-1} > \varepsilon_k$).

To find the order of convergence, note that $\varepsilon_{k+1} \sim \varepsilon_k\varepsilon_{k-1}$ suggests a power-law relation of the form

$$|\varepsilon_k| = |\varepsilon_{k-1}|^\alpha \left|\frac{f''(x_*)}{2f'(x_*)}\right|^\beta \quad \Longrightarrow \quad |\varepsilon_{k-1}| = |\varepsilon_k|^{1/\alpha} \left|\frac{f''(x_*)}{2f'(x_*)}\right|^{-\beta/\alpha}.$$

Putting this in both sides of the previous equation gives

$$|\varepsilon_k|^\alpha \left|\frac{f''(x_*)}{2f'(x_*)}\right|^\beta = |\varepsilon_k|^{(1+\alpha)/\alpha} \left|\frac{f''(x_*)}{2f'(x_*)}\right|^{(\alpha-\beta)/\alpha}.$$

Equating powers gives

$$\alpha = \frac{1+\alpha}{\alpha} \quad \Longrightarrow \alpha = \frac{1+\sqrt{5}}{2}, \quad \beta = \frac{\alpha-\beta}{\alpha} \quad \Longrightarrow \beta = \frac{\alpha}{\alpha+1} = \frac{1}{\alpha}.$$

It follows that

$$\lim_{k\to\infty} \frac{|x_* - x_{k+1}|}{|x_* - x_k|^\alpha} = \lim_{k\to\infty} \frac{|\varepsilon_{k+1}|}{|\varepsilon_k|^\alpha} = \left|\frac{f''(x_*)}{2f'(x_*)}\right|^{1/\alpha},$$

so the secant method has order of convergence α.

Knowledge checklist

Key topics:

1. Polynomial interpolation, including the Lagrange and Newton divided-difference forms of polynomial interpolation.
2. Interpolation error estimates, truncation error, and the significance of node placement (e.g. the Runge phenomenon).
3. Numerical rootfinding methods, including interval bisection and fixed point iteration, with discussion of existence, uniqueness, and orders of convergence (linear, superlinear).

Key skills:

- Constructing and analyzing polynomial interpolants for a given data set and function, using Taylor, Lagrange, and Newton methods.
- Estimating approximation errors and choosing optimal interpolation nodes to improve numerical stability and convergence.
- Implementing and evaluating iterative algorithms for solving nonlinear equations, including measuring and understanding convergence rates.

3 Linear Algebra

3.1 Systems of Linear Equations

The central goal of this chapter is to answer the following seemingly straightforward question:

How do we solve a linear system numerically?

Linear systems of the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2, \\ \vdots \quad \vdots \quad & \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned}$$

occur in many applications (often with very large n). It is convenient to express this in matrix form:

$$A\mathbf{x} = \mathbf{b},$$

where A is an $n \times n$ square matrix with elements a_{ij}, and $\mathbf{x}$, $\mathbf{b}$ are $n \times 1$ vectors.

We will need some basic facts from linear algebra:

1. $A^\top$ is the **transpose** of A, so $(a^\top)_{ij} = a_{ji}$.
2. A is **symmetric** if $A = A^\top$.
3. A is **non-singular** iff there exists a solution $\mathbf{x} \in \mathbb{R}^n$ for every $\mathbf{b} \in \mathbb{R}^n$.
4. A is non-singular iff $\det(A) \neq 0$.
5. A is non-singular iff there exists a unique **inverse** A^{-1} such that $AA^{-1} = A^{-1}A = I$.

It follows from fact 5 above that $A\mathbf{x} = \mathbf{b}$ has a unique solution iff A is non-singular, given by $\mathbf{x} = A^{-1}\mathbf{b}$.

In this chapter, we will see how to solve $A\mathbf{x} = \mathbf{b}$ both **efficiently** and **accurately**.

Although this seems like a conceptually easy problem (just use Gaussian elimination!), it is actually a hard one when n gets large. Nowadays, linear systems with $n = 1$ million arise

routinely in computational problems. And even for small n there are some potential pitfalls, as we will see.

> **Note**
>
> If A is instead rectangular $(m \times n)$, then there are different numbers of equations and unknowns, and we do not expect a unique solution. Nevertheless, we can still look for an approximate solution in this case and there are methods for this problem in the course reading list.

Many algorithms are based on the idea of rewriting $A\mathbf{x} = \mathbf{b}$ in a form where the matrix is easier to invert. Easiest to invert are diagonal matrices, followed by orthogonal matrices (where $A^{-1} = A^{\top}$). However, the most common method for solving $A\mathbf{x} = \mathbf{b}$ transforms the system to *triangular* form.

3.2 Triangular systems

If the matrix A is triangular, then $A\mathbf{x} = \mathbf{b}$ is straightforward to solve.

A matrix L is called **lower triangular** if all entries above the diagonal are zero:

$$L = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ l_{n1} & \cdots & \cdots & l_{nn} \end{pmatrix}.$$

The determinant is just

$$\det(L) = l_{11} l_{22} \cdots l_{nn},$$

so the matrix will be non-singular iff all of the diagonal elements are non-zero.

Example 3.1: Solve $L\mathbf{x} = \mathbf{b}$ for $n = 4$.

The system is

$$\begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

which is equivalent to

$$\begin{aligned} l_{11}x_1 &= b_1, \\ l_{21}x_1 + l_{22}x_2 &= b_2, \\ l_{31}x_1 + l_{32}x_2 + l_{33}x_3 &= b_3, \\ l_{41}x_1 + l_{42}x_2 + l_{43}x_3 + l_{44}x_4 &= b_4. \end{aligned}$$

We can just solve step-by-step:

$$x_1 = \frac{b_1}{l_{11}}, \; x_2 = \frac{b_2 - l_{21}x_1}{l_{22}},$$

$$x_3 = \frac{b_3 - l_{31}x_1 - l_{32}x_2}{l_{33}}, \; x_4 = \frac{b_4 - l_{41}x_1 - l_{42}x_2 - l_{43}x_3}{l_{44}}.$$

This is fine since we know that l_{11}, l_{22}, l_{33}, l_{44} are all non-zero when a solution exists.

In general, any lower triangular system $L\mathbf{x} = \mathbf{b}$ can be solved by **forward substitution**

$$x_j = \frac{b_j - \sum_{k=1}^{j-1} l_{jk}x_k}{l_{jj}}, \quad j = 1, \ldots, n.$$

Similarly, an **upper triangular** matrix U has the form

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{pmatrix},$$

and an upper-triangular system $U\mathbf{x} = \mathbf{b}$ may be solved by **backward substitution**

$$x_j = \frac{b_j - \sum_{k=j+1}^{n} u_{jk}x_k}{u_{jj}}, \quad j = n, \ldots, 1.$$

To estimate the computational cost of forward substitution, we can count the number of floating-point operations ($+$, $-$, $\times$, $\div$).

Example 3.2: Number of operations required for forward substitution.

Consider each x_j. We have - $j = 1$: 1 division - $j = 2$: 1 division + [1 subtraction + 1 multiplication] - $j = 3$: 1 division + 2 $\times$[1 subtraction + 1 multiplication] - $\vdots$ - $j = n$: 1 division + $(n-1)$ $\times$[1 subtraction + 1 multiplication]

So the total number of operations required is

$$\sum_{j=1}^{n} \Big(1 + 2(j-1)\Big) = 2\sum_{j=1}^{n} j - \sum_{j=1}^{n} 1 = n(n+1) - n = n^2.$$

So solving a triangular system by forward (or backward) substitution takes n^2 operations. We may say that the **computational complexity** of the algorithm is n^2.

> **Note**
>
> In practice, this is only a rough estimate of the computational cost, because reading from and writing to the computer's memory also take time. This can be estimated given a "memory model", but this depends on the particular computer.

3.3 Gaussian elimination

If our matrix A is not triangular, we can try to transform it to triangular form. **Gaussian elimination** uses elementary row operations to transform the system to upper triangular form $U\mathbf{x} = \mathbf{y}$.

Elementary row operations include swapping rows and adding multiples of one row to another. They won't change the solution $\mathbf{x}$, but will change the matrix A and the right-hand side $\mathbf{b}$.

Example 3.3: Transform to upper triangular form the system

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 0, \\ x_1 - 2x_2 + 2x_3 &= 4, \\ 2x_1 + 12x_2 - 2x_3 &= 4. \end{aligned}$$

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 1 & -2 & 2 \\ 2 & 12 & -2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ 4 \\ 4 \end{pmatrix}.$$

Stage 1. Subtract 1 times equation 1 from equation 2, and 2 times equation 1 from equation 3, so as to eliminate x_1 from equations 2 and 3:

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 0, \\ -4x_2 + x_3 &= 4, \\ 8x_2 - 4x_3 &= 4. \end{aligned}$$

$$A^{(2)} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 8 & -4 \end{pmatrix} \quad \mathbf{b}^{(2)} = \begin{pmatrix} 0 \\ 4 \\ 4 \end{pmatrix}, \quad m_{21} = 1, \quad m_{31} = 2.$$

Stage 2. Subtract -2 times equation 2 from equation 3, to eliminate x_2 from equation 3:

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 0, \\ -4x_2 + x_3 &= 4, \\ -2x_3 &= 12. \end{aligned}$$

$$A^{(3)} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 0 & -2 \end{pmatrix} \quad \mathbf{b}^{(3)} = \begin{pmatrix} 0 \\ 4 \\ 12 \end{pmatrix}, \quad m_{32} = -2.$$

Now the system is upper triangular, and back substitution gives $x_1 = 11$, $x_2 = -\frac{5}{2}$, $x_3 = -6$.

We can write the general algorithm as follows.

Algorithm 3.1: Gaussian elimination

Let $A^{(1)} = A$ and $\mathbf{b}^{(1)} = \mathbf{b}$. Then for each k from 1 to $n-1$, compute a new matrix $A^{(k+1)}$ and right-hand side $\mathbf{b}^{(k+1)}$ by the following procedure:

1. Define the row multipliers

$$m_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \quad i = k+1, \ldots, n.$$

2. Use these to remove the unknown x_k from equations $k+1$ to n, leaving

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - m_{ik} a_{kj}^{(k)}, \quad b_i^{(k+1)} = b_i^{(k)} - m_{ik} b_k^{(k)}, \quad i, j = k+1, \ldots, n.$$

The final matrix $A^{(n)} = U$ will then be upper triangular.

This procedure will work providing $a_{kk}^{(k)} \neq 0$ for every k. (We will worry about this later.)

What about the computational cost of Gaussian elimination?

Example 3.4: Number of operations required to find U.

Computing $A^{(k+1)}$ requires: - $n-(k+1)+1 = n-k$ divisions to compute m_{ik}. - $(n-k)^2$ subtractions and the same number of multiplications to compute $a_{ij}^{(k+1)}$.
So in total $A^{(k+1)}$ requires $2(n-k)^2 + n - k$ operations. Overall, we need to compute $A^{(k+1)}$ for $k = 1, \ldots, n-1$, so the total number of operations is

$$\begin{aligned} N &= \sum_{k=1}^{n-1} \left(2n^2 + n - (4n+1)k + 2k^2\right) \\ &= n(2n+1)\sum_{k=1}^{n-1} 1 - (4n+1)\sum_{k=1}^{n-1} k + 2\sum_{k=1}^{n-1} k^2. \end{aligned}$$

Recalling that

$$\sum_{k=1}^{n} k = \tfrac{1}{2}n(n+1), \ \sum_{k=1}^{n} k^2 = \tfrac{1}{6}n(n+1)(2n+1),$$

we find

$$\begin{aligned} N &= n(2n+1)(n-1) - \tfrac{1}{2}(4n+1)(n-1)n + \tfrac{1}{3}(n-1)n(2n-1) \\ &= \tfrac{2}{3}n^3 - \tfrac{1}{2}n^2 - \tfrac{1}{6}n. \end{aligned}$$

So the number of operations required to find U is $\mathcal{O}(n^3)$.

It is known that $\mathcal{O}(n^3)$ is not optimal, and the best theoretical algorithm known for inverting a matrix takes $\mathcal{O}(n^{2.3728639})$ operations. However, algorithms achieving this bound are highly impractical for most real-world uses due to massive constant factors and implementation overhead. It remains an open conjecture that there exists an $\mathcal{O}(n^{2+\epsilon})$ algorithm, for ϵ arbitrarily small.

3.4 LU decomposition

In Gaussian elimination, both the final matrix U and the sequence of row operations are determined solely by A, and do not depend on $\mathbf{b}$. We will see that the sequence of row operations that transforms A to U is equivalent to left-multiplying by a matrix F, so that

$$FA = U, \qquad U\mathbf{x} = F\mathbf{b}.$$

To see this, note that step k of Gaussian elimination can be written in the form

$$A^{(k+1)} = F^{(k)}A^{(k)}, \quad \mathbf{b}^{(k+1)} = F^{(k)}\mathbf{b}^{(k)},$$

where

$$F^{(k)} := \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & 1 & \ddots & & \vdots \\ \vdots & & -m_{k+1,k} & \ddots & \ddots & \vdots \\ \vdots & & \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & -m_{n,k} & \cdots & 0 & 1 \end{pmatrix}.$$

Multiplying by $F^{(k)}$ has the effect of subtracting m_{ik} times row k from row i, for $i = k+1, \ldots, n$.

> **Note**
>
> A matrix with this structure (the identity except for a single column below the diagonal) is called a **Frobenius matrix**.

Example 3.5

You can check in the earlier example that

$$F^{(1)}A = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 1 & -2 & 2 \\ 2 & 12 & -2 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 8 & -4 \end{pmatrix} = A^{(2)},$$

and

$$F^{(2)}A^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 8 & -4 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 0 & -2 \end{pmatrix} = A^{(3)} = U.$$

It follows that

$$U = A^{(n)} = F^{(n-1)}F^{(n-2)} \cdots F^{(1)}A.$$

Now the $F^{(k)}$ are invertible, and the inverse is just given by adding rows instead of subtracting:

$$(F^{(k)})^{-1} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & 1 & \ddots & & \vdots \\ \vdots & & m_{k+1,k} & \ddots & \ddots & \vdots \\ \vdots & & \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & m_{n,k} & \cdots & 0 & 1 \end{pmatrix}.$$

So we could write

$$A = (F^{(1)})^{-1}(F^{(2)})^{-1} \cdots (F^{(n-1)})^{-1}U.$$

Since the successive operations don't "interfere" with each other, we can write

$$(F^{(1)})^{-1}(F^{(2)})^{-1}\cdots(F^{(n-1)})^{-1} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ m_{2,1} & 1 & \ddots & & & \vdots \\ m_{3,1} & m_{3,2} & 1 & \ddots & & \vdots \\ m_{4,1} & m_{4,2} & m_{4,3} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & & 1 & 0 \\ m_{n,1} & m_{n,2} & m_{n,3} & \cdots & m_{n,n-1} & 1 \end{pmatrix} := L.$$

Thus we have established the following result.

Theorem 3.1: LU decomposition

Let U be the upper triangular matrix from Gaussian elimination of A (without pivoting), and let L be the unit lower triangular matrix above. Then

$$A = LU.$$

Note

Unit lower triangular means that there are all 1's on the diagonal.

The theorem above says that Gaussian elimination is equivalent to factorising A as the product of a lower triangular and an upper triangular matrix. This is not at all obvious from the algorithm! The decomposition is unique up to a scaling LD, $D^{-1}U$ for some diagonal matrix D.

The system $A\mathbf{x} = \mathbf{b}$ becomes $LU\mathbf{x} = \mathbf{b}$, which we can readily solve by setting $U\mathbf{x} = \mathbf{y}$. We first solve $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$, then $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$. Both are triangular systems.

Moreover, if we want to solve several systems $A\mathbf{x} = \mathbf{b}$ with different $\mathbf{b}$ but the same matrix, we just need to compute L and U once. This saves time because, although the initial LU factorisation takes $\mathcal{O}(n^3)$ operations, the evaluation takes only $\mathcal{O}(n^2)$.

Note

This matrix factorisation viewpoint dates only from the 1940s, and LU decomposition was introduced by Alan Turing in a 1948 paper (*Q. J. Mechanics Appl. Mat.* **1**, 287). Other common factorisations used in numerical linear algebra are **QR** (which we will see later) and *Cholesky*.

Example 3.6

Solve our earlier example by LU decomposition.

$$\begin{pmatrix} 1 & 2 & 1 \\ 1 & -2 & 2 \\ 2 & 12 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 4 \end{pmatrix}.$$

We apply Gaussian elimination as before, but ignore $\mathbf{b}$ (for now), leading to

$$U = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 0 & -2 \end{pmatrix}.$$

As we apply the elimination, we record the multipliers so as to construct the matrix

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & -2 & 1 \end{pmatrix}.$$

Thus we have the factorisation/decomposition

$$\begin{pmatrix} 1 & 2 & 1 \\ 1 & -2 & 2 \\ 2 & 12 & -2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & -2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 0 & -2 \end{pmatrix}.$$

With the matrices L and U, we can readily solve for any right-hand side $\mathbf{b}$. We illustrate for our particular $\mathbf{b}$. Firstly, solve $L\mathbf{y} = \mathbf{b}$:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & -2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 4 \end{pmatrix}$$

$$\implies y_1 = 0,\ y_2 = 4 - y_1 = 4,\ y_3 = 4 - 2y_1 + 2y_2 = 12.$$

Notice that $\mathbf{y}$ is the right-hand side $\mathbf{b}^{(3)}$ constructed earlier. Then, solve $U\mathbf{x} = \mathbf{y}$:

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 0 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 12 \end{pmatrix}$$

$$\implies x_3 = -6,\ x_2 = -\tfrac{1}{4}(4 - x_3) = -\tfrac{5}{2},\ x_1 = -2x_2 - x_3 = 11.$$

3.5 Vector norms

To measure the error when the solution is a vector, as opposed to a scalar, we usually summarize the error in a single number called a **norm**. A **norm** effectively gives us a way to define a notion of distance in higher dimensions.

A **vector norm** on $\mathbb{R}^n$ is a real-valued function that satisfies: 1. $\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ for every $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ (**triangle inequality**). 2. $\|\alpha\mathbf{x}\| = |\alpha| \, \|\mathbf{x}\|$ for every $\mathbf{x} \in \mathbb{R}^n$ and every $\alpha \in \mathbb{R}$. 3. $\|\mathbf{x}\| \geq 0$ for every $\mathbf{x} \in \mathbb{R}^n$, and $\|\mathbf{x}\| = 0$ implies $\mathbf{x} = 0$.

Example 3.7: There are three common examples:

1. The ℓ_2**-norm**

$$\|\mathbf{x}\|_2 := \sqrt{\sum_{k=1}^{n} x_k^2} = \sqrt{\mathbf{x}^\top \mathbf{x}}.$$

   This is just the usual Euclidean length of $\mathbf{x}$.

2. The ℓ_1**-norm**

$$\|\mathbf{x}\|_1 := \sum_{k=1}^{n} |x_k|.$$

   This is sometimes known as the **taxicab** or **Manhattan** norm, because it corresponds to the distance that a taxi has to drive on a rectangular grid of streets to get to $\mathbf{x} \in \mathbb{R}^2$.

3. The ℓ_∞**-norm**

$$\|\mathbf{x}\|_\infty := \max_{k=1,\ldots,n} |x_k|.$$

   This is sometimes known as the **maximum** norm.

The norms in the example above are all special cases of the ℓ_p-norm,

$$\|\mathbf{x}\|_p = \left(\sum_{k=1}^{n} |x_k|^p \right)^{1/p},$$

which is a norm for any real number $p \geq 1$. Increasing p means that more and more emphasis is given to the maximum element $|x_k|$.

Example 3.8: Consider the vectors $\mathbf{a} = (1, -2, 3)^\top$, $\mathbf{b} = (2, 0, -1)^\top$, and $\mathbf{c} = (0, 1, 4)^\top$.

The ℓ_1-, ℓ_2-, and ℓ_∞-norms are:

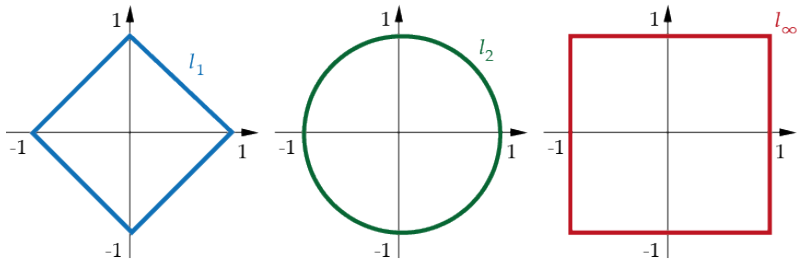
$$\begin{aligned} \|\mathbf{a}\|_1 &= 1 + 2 + 3 = 6 \\ \|\mathbf{b}\|_1 &= 2 + 0 + 1 = 3 \\ \|\mathbf{c}\|_1 &= 0 + 1 + 4 = 5 \end{aligned}$$

$$\begin{aligned} \|\mathbf{a}\|_2 &= \sqrt{1 + 4 + 9} \approx 3.74 \\ \|\mathbf{b}\|_2 &= \sqrt{4 + 0 + 1} \approx 2.24 \\ \|\mathbf{c}\|_2 &= \sqrt{0 + 1 + 16} \approx 4.12 \end{aligned}$$

$$\begin{aligned} \|\mathbf{a}\|_\infty &= \max\{1, 2, 3\} = 3 \\ \|\mathbf{b}\|_\infty &= \max\{2, 0, 1\} = 2 \\ \|\mathbf{c}\|_\infty &= \max\{0, 1, 4\} = 4 \end{aligned}$$

Notice that, for a single vector $\mathbf{x}$, the norms satisfy the ordering $\|\mathbf{x}\|_1 \geq \|\mathbf{x}\|_2 \geq \|\mathbf{x}\|_\infty$, but that vectors may be ordered differently by different norms.

Example 3.9: Sketch the 'unit circles' $\{\mathbf{x} \in \mathbb{R}^2 : \|\mathbf{x}\|_p = 1\}$ for $p = 1, 2, \infty$.



3.6 Matrix norms

We also use norms to measure the "size" of matrices. Since the set $\mathbb{R}^{n\times n}$ of $n \times n$ matrices with real entries is a vector space, we could just use a vector norm on this space. But usually we add an additional axiom.

A **matrix norm** is a real-valued function $\|\cdot\|$ on $\mathbb{R}^{n\times n}$ that satisfies:

1. $\|A + B\| \leq \|A\| + \|B\|$ for every $A, B \in \mathbb{R}^{n\times n}$.

2. $\|\alpha A\| = |\alpha| \, \|A\|$ for every $A \in \mathbb{R}^{n \times n}$ and every $\alpha \in \mathbb{R}$.
3. $\|A\| \geq 0$ for every $A \in \mathbb{R}^{n \times n}$ and $\|A\| = 0$ implies $A = 0$.
4. $\|AB\| \leq \|A\| \|B\|$ for every $A, B \in \mathbb{R}^{n \times n}$ (**consistency**).

> **Note**
>
> We usually want this additional axiom because matrices are more than just vectors. Some books call this a **submultiplicative norm** and define a "matrix norm" to satisfy just the first three properties, perhaps because (4) only works for square matrices.

Example 3.10: Frobenius norm

If we treat a matrix as a big vector with n^2 components, then the ℓ_2-norm is called the **Frobenius norm** of the matrix:

$$\|A\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij}^2}.$$

This norm is rarely used in numerical analysis because it is not induced by any vector norm (as we are about to define).

The most important matrix norms are so-called **induced** or **operator** norms. Remember that A is a linear map on $\mathbb{R}^n$, meaning that it maps every vector to another vector. So we can measure the size of A by how much it can stretch vectors with respect to a given vector norm. Specifically, if $\|\cdot\|_p$ is a vector norm, then the **induced** norm is defined as

$$\|A\|_p := \sup_{\mathbf{x} \neq \mathbf{0}} \frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_p} = \max_{\|\mathbf{x}\|_p = 1} \|A\mathbf{x}\|_p.$$

To see that the two definitions here are equivalent, use the fact that $\|\cdot\|_p$ is a vector norm. So by property (2) we have

$$\sup_{\mathbf{x} \neq \mathbf{0}} \frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_p} = \sup_{\mathbf{x} \neq \mathbf{0}} \left\| A \frac{\mathbf{x}}{\|\mathbf{x}\|_p} \right\|_p = \sup_{\|\mathbf{y}\|_p = 1} \|A\mathbf{y}\|_p = \max_{\|\mathbf{y}\|_p = 1} \|A\mathbf{y}\|_p.$$

> **Note**
>
> Usually we use the same notation for the induced matrix norm as for the original vector norm. The meaning should be clear from the context.

Example 3.11

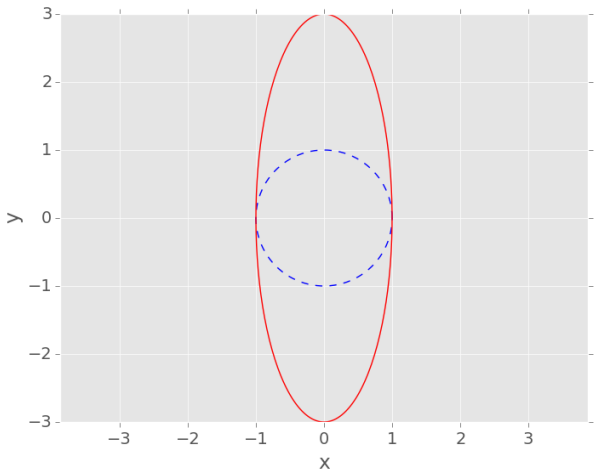
Let

$$A = \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix}.$$

In the ℓ_2-norm, a unit vector in $\mathbb{R}^2$ has the form $\mathbf{x} = (\cos\theta, \sin\theta)^\top$, so the image of the unit circle is

$$A\mathbf{x} = \begin{pmatrix} \sin\theta \\ 3\cos\theta \end{pmatrix}.$$

This is illustrated below:



The induced matrix norm is the maximum stretching of this unit circle, which is

$$\|A\|_2 = \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2 = \max_\theta \left(\sin^2\theta + 9\cos^2\theta\right)^{1/2} = \max_\theta \left(1 + 8\cos^2\theta\right)^{1/2} = 3.$$

Theorem 3.2: Induced norms are matrix norms

The induced norm corresponding to any vector norm is a matrix norm, and the two norms satisfy $\|A\mathbf{x}\| \leq \|A\|\|\mathbf{x}\|$ for any matrix $A \in \mathbb{R}^{n\times n}$ and any vector $\mathbf{x} \in \mathbb{R}^n$.

Proof:
Properties (1)-(3) follow from the fact that the vector norm satisfies the corresponding properties. To show (4), note that, by the definition above, we have for any vector $\mathbf{y} \in \mathbb{R}^n$ that

$$\|A\| \geq \frac{\|A\mathbf{y}\|}{\|\mathbf{y}\|} \quad \Longrightarrow \quad \|A\mathbf{y}\| \leq \|A\|\|\mathbf{y}\|.$$

Taking $\mathbf{y} = B\mathbf{x}$ for some $\mathbf{x}$ with $\|\mathbf{x}\| = 1$, we get

$$\|AB\mathbf{x}\| \leq \|A\|\|B\mathbf{x}\| \leq \|A\|\|B\|.$$

This holds in particular for the vector $\mathbf{x}$ that maximises $\|AB\mathbf{x}\|$, so

$$\|AB\| = \max_{\|\mathbf{x}\|=1} \|AB\mathbf{x}\| \leq \|A\|\|B\|.$$

It is cumbersome to compute the induced norms from their definition, but fortunately there are some very useful alternative formulae.

Theorem 3.3: Matrix norms induced by ℓ_1 and ℓ_∞

The matrix norms induced by the ℓ_1-norm and ℓ_∞-norm satisfy

$$\|A\|_1 = \max_{j=1,\ldots,n} \sum_{i=1}^{n} |a_{ij}|, \quad \text{(maximum column sum)}$$

$$\|A\|_\infty = \max_{i=1,\ldots,n} \sum_{j=1}^{n} |a_{ij}|. \quad \text{(maximum row sum)}$$

Proof:
We will prove the result for the ℓ_1-norm as an illustration of the method:

$$\|A\mathbf{x}\|_1 = \sum_{i=1}^{n} \left| \sum_{j=1}^{n} a_{ij}x_j \right| \leq \sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|\,|x_j| = \sum_{j=1}^{n} |x_j| \sum_{i=1}^{n} |a_{ij}|.$$

If we let

$$c = \max_{j=1,\ldots,n} \sum_{i=1}^{n} |a_{ij}|,$$

then

$$\|A\mathbf{x}\|_1 \leq c\|\mathbf{x}\|_1 \quad \Longrightarrow \quad \|A\|_1 \leq c.$$

Now let m be the column where the maximum sum is attained. If we choose $\mathbf{y}$ to be the vector with components $y_k = \delta_{km}$, then we have $\|A\mathbf{y}\|_1 = c$. Since $\|\mathbf{y}\|_1 = 1$, we must have that

$$\max_{\|\mathbf{x}\|_1=1} \|A\mathbf{x}\|_1 \geq \|A\mathbf{y}\|_1 = c \quad \Longrightarrow \quad \|A\|_1 \geq c.$$

The only way to satisfy both inequalities is if $\|A\|_1 = c$.

Example 3.12

For the matrix

$$A = \begin{pmatrix} -7 & 3 & -1 \\ 2 & 4 & 5 \\ -4 & 6 & 0 \end{pmatrix}$$

we have

$$\|A\|_1 = \max\{13, 13, 6\} = 13, \qquad \|A\|_\infty = \max\{11, 11, 10\} = 11.$$

What about the matrix norm induced by the ℓ_2-norm? This turns out to be related to the eigenvalues of A. Recall that $\lambda \in \mathbb{C}$ is an **eigenvalue** of A with associated **eigenvector u** if

$$A\mathbf{u} = \lambda\mathbf{u}.$$

We define the **spectral radius** $\rho(A)$ of A to be the maximum $|\lambda|$ over all eigenvalues λ of A.

Theorem 3.4: Spectral norm

The matrix norm induced by the ℓ_2-norm satisfies

$$\|A\|_2 = \sqrt{\rho(A^\top A)}.$$

As a result of the theorem above, this norm is sometimes known as the **spectral norm**.

Example 3.13

For our matrix

$$A = \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix},$$

we have

$$A^\top A = \begin{pmatrix} 0 & 3 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix} = \begin{pmatrix} 9 & 0 \\ 0 & 1 \end{pmatrix}.$$

We see that the eigenvalues of $A^\top A$ are $\lambda = 1, 9$, so $\|A\|_2 = \sqrt{9} = 3$ (as we calculated earlier).

Proof:
We want to show that

$$\max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2 = \max\{\sqrt{|\lambda|} \,:\, \lambda \text{ eigenvalue of } A^\top A\}.$$

For A real, $A^\top A$ is symmetric, so has real eigenvalues $\lambda_1 \le \lambda_2 \le \ldots \le \lambda_n$ with corresponding orthonormal eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_n$ in $\mathbb{R}^n$. (Orthonormal means that $\mathbf{u}_j^\top \mathbf{u}_k = \delta_{jk}$.) Note also

that all of the eigenvalues are non-negative, since

$$A^\top A\mathbf{u}_1 = \lambda_1 \mathbf{u}_1 \quad \implies \lambda_1 = \frac{\mathbf{u}_1^\top A^\top A\mathbf{u}_1}{\mathbf{u}_1^\top \mathbf{u}_1} = \frac{\|A\mathbf{u}_1\|_2^2}{\|\mathbf{u}_1\|_2^2} \geq 0.$$

So we want to show that $\|A\|_2 = \sqrt{\lambda_n}$. The eigenvectors form a basis, so every vector $\mathbf{x} \in \mathbb{R}^n$ can be expressed as a linear combination $\mathbf{x} = \sum_{k=1}^n \alpha_k \mathbf{u}_k$. Therefore

$$\|A\mathbf{x}\|_2^2 = \mathbf{x}^\top A^\top A\mathbf{x} = \mathbf{x}^\top \sum_{k=1}^n \alpha_k \lambda_k \mathbf{u}_k = \sum_{j=1}^n \alpha_j \mathbf{u}_j^\top \sum_{k=1}^n \alpha_k \lambda_k \mathbf{u}_k = \sum_{k=1}^n \alpha_k^2 \lambda_k,$$

where the last step uses orthonormality of the $\mathbf{u}_k$. It follows that

$$\|A\mathbf{x}\|_2^2 \leq \lambda_n \sum_{k=1}^n \alpha_k^2.$$

But if $\|\mathbf{x}\|_2 = 1$, then $\|\mathbf{x}\|_2^2 = \sum_{k=1}^n \alpha_k^2 = 1$, so $\|A\mathbf{x}\|_2^2 \leq \lambda_n$. To show that the maximum of $\|A\mathbf{x}\|_2^2$ is equal to λ_n, we can choose $\mathbf{x}$ to be the corresponding eigenvector $\mathbf{x} = \mathbf{u}_n$. In that case, $\alpha_1 = \ldots = \alpha_{n-1} = 0$ and $\alpha_n = 1$, so $\|A\mathbf{x}\|_2^2 = \lambda_n$.

3.7 Conditioning

Some linear systems are inherently more difficult to solve than others, because the solution is sensitive to small perturbations in the input. We will examine how to quantify this sensitivity and how to adjust our methods to control for it.

Example 3.14

Consider the linear system

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \implies \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

If we add a small rounding error $0 < \delta \ll 1$ to the data b_1 then

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+\delta \\ 1 \end{pmatrix} \implies \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \delta \\ 1 \end{pmatrix}.$$

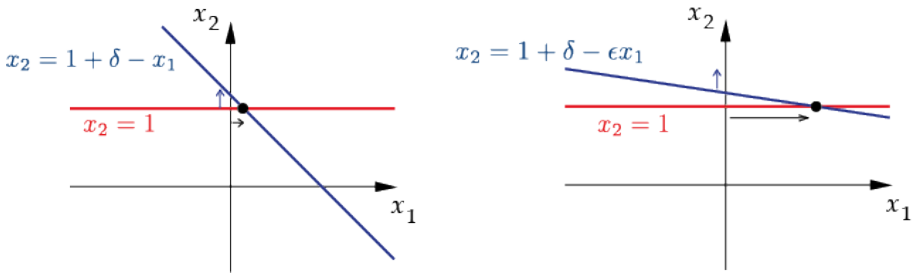
The solution is within rounding error of the true solution, so the system is called **well conditioned**.

Example 3.15

Now let $\epsilon \ll 1$ be a fixed positive number, and consider the linear system

$$\begin{pmatrix} \epsilon & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+\delta \\ 1 \end{pmatrix} \implies \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \delta/\epsilon \\ 1 \end{pmatrix}.$$

The true solution is still $(0,1)^\top$, but if the error δ is as big as the matrix entry ϵ, then the solution for x_1 will be completely wrong. This system is much more sensitive to errors in $\mathbf{b}$, so is called **ill-conditioned**.
Graphically, this system (right) is more sensitive to δ than the first system (left) because the two lines are closer to parallel:



To measure the **conditioning** of a linear system, consider the following estimate of the ratio of the relative errors in the output (x) versus the input (b):

$$\begin{aligned} \frac{|\text{relative error in } \mathbf{x}|}{|\text{relative error in } \mathbf{b}|} &= \frac{\|\delta\mathbf{x}\|/\|\mathbf{x}\|}{\|\delta\mathbf{b}\|/\|\mathbf{b}\|} = \left(\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|}\right)\left(\frac{\|\mathbf{b}\|}{\|\delta\mathbf{b}\|}\right) \\ &= \left(\frac{\|A^{-1}\delta\mathbf{b}\|}{\|\mathbf{x}\|}\right)\left(\frac{\|\mathbf{b}\|}{\|\delta\mathbf{b}\|}\right) \\ &\leq \frac{\|A^{-1}\|\|\delta\mathbf{b}\|}{\|\mathbf{x}\|}\left(\frac{\|\mathbf{b}\|}{\|\delta\mathbf{b}\|}\right) \\ &= \frac{\|A^{-1}\|\|\mathbf{b}\|}{\|\mathbf{x}\|} = \frac{\|A^{-1}\|\|A\mathbf{x}\|}{\|\mathbf{x}\|} \\ &\leq \|A^{-1}\|\|A\|. \end{aligned}$$

We define the **condition number** of a matrix A in some induced norm $\|\cdot\|_*$ to be

$$\kappa_*(A) = \|A^{-1}\|_*\|A\|_*.$$

If $\kappa_*(A)$ is large, then the solution will be sensitive to errors in $\mathbf{b}$, at least for some $\mathbf{b}$. A large condition number means that the matrix is close to being non-invertible (i.e. two rows are close to being linearly dependent).

Note

This is a "worst case" amplification of the error by a given matrix. The actual result will depend on $\delta\mathbf{b}$ (which we usually don't know if it arises from previous rounding error).

Note that $\det(A)$ will tell you whether a matrix is singular or not, but not whether it is ill-conditioned. Since $\det(\alpha A) = \alpha^n \det(A)$, the determinant can be made arbitrarily large or small by scaling (which does not change the condition number). For instance, the matrix

$$\begin{pmatrix} 10^{-50} & 0 \\ 0 & 10^{-50} \end{pmatrix}$$

has tiny determinant but is well-conditioned.

Example 3.16

Return to our earlier examples and consider the condition numbers in the 1-norm. We have (assuming $0 < \epsilon \ll 1$) that

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \implies A^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \implies \|A\|_1 = \|A^{-1}\|_1 = 2 \implies \kappa_1(A) = 4,$$

$$B = \begin{pmatrix} \epsilon & 1 \\ 0 & 1 \end{pmatrix} \implies B^{-1} = \frac{1}{\epsilon}\begin{pmatrix} 1 & -1 \\ 0 & \epsilon \end{pmatrix}$$

$$\implies \|B\|_1 = 2,\ \|B^{-1}\|_1 = \frac{1+\epsilon}{\epsilon} \implies \kappa_1(B) = \frac{2(1+\epsilon)}{\epsilon}.$$

For matrix B, $\kappa_1(B) \to \infty$ as $\epsilon \to 0$, showing that the matrix B is ill-conditioned.

Example 3.17

The **Hilbert matrix** H_n is the $n \times n$ symmetric matrix with entries

$$(h_n)_{ij} = \frac{1}{i+j-1}.$$

These matrices are notoriously ill-conditioned. For example, $\kappa_2(H_5) \approx 4.8 \times 10^5$, and $\kappa_2(H_{20}) \approx 2.5 \times 10^{28}$. Solving an associated linear system in floating-point arithmetic would be hopeless.

A practical limitation of the condition number is that you have to know A^{-1} before you can calculate it. We can always estimate $\|A^{-1}\|$ by taking some arbitrary vectors $\mathbf{x}$ and using

$$\|A^{-1}\| \geq \frac{\|\mathbf{x}\|}{\|\mathbf{b}\|}.$$

3.8 Iterative methods

For large systems, the $\mathcal{O}(n^3)$ cost of Gaussian elimination is prohibitive. Fortunately, many such systems that arise in practice are **sparse**, meaning that most of the entries of the matrix A are zero. In this case, we can often use iterative algorithms to do better than $\mathcal{O}(n^3)$.

In this course, we will only study algorithms for symmetric positive definite matrices. A matrix A is called **symmetric positive definite** (or **SPD**) if $\mathbf{x}^\top A\mathbf{x} > 0$ for every vector $\mathbf{x} \neq 0$.

> **Note**
>
> Recall that a symmetric matrix has real eigenvalues. It is positive definite iff all of its eigenvalues are positive.

Example 3.18

Show that the following matrix is SPD:

$$A = \begin{pmatrix} 3 & 1 & -1 \\ 1 & 4 & 2 \\ -1 & 2 & 5 \end{pmatrix}.$$

With $\mathbf{x} = (x_1, x_2, x_3)^\top$, we have

$$\begin{aligned} \mathbf{x}^\top A\mathbf{x} &= 3x_1^2 + 4x_2^2 + 5x_3^2 + 2x_1x_2 + 4x_2x_3 - 2x_1x_3 \\ &= x_1^2 + x_2^2 + 2x_3^2 + (x_1 + x_2)^2 + (x_1 - x_3)^2 + 2(x_2 + x_3)^2. \end{aligned}$$

This is positive for any non-zero vector $\mathbf{x} \in \mathbb{R}^3$, so A is SPD (eigenvalues 1.29, 4.14 and 6.57).

If A is SPD, then solving $A\mathbf{x} = \mathbf{b}$ is equivalent to minimizing the quadratic functional

$$f : \mathbb{R}^n \to \mathbb{R}, \qquad f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^\top A\mathbf{x} - \mathbf{b}^\top \mathbf{x}.$$

When A is SPD, this functional behaves like a U-shaped parabola, and has a unique finite global minimizer $\mathbf{x}_*$ such that $f(\mathbf{x}_*) < f(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \neq \mathbf{x}_*$.

To find $\mathbf{x}_*$, we need to set $\nabla f = \mathbf{0}$. We have

$$f(\mathbf{x}) = \tfrac{1}{2}\sum_{i=1}^{n} x_i \left(\sum_{j=1}^{n} a_{ij}x_j \right) - \sum_{j=1}^{n} b_j x_j$$

so

$$\begin{aligned}\frac{\partial f}{\partial x_k} &= \tfrac{1}{2}\left(\sum_{i=1}^{n} x_i a_{ik} + \sum_{j=1}^{n} a_{kj} x_j\right) - b_k \\ &= \tfrac{1}{2}\left(\sum_{i=1}^{n} a_{ki} x_i + \sum_{j=1}^{n} a_{kj} x_j\right) - b_k = \sum_{j=1}^{n} a_{kj} x_j - b_k.\end{aligned}$$

In the penultimate step we used the symmetry of A to write $a_{ik} = a_{ki}$. It follows that

$$\nabla f = A\mathbf{x} - \mathbf{b},$$

so locating the minimum of $f(\mathbf{x})$ is indeed equivalent to solving $A\mathbf{x} = \mathbf{b}$.

> **Note**
>
> Minimizing functions is a vast sub-field of numerical analysis known as **optimization**. We will only cover this specific case.

A popular class of methods for optimization are **line search** methods, where at each iteration the search is restricted to a single **search direction** $\mathbf{d}_k$. The iteration takes the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k.$$

The **step size** α_k is chosen by minimizing $f(\mathbf{x})$ along the line $\mathbf{x} = \mathbf{x}_k + \alpha \mathbf{d}_k$. For our functional above, we have

$$f(\mathbf{x}_k + \alpha \mathbf{d}_k) = (\tfrac{1}{2}\mathbf{d}_k^\top A \mathbf{d}_k)\alpha^2 + \mathbf{d}_k^\top (A\mathbf{x}_k - \mathbf{b})\alpha + \tfrac{1}{2}\mathbf{x}_k^\top A \mathbf{x}_k - \mathbf{b}^\top \mathbf{x}_k.$$

This is a quadratic in α, and the coefficient of α^2 is positive because A is positive definite. It is therefore a U-shaped parabola and achieves its minimum when

$$\frac{\partial f}{\partial \alpha} = \mathbf{d}_k^\top A \mathbf{d}_k \alpha + \mathbf{d}_k^\top (A\mathbf{x}_k - \mathbf{b}) = 0.$$

Defining the **residual** $\mathbf{r}_k := A\mathbf{x}_k - \mathbf{b}$, we see that the desired choice of step size is

$$\alpha_k = -\frac{\mathbf{d}_k^\top \mathbf{r}_k}{\mathbf{d}_k^\top A \mathbf{d}_k}.$$

Different line search methods differ in how the search direction $\mathbf{d}_k$ is chosen at each iteration. For example, the **method of steepest descent** sets

$$\mathbf{d}_k = -\nabla f(\mathbf{x}_k) = -\mathbf{r}_k,$$

where we have remembered the gradient formula above.

Example 3.19

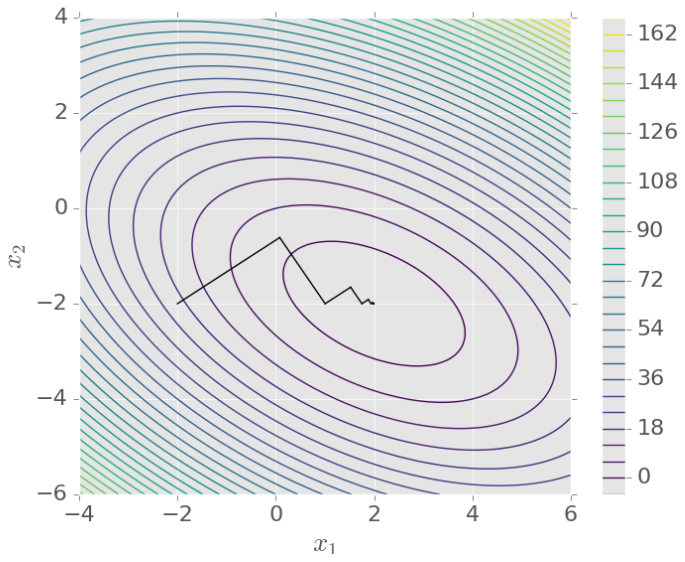
Use the method of steepest descent to solve the system

$$\begin{pmatrix} 3 & 2 \\ 2 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ -8 \end{pmatrix}.$$

Starting from $\mathbf{x}_0 = (-2, -2)^\top$, we get

$$\mathbf{d}_0 = \mathbf{b} - A\mathbf{x}_0 = \begin{pmatrix} 12 \\ 8 \end{pmatrix} \implies \alpha_0 = \frac{\mathbf{d}_0^\top \mathbf{d}_0}{\mathbf{d}_0^\top A \mathbf{d}_0} = \frac{208}{1200}$$
$$\implies \mathbf{x}_1 = \mathbf{x}_0 + \alpha_0 \mathbf{d}_0 \approx \begin{pmatrix} 0.08 \\ -0.613 \end{pmatrix}.$$

Continuing the iteration, $\mathbf{x}_k$ proceeds towards the solution $(2, -2)^\top$ as illustrated below. The coloured contours show the value of $f(x_1, x_2)$.



Unfortunately, the method of steepest descent can be slow to converge. In the **conjugate gradient method**, we still take $\mathbf{d}_0 = -\mathbf{r}_0$, but subsequent search directions $\mathbf{d}_k$ are chosen to be A**-conjugate**, meaning that

$$\mathbf{d}_{k+1}^\top A \mathbf{d}_k = 0.$$

This means that minimization in one direction does not undo the previous minimizations.

In particular, we construct $\mathbf{d}_{k+1}$ by writing

$$\mathbf{d}_{k+1} = -\mathbf{r}_{k+1} + \beta_k \mathbf{d}_k,$$

then choosing the scalar β_k such that $\mathbf{d}_{k+1}^\top A\mathbf{d}_k = 0$. This gives

$$0 = (-\mathbf{r}_{k+1} + \beta_k \mathbf{d}_k)^\top A\mathbf{d}_k = -\mathbf{r}_{k+1}^\top A\mathbf{d}_k + \beta_k \mathbf{d}_k^\top A\mathbf{d}_k$$

and hence

$$\beta_k = \frac{\mathbf{r}_{k+1}^\top A\mathbf{d}_k}{\mathbf{d}_k^\top A\mathbf{d}_k}.$$

Thus we get the basic conjugate gradient algorithm.

Algorithm 3.2: Conjugate gradient method

Start with an initial guess $\mathbf{x}_0$ and initial search direction $\mathbf{d}_0 = -\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$. For each $k = 0, 1, \ldots$, do the following:

1. Compute step size
$$\alpha_k = -\frac{\mathbf{d}_k^\top \mathbf{r}_k}{\mathbf{d}_k^\top A\mathbf{d}_k}.$$
2. Compute $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$.
3. Compute residual $\mathbf{r}_{k+1} = A\mathbf{x}_{k+1} - \mathbf{b}$.
4. If $\|\mathbf{r}_{k+1}\| <$ tolerance, output $\mathbf{x}_{k+1}$ and stop.
5. Determine new search direction
$$\mathbf{d}_{k+1} = -\mathbf{r}_{k+1} + \beta_k \mathbf{d}_k \quad \text{where} \quad \beta_k = \frac{\mathbf{r}_{k+1}^\top A\mathbf{d}_k}{\mathbf{d}_k^\top A\mathbf{d}_k}.$$

Example 3.20

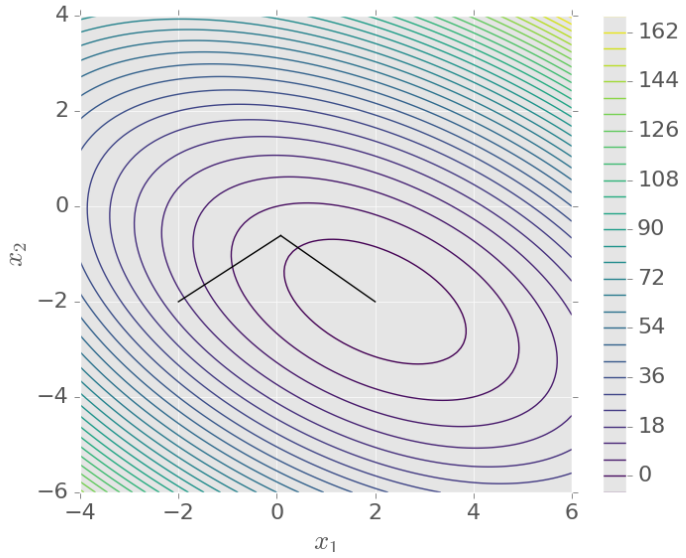
Solve our previous example with the conjugate gradient method.
Starting with $\mathbf{x}_0 = (-2, -2)^\top$, the first step is the same as in steepest descent, giving $\mathbf{x}_1 = (0.08, -0.613)^\top$. But then we take

$$\mathbf{r}_1 = A\mathbf{x}_1 - \mathbf{b} = \begin{pmatrix} -2.99 \\ 4.48 \end{pmatrix}, \quad \beta_0 = \frac{\mathbf{r}_1^\top A\mathbf{d}_0}{\mathbf{d}_0^\top A\mathbf{d}_0} = 0.139, \quad \mathbf{d}_1 = -\mathbf{r}_1 + \beta_0 \mathbf{d}_0 = \begin{pmatrix} 4.66 \\ -3.36 \end{pmatrix}.$$

The second iteration then gives

$$\alpha_1 = -\frac{\mathbf{d}_1^\top \mathbf{r}_1}{\mathbf{d}_1^\top A\mathbf{d}_1} = 0.412 \implies \mathbf{x}_2 = \mathbf{x}_1 + \alpha_1 \mathbf{d}_1 = \begin{pmatrix} 2 \\ -2 \end{pmatrix}.$$

This time there is no zig-zagging and the solution is reached in just two iterations:



In exact arithmetic, the conjugate gradient method will always give the exact answer in n iterations – one way to see this is to use the following.

Theorem 3.5

The residuals $\mathbf{r}_k := A\mathbf{x}_k - \mathbf{b}$ at each stage of the conjugate gradient method are mutually orthogonal, meaning $\mathbf{r}_j^\top \mathbf{r}_k = 0$ for $j = 0, \ldots, k-1$.

After n iterations, the only residual vector that can be orthogonal to all of the previous ones is $\mathbf{r}_n = \mathbf{0}$, so $\mathbf{x}_n$ must be the exact solution.

In practice, conjugate gradients is not competitive as a direct method. It is computationally intensive, and rounding errors can destroy the orthogonality, meaning that more than n iterations may be required. Instead, its main use is for large sparse systems. For suitable matrices (perhaps after **preconditioning**), it can converge very rapidly.

We can save computation by using the alternative formulae

$$\mathbf{r}_{k+1} = \mathbf{r}_k + \alpha_k A\mathbf{d}_k, \quad \alpha_k = \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{d}_k^\top A\mathbf{d}_k}, \quad \beta_k = \frac{\mathbf{r}_{k+1}^\top \mathbf{r}_{k+1}}{\mathbf{r}_k^\top \mathbf{r}_k}.$$

With these formulae, each iteration requires only one matrix-vector product, two vector-vector products, and three vector additions. Compare this to the basic algorithm above which requires two matrix-vector products, four vector-vector products and three vector additions.

Knowledge checklist

Key topics:

1. Direct and iterative methods for solving linear systems: triangular systems, Gaussian elimination, LU decomposition, and iterative algorithms.
2. Vector and matrix norms, including induced matrix norms and their role in error analysis.
3. Conditioning and the condition number: sensitivity of solutions to input errors and implications for numeric stability.

Key skills:

- Formulate and solve linear systems using direct and iterative methods (e.g., Gaussian elimination, LU decomposition).
- Apply norms to measure errors and conditioning, and calculate condition numbers.
- Analyze computational complexity and efficiency of matrix algorithms.

4 Calculus

4.1 Differentiation

How do we differentiate functions numerically?

4.1.1 Basics

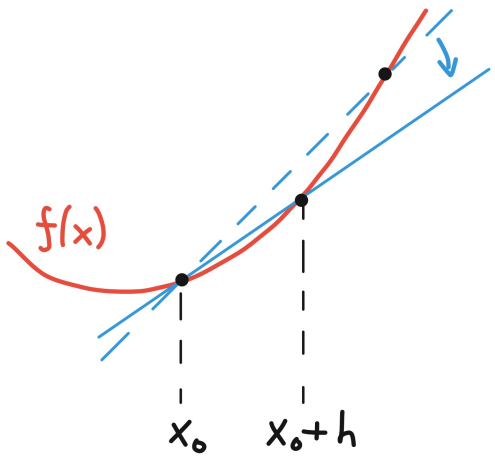
The definition of the derivative as

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h},$$

suggests an obvious approximation: just pick some small finite h to give the estimate

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

For $h > 0$ this is called a **forward difference** (and, for $h < 0$, a **backward difference**). It is an example of a **finite-difference formula**.



Of course, what we are doing with the forward difference is approximating $f'(x_0)$ by the slope of the linear interpolant for f at the nodes x_0 and $x_1 = x_0 + h$. So we could also have derived the forward difference formula by starting with the Lagrange form of the interpolating polynomial,

$$f(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1) + \frac{f''(\xi)}{2}(x - x_0)(x - x_1)$$

for some $\xi \in [x_0, x_1]$. Differentiating – and remembering that ξ depends on x, so that we need to use the chain rule – we get

$$\begin{aligned} f'(x) &= \frac{1}{x_0 - x_1} f(x_0) + \frac{1}{x_1 - x_0} f(x_1) + \frac{f''(\xi)}{2}(2x - x_0 - x_1) \\ &\quad + \frac{f'''(\xi)}{2}\left(\frac{\mathrm{d}\xi}{\mathrm{d}x}\right)(x - x_0)(x - x_1), \\ \implies f'(x_0) &= \frac{f(x_1) - f(x_0)}{x_1 - x_0} + f''(\xi)\frac{x_0 - x_1}{2}. \end{aligned}$$

Equivalently,

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - f''(\xi)\frac{h}{2}.$$

This shows that the **truncation error** for our forward difference approximation is $-f''(\xi)h/2$, for some $\xi \in [x_0, x_0 + h]$. In other words, a smaller interval or a less "wiggly" function will lead to a better estimate, as you would expect.

Another way to estimate the truncation error is to use Taylor's theorem, which tells us that

$$f(x_0 + h) = f(x_0) + hf'(x_0) + h^2 \frac{f''(\xi)}{2},$$

for some ξ between x_0 and $x_0 + h$. Rearranging this will give back the forward difference error formula.

Example 4.1: Derivative of $f(x) = \log(x)$ at $x_0 = 2$.

Using a forward-difference, we get the following sequence of approximations:

h	Forward difference	Truncation error
1	0.405465	0.0945349
0.1	0.487902	0.0120984
0.01	0.498754	0.00124585
0.001	0.499875	0.000124958

Indeed the error is linear in h, and we estimate that it is approximately $0.125h$ when h is small. This agrees with the error formula above, since $f''(x) = -x^{-2}$, so we expect $-f''(\xi)/2 \approx \frac{1}{8}$.

Since the error is linearly proportional to h, the approximation is called **linear**, or **first order**.

4.1.2 Higher-order finite differences

To get a higher-order approximation, we can differentiate a higher degree interpolating polynomial. This means that we need more nodes.

Example 4.2: Central difference

Take three nodes x_0, $x_1 = x_0 + h$, and $x_2 = x_0 + 2h$. Then the Lagrange form of the interpolating polynomial is

$$\begin{aligned} f(x) = &\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) \\ &+ \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2) \\ &+ \frac{f'''(\xi)}{3!}(x-x_0)(x-x_1)(x-x_2). \end{aligned}$$

Differentiating, we get

$$\begin{aligned} f'(x) = &\frac{2x-x_1-x_2}{(x_0-x_1)(x_0-x_2)} f(x_0) \\ &+ \frac{2x-x_0-x_2}{(x_1-x_0)(x_1-x_2)} f(x_1) \\ &+ \frac{2x-x_0-x_1}{(x_2-x_0)(x_2-x_1)} f(x_2) \\ &+ \frac{f'''(\xi)}{6}\Big((x-x_1)(x-x_2) + (x-x_0)(x-x_2) + (x-x_0)(x-x_1)\Big) \\ &+ \frac{f^{(4)}(\xi)}{6}\left(\frac{\mathrm{d}\xi}{\mathrm{d}x}\right)(x-x_0)(x-x_1)(x-x_2). \end{aligned}$$

Now substitute in $x = x_1$ to evaluate this at the central point:

$$\begin{aligned} f'(x_1) &= \frac{x_1 - x_2}{(x_0 - x_1)(x_0 - x_2)} f(x_0) \\ &\quad + \frac{2x_1 - x_0 - x_2}{(x_1 - x_0)(x_1 - x_2)} f(x_1) + \frac{x_1 - x_0}{(x_2 - x_0)(x_2 - x_1)} f(x_2) \\ &\quad + \frac{f'''(\xi)}{6}(x_1 - x_0)(x_1 - x_2), \\ &= \frac{-h}{2h^2} f(x_0) + 0 + \frac{h}{2h^2} f(x_2) - \frac{f'''(\xi)}{6} h^2 \\ &= \frac{f(x_1 + h) - f(x_1 - h)}{2h} - \frac{f'''(\xi)}{6} h^2. \end{aligned}$$

This is called a **central difference** approximation for $f'(x_1)$, and is frequently used in practice.

To see the quadratic behaviour of the truncation error, go back to our earlier example.

Example 4.3: Derivative of $f(x) = \log(x)$ at $x = 2$

h	Forward difference	Truncation error	Central difference	Truncation error
1	0.405465	0.0945349	0.549306	-0.0493061
0.1	0.487902	0.0120984	0.500417	-0.000417293
0.01	0.498754	0.00124585	0.500004	-4.16673e-06
0.001	0.499875	0.000124958	0.500000	-4.16666e-08

The truncation error for the central difference is about $0.04h^2$, which agrees with the formula since $f'''(\xi) \approx 2/2^3 = \frac{1}{4}$ when h is small.

4.1.3 Rounding error

The problem with numerical differentiation is that it involves subtraction of nearly-equal numbers. As h gets smaller, the problem gets worse.

To quantify this for the central difference, suppose that we have the correctly rounded values of $f(x_1 \pm h)$, so that

$$\text{fl}[f(x_1 + h)] = (1 + \delta_1) f(x_1 + h), \qquad \text{fl}[f(x_1 - h)] = (1 + \delta_2) f(x_1 - h),$$

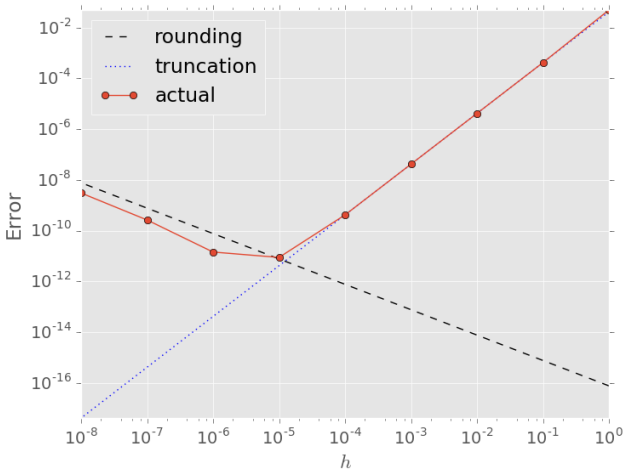
where $|\delta_1|, |\delta_2| \leq \epsilon_{\mathrm{M}}$. Ignoring the rounding error in dividing by $2h$, we then have that

$$\begin{aligned}
&\left| f'(x_1) - \frac{\mathrm{fl}[f(x_1+h)] - \mathrm{fl}[f(x_1-h)]}{2h} \right| \\
=&\left| -\frac{f'''(\xi)}{6}h^2 - \frac{\delta_1 f(x_1+h) - \delta_2 f(x_1-h)}{2h} \right| \\
\leq&\frac{|f'''(\xi)|}{6}h^2 + \epsilon_{\mathrm{M}}\frac{|f(x_1+h)| + |f(x_1-h)|}{2h} \\
\leq&\frac{h^2}{6}\max_{[x_1-h,x_1+h]}|f'''(\xi)| + \frac{\epsilon_{\mathrm{M}}}{h}\max_{[x_1-h,x_1+h]}|f(\xi)|.
\end{aligned}$$

The first term is the truncation error, which tends to zero as $h \to 0$. But the second term is the rounding error, which tends to infinity as $h \to 0$.

Example 4.4: Derivative of $f(x) = \log(x)$ at $x = 2$ again

Here is a comparison of the terms in the inequality above (the red points are the left-hand side), shown on logarithmic scales, using $\xi = 2$ to estimate the maxima.



You see that once h is small enough, rounding error takes over and the error in the computed derivative starts to increase again.

4.1.4 Richardson extrapolation

Finding higher-order formulae by differentiating Lagrange polynomials is tedious, and there is a simpler trick to obtain higher-order formulae, called **Richardson extrapolation**.

We begin from the central-difference formula. Since we will use formulae with different h, let us define the notation

$$D_h := \frac{f(x_1+h) - f(x_1-h)}{2h}.$$

Now use Taylor's theorem to expand more terms in the truncation error:

$$\begin{aligned} f(x_1 \pm h) = f(x_1) \pm f'(x_1)h + \frac{f''(x_1)}{2}h^2 \pm \frac{f'''(x_1)}{3!}h^3 \\ + \frac{f^{(4)}(x_1)}{4!}h^4 \pm \frac{f^{(5)}(x_1)}{5!}h^5 + \mathcal{O}(h^6). \end{aligned}$$

Substituting into the formula for D_h, the even powers of h cancel and we get

$$\begin{aligned} D_h &= \frac{1}{2h}\Big(2f'(x_1)h + 2f'''(x_1)\frac{h^3}{6} + 2f^{(5)}(x_1)\frac{h^5}{120} + \mathcal{O}(h^7)\Big) \\ &= f'(x_1) + f'''(x_1)\frac{h^2}{6} + f^{(5)}(x_1)\frac{h^4}{120} + \mathcal{O}(h^6). \end{aligned}$$

> **Note**
>
> You may not have seen the **big-Oh notation**. When we write $f(x) = \mathcal{O}(g(x))$, we mean
>
> $$\lim_{x\to 0} \frac{|f(x)|}{|g(x)|} \leq M < \infty.$$
>
> So the error is $\mathcal{O}(h^6)$ if it gets smaller at least as fast as h^6 as $h \to 0$ (essentially, it contains no powers of h less than 6).

The leading term in the error here has the same coefficient $h^2/6$ as the truncation error we derived earlier, although we have now expanded the error to higher powers of h.

The trick is to apply the same formula with different step-sizes, typically h and $h/2$:

$$\begin{aligned} D_h &= f'(x_1) + f'''(x_1)\frac{h^2}{6} + f^{(5)}(x_1)\frac{h^4}{120} + \mathcal{O}(h^6), \\ D_{h/2} &= f'(x_1) + f'''(x_1)\frac{h^2}{2^2(6)} + f^{(5)}(x_1)\frac{h^4}{2^4(120)} + \mathcal{O}(h^6). \end{aligned}$$

We can then eliminate the h^2 term by simple algebra:

$$\begin{aligned} D_h - 2^2 D_{h/2} &= -3f'(x_1) + \left(1 - \frac{2^2}{2^4}\right) f^{(5)}(x_1)\frac{h^4}{120} + \mathcal{O}(h^6), \\ \implies \quad D_h^{(1)} &:= \frac{2^2 D_{h/2} - D_h}{3} = f'(x_1) - f^{(5)}(x_1)\frac{h^4}{480} + \mathcal{O}(h^6). \end{aligned}$$

The new formula $D_h^{(1)}$ is 4th-order accurate.

Example 4.5: Derivative of $f(x) = \log(x)$ at $x = 2$ (central difference).

h	D_h	Error	$D_h^{(1)}$	Error
1.0	0.5493061443	0.04930614433	0.4979987836	0.00200121642
0.1	0.5004172928	0.00041729278	0.4999998434	1.56599487e-07
0.01	0.5000041667	4.16672916e-06	0.5000000000	1.56388791e-11
0.001	0.5000000417	4.16666151e-08	0.5000000000	9.29256672e-14

In fact, we could have applied this **Richardson extrapolation** procedure without knowing the coefficients of the error series. If we have some general order-n approximation

$$D_h = f'(x) + Ch^n + \mathcal{O}(h^{n+1}),$$

then we can always evaluate it with $h/2$ to get

$$D_{h/2} = f'(x) + C\frac{h^n}{2^n} + \mathcal{O}(h^{n+1})$$

and then eliminate the h^n term to get a new approximation

$$D_h^{(1)} := \frac{2^n D_{h/2} - D_h}{2^n - 1} = f'(x) + \mathcal{O}(h^{n+1}).$$

Note

The technique is used not only in differentiation but also in **Romberg integration** and the **Bulirsch-Stoer method** for solving ODEs.

Note

There is nothing special about taking $h/2$; we could have taken $h/3$ or even $2h$, and modified the formula accordingly. But $h/2$ is usually convenient.

Furthermore, Richardson extrapolation can be applied iteratively. In other words, we can now combine $D_h^{(1)}$ and $D_{h/2}^{(1)}$ to get an even higher order approximation $D_h^{(2)}$, and so on.

Example 4.6: Iterated Richardson extrapolation for central differences.

From

$$D_h^{(1)} = f'(x_1) + C_1 h^4 + \mathcal{O}(h^6),$$
$$D_{h/2}^{(1)} = f'(x_1) + C_1\frac{h^4}{2^4} + \mathcal{O}(h^6),$$

we can eliminate the h^4 term to get the 6th-order approximation

$$D_h^{(2)} := \frac{2^4 D_{h/2}^{(1)} - D_h^{(1)}}{2^4 - 1}.$$

i Note

Lewis Fry Richardson (1881–1953) was from Newcastle and an undergraduate there (when it was still a College of Durham). He was the first person to apply mathematics (finite differences) to weather prediction, and was ahead of his time: in the absence of electronic computers, he estimated that 60,000 people would be needed to predict the next day's weather!

4.2 Numerical integration

How do we calculate integrals numerically?

The definite integral

$$I(f) := \int_a^b f(x)\,\mathrm{d}x$$

can usually not be evaluated in closed form. To approximate it numerically, we can use a **quadrature formula**

$$I_n(f) := \sum_{k=0}^{n} \sigma_k f(x_k),$$

where $x_0, \ldots, x_n$ are a set of **nodes** and $\sigma_0, \ldots, \sigma_n$ are a set of corresponding **weights**.

i Note

The nodes are also known as **quadrature points** or **abscissas**, and the weights as **coefficients**.

Example 4.7: The trapezium rule

$$I_1(f) = \frac{b-a}{2}\Big(f(a) + f(b)\Big).$$

This is the quadrature formula above with $x_0 = a$, $x_1 = b$, $\sigma_0 = \sigma_1 = \frac{1}{2}(b-a)$.

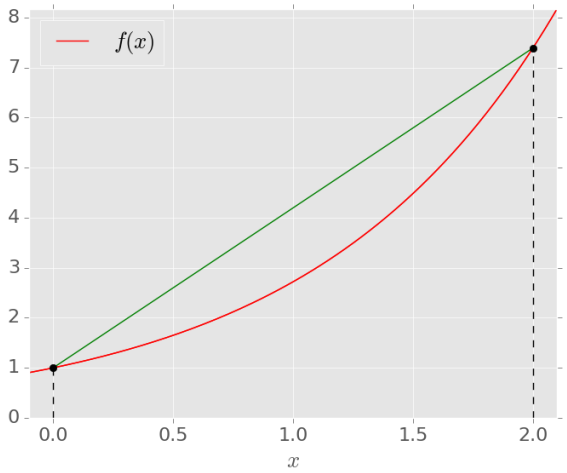
For example, with $a = 0$, $b = 2$, $f(x) = \mathrm{e}^x$, we get

$$I_1(f) = \frac{2-0}{2}(\mathrm{e}^0 + \mathrm{e}^2) = 8.389 \quad \text{to 4 s.f.}$$

The exact answer is

$$I(f) = \int_0^2 \mathrm{e}^x \,\mathrm{d}x = \mathrm{e}^2 - \mathrm{e}^0 = 6.389 \quad \text{to 4 s.f.}$$

Graphically, $I_1(f)$ measures the area under the straight line that interpolates f at the ends:



4.2.1 Newton-Cotes formulae

We can derive a family of "interpolatory" quadrature formulae by integrating interpolating polynomials of different degrees. We will also get error estimates using the interpolation error theorem.

Let $x_0, \ldots, x_n \in [a, b]$, where $x_0 < x_1 < \cdots < x_n$, be a set of $n + 1$ nodes, and let $p_n \in \mathcal{P}_n$ be the polynomial that interpolates f at these nodes. This may be written in Lagrange form as

$$p_n(x) = \sum_{k=0}^{n} f(x_k)\ell_k(x), \quad \text{where} \quad \ell_k(x) = \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x - x_j}{x_k - x_j}.$$

To approximate $I(f)$, we integrate $p_n(x)$ to define the quadrature formula

$$I_n(f) := \int_a^b \sum_{k=0}^{n} f(x_k)\ell_k(x)\,\mathrm{d}x = \sum_{k=0}^{n} f(x_k)\int_a^b \ell_k(x)\,\mathrm{d}x.$$

In other words,

$$I_n(f) := \sum_{k=0}^{n} \sigma_k f(x_k), \quad \text{where} \quad \sigma_k = \int_a^b \ell_k(x)\,\mathrm{d}x.$$

When the nodes are equidistant, this is called a **Newton-Cotes formula**. If $x_0 = a$ and $x_n = b$, it is called a **closed Newton-Cotes formula**.

> **Note**
>
> An **open Newton-Cotes formula** has nodes $x_i = a + (i+1)h$ for $h = (b-a)/(n+2)$.

Example 4.8: Trapezium rule

This is the closed Newton-Cotes formula with $n = 1$. To see this, let $x_0 = a$, $x_1 = b$. Then

$$\begin{aligned}
\ell_0(x) = \frac{x-b}{a-b} \implies \sigma_0 &= \int_a^b \ell_0(x)\,\mathrm{d}x \\
&= \frac{1}{a-b}\int_a^b (x-b)\,\mathrm{d}x \\
&= \frac{1}{2(a-b)}(x-b)^2\Big|_a^b \\
&= \frac{b-a}{2},
\end{aligned}$$

and

$$\begin{aligned}
\ell_1(x) = \frac{x-a}{b-a} \implies \sigma_1 &= \int_a^b \ell_1(x)\,\mathrm{d}x \\
&= \frac{1}{b-a}\int_a^b (x-a)\,\mathrm{d}x \\
&= \frac{1}{2(b-a)}(x-a)^2\Big|_a^b = \frac{b-a}{2}.
\end{aligned}$$

This gives

$$I_1(f) = \sigma_0 f(a) + \sigma_1 f(b) = \frac{b-a}{2}(f(a) + f(b)).$$

Theorem 4.1

Let f be continuous on $[a,b]$ with $n+1$ continuous derivatives on (a,b). Then the

Newton-Cotes formula above satisfies the error bound

$$|I(f) - I_n(f)| \leq \frac{\max_{\xi\in[a,b]} |f^{(n+1)}(\xi)|}{(n+1)!} \int_a^b |(x-x_0)(x-x_1)\cdots(x-x_n)|\,\mathrm{d}x.$$

Proof:
First note that the error in the Newton-Cotes formula may be written

$$\begin{aligned} |I(f) - I_n(f)| &= \left| \int_a^b f(x)\,\mathrm{d}x - \int_a^b p_n(x)\,\mathrm{d}x \right| \\ &= \left| \int_a^b [f(x) - p_n(x)]\,\mathrm{d}x \right| \\ &\leq \int_a^b |f(x) - p_n(x)|\,\mathrm{d}x. \end{aligned}$$

Now recall the interpolation error theorem, which says that, for each $x \in [a,b]$, we can write

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)(x-x_1)\cdots(x-x_n)$$

for some $\xi \in (a,b)$. The theorem simply follows by inserting this into the inequality above. □

Example 4.9: Trapezium rule error

Let $M_2 = \max_{\xi\in[a,b]} |f''(\xi)|$. Here the theorem reduces to

$$\begin{aligned} |I(f) - I_1(f)| &\leq \frac{M_2}{(1+1)!} \int_a^b |(x-a)(x-b)|\,\mathrm{d}x \\ &= \frac{M_2}{2!} \int_a^b (x-a)(b-x)\,\mathrm{d}x \\ &= \frac{(b-a)^3}{12} M_2. \end{aligned}$$

For our earlier example with $a = 0$, $b = 2$, $f(x) = \mathrm{e}^x$, the estimate gives

$$|I(f) - I_1(f)| \leq \tfrac{1}{12}(2^3)\mathrm{e}^2 \approx 4.926.$$

This is an overestimate of the actual error which was ≈ 2.000.

The theorem suggests that the accuracy of I_n is limited both by the smoothness of f (outside our control) and by the location of the nodes x_k. If the nodes are free to be chosen, then we can use **Gaussian integration** (more to follow on that topic).

Note

As with interpolation, taking a high n is not usually a good idea. One can prove for the closed Newton-Cotes formula that

$$\sum_{k=0}^{n} |\sigma_k| \to \infty \quad \text{as} \quad n \to \infty.$$

This makes the quadrature vulnerable to rounding errors for large n.

4.2.2 Composite Newton-Cotes formulae

Since the Newton-Cotes formulae are based on polynomial interpolation at equally-spaced points, the results do not converge as the number of nodes increases. A better way to improve accuracy is to divide the interval $[a, b]$ into m subintervals $[x_{i-1}, x_i]$ of equal length

$$h := \frac{b-a}{m},$$

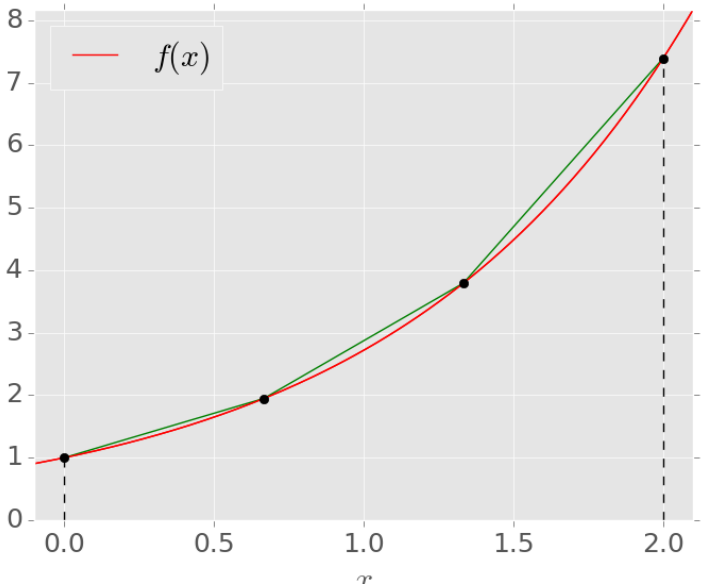
and use a Newton-Cotes formula of small degree n on each subinterval.

Example 4.10: Composite trapezium rule

Applying the trapezium rule $I_1(f)$ on each subinterval gives

$$\begin{aligned} C_{1,m}(f) &= \frac{h}{2}\left[f(x_0) + f(x_1) + f(x_1) + f(x_2) + \ldots + f(x_{m-1}) + f(x_m)\right], \\ &= h\left[\tfrac{1}{2}f(x_0) + f(x_1) + f(x_2) + \ldots + f(x_{m-1}) + \tfrac{1}{2}f(x_m)\right]. \end{aligned}$$

We are effectively integrating a piecewise-linear approximation of $f(x)$; here we show $m = 3$ for our test problem $f(x) = \mathrm{e}^x$ on $[0, 2]$:



Look at what happens as we increase m for our test problem:

m	h	$C_{1,m}(f)$	$\lvert I(f) - C_{1,m}(f)\rvert$
1	2	8.389	2.000
2	1	6.912	0.524
4	0.5	6.522	0.133
8	0.25	6.422	0.033
16	0.125	6.397	0.008
32	0.0625	6.391	0.002

When we halve the sub-interval h, the error goes down by a factor 4, suggesting that we have quadratic convergence, i.e., $\mathcal{O}(h^2)$.
To show this theoretically, we can apply the Newton-Cotes error theorem in each subinterval. In $[x_{i-1}, x_i]$ we have

$$|I(f) - I_1(f)| \leq \frac{\max_{\xi\in[x_{i-1},x_i]} |f''(\xi)|}{2!} \int_{x_{i-1}}^{x_i} |(x - x_{i-1})(x - x_i)| \, \mathrm{d}x$$

Note that

$$\begin{aligned}
\int_{x_{i-1}}^{x_i} |(x - x_{i-1})(x - x_i)| \, \mathrm{d}x &= \int_{x_{i-1}}^{x_i} (x - x_{i-1})(x_i - x) \, \mathrm{d}x \\
&= \int_{x_{i-1}}^{x_i} \big[-x^2 + (x_{i-1} + x_i)x - x_{i-1}x_i \big] \, \mathrm{d}x \\
&= \big[-\tfrac{1}{3}x^3 + \tfrac{1}{2}(x_{i-1} + x_i)x^2 - x_{i-1}x_i x \big]_{x_{i-1}}^{x_i} \\
&= \tfrac{1}{6}x_i^3 - \tfrac{1}{2}x_{i-1}x_i^2 + \tfrac{1}{2}x_{i-1}^2 x_i - \tfrac{1}{6}x_{i-1}^3 \\
&= \tfrac{1}{6}(x_i - x_{i-1})^3 = \tfrac{1}{6}h^3.
\end{aligned}$$

So overall

$$\begin{aligned}
|I(f) - C_{1,m}(f)| &\leq \tfrac{1}{2} \max_i \left(\max_{\xi\in[x_{i-1},x_i]} |f''(\xi)| \right) m \frac{h^3}{6} \\
&= \frac{mh^3}{12} \max_{\xi\in[a,b]} |f''(\xi)| = \frac{b-a}{12} h^2 \max_{\xi\in[a,b]} |f''(\xi)|.
\end{aligned}$$

As long as f is sufficiently smooth, this shows that the composite trapezium rule will converge as $m \to \infty$. Moreover, the convergence will be $\mathcal{O}(h^2)$.

4.2.3 Exactness

From the Newton-Cotes error theorem, we see that the Newton-Cotes formula $I_n(f)$ will give the exact answer if $f^{(n+1)} = 0$. In other words, it will be exact if $f \in \mathcal{P}_n$.

Example 4.11

The trapezium rule $I_1(f)$ is exact for all linear polynomials $f \in \mathcal{P}_1$.

The **degree of exactness** of a quadrature formula is the largest integer n for which the formula is exact for all polynomials in $\mathcal{P}_n$.

To check whether a quadrature formula has degree of exactness n, it suffices to check whether it is exact for the basis 1, x, $x^2, \ldots, x^n$.

Example 4.12: Simpson's rule

This is the $n = 2$ closed Newton-Cotes formula

$$I_2(f) = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right],$$

derived by integrating a quadratic interpolating polynomial. Let us find its degree of exactness:

$$\begin{aligned}
I(1) &= \int_a^b \mathrm{d}x = (b-a),\\
I_2(1) &= \frac{b-a}{6}[1+4+1] = b-a = I(1),\\
I(x) &= \int_a^b x\,\mathrm{d}x = \frac{b^2-a^2}{2},\\
I_2(x) &= \frac{b-a}{6}\left[a + 2(a+b) + b\right] = \frac{(b-a)(b+a)}{2} = I(x),\\
I(x^2) &= \int_a^b x^2\,\mathrm{d}x = \frac{b^3-a^3}{3},\\
I_2(x^2) &= \frac{b-a}{6}\left[a^2 + (a+b)^2 + b^2\right] = \frac{2(b^3-a^3)}{6} = I(x^2),\\
I(x^3) &= \int_a^b x^3\,\mathrm{d}x = \frac{b^4-a^4}{4},\\
I_2(x^3) &= \frac{b-a}{6}\left[a^3 + \tfrac{1}{2}(a+b)^3 + b^3\right] = \frac{b^4-a^4}{4} = I(x^3).
\end{aligned}$$

This shows that the degree of exactness is at least 3 (contrary to what might be expected from the interpolation picture). You can verify that $I_2(x^4) \neq I(x^4)$, so the degree of exactness is exactly 3.

This shows that the term $f'''(\xi)$ in the error formula for Simpson's rule is misleading. In fact, it is possible to write an error bound proportional to $f^{(4)}(\xi)$.

In terms of degree of exactness, Simpson's formula does better than expected. In general, Newton-Cotes formulae with even n have degree of exactness $n+1$. But this is by no means the highest possible (see next section).

4.2.4 Gaussian quadrature

The idea of **Gaussian quadrature** is to choose not only the weights σ_k but also the nodes x_k, in order to achieve the highest possible degree of exactness.

Firstly, we will illustrate the brute force **method of undetermined coefficients**.

Example 4.13: Gaussian quadrature formula $G_1(f) = \sum_{k=0}^{1} \sigma_k f(x_k)$ on the interval $[-1, 1]$

Here we have four unknowns x_0, x_1, σ_0 and σ_1, so we can impose four conditions:

$$G_1(1) = I(1) \implies \sigma_0 + \sigma_1 = \int_{-1}^{1} \mathrm{d}x = 2,$$

$$G_1(x) = I(x) \implies \sigma_0 x_0 + \sigma_1 x_1 = \int_{-1}^{1} x \,\mathrm{d}x = 0,$$

$$G_1(x^2) = I(x^2) \implies \sigma_0 x_0^2 + \sigma_1 x_1^2 = \int_{-1}^{1} x^2 \,\mathrm{d}x = \tfrac{2}{3},$$

$$G_1(x^3) = I(x^3) \implies \sigma_0 x_0^3 + \sigma_1 x_1^3 = \int_{-1}^{1} x^3 \,\mathrm{d}x = 0.$$

To solve this system, the symmetry suggests that $x_1 = -x_0$ and $\sigma_0 = \sigma_1$. This will automatically satisfy the equations for x and x^3, leaving the two equations

$$2\sigma_0 = 2, \qquad 2\sigma_0 x_0^2 = \tfrac{2}{3},$$

so that $\sigma_0 = \sigma_1 = 1$ and $x_1 = -x_0 = 1/\sqrt{3}$. The resulting Gaussian quadrature formula is

$$G_1(f) = f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right).$$

This formula has degree of exactness 3.

In general, the Gaussian quadrature formula with n nodes will have degree of exactness $2n+1$.

The method of undetermined coefficients becomes unworkable for larger numbers of nodes, because of the nonlinearity of the equations. A much more elegant method uses orthogonal polynomials. In addition to what we learned before, we will need the following result.

Theorem 4.2

If $\{\phi_0, \phi_1, \ldots, \phi_n\}$ is a set of orthogonal polynomials on $[a, b]$ under the inner product $(f, g) = \int_a^b f(x)g(x)w(x)\,\mathrm{d}x$ and ϕ_k is of degree k for each $k = 0, 1, \ldots, n$, then ϕ_k has k distinct real roots, and these roots lie in the interval $[a, b]$.

Proof:
Let $x_1, \ldots, x_j$ be the points where $\phi_k(x)$ changes sign in $[a, b]$. If $j = k$ then we are done. Otherwise, suppose $j < k$, and consider the polynomial

$$q_j(x) = (x - x_1)(x - x_2)\cdots(x - x_j).$$

Since q_j has lower degree than ϕ_k, they must be orthogonal, meaning

$$(q_j, \phi_k) = 0 \implies \int_a^b q_j(x)\phi_k(x)w(x)\,\mathrm{d}x = 0.$$

On the other hand, notice that the product $q_j(x)\phi_k(x)$ cannot change sign in $[a, b]$, because each sign change in $\phi_k(x)$ is cancelled out by one in $q_j(x)$. This means that

$$\int_a^b q_j(x)\phi_k(x)w(x)\,\mathrm{d}x \neq 0,$$

which is a contradiction. □

Remarkably, these roots are precisely the optimum choice of nodes for a quadrature formula to approximate the (weighted) integral

$$I_w(f) = \int_a^b f(x)w(x)\,\mathrm{d}x.$$

Theorem 4.3: Gaussian quadrature

Let ϕ_{n+1} be a polynomial in $\mathcal{P}_{n+1}$ that is orthogonal on $[a, b]$ to all polynomials in $\mathcal{P}_n$, with respect to the weight function $w(x)$. If $x_0, x_1, \ldots, x_n$ are the roots of ϕ_{n+1}, then the quadrature formula

$$G_{n,w}(f) := \sum_{k=0}^{n} \sigma_k f(x_k), \qquad \sigma_k = \int_a^b \ell_k(x)w(x)\,\mathrm{d}x$$

approximates $I_w(f)$ with degree of exactness $2n + 1$ (the largest possible).

Like Newton-Cotes, we see that Gaussian quadrature is based on integrating an interpolating polynomial, but now the nodes are the roots of an orthogonal polynomial, rather than equally spaced points.

Example 4.14: Gaussian quadrature with $n = 1$ on $[-1, 1]$ and $w(x) = 1$ (again)

To find the nodes x_0, x_1, we need to find the roots of the orthogonal polynomial $\phi_2(x)$. For this inner product, we already computed this (Legendre polynomial) in Chapter 3, where we found

$$\phi_2(x) = x^2 - \tfrac{1}{3}.$$

Thus the nodes are $x_0 = -1/\sqrt{3}, x_1 = 1/\sqrt{3}$. Integrating the Lagrange polynomials gives

the corresponding weights

$$\sigma_0 = \int_{-1}^{1} \ell_0(x)\,\mathrm{d}x = \int_{-1}^{1} \frac{x - \frac{1}{\sqrt{3}}}{-\frac{2}{\sqrt{3}}}\,\mathrm{d}x = -\tfrac{\sqrt{3}}{2}\left[\tfrac{1}{2}x^2 - \tfrac{1}{\sqrt{3}}x\right]_{-1}^{1} = 1,$$

$$\sigma_1 = \int_{-1}^{1} \ell_1(x)\,\mathrm{d}x = \int_{-1}^{1} \frac{x + \frac{1}{\sqrt{3}}}{\frac{2}{\sqrt{3}}}\,\mathrm{d}x = \tfrac{\sqrt{3}}{2}\left[\tfrac{1}{2}x^2 + \tfrac{1}{\sqrt{3}}x\right]_{-1}^{1} = 1,$$

as before.

i Note

Using an appropriate weight function $w(x)$ can be useful for integrands with a singularity, since we can incorporate this in $w(x)$ and still approximate the integral with $G_{n,w}$.

Example 4.15: Gaussian quadrature for $\int_0^1 \cos(x)x^{-1/2}\,\mathrm{d}x$, with $n = 0$

This is a Fresnel integral, with exact value 1.80905... Let us compare the effect of using an appropriate weight function.

1. *Unweighted quadrature ($w(x) \equiv 1$).* The orthogonal polynomial of degree 1 is

$$\phi_1(x) = x - \frac{\int_0^1 x\,\mathrm{d}x}{\int_0^1 \mathrm{d}x} = x - \tfrac{1}{2} \implies x_0 = \tfrac{1}{2}.$$

   The corresponding weight may be found by imposing $G_0(1) = I(1)$, which gives $\sigma_0 = \int_0^1 \mathrm{d}x = 1$. Then our estimate is

$$G_0\left(\frac{\cos(x)}{\sqrt{x}}\right) = \frac{\cos\left(\frac{1}{2}\right)}{\sqrt{\frac{1}{2}}} = 1.2411\ldots$$

2. *Weighted quadrature with $w(x) = x^{-1/2}$.* This time we get

$$\phi_1(x) = x - \frac{\int_0^1 x^{1/2}\,\mathrm{d}x}{\int_0^1 x^{-1/2}\,\mathrm{d}x} = x - \frac{2/3}{2} \implies x_0 = \tfrac{1}{3}.$$

   The corresponding weight is $\sigma_0 = \int_0^1 x^{-1/2}\,\mathrm{d}x = 2$, so the new estimate is the more accurate

$$G_{0,w}(\cos(x)) = 2\cos\left(\tfrac{1}{3}\right) = 1.8899\ldots$$

Proof:

First, recall that any interpolatory quadrature formula based on $n+1$ nodes will be exact for all polynomials in $\mathcal{P}_n$ (this follows from the Newton-Cotes theorem, which can be modified to include the weight function $w(x)$). So in particular, $G_{n,w}$ is exact for $p_n \in \mathcal{P}_n$.

Now let $p_{2n+1} \in \mathcal{P}_{2n+1}$. The trick is to divide this by the orthogonal polynomial ϕ_{n+1} whose roots are the nodes. This gives

$$p_{2n+1}(x) = \phi_{n+1}(x)q_n(x) + r_n(x) \quad \text{for some} \quad q_n, r_n \in \mathcal{P}_n.$$

Then

$$\begin{aligned} G_{n,w}(p_{2n+1}) &= \sum_{k=0}^{n} \sigma_k p_{2n+1}(x_k) = \sum_{k=0}^{n} \sigma_k \Big[\phi_{n+1}(x_k)q_n(x_k) + r_n(x_k)\Big] \\ &= \sum_{k=0}^{n} \sigma_k r_n(x_k) = I_w(r_n), \end{aligned}$$

where we have used the fact that $G_{n,w}$ is exact for $r_n \in \mathcal{P}_n$. Now, since q_n has lower degree than ϕ_{n+1}, it must be orthogonal to ϕ_{n+1}, so

$$I_w(\phi_{n+1}q_n) = \int_a^b \phi_{n+1}(x)q_n(x)w(x)\,\mathrm{d}x = 0$$

and hence

$$\begin{aligned} G_{n,w}(p_{2n+1}) &= I_w(r_n) + 0 = I_w(r_n) + I_w(\phi_{n+1}q_n) \\ &= I_w(\phi_{n+1}q_n + r_n) = I_w(p_{2n+1}). \end{aligned}$$

Unlike Newton-Cotes formulae with equally-spaced points, it can be shown that $G_{n,w}(f) \to I_w(f)$ as $n \to \infty$, for any continuous function f. This follows (with a bit of analysis) from the fact that all of the weights σ_k are positive, along with the fact that they sum to a fixed number $\int_a^b w(x)\,\mathrm{d}x$. For Newton-Cotes, the signed weights still sum to a fixed number, but $\sum_{k=0}^{n} |\sigma_k| \to \infty$, which destroys convergence.

Not surprisingly, we can derive an error formula that depends on $f^{(2n+2)}(\xi)$ for some $\xi \in (a,b)$. To do this, we will need the following result from calculus.

Theorem 4.4: Mean value theorem for integrals

If f, g are continuous on $[a,b]$ and $g(x) \geq 0$ for all $x \in [a,b]$, then there exists $\xi \in (a,b)$ such that

$$\int_a^b f(x)g(x)\,\mathrm{d}x = f(\xi)\int_a^b g(x)\,\mathrm{d}x.$$

Proof:
Let m and M be the minimum and maximum values of f on $[a,b]$, respectively. Since $g(x) \geq 0$, we have that

$$m\int_a^b g(x)\,\mathrm{d}x \leq \int_a^b f(x)g(x)\,\mathrm{d}x \leq M\int_a^b g(x)\,\mathrm{d}x.$$

Now let $I = \int_a^b g(x)\,\mathrm{d}x$. If $I = 0$ then $g(x) \equiv 0$, so $\int_a^b f(x)g(x)\,\mathrm{d}x = 0$ and the theorem holds for every $\xi \in (a, b)$. Otherwise, we have

$$m \leq \frac{1}{I}\int_a^b f(x)g(x)\,\mathrm{d}x \leq M.$$

By the Intermediate Value Theorem, $f(x)$ attains every value between m and M somewhere in (a, b), so in particular there exists $\xi \in (a, b)$ with

$$f(\xi) = \frac{1}{I}\int_a^b f(x)g(x)\,\mathrm{d}x.$$

Theorem 4.5: Error estimate for Gaussian quadrature

Let $\phi_{n+1} \in \mathcal{P}_{n+1}$ be monic and orthogonal on $[a, b]$ to all polynomials in $\mathcal{P}_n$, with respect to the weight function $w(x)$. Let $x_0, x_1, \ldots, x_n$ be the roots of ϕ_{n+1}, and let $G_{n,w}(f)$ be the Gaussian quadrature formula defined above. If f has $2n + 2$ continuous derivatives on (a, b), then there exists $\xi \in (a, b)$ such that

$$I_w(f) - G_{n,w}(f) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!}\int_a^b \phi_{n+1}^2(x)w(x)\,\mathrm{d}x.$$

Proof:
A neat trick is to use Hermite interpolation. Since the x_k are distinct, there exists a unique polynomial p_{2n+1} such that

$$p_{2n+1}(x_k) = f(x_k), \quad p'_{2n+1}(x_k) = f'(x_k) \quad \text{for } k = 0, \ldots, n.$$

In addition (see problem sheet), there exists $\lambda \in (a, b)$, depending on x, such that

$$f(x) - p_{2n+1}(x) = \frac{f^{(2n+2)}(\lambda)}{(2n+2)!}\prod_{i=0}^{n}(x - x_i)^2.$$

Now we know that $(x - x_0)(x - x_1)\cdots(x - x_n) = \phi_{n+1}(x)$, since we fixed ϕ_{n+1} to be monic. Hence

$$\int_a^b f(x)w(x)\,\mathrm{d}x - \int_a^b p_{2n+1}(x)w(x)\,\mathrm{d}x = \int_a^b \frac{f^{(2n+2)}(\lambda)}{(2n+2)!}\phi_{n+1}^2(x)w(x)\,\mathrm{d}x.$$

Now we know that $G_{n,w}$ must be exact for p_{2n+1}, so

$$\int_a^b p_{2n+1}(x)w(x)\,\mathrm{d}x = G_{n,w}(p_{2n+1}) = \sum_{k=0}^{n}\sigma_k p_{2n+1}(x_k) = \sum_{k=0}^{n}\sigma_k f(x_k) = G_{n,w}(f).$$

For the right-hand side, we can't take $f^{(2n+2)}(\lambda)$ outside the integral since λ depends on x. But $\phi_{n+1}^2(x)w(x) \geq 0$ on $[a, b]$, so we can apply the mean value theorem for integrals and get

$$I_w(f) - G_{n,w}(f) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!}\int_a^b \phi_{n+1}^2(x)w(x)\,\mathrm{d}x$$

for some $\xi \in (a, b)$ that does not depend on x.

5 Differential Equations

How can computers solve differential equations?

Almost all differential equations which arise in mathematics and its applications do not have exact analytical solutions. This is a central motivation for numerical analysis, as well as the historical development of increasingly powerful computers. Most scientific computing languages have pre-built packages for solving differential equations quickly and accurately using numerical approximations, and there are entire classes of software packages designed to do this for specialised industries, such as aerospace or finance. The goal of this Chapter is to understand the fundamentals of numerical timestepping, so that you can understand properties of these numerical methods, and hence so you can choose which to use for a given problem.

> **Note**
>
> This topic is vast, with many textbooks detailing aspects of numerical differential equations from a variety of theoretical or applied perspectives. We will focus on building up the mathematical terminology of different classes of solvers, such as those available in MATLAB and described in detail here, as well as the basics of convergence theory and error analysis.

> **Note**
>
> Symbolic tools can solve many of the analytically-tractable classes of differential equations using a variety of algorithms and heuristics, but these are such a limited set of equations that they are not as frequently used outside of theoretical areas. The MATLAB function `desolve` can be used to solve a reasonably large class of ODEs symbolically.

5.1 Basic Concepts

We will focus on general systems of n first-order differential equations of the form

$$\dot{\mathbf{u}} = \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{u}), \quad \mathbf{u}(t_0) = \mathbf{u}_0 \in \mathbb{R}^n.$$

Many general classes of equations can be written in this form, and such systems also arise when discretising partial differential equations, as well as other kinds of models involving derivatives. When $\mathbf{f}(t, \mathbf{u}) = \mathbf{f}(\mathbf{u})$ (i.e. does not depend on time) we call the system **autonomous**.

It is reasonable to then ask: Why do we only focus on first order differential equations? This ie because any ordinary differential equation of order n can be written as a system of n first-order ordinary differential equations by introducing new variables to represent each derivative up to order $n-1$. This is a standard step in both numerical and analytical solution methods as most modern solvers only handle systems of first-order ODEs.

Suppose we have an nth order ODE:

$$u^{(n)} = f\big(t, u, u', u'', \ldots, u^{(n-1)}\big),$$

where $u^{(n)} := d^n u(t)/du^n$. Now introduce the auxiliary variables:

$$y_0 = u, \quad y_1 = u', \quad \ldots, \quad y_{n-1} = u^{(n-1)}$$

Then, the new system is:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= y_2 \\ &\vdots \\ y_{n-2}' &= y_{n-1} \\ y_{n-1}' &= f\big(t, y_0, y_1, \ldots, y_{n-1}\big) \end{aligned}$$

This approach allows the use of vectorized notation and standard solution methods for systems of first-order ODEs.

Example 5.1

Convert the third-order ODE

$$u''' = t + 2u + 3u''$$

with initial conditions $u(0) = 1$, $u'(0) = 2$, $u''(0) = 3$, into a system of three first-order ODEs.
Let

$$y_0 = u \qquad y_1 = u' \qquad y_2 = u'',$$

and then deduce that

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= y_2 \\ y_2' &= t + 2y_0 + 3y_2 \end{aligned}$$

with initial conditions

$$y_0(0) = 1, \quad y_1(0) = 2, \quad y_2(0) = 3.$$

We can also represent this system in matrix form as follows:

$$\frac{d}{dt}\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 2 & 0 & 3 \end{bmatrix}\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ t \end{bmatrix}.$$

5.1.1 Preliminary ODE theory

Before discussing practical aspects of solving such equations, recall the basic existence and uniqueness theory. For ODEs this theory is not too complicated, provided the right-hand side is sufficiently well behaved.

Theorem 5.1

Picard–Lindelöf theorem.
Let $\mathcal{D} \subset \mathbb{R} \times \mathbb{R}^n$ be an open rectangle with interior point $(t_0, \mathbf{u}_0)$. Suppose $\mathbf{f} : \mathcal{D} \to \mathbb{R}^n$ is continuous in t and Lipschitz continuous in $\mathbf{u}$ for all $(t, \mathbf{u}) \in \mathcal{D}$. Then there exists $\varepsilon > 0$ such that the initial-value problem above has a unique solution $\mathbf{u}(t)$ for $t \in [t_0 - \varepsilon, t_0 + \varepsilon]$.

Essentially, this says that if $\mathbf{f}$ is sufficiently nice then the ODE has a unique solution for each initial condition. The result follows from the contraction-mapping theorem or via an iterative scheme. If $\mathbf{f}$ is globally Lipschitz (same Lipschitz constant for all $\mathbf{u} \in \mathbb{R}^n$) and continuous for all $t \in \mathbb{R}$, then the solution exists for all t. For autonomous systems, solution curves $\mathbf{u}(t) \in \mathbb{R}^n$ cannot cross.

While this is theoretical, the theorem is important: there are ODEs without solutions or with non-unique solutions, and for PDEs existence and uniqueness are often much harder (and can fail).

i Millennium Prize Problem: Navier–Stokes Equations

The Clay Mathematics Institute has offered a $1,000,000 prize for resolving one of the most important open problems in mathematics:
Do smooth and uniquely determined solutions always exist for the three-dimensional, incompressible Navier–Stokes equations, given reasonable initial data?
The Navier-Stokes equations are the fundamental equations of fluid mechanics. Resolving this longstanding problem would be a huge achievement in pure mathematics and would also revolutionise our understanding of turbulence in physics.

Example 5.2

The ODE given by

$$\frac{\mathrm{d}^2 u}{\mathrm{d}t^2} = \sqrt{u}, \quad u(0) = \frac{\mathrm{d}u}{\mathrm{d}t}(0) = 0,$$

which is equivalent to the first-order system

$$\frac{\mathrm{d}u}{\mathrm{d}t} = v, \quad \frac{\mathrm{d}v}{\mathrm{d}t} = \sqrt{u}, \quad u(0) = v(0) = 0,$$

is a famous example of a system with non-unique solutions. Namely, for any $T > 0$, there are infinitely many solutions to this equation given by

$$u(t) = \begin{cases} 0 & t < T, \\ \frac{1}{144}(t-T)^4 & t \geq T, \end{cases}$$

in addition to the solution $u(t) = 0$. This is also known as Norton's Dome, which has an amusing interpretation as a model in Newtonian mechanics that breaks the notion of causality, as it represents a situation where a particle can spontaneously start moving after an arbitrary amount of time.

5.2 Finite Difference Methods

Moving to practical questions, we now consider how to approximate solutions to the general ODE system using a computer. We have already seen in the previous chapter how to approximate the derivative using what are known as **finite differences.** These approximations are the main ingredients we will use to develop our numerical approaches.

The most well-known numerical method is commonly referred to as the **forward Euler method**, which is given by taking the forward difference operator on the left-hand side of the ODE system and rearranging the equation to obtain:

$$\mathbf{u}(t + \Delta t) = \mathbf{u}(t) + \Delta t \mathbf{f}(\mathbf{u}(t)),$$

where we are now using Δt to denote a small time step, rather than h. This is essentially the same approximation for the gradient of a function illustrated in Chapter 4, except now for the solution of an ODE. We can then iterate this formula starting at the initial condition to find an approximate solution at an arbitrary time t.

We can use a more natural notation for this approximation by taking $t \approx n\Delta t$ and letting $\mathbf{u}_n \approx \mathbf{u}(n\Delta t)$. The forward Euler method can then be written as the iteration:

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \mathbf{f}(\mathbf{u}_n).$$

How accurate is this method for approximating the true solution? How can we know that this method is **convergent**; that is, if we take $\Delta t \to 0$, can we ensure that $\mathbf{u}_n \to \mathbf{u}(t)$? These are more subtle issues compared to numerical differentiation, as here we are using previous approximations for each subsequent timestep, and the dynamics of these schemes can play important roles in determining convergence. Let's consider a simple example to illustrate these ideas.

5.2.1 The van der Pol Oscillator

The van der Pol oscillator is given by

$$\frac{\mathrm{d}^2 u}{\mathrm{d}t^2} - C\left(1 - u^2\right)\frac{\mathrm{d}u}{\mathrm{d}t} + u = 0,$$

which can be converted to the first-order system:

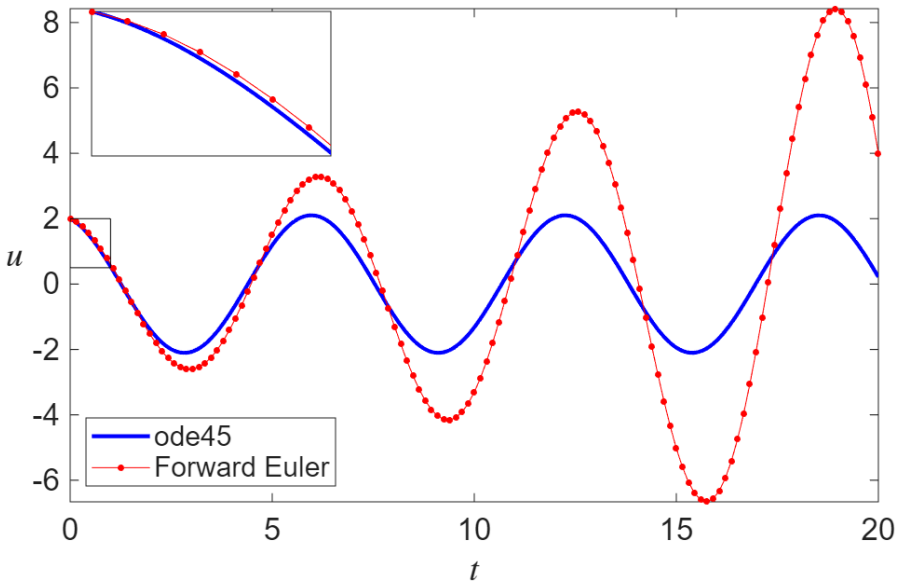
$$\frac{\mathrm{d}u}{\mathrm{d}t} = v, \quad \frac{\mathrm{d}v}{\mathrm{d}t} = C\left(1 - u^2\right)\frac{\mathrm{d}u}{\mathrm{d}t} - u.$$

For $C = 0$, this is the simple harmonic oscillator. For $C > 0$, the extra term means that for $u > 1$, the oscillations are damped, but for $u < 1$, there is an extra force due to "negative damping" driving the system away from $u = 0$.

We can first solve the simple case of $C = 0$ using the forward Euler scheme. The code for this can be compared with a high-order scheme that MATLAB has built-in called `ode45`.

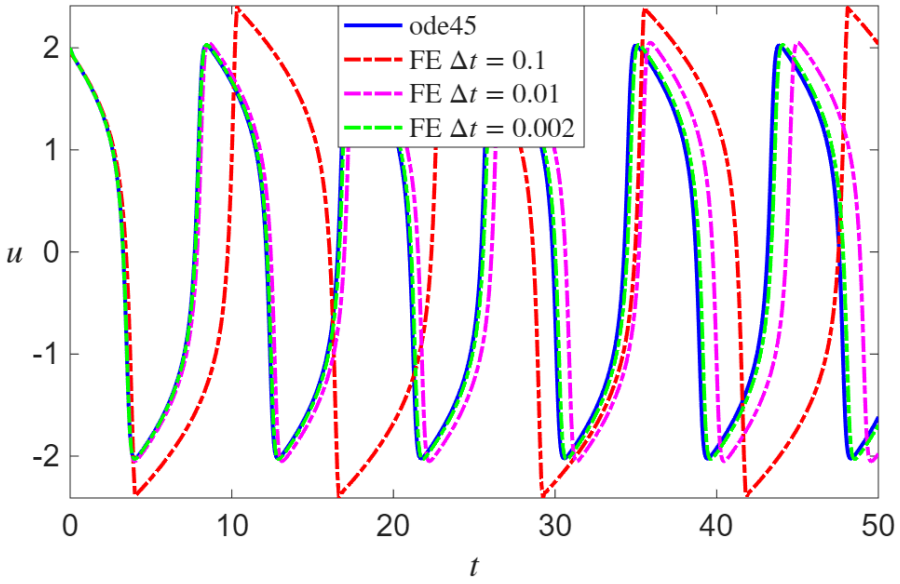
```
% Forward Euler implementation for the simple harmonic oscillator
% Solving the van der Pol equation using Matlab's ode45 command

C = 0; % Parameter C in the model.
% The ODE rhs function as an "anonymous function".
f = @(t,u)  [u(2); -C*(u(1)^2 - 1).*u(2) - u(1)];

U0 = [2; -0.65];  % Initial condition
tspan = linspace(0,20,1e3); % Time span

% Set low tolerances to ensure an accurate solution
options = odeset('RelTol',1e-11,'AbsTol',1e-11);
[~,u_ode45] = ode45(f, tspan, U0);

% Forward Euler setup (timestep, vector of solution points etc)
dt = 0.15; n = round(tspan(end)/dt); u_Euler = NaN(2,n);

u_Euler(:,1) = U0; % Initial condition for Euler method
for i = 1:n-1
    t = (i-1) * dt; % Current time (NB: Unused currently!)
    u_Euler(:,i+1) = u_Euler(:,i) + dt * f(t, u_Euler(:,i));
end
```

Running this code and plotting the outputs, we obtain the following graph:



The inset shows that the first few steps of the method match the solution reasonably well. However, over time it appears that the error builds up, and the amplitudes grow (despite the correct solution having the same amplitude for all time). Reducing the timestep will improve this, but eventually the amplitude will always begin to grow, at a rate which will become approximately exponential.

We can see how this scheme behaves with the timestep more clearly by considering the nonlinear van der Pol oscillator with $C = 3$, shown below for four different choices of timestep Δt:



We observe convergence towards a similar solution profile as Δt decreases. However, there is still a buildup of error as t increases, meaning that we will need to consider local errors over one timestep, as well as global errors over iterative schemes for many timesteps.

We can formalize these ideas in terms of the **local truncation error (LTE),** which is essentially how much the approximation given in the forward Euler method fails to exactly satisfy the ODE. We can compute this by substituting in the true solution, $\mathbf{u}(t)$, and using Taylor series expansions to determine the error.

Example 5.3

LTE of forward Euler
Expanding $\mathbf{u}(t)$ in a Taylor series, we find:

$$\mathbf{u}_{n+1} = \mathbf{u}(t + \Delta t) = \mathbf{u}(t) + \Delta t \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} + O(\Delta t^2) = \mathbf{u}_n + \Delta t \mathbf{f}(\mathbf{u}_n),$$

which implies that this method has an LTE of $O(\Delta t^2)$. Note, however, that we can do precisely the same calculation before rearranging (via the approximation of the derivative directly) as:

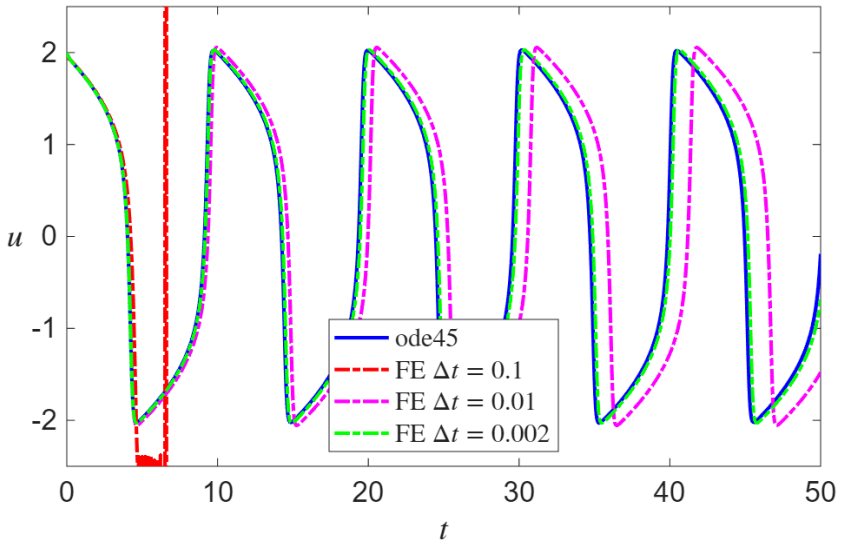
$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\Delta t} = \frac{\mathbf{u}(t + \Delta t) - \mathbf{u}(t)}{\Delta t} = \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} + O(\Delta t) = \mathbf{f}(\mathbf{u}(t)),$$

which implies an LTE of $O(\Delta t)$. These two definitions are both used, despite being somewhat inconsistent. We will adopt the former definition, which is sometimes called the **single-step error.**

An integration scheme which has an LTE of the form $O(\Delta t)$ or smaller is called **consistent.** Essentially, a consistent scheme is one where the approximations are equivalent to a collection of Taylor series approximations, and hence, subject to various smoothness assumptions, we expect to be able to make the error tend to 0 for small enough time steps. However, consistency is not enough to ensure that a numerical scheme converges to the analytical solution of the original ODE.

5.3 Stability of Finite Difference Schemes

A consistent scheme may fail to converge to the true solution if the method is not stable. To illustrate stability, we can explore exactly the plot above of the van der Pol oscillator, but for $\$C = 4$ instead. The code gives us this output:



The solutions for $\Delta t \leq 0.01$ appear very much as they did before, but the solution for $\Delta t = 0.1$ now rapidly increases. MATLAB says that it has reached $u_{70} \approx 10^{172}$, meaning that after 70 timesteps (i.e., $t \approx 7$), it has blown up in such a way that numerical calculations on these values are no longer meaningful. This illustrates a general property of numerical methods for differential equations known as **stability**, which is essentially the idea that we want numerical methods not to make successive iterations worse by compounding small errors. To be precise, let's focus on a particular equation which serves as a model for this phenomenon.

5.3.1 The Dahlquist Problem

We consider the **Dahlquist test problem**, which is the following simple linear equation:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \lambda u,$$

where $\lambda \in \mathbb{C}$ is a given complex parameter. While this equation is simple, it gives insight into any solution of our original problem if we "linearize" the equations around a single point in time. Importantly, this problem also allows us to make analytical progress in determining the stability of a general numerical scheme. Analytically, we expect the ODE to have decaying solutions for $\Re(\lambda) < 0$, but the iterations themselves can sometimes cause the opposite behavior of solutions growing without bound. It is easiest to see this by working with an example.

Example 5.4

Stability of forward Euler

Let's apply the forward Euler method to the Dahlquist problem. We have the iterations:

$$u_{n+1} = u_n + \Delta t \lambda u_n = (1 + \Delta t \lambda) u_n = (1 + \Delta t \lambda)^n u_0,$$

where we have explicitly "solved" the difference equation by iterating it back to the initial condition. Importantly, this will now blow up if $|1 + \Delta t \lambda| \geq 1$, meaning that for real λ, stability requires

$$-2 < \Delta t \lambda < 0.$$

Example 5.5

A consistent but unstable method
Consider the following numerical method for the test problem:

$$u_{n+1} = -4u_n + 5u_{n-1} + \Delta t(4\lambda u_n + 2\lambda u_{n-1}),$$

which can be rewritten as:

$$u_{n+1} = 4(\Delta t \lambda - 1)u_n + (5 + 2\Delta t \lambda)u_{n-1}.$$

We can solve this difference equation by rewriting it as a system, or by noting that it must have two solutions of the form $u_n = C\mu^n$ for some μ (analogous to how linear ODEs have solutions of the form $e^{\mu t}$). Substituting this in, and dividing by u_n, we find that μ satisfies:

$$\mu^2 = 4(\Delta t \lambda - 1)\mu + (5 + 2\Delta t \lambda),$$

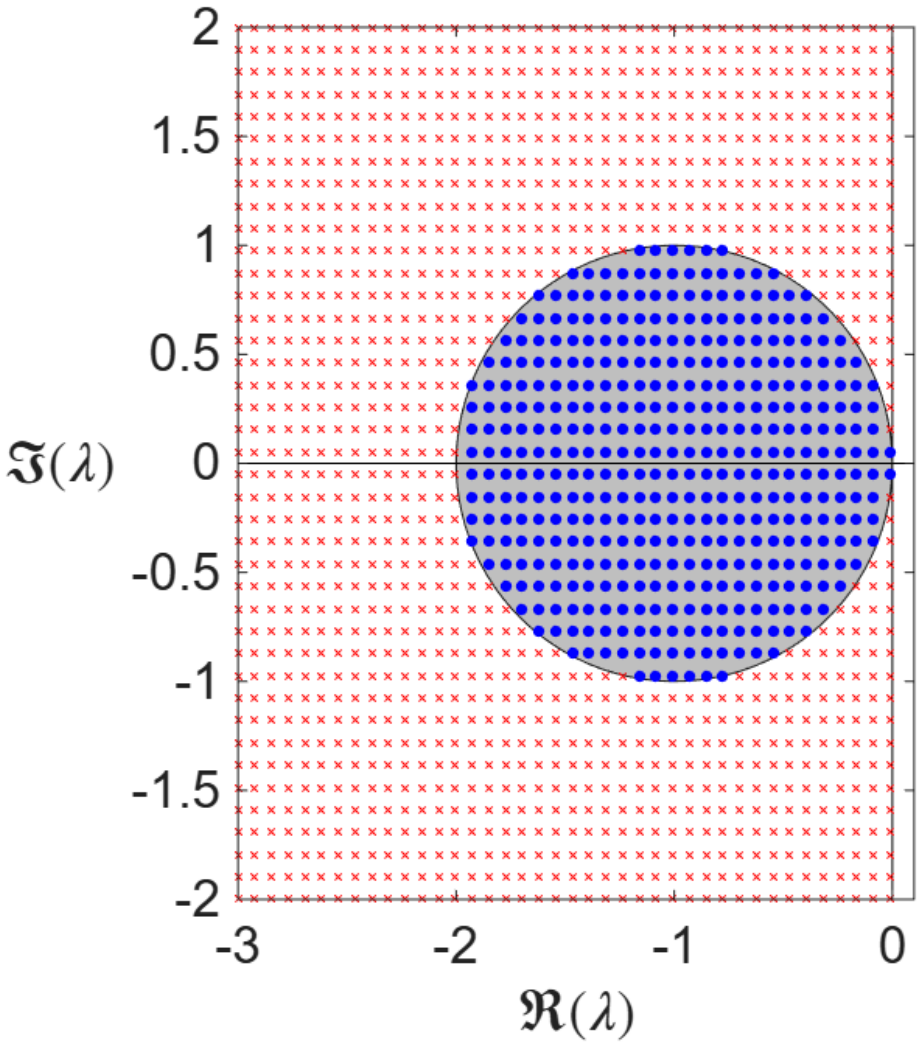
which simplifies to:

$$\mu = 2(\Delta t \lambda - 1) \pm \sqrt{4(\Delta t \lambda)^2 - 6\Delta t \lambda + 9}.$$

For $|\Delta t| \ll 1$, we see that one of these solutions approaches $\mu \approx -5$, so that $u_n \sim (-5)^n$, implying that we expect this scheme to be unstable for all λ as long as Δt is sufficiently small. With more work, we can in fact show that at least one of the solutions μ will always have $|\mu| \geq \sqrt{19} > 1$ for all $\lambda \in \mathbb{C}$ and $\Delta t \in \mathbb{R}$, so this scheme is *always* unstable.

5.3.2 Stability Regions and A-Stability

For a given numerical method, we can use the Dahlquist problem to define **stability regions** for complex λ. These essentially tell us for what values of λ we expect the numerical method to behave well. Importantly, this also gives us insight into how they impact the growth of truncation errors, which is why these stability regions apply to the method more generally.



We see that, as long as $\Re(\lambda) < 0$, then for sufficiently small Δt, this scheme is stable. In practice, this criterion means that we have to take extremely small time steps. A numerical method that is stable for all $\Re(\lambda) < 0$ is called **A-Stable**, which is a very desirable property. For such methods, we can (usually) mostly ignore aspects of stability and decide on suitable timesteps based solely on accuracy considerations.

The simplest example of an A-Stable alternative is to consider a slight modification of the Euler scheme, where instead of taking a forward derivative, we take a backward derivative. This results in the **backward Euler** scheme:

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \mathbf{f}(\mathbf{u}_{n+1}).$$

While this method looks very similar, in practice it is harder to implement as it is an **implicit** scheme, where the unknown $\mathbf{u}_{n+1}$ appears inside of the function on the right-hand side. This

means that to actually take a step, we have to solve the problem:

$$\mathbf{G}(\mathbf{u}_{n+1}) := \mathbf{u}_{n+1} - \Delta t \mathbf{f}(\mathbf{u}_{n+1}) = \mathbf{u}_n \implies \mathbf{u}_{n+1} = \mathbf{G}^{-1}(\mathbf{u}_n),$$

where $\mathbf{G}^{-1}$ is the inverse of the function $\mathbf{G}$. In practice, this becomes a (possibly high-dimensional) rootfinding problem, though for some special functions $\mathbf{f}$, it may be possible to evaluate it directly (e.g., when $\mathbf{f}$ is a linear function).

You can show (try this yourself) that this scheme has the same LTE as the forward Euler method. But its real advantage is in its stability.

Example 5.6

Stability of backward Euler
As before, we have the iterations:

$$u_{n+1} - \Delta t \lambda u_{n+1} = u_n \implies u_{n+1} = \frac{1}{1 - \Delta t \lambda} u_n = \left| \frac{1}{1 - \Delta t \lambda} \right|^n u_0.$$

Now, for any complex number z with $\Re(z) < 0$, we have that $|1 - z|^2 = 1 - 2\Re(z) + z^2 > 0$. Hence, as $\Delta t > 0$, if $\Re(\lambda) < 0$, then we have that this scheme is A-Stable.

5.3.3 Convergence

In general, it is difficult to prove that a numerical scheme converges in a suitable sense to the actual solution of an ODE system, as this requires some heavy machinery from analysis involving how the solutions of difference equations can embed into spaces of functions properly. One way to short-circuit these deep issues, however, is the use of the following theorem.

Theorem 5.2

Lax Equivalence Theorem
Assume that the system satisfies the conditions of the Picard–Lindelöf theorem. A finite difference method converges (in a suitable sense) to the unique solution of the system

$$\dot{\mathbf{u}} = \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{u}), \quad \mathbf{u}(t_0) = \mathbf{u}_0 \in \mathbb{R}^N,$$

as $\Delta t \to 0$ if the method is both consistent and stable.

One important caveat here is that this theorem only holds in the theoretical case of infinite-precision arithmetic; rounding error can build up if we need to simulate the equation for a long time interval, or to use very small time steps (as we have to for methods that are not A-stable). There is a related concept known as **global truncation error**, which is one way to keep track of how the LTE can build up over many timesteps, but we will not discuss this further except to say that the LTE being sufficiently small is usually the most desirable measure of accuracy.

5.4 Higher-Order Methods

So far we have discussed just the Euler stepping methods, which both have an LTE of $O(\Delta t)$. There are a variety of higher-order methods commonly used, which broadly fall into classes of **explicit** or **implicit** as described before, as well as **single-step** and **multi-step**. Here we will focus on explicit methods and give an example from each class. For simplicity of notation, let's consider a single ODE (i.e., $N = 1$), and let's only consider autonomous problems where $\dot{u} = f(u)$. While the basic ideas will be essentially identical for larger systems, adding in explicit time-dependence will make the formulae messier (but conceptually the same).

5.4.1 Single-step (Runge-Kutta) methods

One can take a single integration step broken up over different points within an interval. Let $t_n = n\Delta t$ be the current time corresponding to the approximate solution u_n. Then these methods can be written as:

$$u_{n+1} = u_n + \Delta t \sum_{i=1}^{s} a_i k_i, \quad k_1 = f(u_n), \quad k_i = f\left(u_n + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j\right).$$

These schemes are typically referred to as explicit Runge-Kutta methods (specifically having order s). They involve taking a single timestep using multiple function evaluations, essentially in order to get a better approximation of the derivative. They are derived in a similar way to how higher-order differentiation schemes can be derived; namely by taking different points within the interval of one timestep, and considering function evaluations and Taylor series on those points. It is easiest to see this with an example.

Example 5.7

Derivation of RK2 Methods
Let's consider three time points $t_n, t_n + \Delta t/2$, and $t_{n+1} = t_n + \Delta t$. In this case, we can expand the Taylor series for $u(t_n + \Delta t)$ as:

$$\begin{aligned} u(t_n + \Delta t) &= u(t_n) + \Delta t \dot{u}(t_n) + \frac{\Delta t^2}{2}\ddot{u}(t_n) + O(\Delta t^3) \\ &= u(t_n) + \Delta t f(u(t_n)) + \frac{\Delta t^2}{2} f_u(u(t_n))\dot{u}(t_n) + O(\Delta t^3), \end{aligned}$$

where we have used the chain rule to evaluate $\ddot{u}(t_n)$. The goal will be to derive a formula which includes the $O(\Delta t^2)$ term explicitly, so that the method is exact up to this order.

We perform a Taylor expansion of f evaluated at the midpoint as:

$$\begin{aligned} &f(u(t_n) + (\Delta t/2) f(u(t_n))) \\ =&f(u(t_n)) + \frac{\Delta t}{2} f_u(u(t_n))\dot{u}(t_n) + O(\Delta t^2). \end{aligned}$$

Now, if we multiply this expansion by Δt, we see that it exactly matches the expansion of $u(t + \Delta t)$ up to $O(\Delta t^2)$. So if we consider the numerical scheme:

$$u_{n+1} = u_n + \Delta t f(u_n + (\Delta t/2) f(u_n)),$$

we see that this scheme has an LTE of $O(\Delta t^2)$, which is an improvement over the Euler methods at the cost of one additional function evaluation.

One can work out that this method also very slightly improves the region of numerical stability compared to forward Euler, though it is not an A-stable method and hence will require sufficiently small timesteps to be stable.

5.4.2 Linear multistep methods

One disadvantage of the Runge-Kutta methods is that they require multiple function evaluations to arrive at a single timestep with a desired accuracy. A different approach is to use information from multiple timesteps to predict the next one in such a way that higher-order accuracy is maintained. Such methods are known as linear multistep methods, which have the general form:

$$u_{n+s} = \Delta t \sum_{j=0}^{s} \beta_j f(u_{n+j}),$$

where β_j are constants chosen to ensure the scheme is consistent, has a good LTE, etc. If $\beta_k = 0$, then the method is explicit, as everything on the right-hand side involves earlier function values. An example of such a method is the two-step **Adams-Bashforth** scheme given by:

$$u_{n+2} = u_{n+1} + \frac{\Delta t}{2} \left(3f(u_{n+1}) - f(u_n)\right).$$

One can work out that this method is second-order accurate, just like the RK2 method above, but only requires one new function evaluation at each timestep as opposed to two.

One disadvantage to these methods is that the initial-value problem given in the ODE system is not enough data to start them, as it only contains the solution at one time point u_0. So a typical method is to use a high-order Runge-Kutta scheme to start a multistep method.

5.5 Beyond the Basics

What we have covered here is a very brief sketch of the core ideas which were all mostly well-understood before 1960 or so. Since then, enormous progress has been made on a number of topics to develop increasingly sophisticated algorithms for simulating differential equations. A key idea from a "user" perspective is to be aware of how the stability of a method is important, but often difficult to ensure. There are no explicit methods of the forms described which are A-stable, and all implicit methods require some form of (often difficult) rootfinding.

> **Note**
>
> A **stiff** differential equation is one which is difficult to integrate from a stability perspective; often these correspond to equations which have large (in magnitude) values of λ when linearized, though making the idea precise to cover all known cases is not so easy.

6 Further Topics

6.1 Partial Differential Equations

6.2 Random Number Generation

How do we generate random numbers on a computer?

The ability to simulate random numbers on a computer is essential for modelling and understanding systems where chance and variability play a central role. Robust and repeatable random number generation is thus central to statistics, finance, queuing systems, cryptography, and a host of algorithms and other applications.

But what exactly do we mean by randomness or random numbers? For our purposes, we will consider "random numbers" to mean sequences of numbers that mimic the statistical properties of realisations of random variables drawn with a given distribution.

6.2.1 Uniform Random Numbers

If we know how to create a sample from a uniform distribution, then we can obtain a sample with a given distribution F using the following result:

Theorem 6.1: The Inverse Transform Method

Let U be a uniformly distributed random variable, and let $F(x)$ be a c.d.f. If F is invertible, then $X = F^{-1}(U)$ is distributed according to F.

Therefore it is often enough to have samples from the uniform distribution and hence this distribution plays a central role in the theory of random number generation.

In some sense, there are no *truly* random number generators. Computers can only execute algorithms, which are deterministic instructions, and thus they can only yield samples which appear random. We call these numbers **pseudorandom numbers**, and the algorithms which produce these numbers are called **pseudorandom number generators**.

The theoretical wish

A generator of **genuine** random numbers is an algorithm that produces a sequence of random variables U_1, U_2, ... which satisfies:

1. Each U_i is uniformly distributed between 0 and 1.
2. The U_i are mutually independent.

Property (ii) is the more important one since the normalisation in (i) is convenient but not crucial. Property (ii) implies that all pairs of values are uncorrelated and, more generally, that the value of U_i should not be predictable from $U_1, \ldots, U_{i-1}$. Of course, the properties listed above are those of authentically random numbers; the goal is to come as close as possible to these properties with our artificially generated pseudorandom numbers.

6.2.2 Linear Congruential Generators

An important and simple class of generators are the **linear congruential generators**, abbreviated as LCGs.
We need the modulo operation in order to define this class of generators.

Definition 6.1

For nonnegative integers x and m, we call $y = x \bmod m$ the integer remainder from the division x/m; we will write this as

$$y = x \mod m.$$

Definition 6.2

A **linear congruential generator (LCG)** is an iteration of the form

$$x_{i+1} = (ax_i + c) \mod m$$
$$u_{i+1} = \frac{x_{i+1}}{m} \in (0, 1),$$

where a, c, and m are integers.

Example 6.1

Choose $a = 6$, $c = 0$, and $m = 11$. Starting from $x_0 = 1$, which is called the *seed*, gives

$$1, 6, 3, 7, 9, 10, 5, 8, 4, 2, 1, 6, \ldots$$

Choosing $a = 3$ yields the sequence

$$1, 3, 9, 5, 4, 1, \ldots$$

whereas the seed $x_0 = 2$ results in

$$2, 6, 7, 10, 8, 2, \ldots$$

Conditions for a Full Period I

Theorem 6.2

Suppose $c \neq 0$. The generator has full period (that is, the number of distinct values generated from any seed x_0 is $m - 1$) if and only if the following conditions hold:

1. c and m are relatively prime (their only common divisor is 1).
2. Every prime number that divides m divides $a - 1$ as well.
3. $a - 1$ is divisible by 4 if m is.

If m is a power of 2, the generator has full period if c is odd and $a = 4n + 1$ for some integer n.

Example 6.2

The Borland C++ LCG has parameters

$$m = 2^{32}, \quad a = 22695477 = 1 + 4 \times 5673869, \quad c = 1.$$

Hence, by the corollary above, the LCG has full period.

Conditions for a Full Period II

If $c = 0$ and m is a prime, then full period is achieved from any $x_0 \neq 0$ when

1. $a^{m-1} - 1$ is a multiple of m,
2. $a^j - 1$ is not a multiple of m for $j = 1, \ldots, m - 2$.

If a satisfies these two properties it is called a *primitive root* of m. In this situation, the sequence $\{x_i\}_{i \geq 1}$ is of the form

$$x_0, ax_0, a^2x_0, a^3x_0, \ldots \mod m,$$

given that $c = 0$. The sequence returns to x_0 for the first time for the smallest k which satisfies $a^k x_0 \mod m = x_0$. This is the smallest k for which $a^k \mod m = 1$, that is $a^k - 1$ is a multiple of m.

Hence, the definition of a prime root coincides with the requirement that the series does not return to x_0 until $a^{m-1}x_0$.

Example 6.3: Examples of LCG Parameters

Modulus m	Multiplier a	Reference
$2^{31} - 1$	16807	Lewis, Goodman, and Miller, Park and Miller
(= 2147483647)	39373	L'Ecuyer
	742938285	Fishman and Moore
	950706376	Fishman and Moore
	1226874159	Fishman and Moore
2147483399	40692	L'Ecuyer
2147483563	40014	L'Ecuyer

Exercise 6.1

Define an LCG with parameters

$$c = 0, \quad m = 2^3 - 1 = 7, \quad a = 3.$$

Does this LCG have full period?

Note

Earlier versions of Microsoft Excel (prior to 2003) relied on a linear congruential generator (LCG) for its RAND() function, which produced predictable and statistically non-random outputs that undermined the reliability of numerical simulations, Monte Carlo methods, and statistical sampling. This flaw was widely recognized in the simulation community during the late 1990s and early 2000s, leading Microsoft to update Excel's random number algorithm in the 2003 release.

In applications, the following considerations are typically the most important when considering the appropriateness of a random number generating scheme:

- Reproducibility
- Speed
- Portability
- Period Length (if any)
- Randomness (i.e. robust statistical properties)

Linear congruential generators are no longer (and have not been for some time) used practically,

although they are a useful illustration of pseudorandom number generation. The Mersenne Twister family of algorithms is among the most popular in practice, and modern implementations are appropriate for most applications. It is notable for its extremely long period ($2^{19937} - 1$), strong statistical properties, and fast performance (see *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator* by Matsumoto and Nishimura, 1998).

6.2.3 Testing uniformity

Previously, we have only checked for a full period, to see whether a pseudorandom number generator is reasonable or not. In this section, we demonstrate more elaborate means to test the quality of a pseudorandom number generator.

Given a sample from a supposedly uniform distribution, one can use statistical tests to reject the hypothesis of uniformity. The samples provided by a computer are fake since they are totally deterministic, and therefore not random (as such they cannot be uniformly distributed). However, they are so "well chosen", that they might appear random. Hence we require statistical tests of randomness in order to judge the quality of a candidate PRNG.

6.2.3.1 Chi-Squared Test

Definition 6.3

Let $k \geq 1$, and let $X_1, \ldots, X_k$ be a sequence of i.i.d. standard normally distributed random variables. The distribution of the sum of squares

$$S = X_1^2 + \cdots + X_k^2$$

is called **chi-square** with k degrees of freedom.

The probability density function of $\chi(k)$ is given by

$$f(x, k) = \frac{1}{2^{k/2}\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ denotes the gamma function, which is defined by

$$\Gamma(\xi) = \int_0^\infty x^{\xi-1} e^{-x} \, \mathrm{d}x.$$

For integers $n \in \mathbb{N}$, we can write the Gamma function in terms of the factorial function as follows:

$$\Gamma(n) = (n-1)! = (n-1)(n-2)\ldots 2 \cdot 1.$$

Let

$$X_1, X_2, \ldots, X_n$$

be a sample.

The **chi-squared test for uniformity** is constituted by the following: - The null hypothesis H_0: The sample is from a uniform distribution against the alternative hypothesis H_a that it is not. - The test statistic

$$T = \frac{k}{n} \sum_{j=1}^{k} \left(n_j - \frac{n}{k} \right)^2,$$

where k is the number of equidistant partitions ("bins", to be chosen) of the unit interval, given by

$$[0, 1/k), \quad [1/k, 2/k), \ldots, [(k-1)/k, 1].$$

and n_j is the number of observations in the jth bin. - The confidence level α (to be chosen).

The following is given without proof.

Theorem 6.3

As $n \to \infty$, T converges, in distribution, to the chi-square distribution χ_{k-1} with $k-1$ degrees of freedom.

6.2.3.2 The Kolmogorov–Smirnov Test

Another simple test uses the empirical distribution function of the sample. The Kolmogorov–Smirnov test is based on the following intuition: If the sample is uniformly distributed, the deviation of the empirical distribution function from the theoretical distribution function, as given in Lemma 4.2, should be small.

Definition 6.4

If $x = (x_1, \ldots, x_n)$ is a sample, then the **empirical distribution function** of x is given by

$$F_n(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{(-\infty, x)}(x_i), \quad x \in \mathbb{R}.$$

The **Kolmogorov–Smirnov test for uniformity** is constituted by the following: - The null hypothesis H_0: The sample is from a given distribution F against the alternative hypothesis H_a that it is not. - The test statistic

$$D_n = \sup_{x \in \mathbb{R}} |F_n(x) - F(x)|,$$

where n is the sample size. - The confidence level α (to be chosen).

As $n \to \infty$, $\sqrt{n} D_n$ converges to

$$\sup_{t \in \mathbb{R}} |B(F(t))|,$$

in distribution, where B is a Brownian bridge (i.e., the quantity $\sup_{t\in\mathbb{R}} |B(F(t))|$ is a random variable).

For $F(t) = t$, the uniform distribution, the critical values of the Kolmogorov statistics D_n are known. For large n, the statistics converge in distribution to the so-called **Kolmogorov distribution**, which satisfies

$$K = \sup_{t\in[0,1]} |B(t)|.$$

In fact, it can be shown that

$$\mathbb{P}[K \leq x] = 1 - 2\sum_{k=1}^{\infty} (-1)^{k-1} e^{-2k^2x^2}, \quad x \in \mathbb{R}^+.$$

6.3 Stochastic Processes