Numerical Analysis II – Lecture Notes

Anthony Yeates
Durham University

March 12, 2018

Contents

0 What is numerical analysis?

> *Numerical analysis is the study of algorithms*
> *for the problems of continuous mathematics.*
>
> (from *The definition of numerical analysis*, L. N. Trefethen)

The goal is to devise algorithms that give quick and accurate answers to mathematical problems for scientists and engineers, nowadays using computers.

The word *continuous* is important: numerical analysis concerns real (or complex) variables, as opposed to discrete variables, which are the domain of computer science.

0.1 Direct or iterative methods?

Some problems can be solved by a finite sequence of elementary operations: a *direct method*.

Example → Solve a system of simultaneous linear equations.

$$\begin{array}{rcrcl} x & + & 2y & = & 0 \\ 2x & - & \pi y & = & 1 \end{array} \quad \rightarrow \quad \begin{pmatrix} 1.00 & 2.00 \\ 2.00 & -3.14 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0.00 \\ 1.00 \end{pmatrix} \quad \rightarrow \quad \text{Gaussian elimination . . .}$$

Even in this simple example, we hit upon one problem: π is a transcendental number that can't be represented exactly in a computer with finite memory. Instead, we will see that the computer uses a *floating-point* approximation, which incurs *rounding error*.

In direct methods, we only have to worry about rounding error, and computational time/memory. But, unfortunately, most problems of continuous mathematics cannot be solved by a finite algorithm.

Example → Evaluate $\sin(1.2)$.
We could do this with a Maclaurin series:

$$\sin(1.2) = 1.2 - \frac{(1.2)^3}{3!} + \frac{(1.2)^5}{5!} - \frac{(1.2)^7}{7!} + \dots .$$

To 8 decimal places, we get the partial sums

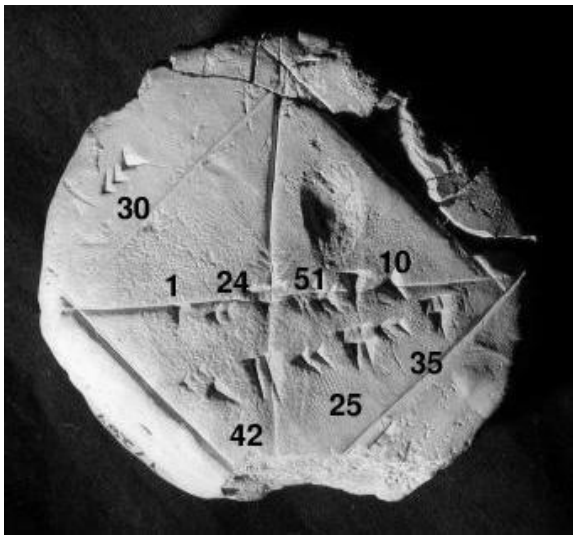
$$\begin{array}{l} 1.2 \\ 0.912 \\ 0.932736 \\ 0.93202505 \\ 0.93203927 \\ 0.93203908 \\ 0.93203909 \end{array}$$

This is an *iterative method* – keep adding an extra term to improve the approximation. Iterative methods are the only option for the majority of problems in numerical analysis, and may actually be quicker even when a direct method exists.

▶ The word "iterative" derives from the latin *iterare*, meaning "to repeat".

Even if our computer could do exact real arithmetic, there would still be an error resulting from stopping our iterative process at some finite point. This is called *truncation error*. We will be concerned with controlling this error and designing methods which converge as fast as possible.

Example → The famous $\sqrt{2}$ tablet from the Yale Babylonian Collection (photo: Bill Casselman, http://www.math.ubc.ca/~cass/Euclid/ybc/ybc.html).



This is one of the oldest extant mathematical diagrams, dated approximately to 1800-1600 BC. The numbers along the diagonal of the square approximate $\sqrt{2}$ in base 60 (sexagesimal):

$$1 + \frac{24}{60} + \frac{51}{60^2} + \frac{10}{60^3} = 1.41421296 \quad \text{to 9 s.f.}$$

This is already a good approximation to the true value $\sqrt{2} = 1.41421356$ – much better than could be achieved by ruler and pencil measurements!

► The other numbers on the tablet relate to the calculation of the diagonal length for a square of side 30, which is

$$30\sqrt{2} \approx 42 + \frac{25}{60} + \frac{35}{60^2}.$$

Example → Iterative method for $\sqrt{2}$.
It is probable that the Babylonians used something like the following iterative method. Start with an initial guess $x_0 = 1.4$. Then iterate the following formula:

$$x_{k+1} = \frac{x_k}{2} + \frac{1}{x_k} \quad \Longrightarrow \quad \begin{aligned} x_1 &= 1.4142857\ldots \\ x_2 &= 1.414213564\ldots \end{aligned}$$

► This method is also known as *Heron's method*, after a Greek mathematician who described it in the first century AD.

► Notice that the method converges extremely rapidly! We will explain this later in the course when we discuss rootfinding for nonlinear equations.

0.2 Course outline

In this course, we will learn how to do many common calculations quickly and accurately. In particular:

1. **Floating-point arithmetic** (*How do we represent real numbers on a computer?*)
2. **Polynomial interpolation** (*How do we represent mathematical functions on a computer?*)
3. **Numerical differentiation** (*How do we calculate derivatives?*)
4. **Nonlinear equations** (*How do we find roots of nonlinear equations?*)
5. **Linear equations** (*How do we solve linear systems?*)
6. **Least-squares approximation** (*How do we find approximate solutions to overdetermined systems?*)
7. **Numerical integration** (*How do we calculate integrals?*)

One area we won't cover is how to solve differential equations. This is such an important topic that it has its own course *Numerical Differential Equations III/IV*.

1 Floating-point arithmetic

How do we represent numbers on a computer?

Integers can be represented exactly, up to some maximum size.

Example → 64-bit integers.
If 1 bit (binary digit) is used to store the sign ±, 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 Python, this is not really a worry. Even though the maximum size of a normal (32-bit) integer is $2^{31} - 1$, larger results will be automatically promoted to "long" integers.

By contrast, only a subset of real numbers within any given interval can be represented exactly.

1.1 Fixed-point numbers

In everyday life, we tend to use a *fixed point* representation

$$x = \pm(d_1 d_2 \cdots d_{k-1}.d_k \cdots d_n)_\beta, \quad \text{where} \quad d_1, \ldots, d_n \in \{0, 1, \ldots, \beta - 1\}. \tag{1.1}$$

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

Example → $(10.1)_2 = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} = 2.5$.

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 $\beta - 1$.

Example → $3.1999\cdots = 3.2$.

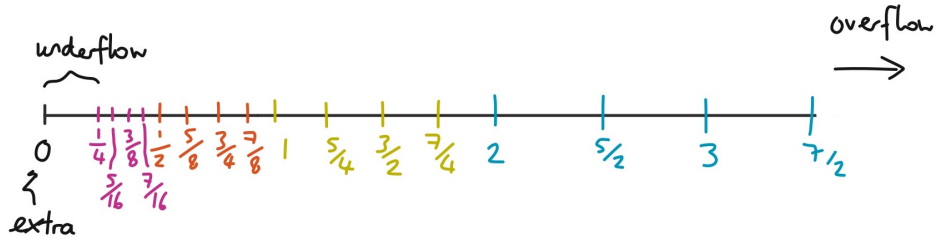
1.2 Floating-point numbers

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

$$F = \left\{ \pm (0.d_1 d_2 \cdots d_m)_\beta \beta^e \mid \beta, d_i, e \in \mathbb{Z},\ 0 \leq d_i \leq \beta - 1,\ e_{\min} \leq e \leq e_{\max} \right\}. \tag{1.2}$$

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*.

Example → Floating-point number system with $\beta = 2$, $m = 3$, $e_{\min} = -1$, $e_{\max} = 2$.



► 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 $(0.100)_2 2^{-1} = \frac{1}{2}(\frac{1}{2}) = \frac{1}{4}$.

Example → IEEE standard (1985) for double-precision (64-bit) arithmetic.
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

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

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 $(0.1 \cdots 1)_2 2^{1024} \approx 1.798 \times 10^{308}$. ► IEEE stands for Institute of Electrical and Electronics Engineers. This is the default system in Python/numpy. 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.

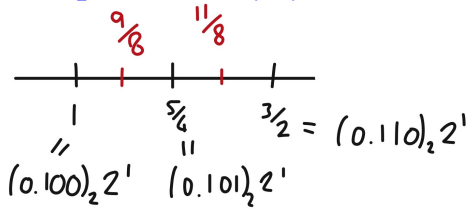
► e.g. in Python, `2.0**1025` leads to an exception.

► **e.g. 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.

► 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. ► This avoids statistical bias or prolonged drift.

Example → Our toy system from earlier.



$\frac{9}{8} = (1.001)_2$ 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 $\frac{3}{2}$.

► **e.g. 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.

Example → 3.1056, 31.050, 0.031056, 0.031050, and 3105.0 all have 5 significant figures (s.f.).

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

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

$$\mathrm{fl}(x) = x(1+\delta), \tag{1.3}$$

where the relative error incurred by rounding is

$$|\delta| = \frac{|\mathrm{fl}(x) - x|}{|x|}. \tag{1.4}$$

► 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 this 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

$$|\mathrm{fl}(x) - x| \le \tfrac{1}{2}\beta^{-m}\beta^e \quad \Longrightarrow \quad |\delta| \le \tfrac{1}{2}\beta^{1-m}. \tag{1.5}$$

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

$$|\delta| \le \epsilon_{\mathrm{M}}. \tag{1.6}$$

► Sometimes the machine epsilon is defined without the factor $\frac{1}{2}$. For example, in Python, `print(np.finfo(np.float64).eps)`.

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

Example → For our system with $\beta = 2$, $m = 3$, we have $\epsilon_M = \frac{1}{2}.2^{1-3} = \frac{1}{8}$. For IEEE double precision, we have $\beta = 2$ and $m = 53$ (including the hidden bit), so $\epsilon_M = \frac{1}{2}.2^{1-53} = 2^{-53} \approx 1.11 \times 10^{-16}$.

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

Example → Our toy system again ($\beta = 2$, $m = 3$, $e_{\min} = -1$, $e_{\max} = 2$).
Let us multiply $x = \frac{5}{8}$ and $y = \frac{7}{8}$. We have

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

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

► 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

$$\mathrm{fl}(x \circ y) = (x \circ y)(1 + \delta), \quad |\delta| \leq \epsilon_M. \tag{1.7}$$

1.5 Loss of significance

You might think that (1.7) 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 $|\delta_1|, |\delta_2| \leq \epsilon_M$. In that case, there is an error even before we round the result, since

$$\bar{x} \pm \bar{y} = x(1 + \delta_1) \pm y(1 + \delta_2) \tag{1.8}$$

$$= (x \pm y)\left(1 + \frac{x\delta_1 \pm y\delta_2}{x \pm y}\right). \tag{1.9}$$

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.

Example → Quadratic formula for solving $x^2 - 56x + 1 = 0$.
To 4 s.f., the roots are

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

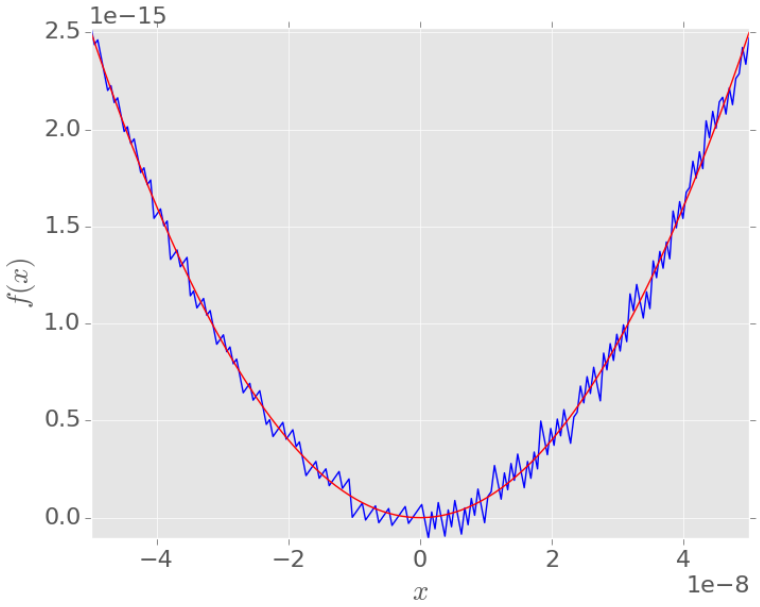
However, working to 4 s.f. we would compute $\sqrt{783} = 27.98$, which would lead to the results

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

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 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.

Example → Evaluate $f(x) = e^x - \cos(x) - x$ for x very near zero.
Let us plot this function in the range $-5\times10^{-8} \le x \le 5\times10^{-8}$ – 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.

► 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 $(a + b) + c = a + (b + c)$.

Example → In 2-digit decimal arithmetic,

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

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

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

The moral of the story is that sometimes care is needed.

2 Polynomial interpolation

How do we represent mathematical functions on a computer?

If f is a polynomial of degree n,

$$f(x) = p_n(x) = a_0 + a_1 x + \ldots + a_n x^n, \tag{2.1}$$

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. The idea in this chapter is to find a polynomial that approximates a general function f. For a continuous function f on a bounded interval, this is always possible if you take a high enough degree polynomial:

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*

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

► This may be proved using an explicit sequence of polynomials, called Bernstein polynomials. The proof is beyond the scope of this course, but see the extra handout for an outline.

If f is not continuous, then something other than a polynomial is required, since polynomials can't handle asymptotic behaviour.

► 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. In Chapter 6 we will see an alternative approach, appropriate for noisy data, where the overall error $|f(x) - p_n(x)|$ is minimised, without requiring p_n to match f at specific points.

2.1 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.

Example → Calculating $\sin(0.1)$ to 6 s.f. by Taylor series.
We can approximate this using a Taylor series about the point $x_0 = 0$, which is

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

This comes from writing

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

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.

▶ The exact answer is $\sin(0.1) = 0.09983341$.

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.

Example → We can use the Lagrange remainder to bound the error in our approximation. 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

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

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

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

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

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

▶ 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)].

▶ Much of this course will be concerned with truncation error, since this is a property of the numerical algorithm and independent of the floating-point arithmetic used.

Error estimates like the Lagrange remainder will play an important role in this course, so it is important to understand where it comes from. The number ξ will ultimately come from Rollé's theorem, which is a special case of the mean value theorem from 1H calculus;

Theorem 2.3 (Rolle). *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 $f'(\xi) = 0$.*

Proof of Lagrange remainder (Theorem 2.2). The argument goes as follows:

1. Define the "auxilliary" function
$$g(t) = f(t) - p_n(t) - M(t - x_0)^{n+1},$$
where p_n is the Taylor polynomial. By construction, this function satisfies
$$\begin{aligned} g(x_0) &= f(x_0) - p_n(x_0) - M(0)^{n+1} = 0, \\ g'(x_0) &= f'(x_0) - p_n'(x_0) - (n+1)M(0)^n = 0, \\ g''(x_0) &= f''(x_0) - p_n''(x_0) - n(n+1)M(0)^{n-1} = 0, \\ &\vdots \\ g^{(n)}(x_0) &= f^{(n)}(x_0) - p_n^{(n)}(x_0) - (n+1)!M(0) = 0. \end{aligned}$$
2. By a cunning choice of M, we can make $g(x) = 0$ too. Put
$$M = \frac{f(x) - p_n(x)}{(x - x_0)^{n+1}},$$
then $g(x) = f(x) - p_n(x) - M(x - x_0)^{n+1} = 0$.
3. Since $g(x_0) = g(x) = 0$ and $x \neq x_0$, Rolle's theorem implies that there exists ξ_0 between x_0 and x such that $g'(\xi_0) = 0$. But we already know that $g'(x_0) = 0$, so g' has two distinct roots and we can apply Rolle's theorem again. Hence there exists ξ_1 between x_0 and ξ_0 such that $g''(\xi_1) = 0$. We can keep repeating this argument until we get $\xi_{n+1} \equiv \xi$ such that $g^{(n+1)}(\xi) = 0$.
4. We can differentiate $g(t)$ to see that
$$g^{(n+1)}(t) = f^{(n+1)}(t) - p_n^{(n+1)}(t) - M\frac{\mathrm{d}^{n+1}}{\mathrm{d}t^{n+1}}\left[(t - x_0)^{n+1}\right] = f^{(n+1)}(t) - M(n+1)!$$
Substituting ξ and our chosen M gives
$$0 = g^{(n+1)}(\xi) = f^{(n+1)}(\xi) - \frac{f(x) - p_n(x)}{(x - x_0)^{n+1}}(n+1)!$$
which rearranges to give the formula in Theorem 2.2.

□

2.2 Polynomial interpolation

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

$$p_n(x) = a_0 + a_1 x + \ldots + a_n x^n = \sum_{k=0}^{n} a_k x^k \tag{2.2}$$

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 $m = n$.

Example → Linear interpolation ($n = 1$).

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

$$\begin{array}{ccccccccccc} a_0 & + & a_1 x_0 & + & a_2 x_0^2 & + & \ldots & + & a_n x_0^n & = & f(x_0), \\ a_0 & + & a_1 x_1 & + & a_2 x_1^2 & + & \ldots & + & a_n x_1^n & = & f(x_1), \\ \vdots & & \vdots & & \vdots & & & & \vdots & & \vdots \\ a_0 & + & a_1 x_n & + & a_2 x_n^2 & + & \ldots & + & a_n x_n^n & = & f(x_n), \end{array} \tag{2.3}$$

so we have to solve

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}. \tag{2.4}$$

This is called a *Vandermonde matrix*. The determinant of this matrix (problem sheet) is

$$\det(A) = \prod_{0 \le i < j \le n} (x_i - x_j), \tag{2.5}$$

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.

Uniqueness part of Theorem 2.4. 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 $\le n$. 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 $q_n = p_n$. □

► An alternative way to prove existence is to construct the interpolant explicitly, as we will do in Section 2.3.

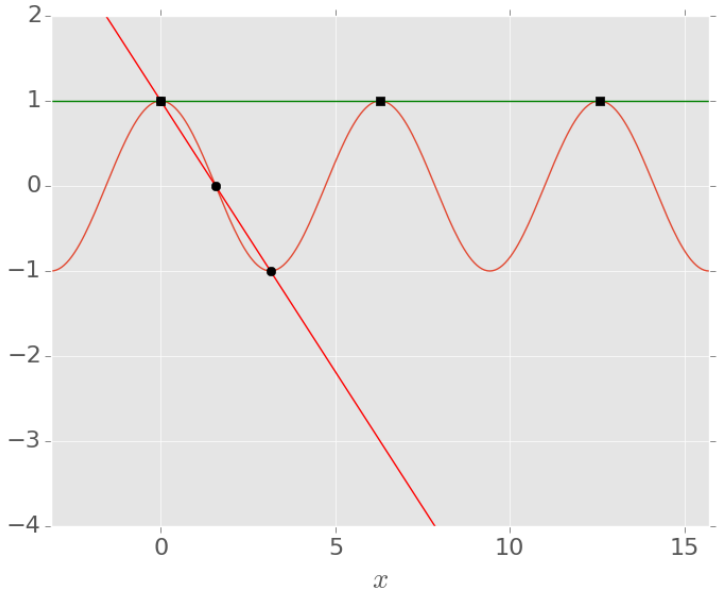
Note that the unique polynomial through $n+1$ points may have degree $< n$.

► This happens when $a_0 = 0$ in the solution to (2.4).

Example → 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 (2.4), e.g. by Gaussian elimination. However, we will see (next term) that this is not recommended. In practice, we choose a different basis for p_n. There are two common choices, due to Lagrange and Newton.

► The Vandermonde matrix arises when we write p_n in the *natural basis* $\{1, x, x^2, \ldots\}$.

2.3 Lagrange form

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,

$$p_n(x) = \sum_{k=0}^{n} f(x_k)\ell_k(x). \tag{2.6}$$

Example → 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

$$\ell_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j}. \tag{2.7}$$

By construction, they have the property that

$$\ell_k(x_i) = \begin{cases} 1 & \text{if } i = k, \\ 0 & \text{otherwise.} \end{cases} \tag{2.8}$$

From this, it follows that the interpolating polynomial may be written as (2.6).

▸ By Theorem 2.4, the Lagrange polynomials are the *unique* polynomials with property (2.8).

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

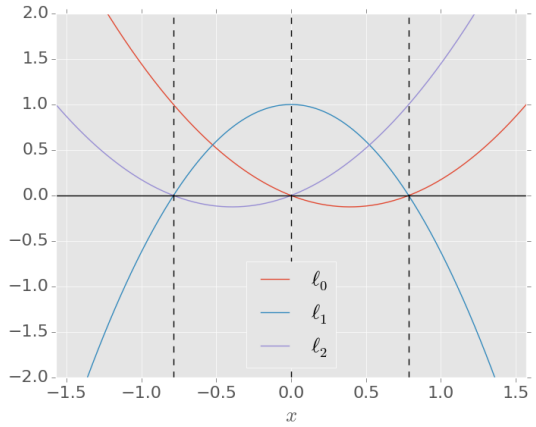
The Lagrange polynomials of degree 2 for these nodes are

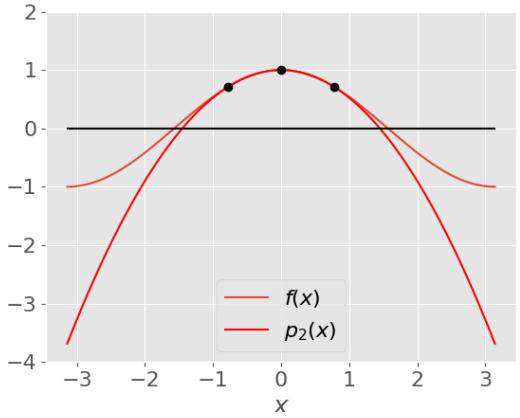
$$\begin{aligned} \ell_0(x) &= \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} = \frac{x(x - \frac{\pi}{4})}{\frac{\pi}{4} \cdot \frac{\pi}{2}}, \\ \ell_1(x) &= \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} = \frac{(x + \frac{\pi}{4})(x - \frac{\pi}{4})}{-\frac{\pi}{4} \cdot \frac{\pi}{4}}, \\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

So the interpolating polynomial is

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

The Lagrange polynomials and the resulting interpolant are shown below:





► 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.

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).

2.4 Newton/divided-difference form

It would be easy to increase the degree of p_n if

$$p_{n+1}(x) = p_n(x) + g_{n+1}(x), \quad \text{where } g_{n+1} \in \mathcal{P}_{n+1}. \tag{2.9}$$

From the interpolation conditions, we know that

$$g_{n+1}(x_i) = p_{n+1}(x_i) - p_n(x_i) = f(x_i) - f(x_i) = 0 \quad \text{for } i = 0, \ldots, n, \tag{2.10}$$

$$\implies \quad g_{n+1}(x) = a_{n+1}(x - x_0) \cdots (x - x_n). \tag{2.11}$$

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

$$p_n(x_{n+1}) + g_{n+1}(x_{n+1}) = f(x_{n+1}) \quad \implies \quad a_{n+1} = \frac{f(x_{n+1}) - p_n(x_{n+1})}{(x_{n+1} - x_0) \cdots (x_{n+1} - x_n)}. \tag{2.12}$$

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

$$n_0(x) = 1, \qquad n_k(x) = \prod_{j=0}^{k-1} (x - x_j) \quad \text{for } k > 0. \tag{2.13}$$

► Proving that this is truly a basis is left to the Problem Sheet.

The *Newton form* of the interpolating polynomial is then

$$p_n(x) = \sum_{k=0}^{n} a_k n_k(x), \qquad a_0 = f(x_0), \qquad a_k = \frac{f(x_k) - p_{k-1}(x_k)}{(x_k - x_0) \cdots (x_k - x_{k-1})} \text{ for } k > 0. \tag{2.14}$$

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

$$f[x_0, x_1, \ldots, x_k] = a_k, \tag{2.15}$$

where a_k is the coefficient in (2.14).

Example → Using (2.14), we find

$$f[x_0] = a_0 = f(x_0), \tag{2.16}$$

$$f[x_0, x_1] = a_1 = \frac{f(x_1) - p_0(x_1)}{x_1 - x_0} = \frac{f(x_1) - a_0}{x_1 - x_0} = \frac{f[x_1] - f[x_0]}{x_1 - x_0}. \tag{2.17}$$

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 → Continuing, we find

$$\begin{aligned} f[x_0, x_1, x_2] = a_2 &= \frac{f(x_2) - p_1(x_2)}{(x_2 - x_0)(x_2 - x_1)} = \frac{f(x_2) - a_0 - a_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)} \\ &= \ldots = \frac{1}{x_2 - x_0}\left(\frac{f[x_2] - f[x_1]}{x_2 - x_1} - \frac{f[x_1] - f[x_0]}{x_1 - x_0}\right) = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}. \end{aligned} \tag{2.18}$$

So again, we subtract and divide. In general, we have the following.

Theorem 2.5. *For $k > 0$, the divided differences satisfy*

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}. \tag{2.19}$$

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

$$f[x_0, x_1, \ldots, x_k] = \frac{f[x_1, \ldots, x_k] - f[x_0, \ldots, x_{k-1}]}{x_k - x_0}. \tag{2.20}$$

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

$$p_k(x) = \frac{(x_k - x)q_{k-1}(x) + (x - x_0)\tilde{q}_{k-1}(x)}{x_k - x_0}, \tag{2.21}$$

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 (2.21) holds, then matching the coefficient of x^k on each side will give (2.20), 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 as (2.21), note that

$$p_k(x_0) = q_{k-1}(x_0) = f(x_0), \tag{2.22}$$

$$p_k(x_k) = \tilde{q}_{k-1}(x_k) = f(x_k), \tag{2.23}$$

$$p_k(x_i) = \frac{(x_k - x_i)q_{k-1}(x_i) + (x_i - x_0)\tilde{q}_{k-1}(x_i)}{x_k - x_0} = f(x_i) \quad \text{for } i = 1, \ldots, k-1. \tag{2.24}$$

Since p_k agrees with f at the $k+1$ nodes, it is the unique interpolant in $\mathcal{P}_k$ (Theorem 2.4). □

Theorem 2.5 gives us our convenient method, which is to construct a *divided-difference table*.

Example → Nodes $\{-1, 0, 1, 2\}$ and data $\{5, 1, 1, 11\}$. 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

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

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

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

The new interpolating polynomial is

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

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

► 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 (cf. Chapter 3). In the limit that the nodes all coincide, the Newton form of $p_n(x)$ becomes the Taylor polynomial.

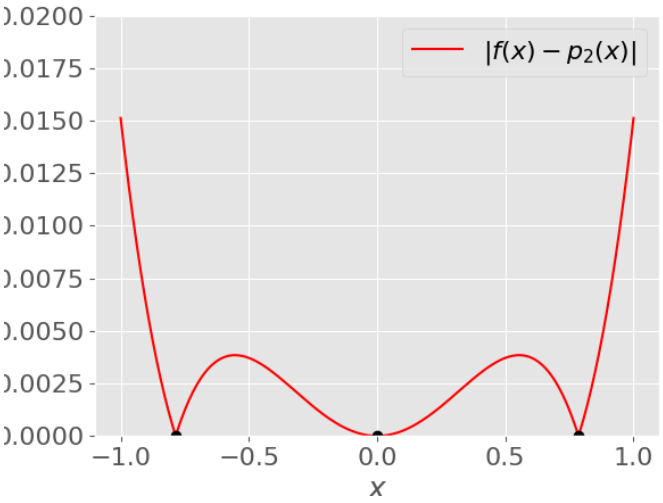
2.5 Interpolation error

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 → Quadratic interpolant for $f(x) = \cos(x)$ with $\{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}$.
From Section 2.3, we have $p_2(x) = \frac{16}{\pi^2}\left(\frac{1}{\sqrt{2}} - 1\right)x^2 + 1$, so the error is

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

This is shown here:



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). *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 (Theorem 2.2). 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 2.6 that this is not always true.

► 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.

Proof of Theorem 2.6. We follow a similar idea as for the Lagrange remainder in Theorem 2.2.

1. Define the "auxilliary" function
$$g(t) = f(t) - p_n(t) - M\prod_{i=0}^{n}(t - x_i).$$
By construction, this function satisfies
$$g(x_i) = f(x_i) - p_n(x_i) = 0 \quad \text{for} \quad i = 0, \ldots, n.$$
2. By a cunning choice of M, we can make $g(x) = 0$ too. Put
$$M = \frac{f(x) - p_n(x)}{\prod_{i=0}^{n}(x - x_i)},$$
then $g(x) = f(x) - p_n(x) - M\prod_{i=0}^{n}(x - x_i) = 0$.
3. Since $g(t)$ has $n + 2$ distinct roots, Rolle's theorem implies that there are $n + 1$ distinct points where $g'(t) = 0$. But then we can apply Rolle's theorem again to see that there are n distinct points where $g''(t) = 0$. Continuing to apply Rolle's theorem in this way, we end up with a single point $t = \xi$ where $g^{(n+1)}(\xi) = 0$.
4. Repeatedly differentiating $g(t)$ gives
$$g^{(n+1)}(t) = f^{(n+1)}(t) - p_n^{(n+1)}(t) - M\frac{\mathrm{d}^{n+1}}{\mathrm{d}t^{n+1}}\Big[\prod_{i=0}^{n}(t - x_i)\Big] = f^{(n+1)}(t) - M(n + 1)!$$
Substituting ξ and our chosen M gives
$$0 = g^{(n+1)}(\xi) = f^{(n+1)}(\xi) - \frac{f(x) - p_n(x)}{\prod_{i=0}^{n}(x - x_i)}(n + 1)!$$
which rearranges to give the formula in Theorem 2.6.

□

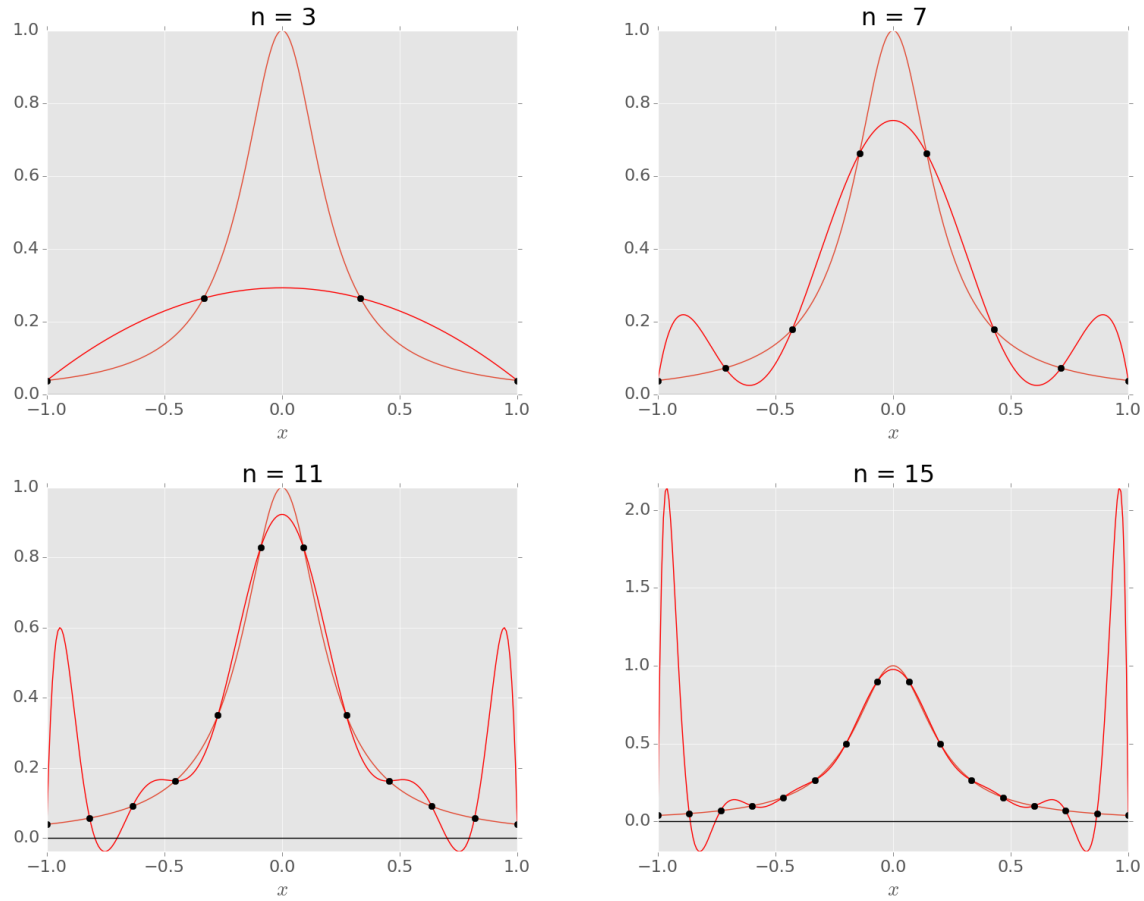
Example → Quadratic interpolant for $f(x) = \cos(x)$ with $\{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}$.
For $n = 2$, Theorem 2.6 says that
$$f(x) - p_2(x) = \frac{f^{(3)}(\xi)}{6}x(x + \tfrac{\pi}{4})(x - \tfrac{\pi}{4}) = \tfrac{1}{6}\sin(\xi)x(x + \tfrac{\pi}{4})(x - \tfrac{\pi}{4}), \quad \text{for some } \xi \in [-\tfrac{\pi}{4}, \tfrac{\pi}{4}].$$
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.

2.6 Convergence and the 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 → The *Runge function* $f(x) = 1/(1 + 25x^2)$ on $[-1, 1]$.
Here are illustrations of p_n for increasing n:



Notice that the 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*.

► 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 http://math.stackexchange.com/questions/775405/.

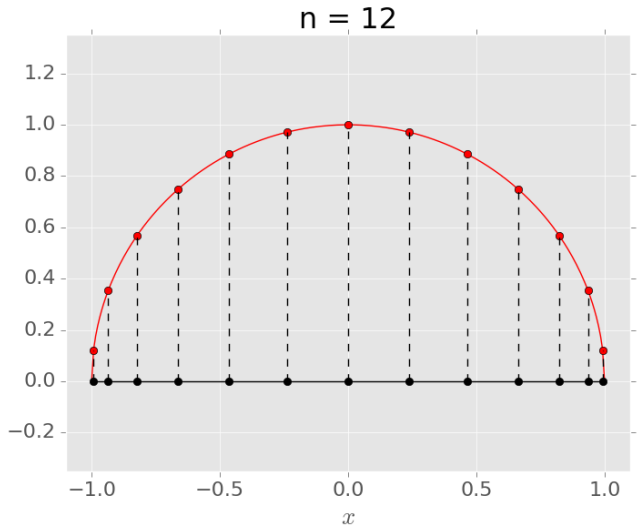
The problem is (largely) coming from the polynomial

$$w(x) = \prod_{i=0}^{n} (x - x_i). \tag{2.25}$$

We can avoid the Runge phenomenon by choosing different nodes x_i.

► If we are forced to keep equally-spaced nodes, then the best option is to use a piecewise interpolant made up of lower-degree polynomials.

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

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

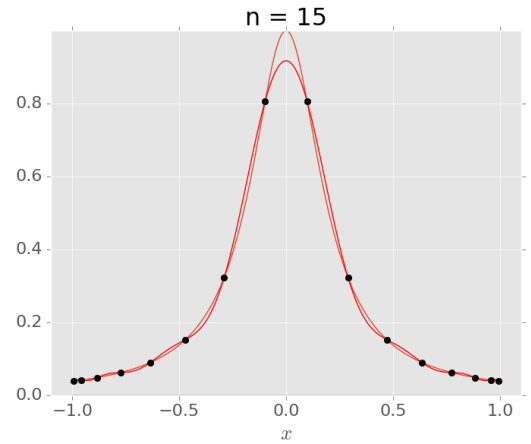
so the corresponding *Chebyshev nodes* are

$$x_j = \cos\left[\frac{(2j+1)\pi}{2(n+1)}\right], \quad j = 0, \ldots, n. \tag{2.26}$$

► Note that the order of the points is decreasing in x. This doesn't matter for any of our interpolation methods.

Example → 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.

Lemma 2.7. *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*

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

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

▶ Technically, these are Chebyshev polynomials *of the first kind.* The second kind are denoted $U_n(t)$ and defined by the recurrence $U_0(t) = 1$, $U_1(t) = 2t$, $U_{n+1}(t) = 2tU_n(t) - U_{n-1}(t)$.

▶ Chebyshev polynomials have many uses because they form an orthogonal basis for $\mathcal{P}_n$ with weight function $(1-t^2)^{-1/2}$ (next term). They (first kind) are solutions to the ODE $(1-t^2)T'' - tT' + n^2T = 0$.

Proof. There are two things to prove here. Firstly that our x_j are zeroes of this function, and secondly that it is a polynomial (not obvious!).

To see that $T_{n+1}(x_j) = 0$, just put them in for $j = 0, \ldots, n$:

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

To see that $T_n \in \mathcal{P}_n$, we will work by induction. Firstly, note that

$$T_0(t) = \cos[0] = 1 \in \mathcal{P}_0, \quad \text{and} \quad T_1(t) = \cos\Big[\arccos(t)\Big] = t \in \mathcal{P}_1.$$

Now

$$\begin{aligned} T_{n+1}(t) + T_{n-1}(t) &= \cos\Big[(n+1)\theta\Big] + \cos\Big[(n-1)\theta\Big] \quad \text{where} \quad \theta = \arccos(t), \\ &= \cos(n\theta)\cos(\theta) - \sin(n\theta)\sin(\theta) + \cos(n\theta)\cos(\theta) + \sin(n\theta)\sin(\theta), \\ &= 2\cos(n\theta)\cos\theta, \\ &= 2tT_n(t). \end{aligned}$$

This gives the recurrence relation

$$T_{n+1}(t) = 2tT_n(t) - T_{n-1}(t). \tag{2.27}$$

It follows that T_{n+1} is a polynomial of degree $n+1$. □

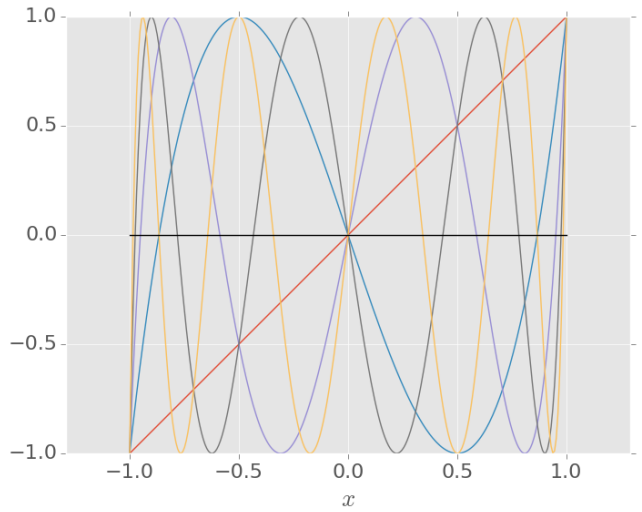
In choosing the Chebyshev nodes, we are choosing the error polynomial $w(x) := \prod_{i=0}^{n}(x - x_i)$ to be $T_{n+1}(x)/2^n$. (This normalisation makes the leading coefficient 1, from (2.27).) This is a good choice because of the following result.

Theorem 2.8 (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* $T_{n+1}(x) = \cos\Big((n+1)\arccos(x)\Big)$.

Before we prove it, look at the first few Chebyshev polynomials T_{n+1}:



The trigonometric nature of T_{n+1} means that it oscillates between ± 1, so $|w(x)|$ will not "blow up" near the ends of the interval. This is the underlying idea of the proof.

Proof. Note that $p(t) = 2^{-n}T_{n+1}(t)$ is a monic polynomial (leading coefficient 1) with correct degree $n+1$ and distinct roots. It attains its maximum absolute value at the $n+2$ points $t_j = \cos\left(\frac{j\pi}{n+1}\right)$, where $p(t_j) = 2^{-n}(-1)^j$.

Now assume there is another monic polynomial $q \in \mathcal{P}_{n+1}$ such that $\max_{[-1,1]} |q(t)| < \max_{[-1,1]} |p(t)|$.

Define the difference $r(t) = p(t) - q(t)$, which is in $\mathcal{P}_n$ since p and q are both monic. Then

$$r(t_j) = p(t_j) - q(t_j) \begin{cases} > 0 & \text{if } j \text{ even,} \\ < 0 & \text{if } j \text{ odd.} \end{cases}$$

By the intermediate value theorem, r must have $n+1$ zeroes, but $r \in \mathcal{P}_n$ so $r \equiv 0$. Hence $q \equiv p$ and $w := p$ minimises $\max_{[-1,1]} |w|$. □

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

2.7 Derivative conditions

A variant of the interpolation problem is to require that the interpolant matches one or more derivatives of f at each of the nodes, in addition to the function values.

► This is sometimes called *osculating* ("kissing") interpolation.

► We already saw the Taylor polynomial, which is an extreme case with many derivative conditions but only one node.

As an example, we consider *Hermite interpolation*, where we look for a polynomial that matches both $f'(x_i)$ and $f(x_i)$ at the nodes $x_i = x_0, \ldots, x_n$. Since there are $2n+2$ conditions, this suggests that we will need a polynomial of degree $2n+1$.

Theorem 2.9 (Hermite interpolation). *Given $n+1$ distinct nodes $x_0, x_1, \ldots, x_n$, there exists a unique polynomial $p_{2n+1} \in \mathcal{P}_{2n+1}$ that interpolates both $f(x)$ and $f'(x)$ at these points.*

Proof. 1. Uniqueness. The proof is similar to Theorem 2.4. Suppose that both p_{2n+1} and q_{2n+1} are polynomials in $\mathcal{P}_{2n+1}$ that interpolate both f and f' at the nodes. Then the difference $r_{2n+1} := p_{2n+1} - q_{2n+1}$ is also in $\mathcal{P}_{2n+1}$. But we have

$$\begin{aligned} r_{2n+1}(x_i) &= p_{2n+1}(x_i) - q_{2n+1}(x_i) = f(x_i) - f(x_i) = 0 \quad \text{for } i = 0, \ldots, n, \\ r'_{2n+1}(x_i) &= p'_{2n+1}(x_i) - q'_{2n+1}(x_i) = f'(x_i) - f'(x_i) = 0 \quad \text{for } i = 0, \ldots, n, \end{aligned}$$

so each node is a root of r_{2n+1} of multiplicity ≥ 2. Therefore r_{2n+1} has $2n+2$ roots, so by the Fundamental Theorem of Algebra $r_{2n+1}(x) \equiv 0$, which implies that $q_n = p_n$.

2. Existence. The simplest way to see that such a polynomial exists is to construct a basis analogous to the Lagrange basis, so that we can write

$$p_{2n+1}(x) = \sum_{k=0}^{n} \Big(f(x_k) h_k(x) + f'(x_k) \hat{h}_k(x) \Big), \tag{2.28}$$

where h_k and $\hat{h}_k$ are basis functions in $\mathcal{P}_{2n+1}$ that satisfy

$$\begin{array}{rclcrcl} h_k(x_j) & = & \delta_{jk}, & & h'_k(x_j) & = & 0, \\ \hat{h}_k(x_j) & = & 0, & & \hat{h}'_k(x_j) & = & \delta_{jk}. \end{array} \tag{2.29}$$

Let's try and construct h_k and $\hat{h}_k$ using the Lagrange basis functions $\ell_k(x)$, which satisfy $\ell_k(x_j) = \delta_{jk}$. To get the correct degree, try writing

$$h_k(x) = \ell_k^2(x)\big(a_k(x - x_k) + b_k\big), \qquad \hat{h}_k(x) = \ell_k^2(x)\big(\hat{a}_k(x - x_k) + \hat{b}_k\big). \tag{2.30}$$

The coefficients a_k, b_k, $\hat{a}_k$, $\hat{b}_k$ need to be chosen to match the conditions (2.29). We have

$$\begin{aligned} h_k(x_k) = 1 \quad &\Longrightarrow \quad b_k = 1, \\ h'_k(x_k) = 0 \quad &\Longrightarrow \quad 2\ell_k(x_k)\ell'_k(x_k) + a_k\ell_k^2(x_k) = 0 \quad \Longrightarrow \; a_k = -2\ell'_k(x_k), \\ \hat{h}_k(x_k) = 0 \quad &\Longrightarrow \quad \hat{b}_k = 0, \\ \hat{h}'_k(x_k) = 1 \quad &\Longrightarrow \quad \hat{a}_k\ell_k^2(x_k) = 1 \quad \Longrightarrow \; \hat{a}_k = 1. \end{aligned}$$

Hence the Hermite basis functions are

$$h_k(x) = \Big(1 - 2(x - x_k)\ell'_k(x_k)\Big)\ell_k^2(x), \tag{2.31}$$

$$\hat{h}_k(x) = (x - x_k)\ell_k^2(x). \tag{2.32}$$

□

Example → Hermite interpolant for $f(x) = \sin(x)$ with $\{0, \frac{\pi}{2}\}$.
We have $x_0 = 0, x_1 = \frac{\pi}{2}$ and $f(x_0) = 0, f(x_1) = 1, f'(x_0) = 1, f'(x_1) = 0$, so

$$p_3(x) = f(x_0)h_0(x) + f'(x_0)\hat{h}_0(x) + f(x_1)h_1(x) + f'(x_1)\hat{h}_1(x) = \hat{h}_0(x) + h_1(x).$$

The Lagrange polynomials are

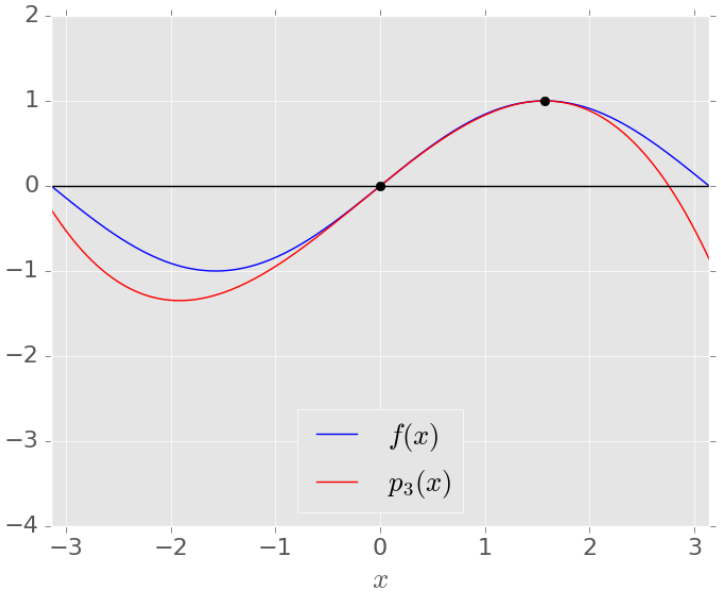
$$\ell_0(x) = \frac{x - x_1}{x_0 - x_1} = 1 - \tfrac{2}{\pi}x, \qquad \ell_1(x) = \frac{x - x_0}{x_1 - x_0} = \tfrac{2}{\pi}x \quad \Longrightarrow \quad \ell_1'(x) = \tfrac{2}{\pi}.$$

So

$$h_1(x) = \Big(1 - 2(x - x_1)\ell_1'(x)\Big)\ell_1^2(x) = \tfrac{4}{\pi^2}x^2(3 - \tfrac{4}{\pi}x), \qquad \hat{h}_0(x) = (x - x_0)\ell_0^2(x) = x(1 - \tfrac{2}{\pi}x)^2,$$

and we get

$$p_3(x) = x(1 - \tfrac{2}{\pi}x)^2 + \tfrac{4}{\pi^2}x^2(3 - \tfrac{4}{\pi}x).$$



A similar error estimate can be derived as for normal interpolation, and analogous interpolants can be found for different sets of derivative conditions (see the problems).

► The Hermite interpolant can also be calculated using a modified divided-difference table.

► A surprising connection (for those taking Algebra II): both the original and Hermite interpolating polynomials can be viewed as the Chinese Remainder Theorem applied to polynomial rings. See the Wikipedia page on *Chinese Remainder Theorem* for details.

3 Differentiation

How do we differentiate functions numerically?

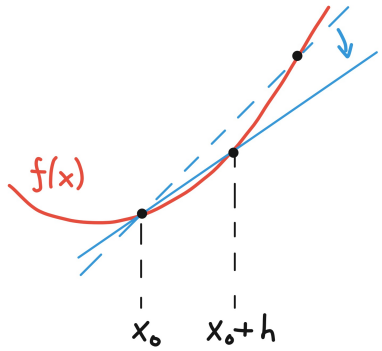
The definition of the derivative as

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}, \tag{3.1}$$

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

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}. \tag{3.2}$$

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 (3.2) by starting with the Lagrange form of the interpolating polynomial,

$$f(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1) + \frac{f''(\xi)}{2}(x - x_0)(x - x_1) \tag{3.3}$$

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

$$f'(x) = \frac{1}{x_0 - x_1} f(x_0) + \frac{1}{x_1 - x_0} f(x_1) + \frac{f''(\xi)}{2}(2x - x_0 - x_1) + \frac{f'''(\xi)}{2}\left(\frac{\mathrm{d}\xi}{\mathrm{d}x}\right)(x - x_0)(x - x_1), \tag{3.4}$$

$$\implies f'(x_0) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} + f''(\xi)\frac{x_0 - x_1}{2}. \tag{3.5}$$

Equivalently,

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - f''(\xi)\frac{h}{2}. \tag{3.6}$$

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 2.2, which tells us that

$$f(x_0 + h) = f(x_0) + hf'(x_0) + h^2 \frac{f''(\xi)}{2} \quad \text{for some } \xi \text{ between } x_0 \text{ and } x_0 + h. \tag{3.7}$$

Rearranging this will give back (3.6).

Example → 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 (3.6), 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*.

3.1 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 → 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

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

Differentiating, we get

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

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

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

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 → 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.

3.2 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

$$\mathrm{fl}[f(x_1 + h)] = (1 + \delta_1) f(x_1 + h), \qquad \mathrm{fl}[f(x_1 - h)] = (1 + \delta_2) f(x_1 - h), \tag{3.8}$$

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

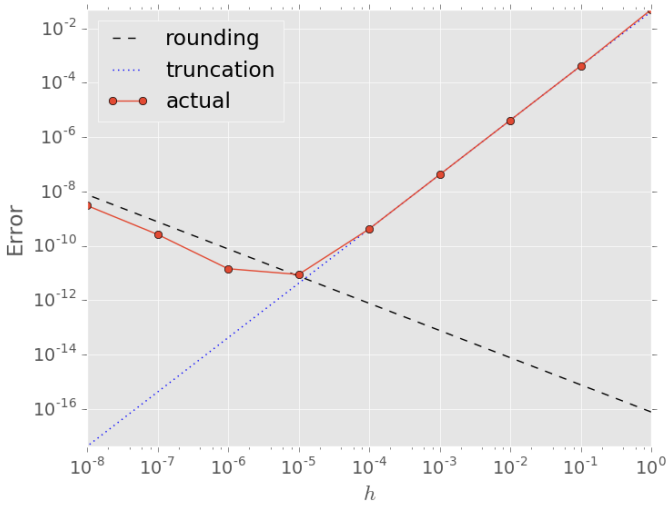
$$\left| f'(x_1) - \frac{\mathrm{fl}[f(x_1 + h)] - \mathrm{fl}[f(x_1 - h)]}{2h} \right| = \left| -\frac{f'''(\xi)}{6} h^2 - \frac{\delta_1 f(x_1 + h) - \delta_2 f(x_1 - h)}{2h} \right| \tag{3.9}$$

$$\leq \frac{|f'''(\xi)|}{6} h^2 + \epsilon_{\mathrm{M}} \frac{|f(x_1 + h)| + |f(x_1 - h)|}{2h} \tag{3.10}$$

$$\leq \frac{h^2}{6} \max_{[x_1 - h, x_1 + h]} |f'''(\xi)| + \frac{\epsilon_{\mathrm{M}}}{h} \max_{[x_1 - h, x_1 + h]} |f(\xi)|. \tag{3.11}$$

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 → Derivative of $f(x) = \log(x)$ at $x = 2$ again.
Here is a comparison of the terms in the above inequality using Python (the red points are the left-hand side), shown on logarithmic scales. To estimate the maxima, I just took $\xi = 2$.



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

3.3 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

$$D_h := \frac{f(x_1+h) - f(x_1-h)}{2h}. \tag{3.12}$$

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

$$f(x_1 \pm h) = f(x_1) \pm f'(x_1)h + \frac{f''(x_1)}{2}h^2 \pm \frac{f'''(x_1)}{3!}h^3 + \frac{f^{(4)}(x_1)}{4!}h^4 \pm \frac{f^{(5)}(x_1)}{5!}h^5 + O(h^6). \tag{3.13}$$

Substituting into (3.12), the even powers of h cancel and we get

$$D_h = \frac{1}{2h}\left(2f'(x_1)h + 2f'''(x_1)\frac{h^3}{6} + 2f^{(5)}(x_1)\frac{h^5}{120} + O(h^7)\right) \tag{3.14}$$

$$= f'(x_1) + f'''(x_1)\frac{h^2}{6} + f^{(5)}(x_1)\frac{h^4}{120} + O(h^6). \tag{3.15}$$

► You may not have seen the *big-Oh notation*. When we write $f(x) = O(g(x))$, we mean

$$\lim_{x\to 0} \frac{|f(x)|}{|g(x)|} \leq M < \infty.$$

So the error is $O(h^6)$ 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 $h/2$:

$$D_h = f'(x_1) + f'''(x_1)\frac{h^2}{6} + f^{(5)}(x_1)\frac{h^4}{120} + O(h^6), \tag{3.16}$$

$$D_{h/2} = f'(x_1) + f'''(x_1)\frac{h^2}{2^2(6)} + f^{(5)}(x_1)\frac{h^4}{2^4(120)} + O(h^6). \tag{3.17}$$

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

$$D_h - 2^2 D_{h/2} = -3f'(x_1) + \left(1 - \frac{2^2}{2^4}\right) f^{(5)}(x_1)\frac{h^4}{120} + O(h^6), \tag{3.18}$$

$$\implies \quad D_h^{(1)} := \frac{2^2 D_{h/2} - D_h}{3} = f'(x_1) - f^{(5)}(x_1)\frac{h^4}{480} + O(h^6). \tag{3.19}$$

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

Example → 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	0.0004172927849	0.4999998434	1.565994869e-07
0.01	0.5000041667	4.166729162e-06	0.5000000000	1.563887908e-11
0.001	0.5000000417	4.166661505e-08	0.5000000000	9.292566716e-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

$$D_h = f'(x) + Ch^n + O(h^{n+1}), \tag{3.20}$$

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

$$D_{h/2} = f'(x) + C\frac{h^n}{2^n} + O(h^{n+1}) \tag{3.21}$$

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

$$D_h^{(1)} := \frac{2^n D_{h/2} - D_h}{2^n - 1} = f'(x) + O(h^{n+1}). \tag{3.22}$$

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

► 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 → Iterated Richardson extrapolation for central differences.
From

$$D_h^{(1)} = f'(x_1) + C_1 h^4 + O(h^6),$$

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

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

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

► Lewis Fry Richardson (1881–1953) was from Newcastle and an undergraduate there. 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 Nonlinear equations

How do we find roots of nonlinear equations?

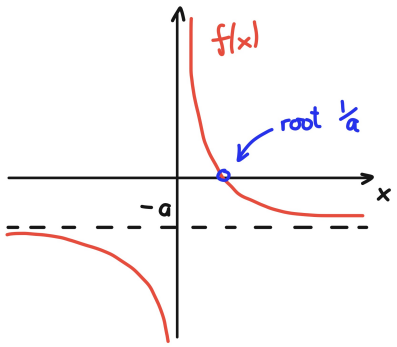
Given a general equation

$$f(x) = 0, \tag{4.1}$$

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



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

4.1 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 4.1 (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 Theorem 4.1 there must be a point $x_* \in [a, b]$ where $f(x_*) = 0$.

We can turn this into the following iterative algorithm:

Algorithm 4.2 (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 $a_{k+1} = a_k$, $b_{k+1} = m_k$.*

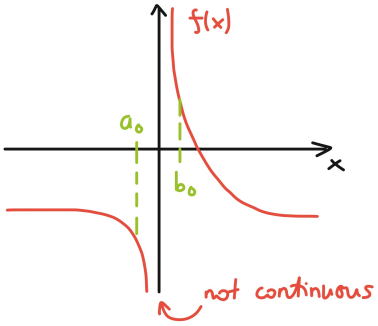
Example → $f(x) = \frac{1}{x} - 0.5$.

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 → 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

$$|b_k - a_k| = \frac{|b_0 - a_0|}{2^k}, \tag{4.2}$$

so the error in the mid-point satisfies

$$|m_k - x_*| \leq \frac{|b_0 - a_0|}{2^{k+1}}. \tag{4.3}$$

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

$$\frac{|b_0 - a_0|}{2^{n+1}} \leq \delta \quad \Longrightarrow \log|b_0 - a_0| - (n+1)\log(2) \leq \log(\delta) \quad \Longrightarrow \quad n \geq \frac{\log|b_0 - a_0| - \log(\delta)}{\log(2)} - 1. \tag{4.4}$$

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

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

► 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.

4.2 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

$$x_{k+1} = g(x_k) \tag{4.5}$$

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 $\rightarrow f(x) = x^2 - 2x - 3$.
Note that the roots are -1 and 3. Consider some different rearrangements, with $x_0 = 0$.

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

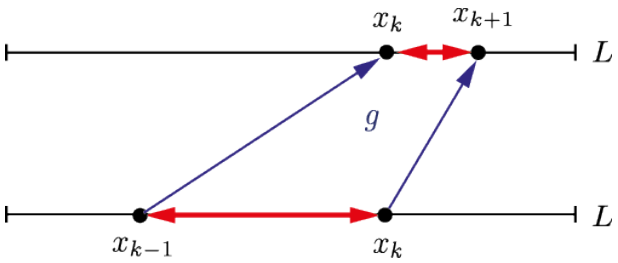
If instead we take $x_0 = 42$, then (a) and (b) still converge to the same roots, (c) now diverges, (d) still diverges, and (e) now converges to the other root $x_k \rightarrow 3$.

In this section, we will consider which iterations will converge, before addressing the *rate* of convergence in Section 4.3.

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

$$|g(x) - g(y)| \leq \lambda |x - y| \tag{4.6}$$

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



Theorem 4.3 (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 $k \rightarrow \infty$.*

Proof. To prove *existence*, consider $h(x) = g(x) - x$. Since $g : L \rightarrow L$ we have $h(a) = g(a) - a \geq 0$ and $h(b) = g(b) - b \leq 0$. Moreover, it follows from the contraction property (4.6) that g is continuous (think of "$\epsilon\delta$"), therefore so is h. So Theorem 4.1 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

$$|x_* - y_*| = |g(x_*) - g(y_*)| \leq \lambda |x_* - y_*| < |x_* - y_*|, \tag{4.7}$$

which is a contradiction.

Finally, to show *convergence*, consider

$$|x_* - x_{k+1}| = |g(x_*) - g(x_k)| \leq \lambda |x_* - x_k| \leq \ldots \leq \lambda^{k+1} |x_* - x_0|. \tag{4.8}$$

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

► Theorem 4.3 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

$$g(x) = g(y) + g'(\xi)(x - y) \qquad \Longrightarrow \qquad |g(x) - g(y)| \leq \left(\max_{\xi \in L} |g'(\xi)| \right) |x - y|. \tag{4.9}$$

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 → 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 Theorem 4.3 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 4.4 (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

$$|x_* - g(x)| = |g(x_*) - g(x)| \leq M |x_* - x| < |x_* - x| \leq \delta, \tag{4.10}$$

so $g(x) \in L$. Hence g is a contraction mapping on L and Theorem 4.3 shows that $x_k \to x_*$. □

Example → Iteration (a) again, $g(x) = \sqrt{2x+3}$.
Here we know that $x_* = 3$, and $|g'(3)| = \frac{1}{3} < 1$, so Theorem 4.4 tells us that the iteration will converge to 3 if x_0 is close enough to 3.

Example → 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 Theorem 4.4 tells us that the iteration will converge to either root if we start close enough.

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

4.3 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 → 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 $\lambda < 1$.

Example → 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 → 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

$$\lim_{k\to\infty} \frac{|x_* - x_{k+1}|}{|x_* - x_k|} = \lambda \quad \text{with} \quad 0 < \lambda < 1. \tag{4.11}$$

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

▶ The constant λ is called the *rate* or *ratio*.

▶ Clearly superlinear convergence is a desirable property for a numerical algorithm.

Theorem 4.5. *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

$$x_* - x_{k+1} = g(x_*) - g(x_k) = g(x_*) - \Big[g(x_*) + g'(\xi_k)(x_k - x_*)\Big] = g'(\xi_k)(x_* - x_k) \tag{4.12}$$

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

$$\lim_{k\to\infty} \frac{|x_* - x_{k+1}|}{|x_* - x_k|} = \lim_{k\to\infty} |g'(\xi_k)| = |g'(x_*)|. \tag{4.13}$$

This proves the result. □

Example → Iteration (a) again, $g(x) = \sqrt{2x+3}$.
We saw before that $g'(3) = \frac{1}{3}$, so Theorem 4.5 shows that convergence will be linear with $\lambda = |g'(3)| = \frac{1}{3}$ as we found numerically.

Example → Iteration (e) again, $g(x) = (x^2+3)/(2x-2)$.
We saw that $g'(-1) = 0$, so Theorem 4.5 shows that convergence will be superlinear, again consistent with our numerical findings.

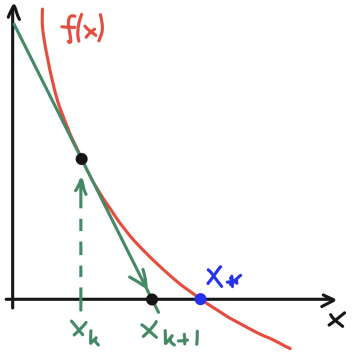
Although it is of limited practical importance in Numerical Analysis, we can further classify superlinear convergence by the *order of convergence*, defined as

$$\alpha = \sup\left\{\beta \,:\, \lim_{k\to\infty} \frac{|x_* - x_{k+1}|}{|x_* - x_k|^\beta} < \infty\right\}. \tag{4.14}$$

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).

4.4 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

$$\frac{0 - f(x_k)}{x_{k+1} - x_k} = f'(x_k) \qquad \Longrightarrow \qquad x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{4.15}$$

► 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:

$$0 \approx f(x_{k+1}) \approx f(x_k) + f'(x_k)(x_{k+1} - x_k). \tag{4.16}$$

The iteration function for Newton's method is

$$g(x) = x - \frac{f(x)}{f'(x)}, \tag{4.17}$$

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

$$g'(x) = 1 - \frac{f'(x)f'(x) - f(x)f''(x)}{[f'(x)]^2} = \frac{f(x)f''(x)}{[f'(x)]^2} \qquad \Longrightarrow g'(x_*) = 0 \quad \text{if } f'(x_*) \neq 0. \tag{4.18}$$

So if $f'(x_*) \neq 0$, Theorem 4.4 shows that the iteration will converge for x_0 close enough to x_*. Moreover, since $g'(x_*) = 0$, Theorem 4.5 shows that this convergence will be superlinear.

Example → 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	3.051757812e-05	0.00390625	0.5
2.0	4.656612873-10	1.525878906e-05	0.5
2.0	1.084202172e-19	2.328396437e-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 $\alpha = 2$).

► 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.

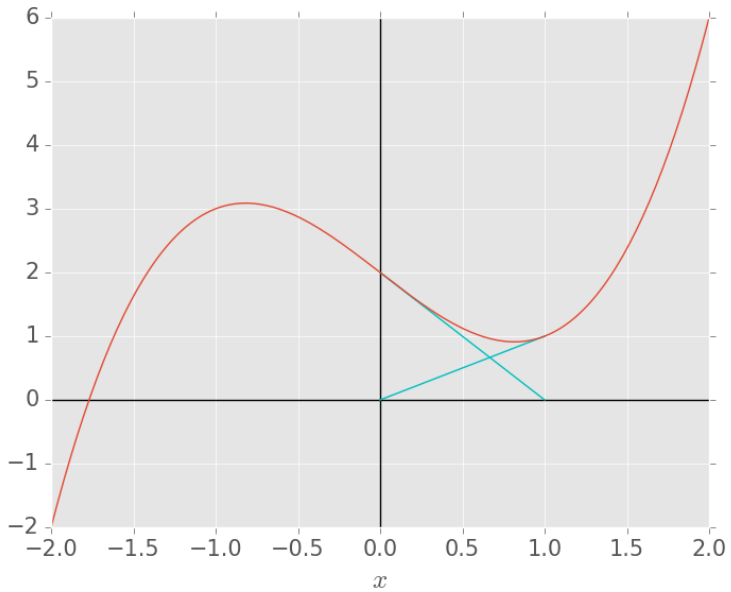
► A robust computer implementation should catch any attempt to take too large a step, and switch to a less sensitive (but slower) algorithm (e.g. bisection).

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

Example → $f(x) = x^3 - 2x + 2$.

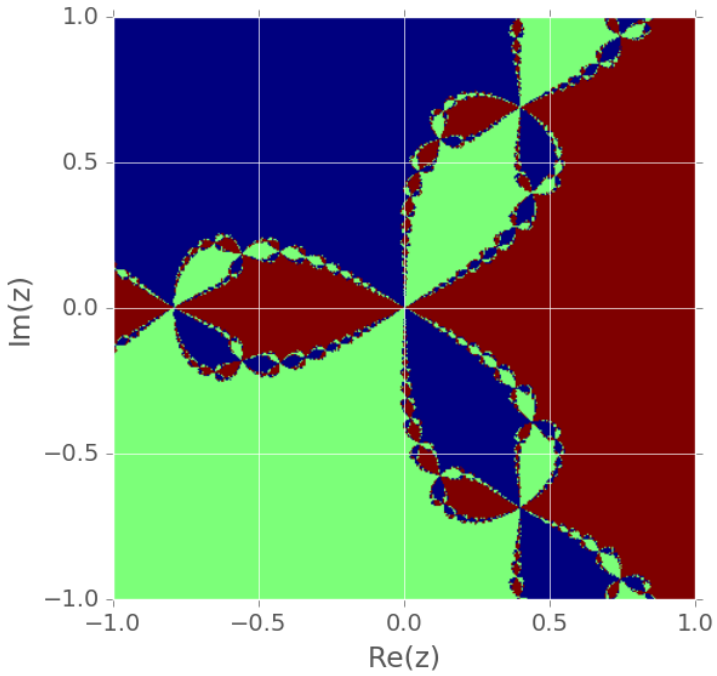
Here $f'(x) = 3x^2 - 2$ so there are turning points at $x = \pm\sqrt{\frac{2}{3}}$ where $f'(x) = 0$, as well as a single real root at $x_* \approx -1.769$. The presence of points where $f'(x) = 0$ means that care is needed in choosing a starting value x_0.

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 $x_0 = -1.0$.

► 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.

4.5 Newton's method for systems

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

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

$$0 \approx f_1(\boldsymbol{x}_{k+1}) \approx f_1(\boldsymbol{x}_k) + \left.\frac{\partial f_1}{\partial x_1}\right|_{\boldsymbol{x}_k} (x_{1,k+1} - x_{1,k}) + \left.\frac{\partial f_1}{\partial x_2}\right|_{\boldsymbol{x}_k} (x_{2,k+1} - x_{2,k}), \tag{4.19}$$

$$0 \approx f_2(\boldsymbol{x}_{k+1}) \approx f_2(\boldsymbol{x}_k) + \left.\frac{\partial f_2}{\partial x_1}\right|_{\boldsymbol{x}_k} (x_{1,k+1} - x_{1,k}) + \left.\frac{\partial f_2}{\partial x_2}\right|_{\boldsymbol{x}_k} (x_{2,k+1} - x_{2,k}). \tag{4.20}$$

In matrix form, we can write

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} f_1(\boldsymbol{x}_k) \\ f_2(\boldsymbol{x}_k) \end{pmatrix} + \begin{pmatrix} \partial f_1/\partial x_1(\boldsymbol{x}_k) & \partial f_1/\partial x_2(\boldsymbol{x}_k) \\ \partial f_2/\partial x_1(\boldsymbol{x}_k) & \partial f_2/\partial x_2(\boldsymbol{x}_k) \end{pmatrix} \begin{pmatrix} x_{1,k+1} - x_{1,k} \\ x_{2,k+1} - x_{2,k} \end{pmatrix}. \tag{4.21}$$

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

$$\boldsymbol{0} = \boldsymbol{f}(\boldsymbol{x}_k) + J(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k). \tag{4.22}$$

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

$$J(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = -\boldsymbol{f}(\boldsymbol{x}_k) \quad \Longrightarrow \quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k - J^{-1}(\boldsymbol{x}_k)\boldsymbol{f}(\boldsymbol{x}_k). \tag{4.23}$$

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

► 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 → 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

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

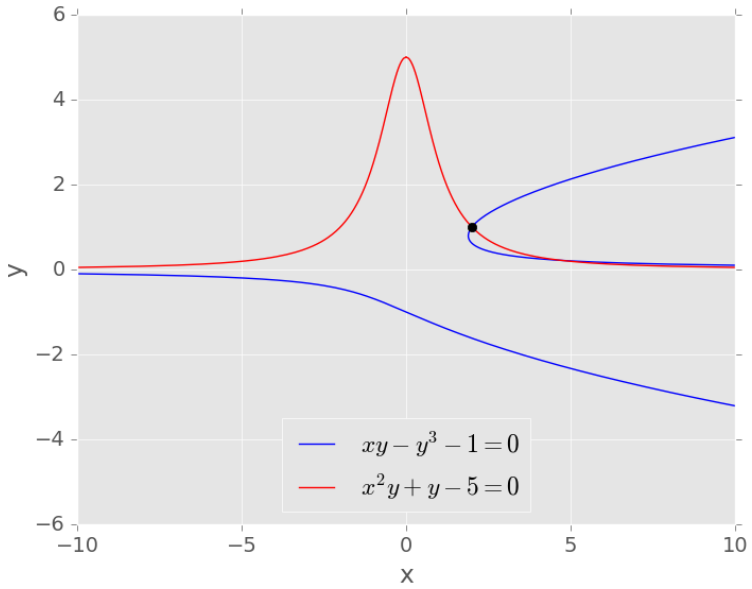
The first iteration of Newton's method gives

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

Subsequent iterations give

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1.54720541 \\ 1.47779333 \end{pmatrix}, \quad \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1.78053503 \\ 1.15886481 \end{pmatrix}, \quad \begin{pmatrix} x_4 \\ y_4 \end{pmatrix} = \begin{pmatrix} 1.952843 \\ 1.02844269 \end{pmatrix}, \quad \begin{pmatrix} x_5 \\ y_5 \end{pmatrix} = \begin{pmatrix} 1.99776297 \\ 1.00124041 \end{pmatrix},$$

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 $\boldsymbol{x}_0$ sufficiently close to $\boldsymbol{x}_*$, provided that $J(\boldsymbol{x}_*)$ is non-singular (i.e., $\det[J(\boldsymbol{x}_*)] \neq 0$).

► In general, finding a good starting point in more than one dimension is difficult, particularly because interval bisection is not available.

4.6 Aitken acceleration

This is a simple trick for accelerating the convergence of a linearly convergent sequence $\{x_k\}$, which (we recall) satisfies

$$\lim_{k\to\infty} \frac{|x_* - x_{k+1}|}{|x_* - x_k|} = \lambda. \tag{4.24}$$

If we had exactly

$$\frac{|x_* - x_{k+1}|}{|x_* - x_k|} = \lambda, \tag{4.25}$$

then we could just take two neighbouring iterates and extrapolate directly to the answer,

$$x_* = \frac{x_{k+1} - \lambda x_k}{1 - \lambda}, \tag{4.26}$$

with no need for any further iteration. Alternatively, we could use two successive iterations to eliminate λ:

$$\frac{x_* - x_{k+2}}{x_* - x_{k+1}} = \frac{x_* - x_{k+1}}{x_* - x_k} \implies (x_* - x_{k+2})(x_* - x_k) = (x_* - x_{k+1})^2 \tag{4.27}$$

$$\implies x_{k+2}x_k - (x_{k+2} + x_k)x_* = x_{k+1}^2 - 2x_{k+1}x_* \tag{4.28}$$

$$\implies x_* = \frac{x_{k+2}x_k - x_{k+1}^2}{x_{k+2} - 2x_{k+1} + x_k} = x_k - \frac{(x_{k+1} - x_k)^2}{x_{k+2} - 2x_{k+1} + x_k}. \tag{4.29}$$

The idea of *Aitken acceleration* is that, even when the ratio between successive errors is not precisely constant, the extrapolation (4.29) may still get closer to x_* than a single step (or even several steps) of the iteration. The idea is to replace every third iterate by

$$\hat{x}_{k+2} = x_k - \frac{(x_{k+1} - x_k)^2}{x_{k+2} - 2x_{k+1} + x_k}. \tag{4.30}$$

► You may see the *forward difference* notation $\Delta x_k := x_{k+1} - x_k$, and $\Delta^2 x_k := \Delta(\Delta x_k)$.

► The method was introduced by Alexander Aitken in 1926. He worked in Edinburgh, and at Bletchley Park during WWII.

Example → $f(x) = \frac{1}{x} - 0.5$ with $g(x) = x + \frac{1}{16}f(x)$.
First we verify that this is a valid iteration. It is clear that $g(2) = 2$, and $g'(x) = 1 + \frac{1}{16}(-\frac{1}{x^2})$, so $g'(2) = 0.984375$, so by Theorem 4.4 the iteration will converge to $x_* = 2$ if we start close enough. Since $g'(2)$ is non-zero, convergence is linear.

But since $g'(2)$ is almost 1, we expect convergence to be slow. In Python, we get (to 7 s.f.):

$$\begin{aligned} x_0 &= 1.5 \\ x_1 &= 1.510417 \\ x_2 &= 1.520546 \\ x_3 &= 1.530400 \\ x_4 &= 1.539989 \\ &\vdots \\ x_{818} &= 1.999999 \end{aligned}$$

It takes 818 iterations to reduce the error to 10^{-6}.

Now we apply Aitken acceleration, and find the following

$$\begin{aligned}
x_0 &= 1.5 \\
x_1 &= 1.510417 \\
x_2 &= 1.520546 \\
\hat{x}_2 &= x_0 - (x_1 - x_0)^2/(x_2 - 2x_1 + x_0) = 1.8776042 \\
x_3 &= 1.8796413 \\
x_4 &= 1.8816423 \\
\hat{x}_4 &= \hat{x}_2 - (x_3 - \hat{x}_2)^2/(x_4 - 2x_3 + \hat{x}_2) = 1.9926343 \\
\vdots &\quad \vdots \\
\hat{x}_8 &= 2.000000
\end{aligned}$$

The error has reduced to 10^{-6} with only 8 evaluations of g.

► Aitken acceleration must be implemented with caution, as rounding error can affect $\Delta^2 x_k$.

When the original sequence $\{x_k\}$ comes from $x_{k+1} = g(x_k)$, we can write

$$\hat{x}_{k+1} = \hat{x}_k - \frac{(x_{k+1} - \hat{x}_k)^2}{x_{k+2} - 2x_{k+1} + \hat{x}_k} \tag{4.31}$$

$$= \hat{x}_k - \frac{\Big(g(\hat{x}_k) - \hat{x}_k\Big)^2}{g(g(\hat{x}_k)) - 2g(\hat{x}_k) + \hat{x}_k} = \hat{x}_k - \frac{\Big(g(\hat{x}_k) - \hat{x}_k\Big)^2}{\Big(g(g(\hat{x}_k)) - g(\hat{x}_k)\Big) - \Big(g(\hat{x}_k) - \hat{x}_k\Big)}. \tag{4.32}$$

Now suppose we are solving $f(x) = 0$ with the iteration function $g(x) = x + f(x)$. Then

$$\hat{x}_{k+1} = \hat{x}_k - \frac{f^2(\hat{x}_k)}{f\Big(f(\hat{x}_k) + \hat{x}_k\Big) - f(\hat{x}_k)}. \tag{4.33}$$

This is called *Steffensen's method.*

► This has the advantage of not requiring f', yet may be shown to converge quadratically for $\hat{x}_0$ close enough to x_* [proof omitted]. On the other hand, it does require two function evaluations per step.

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

$$x_{k+1} = x_k - \frac{f(x_k)}{g_k}, \tag{4.34}$$

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

Example → Steffensen's method:

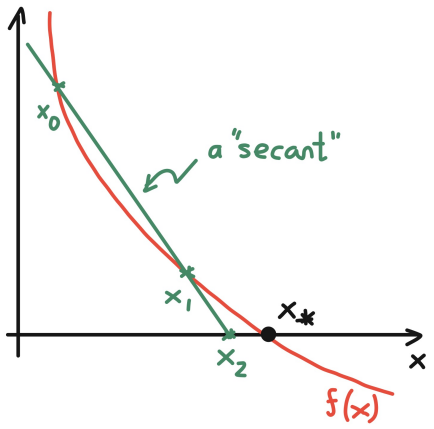
$$g_k = \frac{f\Big(f(x_k) + x_k\Big) - f(x_k)}{f(x_k)}.$$

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

$$g_k = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \qquad \Longrightarrow \qquad x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}. \tag{4.35}$$

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



► The secant method was introduced by Newton.

Example → $f(x) = \frac{1}{x} - 0.5$.
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 4.4 and 4.5 do not apply.

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

Theorem 4.6. *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 $(1 + \sqrt{5})/2 = 1.618\ldots$.*

► 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

$$\varepsilon_k := x_* - x_k. \tag{4.36}$$

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

$$f(x_{k-1}) = -f'(x_*)\varepsilon_{k-1} + \frac{f''(x_*)}{2}\varepsilon_{k-1}^2 + O(\varepsilon_{k-1}^3), \tag{4.37}$$

$$f(x_k) = -f'(x_*)\varepsilon_k + \frac{f''(x_*)}{2}\varepsilon_k^2 + O(\varepsilon_k^3). \tag{4.38}$$

So using the secant formula (4.35) we get

$$\varepsilon_{k+1} = \varepsilon_k - (\varepsilon_k - \varepsilon_{k-1})\frac{-f'(x_*)\varepsilon_k + \frac{f''(x_*)}{2}\varepsilon_k^2 + O(\varepsilon_k^3)}{-f'(x_*)(\varepsilon_k - \varepsilon_{k-1}) + \frac{f''(x_*)}{2}(\varepsilon_k^2 - \varepsilon_{k-1}^2) + O(\varepsilon_{k-1}^3)}, \tag{4.39}$$

$$= \varepsilon_k - \frac{-f'(x_*)\varepsilon_k + \frac{f''(x_*)}{2}\varepsilon_k^2 + O(\varepsilon_k^3)}{-f'(x_*) + \frac{f''(x_*)}{2}(\varepsilon_k + \varepsilon_{k-1}) + O(\varepsilon_{k-1}^2)}, \tag{4.40}$$

$$= \varepsilon_k + \frac{-\varepsilon_k + \frac{1}{2}\varepsilon_k^2 f''(x_*)/f'(x_*) + O(\varepsilon_k^3)}{1 - \frac{1}{2}(\varepsilon_k + \varepsilon_{k-1})f''(x_*)/f'(x_*) + O(\varepsilon_{k-1}^2)}, \tag{4.41}$$

$$= \varepsilon_k + \left(-\varepsilon_k + \frac{f''(x_*)}{2f'(x_*)}\varepsilon_k^2 + O(\varepsilon_k^3)\right)\left(1 + (\varepsilon_k + \varepsilon_{k-1})\frac{f''(x_*)}{2f'(x_*)} + O(\varepsilon_{k-1}^2)\right), \tag{4.42}$$

$$= \varepsilon_k - \varepsilon_k + \frac{f''(x_*)}{2f'(x_*)}\varepsilon_k^2 - \frac{f''(x_*)}{2f'(x_*)}\varepsilon_k(\varepsilon_k + \varepsilon_{k-1}) + O(\varepsilon_{k-1}^3), \tag{4.43}$$

$$= -\frac{f''(x_*)}{2f'(x_*)}\varepsilon_k\varepsilon_{k-1} + O(\varepsilon_{k-1}^3). \tag{4.44}$$

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

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

Equation (4.44) 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

$$|\varepsilon_k| = |\varepsilon_{k-1}|^\alpha \left|\frac{f''(x_*)}{2f'(x_*)}\right|^\beta \quad \Longrightarrow \quad |\varepsilon_{k-1}| = |\varepsilon_k|^{1/\alpha} \left|\frac{f''(x_*)}{2f'(x_*)}\right|^{-\beta/\alpha}. \tag{4.45}$$

Putting this in both sides of (4.44) gives

$$|\varepsilon_k|^\alpha \left|\frac{f''(x_*)}{2f'(x_*)}\right|^\beta = |\varepsilon_k|^{(1+\alpha)/\alpha} \left|\frac{f''(x_*)}{2f'(x_*)}\right|^{(\alpha-\beta)/\alpha}. \tag{4.46}$$

Equating powers gives

$$\alpha = \frac{1+\alpha}{\alpha} \quad \Longrightarrow \alpha = \frac{1+\sqrt{5}}{2}, \quad \beta = \frac{\alpha-\beta}{\alpha} \quad \Longrightarrow \beta = \frac{\alpha}{\alpha+1} = \frac{1}{\alpha}. \tag{4.47}$$

It follows that

$$\lim_{k\to\infty} \frac{|x_* - x_{k+1}|}{|x_* - x_k|^\alpha} = \lim_{k\to\infty} \frac{|\varepsilon_{k+1}|}{|\varepsilon_k|^\alpha} = \left|\frac{f''(x_*)}{2f'(x_*)}\right|^{1/\alpha}, \tag{4.48}$$

so the secant method has order of convergence α. □

5 Linear equations

How do we solve a linear system numerically?

Linear systems of the form

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1, \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2, \\ \vdots & & \vdots & & & & \vdots & = & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \dots & + & a_{nn}x_n & = & b_n \end{array} \tag{5.1}$$

occur in many applications (often with very large n). It is convenient to express (5.1) in the matrix form

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{5.2}$$

where A is an $n \times n$ square matrix with elements a_{ij}, and $\boldsymbol{x}$, $\boldsymbol{b}$ 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 $\boldsymbol{x} \in \mathbb{R}^n$ for every $\boldsymbol{b} \in \mathbb{R}^n$.
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 that (5.2) has a unique solution iff A is non-singular, given by $\boldsymbol{x} = A^{-1}\boldsymbol{b}$.

In this chapter, we will see how to solve (5.2) 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.

► 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 – this will be considered in Section 6.

Many algorithms are based on the idea of rewriting (5.2) 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 $A\boldsymbol{x} = \boldsymbol{b}$ transforms the system to *triangular* form.

5.1 Triangular systems

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

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

$$L = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ l_{n1} & \cdots & \cdots & l_{nn} \end{pmatrix}. \tag{5.3}$$

The determinant is just

$$\det(L) = l_{11} l_{22} \cdots l_{nn}, \tag{5.4}$$

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

Example → Solve $L\boldsymbol{x} = \boldsymbol{b}$ for $n = 4$.
The system is

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

We can just solve step-by-step:

$$x_1 = \frac{b_1}{l_{11}}, \quad x_2 = \frac{b_2 - l_{21}x_1}{l_{22}}, \quad x_3 = \frac{b_3 - l_{31}x_1 - l_{32}x_2}{l_{33}}, \quad x_4 = \frac{b_4 - l_{41}x_1 - l_{42}x_2 - l_{43}x_3}{l_{44}}.$$

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 $L\boldsymbol{x} = \boldsymbol{b}$ can be solved by *forward substitution*

$$x_j = \frac{b_j - \sum_{k=1}^{j-1} l_{jk} x_k}{l_{jj}}, \quad j = 1, \ldots, n. \tag{5.5}$$

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

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{pmatrix}, \tag{5.6}$$

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

$$x_j = \frac{b_j - \sum_{k=j+1}^{n} u_{jk} x_k}{u_{jj}}, \quad j = n, \ldots, 1. \tag{5.7}$$

To estimate the computational cost of forward substitution, we can count the number of floating-point operations (+, −, ×, ÷).

Example → Number of operations required for forward substitution.
Consider each x_j. We have

$j = 1 \rightarrow$ 1 division
$j = 2 \rightarrow$ 1 division + [1 subtraction + 1 multiplication]
$j = 3 \rightarrow$ 1 division + 2 ×[1 subtraction + 1 multiplication]
⋮
$j = n \rightarrow$ 1 division + $(n-1)$ ×[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 say that the *computational complexity* of the algorithm is n^2.

► 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.

5.2 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 $U\boldsymbol{x} = \boldsymbol{y}$.

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

Example → Transform to upper triangular form the system

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 0, \\ x_1 - 2x_2 + 2x_3 &= 4, \\ 2x_1 + 12x_2 - 2x_3 &= 4. \end{aligned} \qquad A = \begin{pmatrix} 1 & 2 & 1 \\ 1 & -2 & 2 \\ 2 & 12 & -2 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 0 \\ 4 \\ 4 \end{pmatrix}.$$

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:

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 0, \\ -4x_2 + x_3 &= 4, \\ 8x_2 - 4x_3 &= 4. \end{aligned} \qquad A^{(2)} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 8 & -4 \end{pmatrix} \quad \boldsymbol{b}^{(2)} = \begin{pmatrix} 0 \\ 4 \\ 4 \end{pmatrix}, \quad m_{21} = 1, \quad m_{31} = 2.$$

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

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 0, \\ -4x_2 + x_3 &= 4, \\ -2x_3 &= 12. \end{aligned} \qquad A^{(3)} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 0 & -2 \end{pmatrix} \quad \boldsymbol{b}^{(3)} = \begin{pmatrix} 0 \\ 4 \\ 12 \end{pmatrix}, \quad m_{32} = -2.$$

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 5.1 (Gaussian elimination). *Let $A^{(1)} = A$ and $\boldsymbol{b}^{(1)} = \boldsymbol{b}$. Then for each k from 1 to $n-1$, compute a new matrix $A^{(k+1)}$ and right-hand side $\boldsymbol{b}^{(k+1)}$ 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 → 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

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

Recalling that

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

we find

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

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

► It is known that $O(n^3)$ is not optimal, and the best theoretical algorithm known for inverting a matrix takes $O(n^{2.3728639})$ operations (although this is not practically useful). But it remains an open conjecture that there exists an $O(n^{2+\epsilon})$ algorithm, for ϵ arbitrarily small.

5.3 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 $\boldsymbol{b}$. 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

$$FA = U, \qquad U\boldsymbol{x} = F\boldsymbol{b}. \tag{5.8}$$

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

$$A^{(k+1)} = F^{(k)}A^{(k)}, \quad \boldsymbol{b}^{(k+1)} = F^{(k)}\boldsymbol{b}^{(k)}, \quad \text{where } F^{(k)} := \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & 1 & \ddots & & \vdots \\ \vdots & & -m_{k+1,k} & \ddots & \ddots & \vdots \\ \vdots & & \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & -m_{n,k} & \cdots & 0 & 1 \end{pmatrix}. \tag{5.9}$$

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

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

Example → You can check in the earlier example that

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

and

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

It follows that

$$U = A^{(n)} = F^{(n-1)}F^{(n-2)}\cdots F^{(1)}A. \tag{5.10}$$

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

$$(F^{(k)})^{-1} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & 1 & \ddots & & \vdots \\ \vdots & & m_{k+1,k} & \ddots & \ddots & \vdots \\ \vdots & & \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & m_{n,k} & \cdots & 0 & 1 \end{pmatrix}. \tag{5.11}$$

So we could write

$$A = (F^{(1)})^{-1}(F^{(2)})^{-1}\cdots(F^{(n-1)})^{-1}U. \tag{5.12}$$

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

$$(F^{(1)})^{-1}(F^{(2)})^{-1}\cdots(F^{(n-1)})^{-1} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ m_{2,1} & 1 & \ddots & & & \vdots \\ m_{3,1} & m_{3,2} & 1 & \ddots & & \vdots \\ m_{4,1} & m_{4,2} & m_{4,3} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & & 1 & 0 \\ m_{n,1} & m_{n,2} & m_{n,3} & \cdots & m_{n,n-1} & 1 \end{pmatrix} := L. \tag{5.13}$$

Thus we have established the following result.

Theorem 5.2 (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* (5.13)*. Then*

$$A = LU.$$

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

► Theorem 5.2 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 Algorithm 5.1! The decomposition is unique up to a scaling LD, $D^{-1}U$ for some diagonal matrix D.

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

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

► 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 → 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 $\boldsymbol{b}$ (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 $\boldsymbol{b}$. We illustrate for our particular $\boldsymbol{b}$. Firstly, solve $L\boldsymbol{y} = \boldsymbol{b}$:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & -2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 4 \end{pmatrix} \quad \Longrightarrow \quad y_1 = 0, \quad y_2 = 4 - y_1 = 4, \quad y_3 = 4 - 2y_1 + 2y_2 = 12.$$

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

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & -4 & 1 \\ 0 & 0 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 12 \end{pmatrix} \quad \Longrightarrow \quad x_3 = -6, \quad x_2 = -\tfrac{1}{4}(4 - x_3) = -\tfrac{5}{2}, \quad x_1 = -2x_2 - x_3 = 11.$$

5.4 Pivoting

Gaussian elimination and LU factorisation will both fail if we ever hit a zero on the diagonal. But this does not mean that the matrix A is singular.

Example → The system

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

obviously has solution $x_1 = x_2 = x_3 = 1$ (the matrix has determinant -6). But Gaussian elimination will fail because $a_{11}^{(1)} = 0$, so we cannot calculate m_{21} and m_{31}. However, we could avoid the problem by changing the order of the equations to get the equivalent system

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}.$$

Now there is no problem with Gaussian elimination (actually the matrix is already upper triangular). Alternatively, we could have rescued Gaussian elimination by swapping columns:

$$\begin{pmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ x_1 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}.$$

Swapping rows or columns is called *pivoting*. It is needed if the "pivot" element is zero, as in the above example. But it is also used to reduce rounding error.

Example → Consider the system

$$\begin{pmatrix} 10^{-4} & 1 \\ -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

1. Using Gaussian elimination with *exact* arithmetic gives

   $$m_{21} = -10^4, \quad a_{22}^{(2)} = 2 + 10^4, \quad b_2^{(2)} = 1 + 10^4.$$

   So backward substitution gives the solution

   $$x_2 = \frac{1 + 10^4}{2 + 10^4} = 0.9999, \quad x_1 = \frac{1 - x_2}{a_{11}} = 10^4 \left(1 - \frac{1 + 10^4}{2 + 10^4}\right) = \frac{10^4}{2 + 10^4} = 0.9998.$$

2. Now do the calculation in 3-digit arithmetic. We have

   $$m_{21} = \mathrm{fl}(-10^4) = -10^4, \quad a_{22}^{(2)} = \mathrm{fl}(2 + 10^4) = 10^4, \quad b_2^{(2)} = \mathrm{fl}(1 + 10^4) = 10^4.$$

   Now backward substitution gives

   $$x_2 = \mathrm{fl}\left(\frac{10^4}{10^4}\right) = 1, \quad x_1 = \mathrm{fl}\left(10^4(1 - 1)\right) = 0.$$

   The large value of m_{21} has caused a rounding error which has later led to a loss of significance during the evaluation of x_1.

3. We do the calculation correctly in 3-digit arithmetic if we first swap the equations,

$$\begin{pmatrix} -1 & 2 \\ 10^{-4} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Now,

$$m_{21} = \text{fl}(-10^{-4}) = -10^{-4}, \quad a_{22}^{(2)} = \text{fl}(1 + 10^{-4}) = 1, \quad b_2^{(2)} = \text{fl}(1 + 10^{-4}) = 1,$$

and

$$x_2 = \text{fl}\left(\frac{1}{1}\right) = 1, \quad x_1 = \text{fl}\left(-[1 - 2(1)]\right) = 1.$$

Now both x_1 and x_2 are correct to 3 significant figures.

So pivoting is used to avoid large multipliers m_{ik}. A common strategy is *partial pivoting*, where we interchange rows at the kth stage of Gaussian elimination to bring the largest element $a_{ik}^{(k)}$ (for $k \leq i \leq n$) to the diagonal position $a_{kk}^{(k)}$. This dramatically improves the stability of Gaussian elimination.

► Gaussian elimination without pivoting is *unstable*: rounding errors can accumulate.

► The ultimate accuracy is obtained by *full pivoting*, where both the rows and columns are swapped to bring the largest element possible to the diagonal.

► If it is not possible to rearrange the columns or rows to remove a zero from position $a_{kk}^{(k)}$, then A is singular.

If pivoting is applied, then the effect of Gaussian elimination is to produce a modified LU factorisation of the form

$$PA = LU, \tag{5.14}$$

where P is a *permutation matrix*. This is a matrix where every row and column has exactly one non-zero element, which is 1.

► The permutation matrices form a (sub)group, so a product of permutation matrices equals another, different, permutation matrix.

In this case, we solve $L\boldsymbol{y} = P\boldsymbol{b}$ then $U\boldsymbol{x} = \boldsymbol{y}$.

Example → Consider again the system

$$\begin{pmatrix} 0 & 3 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}.$$

To swap rows 1 and 2, we can left-multiply A by the permutation matrix

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \implies PA = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad P\boldsymbol{b} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}.$$

Now we find the LU factorisation of PA, which is easy in this case:

$$PA = LU \quad \text{where} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Since $LU\boldsymbol{x} = P\boldsymbol{b}$, we can then solve for $\boldsymbol{x}$ in two steps:

$$L\boldsymbol{y} = P\boldsymbol{b} \implies \boldsymbol{y} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}, \qquad U\boldsymbol{x} = \boldsymbol{y} \implies \begin{pmatrix} 2x_1 \\ 3x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix} \implies \boldsymbol{x} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

5.5 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 *vector norm* on $\mathbb{R}^n$ is a real-valued function that satisfies

$$\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\| \quad \text{for every } \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n, \tag{N1}$$

$$\|\alpha\boldsymbol{x}\| = |\alpha| \, \|\boldsymbol{x}\| \quad \text{for every } \boldsymbol{x} \in \mathbb{R}^n \text{ and every } \alpha \in \mathbb{R}, \tag{N2}$$

$$\|\boldsymbol{x}\| \geq 0 \quad \text{for every } \boldsymbol{x} \in \mathbb{R}^n \text{ and } \|\boldsymbol{x}\| = 0 \implies \boldsymbol{x} = 0. \tag{N3}$$

Property (N1) is called the *triangle inequality*.

Example → There are three common examples:

1. The ℓ_2*-norm*
$$\|\boldsymbol{x}\|_2 := \sqrt{\sum_{k=1}^{n} x_k^2} = \sqrt{\boldsymbol{x}^\top \boldsymbol{x}}.$$
This is just the usual Euclidean length of $\boldsymbol{x}$.
2. The ℓ_1*-norm*
$$\|\boldsymbol{x}\|_1 := \sum_{k=1}^{n} |x_k|.$$
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 $\boldsymbol{x} \in \mathbb{R}^2$.
3. The ℓ_∞*-norm*
$$\|\boldsymbol{x}\|_\infty := \max_{k=1,\ldots,n} |x_k|.$$
This is sometimes known as the *maximum* norm.

We leave the proofs that these satisfy (N1)-(N3) to the problem sheet.

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

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

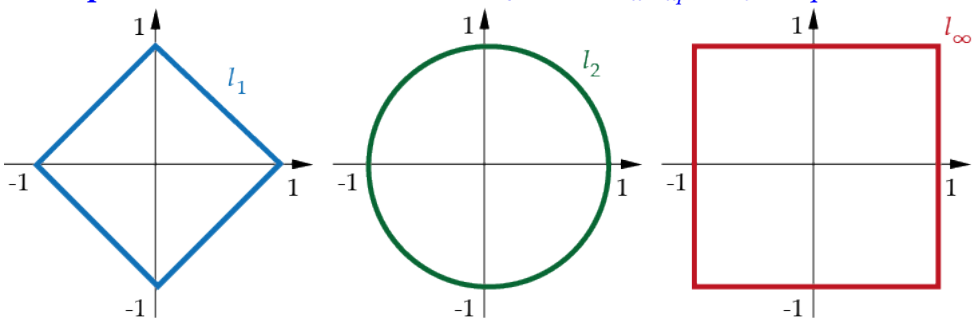
which is a norm for any real number $p \geq 1$. Increasing p means that more and more emphasis is given to the maximum element $|x_k|$.

Example → Consider the vectors $\boldsymbol{a} = (1, -2, 3)^\top$, $\boldsymbol{b} = (2, 0, -1)^\top$, and $\boldsymbol{c} = (0, 1, 4)^\top$.
The ℓ_1-, ℓ_2–, and ℓ_∞-norms are

$$\begin{array}{lll} \|\boldsymbol{a}\|_1 = 1 + 2 + 3 = 6 & \|\boldsymbol{b}\|_1 = 2 + 0 + 1 = 3 & \|\boldsymbol{c}\|_1 = 0 + 1 + 4 = 5 \\ \|\boldsymbol{a}\|_2 = \sqrt{1 + 4 + 9} \approx 3.74 & \|\boldsymbol{b}\|_2 = \sqrt{4 + 0 + 1} \approx 2.24 & \|\boldsymbol{c}\|_2 = \sqrt{0 + 1 + 16} \approx 4.12 \\ \|\boldsymbol{a}\|_\infty = \max\{1, 2, 3\} = 3 & \|\boldsymbol{b}\|_\infty = \max\{2, 0, 1\} = 2 & \|\boldsymbol{c}\|_\infty = \max\{0, 1, 4\} = 4. \end{array}$$

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

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



5.6 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:

$$\|A + B\| \leq \|A\| + \|B\| \quad \text{for every } A, B \in \mathbb{R}^{n\times n}, \tag{M1}$$

$$\|\alpha A\| = |\alpha|\,\|A\| \quad \text{for every } A \in \mathbb{R}^{n\times n} \text{ and every } \alpha \in \mathbb{R}, \tag{M2}$$

$$\|A\| \geq 0 \quad \text{for every } A \in \mathbb{R}^{n\times n} \text{ and } \|A\| = 0 \implies A = 0, \tag{M3}$$

$$\|AB\| \leq \|A\|\|B\| \quad \text{for every } A, B \in \mathbb{R}^{n\times n}. \tag{M4}$$

The new axiom (M4) is called *consistency*.

► 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 (M1), (M2), (M3), perhaps because (M4) only works for square matrices.

Example → 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

$$\|A\|_p := \sup_{\boldsymbol{x}\neq\boldsymbol{0}} \frac{\|A\boldsymbol{x}\|_p}{\|\boldsymbol{x}\|_p} = \max_{\|\boldsymbol{x}\|_p=1} \|A\boldsymbol{x}\|_p. \tag{5.15}$$

To see that the two definitions here are equivalent, use the fact that $\|\cdot\|_p$ is a vector norm. So by (N2) we have

$$\sup_{\boldsymbol{x}\neq\boldsymbol{0}} \frac{\|A\boldsymbol{x}\|_p}{\|\boldsymbol{x}\|_p} = \sup_{\boldsymbol{x}\neq\boldsymbol{0}} \left\| A\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|_p} \right\|_p = \sup_{\|\boldsymbol{y}\|_p=1} \|A\boldsymbol{y}\|_p = \max_{\|\boldsymbol{y}\|_p=1} \|A\boldsymbol{y}\|_p. \tag{5.16}$$

► 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.

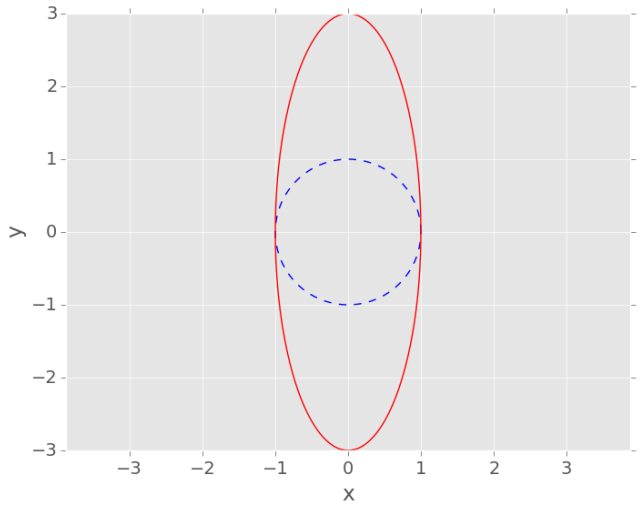
Example → Let

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

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

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

This is illustrated below:



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

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

Theorem 5.3. *The induced norm corresponding to any vector norm is a matrix norm, and the two norms satisfy* $\|A\boldsymbol{x}\| \leq \|A\|\|\boldsymbol{x}\|$ *for any matrix* $A \in \mathbb{R}^{n\times n}$ *and any vector* $\boldsymbol{x} \in \mathbb{R}^n$.

Proof. Properties (M1)-(M3) follow from the fact that the vector norm satisfies (N1)-(N3). To show (M4), note that, by the definition (5.15), we have for any vector $\boldsymbol{y} \in \mathbb{R}^n$ that

$$\|A\| \geq \frac{\|A\boldsymbol{y}\|}{\|\boldsymbol{y}\|} \quad \Longrightarrow \quad \|A\boldsymbol{y}\| \leq \|A\|\|\boldsymbol{y}\|. \tag{5.17}$$

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

$$\|AB\boldsymbol{x}\| \leq \|A\|\|B\boldsymbol{x}\| \leq \|A\|\|B\|. \tag{5.18}$$

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

$$\|AB\| = \max_{\|\boldsymbol{x}\|=1} \|AB\boldsymbol{x}\| \leq \|A\|\|B\|. \tag{5.19}$$

□

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

Theorem 5.4. *The matrix norms induced by the ℓ_1-norm and ℓ_∞-norm satisfy*

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

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

Proof. We will prove the result for the ℓ_1-norm and leave the ℓ_∞-norm to the problem sheet. Starting from the definition of the ℓ_1 vector norm, we have

$$\|A\boldsymbol{x}\|_1 = \sum_{i=1}^{n} \left| \sum_{j=1}^{n} a_{ij}x_j \right| \leq \sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|\,|x_j| = \sum_{j=1}^{n} |x_j| \sum_{i=1}^{n} |a_{ij}|. \tag{5.20}$$

If we let

$$c = \max_{j=1,\ldots,n} \sum_{i=1}^{n} |a_{ij}|, \tag{5.21}$$

then

$$\|A\boldsymbol{x}\|_1 \leq c\|\boldsymbol{x}\|_1 \quad \Longrightarrow \quad \|A\|_1 \leq c. \tag{5.22}$$

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

$$\max_{\|\boldsymbol{x}\|_1=1} \|A\boldsymbol{x}\|_1 \geq \|A\boldsymbol{y}\|_1 = c \quad \Longrightarrow \quad \|A\|_1 \geq c. \tag{5.23}$$

The only way to satisfy both (5.22) and (5.23) is if $\|A\|_1 = c$. □

Example → 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* $\boldsymbol{u}$ if

$$A\boldsymbol{u} = \lambda \boldsymbol{u}. \tag{5.24}$$

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

Theorem 5.5. *The matrix norm induced by the ℓ_2-norm satisfies*

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

▶ As a result this is sometimes known as the *spectral norm.*

Example → 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

$$\max_{\|\boldsymbol{x}\|_2 = 1} \|A\boldsymbol{x}\|_2 = \max\{\sqrt{|\lambda|} \,:\, \lambda \text{ eigenvalue of } A^\top A\}. \tag{5.25}$$

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 $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n$ in $\mathbb{R}^n$. (Orthonormal means that $\boldsymbol{u}_j^\top \boldsymbol{u}_k = \delta_{jk}$.) Note also that all of the eigenvalues are non-negative, since

$$A^\top A\boldsymbol{u}_1 = \lambda_1 \boldsymbol{u}_1 \quad \Longrightarrow \quad \lambda_1 = \frac{\boldsymbol{u}_1^\top A^\top A \boldsymbol{u}_1}{\boldsymbol{u}_1^\top \boldsymbol{u}_1} = \frac{\|A\boldsymbol{u}_1\|_2^2}{\|\boldsymbol{u}_1\|_2^2} \ge 0. \tag{5.26}$$

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

$$\|A\boldsymbol{x}\|_2^2 = \boldsymbol{x}^\top A^\top A \boldsymbol{x} = \boldsymbol{x}^\top \sum_{k=1}^n \alpha_k \lambda_k \boldsymbol{u}_k = \sum_{j=1}^n \alpha_j \boldsymbol{u}_j^\top \sum_{k=1}^n \alpha_k \lambda_k \boldsymbol{u}_k = \sum_{k=1}^n \alpha_k^2 \lambda_k, \tag{5.27}$$

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

$$\|A\boldsymbol{x}\|_2^2 \le \lambda_n \sum_{k=1}^n \alpha_k^2. \tag{5.28}$$

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

5.7 Conditioning

Some linear systems are inherently more difficult to solve than others, because the solution is sensitive to small perturbations in the input.

Example → Consider the linear system

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

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

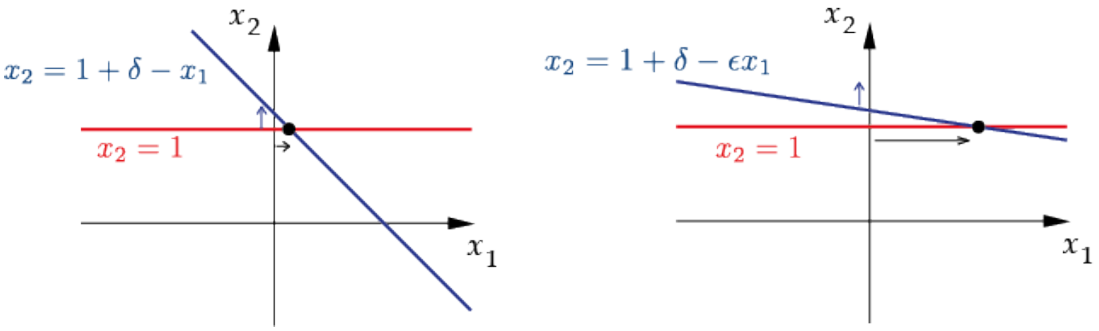
$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+\delta \\ 1 \end{pmatrix} \quad \Longrightarrow \quad \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \delta \\ 1 \end{pmatrix}.$$

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

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

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

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 $\boldsymbol{b}$, 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

$$\frac{|\text{relative error in } \boldsymbol{x}|}{|\text{relative error in } \boldsymbol{b}|} = \frac{\|\delta\boldsymbol{x}\|/\|\boldsymbol{x}\|}{\|\delta\boldsymbol{b}\|/\|\boldsymbol{b}\|} = \left(\frac{\|\delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|}\right)\left(\frac{\|\boldsymbol{b}\|}{\|\delta\boldsymbol{b}\|}\right) = \left(\frac{\|A^{-1}\delta\boldsymbol{b}\|}{\|\boldsymbol{x}\|}\right)\left(\frac{\|\boldsymbol{b}\|}{\|\delta\boldsymbol{b}\|}\right)$$
$$\leq \frac{\|A^{-1}\|\|\delta\boldsymbol{b}\|}{\|\boldsymbol{x}\|}\left(\frac{\|\boldsymbol{b}\|}{\|\delta\boldsymbol{b}\|}\right) = \frac{\|A^{-1}\|\|\boldsymbol{b}\|}{\|\boldsymbol{x}\|} = \frac{\|A^{-1}\|\|A\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \leq \|A^{-1}\|\|A\|. \quad (5.29)$$

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

$$\kappa_*(A) = \|A^{-1}\|_*\|A\|_*. \quad (5.30)$$

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

► This is a "worst case" amplification of the error by a given matrix. The actual result will depend on $\delta \boldsymbol{b}$ (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 → Return to our earlier examples and consider the condition numbers in the 1-norm. We have (assuming $0 < \epsilon \ll 1$) that

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

$$B = \begin{pmatrix} \epsilon & 1 \\ 0 & 1 \end{pmatrix} \implies B^{-1} = \frac{1}{\epsilon}\begin{pmatrix} 1 & -1 \\ 0 & \epsilon \end{pmatrix} \implies \|B\|_1 = 2, \quad \|B^{-1}\|_1 = \frac{1+\epsilon}{\epsilon} \implies \kappa_1(B) = \frac{2(1+\epsilon)}{\epsilon}.$$

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

Example → 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 $\boldsymbol{x}$ and using

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

5.8 Iterative methods

For large systems, the $O(n^3)$ 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 $O(n^3)$.

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

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

Example → Show that the following matrix is SPD:

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

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

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

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

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

$$f : \mathbb{R}^n \to \mathbb{R}, \qquad f(\boldsymbol{x}) = \tfrac{1}{2}\boldsymbol{x}^\top A\boldsymbol{x} - \boldsymbol{b}^\top \boldsymbol{x}. \tag{5.31}$$

When A is SPD, this functional behaves like a U-shaped parabola, and has a unique finite global minimizer $\boldsymbol{x}_*$ such that $f(\boldsymbol{x}_*) < f(\boldsymbol{x})$ for all $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{x} \neq \boldsymbol{x}_*$. To find $\boldsymbol{x}_*$, we need to set $\nabla f = \boldsymbol{0}$. We have

$$f(\boldsymbol{x}) = \tfrac{1}{2}\sum_{i=1}^{n} x_i \left(\sum_{j=1}^{n} a_{ij}x_j \right) - \sum_{j=1}^{n} b_j x_j \tag{5.32}$$

so

$$\frac{\partial f}{\partial x_k} = \tfrac{1}{2}\left(\sum_{i=1}^{n} x_i a_{ik} + \sum_{j=1}^{n} a_{kj}x_j \right) - b_k = \tfrac{1}{2}\left(\sum_{i=1}^{n} a_{ki}x_i + \sum_{j=1}^{n} a_{kj}x_j \right) - b_k = \sum_{j=1}^{n} a_{kj}x_j - b_k. \tag{5.33}$$

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

$$\nabla f = A\boldsymbol{x} - \boldsymbol{b}, \tag{5.34}$$

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

► 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* $\boldsymbol{d}_k$. The iteration takes the form

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k. \tag{5.35}$$

The *step size* α_k is chosen by minimizing $f(\boldsymbol{x})$ along the line $\boldsymbol{x} = \boldsymbol{x}_k + \alpha\boldsymbol{d}_k$. For our functional (5.31), we have

$$f(\boldsymbol{x}_k + \alpha\boldsymbol{d}_k) = \left(\tfrac{1}{2}\boldsymbol{d}_k^\top A\boldsymbol{d}_k\right)\alpha^2 + \left(\tfrac{1}{2}\boldsymbol{d}_k^\top A\boldsymbol{x}_k + \tfrac{1}{2}\boldsymbol{x}_k^\top A\boldsymbol{d}_k - \boldsymbol{b}^\top\boldsymbol{d}_k\right)\alpha + \tfrac{1}{2}\boldsymbol{x}_k^\top A\boldsymbol{x}_k - \boldsymbol{b}^\top\boldsymbol{x}_k. \tag{5.36}$$

Since A is symmetric, we have $\boldsymbol{x}_k^\top A\boldsymbol{d}_k = \boldsymbol{x}_k^\top A^\top \boldsymbol{d}_k = \left(A\boldsymbol{x}_k\right)^\top \boldsymbol{d}_k = \boldsymbol{d}_k^\top A\boldsymbol{x}_k$ and $\boldsymbol{b}^\top\boldsymbol{d}_k = \boldsymbol{d}_k^\top\boldsymbol{b}$, so we can simplify to

$$f(\boldsymbol{x}_k + \alpha\boldsymbol{d}_k) = \left(\tfrac{1}{2}\boldsymbol{d}_k^\top A\boldsymbol{d}_k\right)\alpha^2 + \boldsymbol{d}_k^\top\left(A\boldsymbol{x}_k - \boldsymbol{b}\right)\alpha + \tfrac{1}{2}\boldsymbol{x}_k^\top A\boldsymbol{x}_k - \boldsymbol{b}^\top\boldsymbol{x}_k. \tag{5.37}$$

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

$$\frac{\partial f}{\partial \alpha} = \boldsymbol{d}_k^\top A\boldsymbol{d}_k\alpha + \boldsymbol{d}_k^\top\left(A\boldsymbol{x}_k - \boldsymbol{b}\right) = 0. \tag{5.38}$$

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

$$\alpha_k = -\frac{\boldsymbol{d}_k^\top \boldsymbol{r}_k}{\boldsymbol{d}_k^\top A \boldsymbol{d}_k}. \tag{5.39}$$

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

$$\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k) = -\boldsymbol{r}_k, \tag{5.40}$$

where we have remembered (5.34).

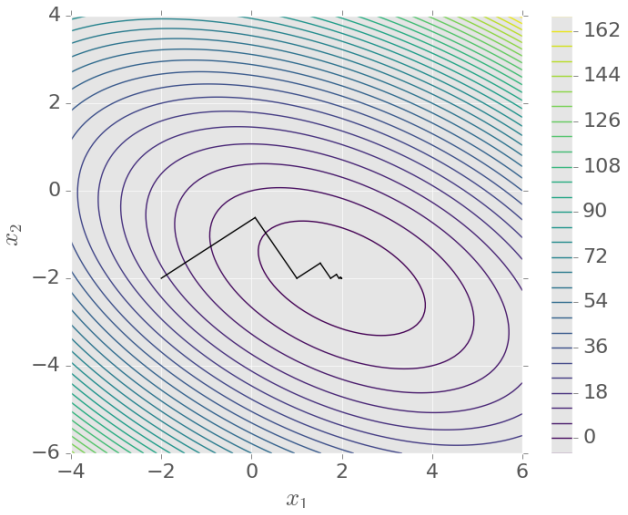
Example → Use the method of steepest descent to solve the system

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

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

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

Continuing the iteration, $\boldsymbol{x}_k$ 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 $\boldsymbol{d}_0 = -\boldsymbol{r}_0$, but subsequent search directions $\boldsymbol{d}_k$ are chosen to be *A-conjugate*, meaning that

$$\boldsymbol{d}_{k+1}^\top A \boldsymbol{d}_k = 0. \tag{5.41}$$

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

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

$$\boldsymbol{d}_{k+1} = -\boldsymbol{r}_{k+1} + \beta_k \boldsymbol{d}_k, \tag{5.42}$$

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

$$0 = \Big(-\boldsymbol{r}_{k+1} + \beta_k \boldsymbol{d}_k \Big)^\top A\boldsymbol{d}_k = -\boldsymbol{r}_{k+1}^\top A\boldsymbol{d}_k + \beta_k \boldsymbol{d}_k^\top A\boldsymbol{d}_k \tag{5.43}$$

and hence

$$\beta_k = \frac{\boldsymbol{r}_{k+1}^\top A\boldsymbol{d}_k}{\boldsymbol{d}_k^\top A\boldsymbol{d}_k}. \tag{5.44}$$

Thus we get the basic conjugate gradient algorithm.

Algorithm 5.6 (Conjugate gradient method)**.** *Start with an initial guess $\boldsymbol{x}_0$ and initial search direction $\boldsymbol{d}_0 = -\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$. For each $k = 0, 1, \ldots$, do the following:*

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

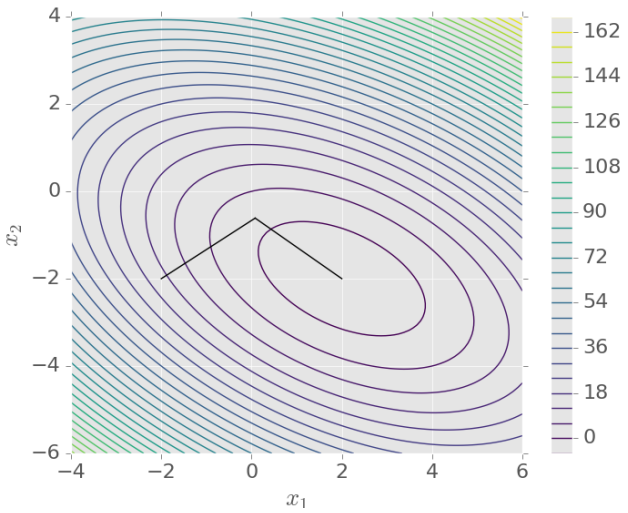
Example → Solve our previous example with the conjugate gradient method.
Starting with $\boldsymbol{x}_0 = (-2, -2)^\top$, the first step is the same as in steepest descent, giving $\boldsymbol{x}_1 = (0.08, -0.613)^\top$. But then we take

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

The second iteration then gives

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

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

Proof. See problem sheet. □

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

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

With these formulae, each iteration requires only one matrix-vector product, two vector-vector products, and three vector additions. Compare this to Algorithm 5.6 which requires two matrix-vector products, four vector-vector products and three vector additions.

6 Least-squares approximation

How do we find approximate solutions to overdetermined systems?

If A is an $m \times n$ rectangular matrix with $m > n$, then the linear system $A\boldsymbol{x} = \boldsymbol{b}$ is *overdetermined* and will usually have no solution. But we can still look for an approximate solution.

6.1 Orthogonality

Recall that the *inner product* between two column vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ is defined as

$$\boldsymbol{x} \cdot \boldsymbol{y} = \boldsymbol{x}^\top \boldsymbol{y} = \sum_{k=1}^{n} x_k y_k. \tag{6.1}$$

This is related to the ℓ_2-norm since $\|\boldsymbol{x}\|_2 = \sqrt{\boldsymbol{x}^\top \boldsymbol{x}}$. The angle θ between $\boldsymbol{x}$ and $\boldsymbol{y}$ is given by $\boldsymbol{x}^\top \boldsymbol{y} = \|\boldsymbol{x}\|_2 \|\boldsymbol{y}\|_2 \cos\theta$.

Two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are *orthogonal* if $\boldsymbol{x}^\top \boldsymbol{y} = 0$ (i.e. they lie at right angles in $\mathbb{R}^n$).

Let $S = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ be a set of n vectors. Then S is called *orthogonal* if $\boldsymbol{x}_i^\top \boldsymbol{x}_j = 0$ for all $i, j \in \{1, 2, \ldots, n\}$ with $i \neq j$.

Theorem 6.1. *An orthogonal set S of n vectors in $\mathbb{R}^n$ is a basis for $\mathbb{R}^n$.*

Proof. We know that a set of n vectors is a basis for $\mathbb{R}^n$ if the vectors are linearly independent. If this is not the case, then some $\boldsymbol{x}_k \in S$ could be expressed as a linear combination of the other members,

$$\boldsymbol{x}_k = \sum_{\substack{i=1 \\ i \neq k}}^{n} c_i \boldsymbol{x}_i. \tag{6.2}$$

Since $\boldsymbol{x}_k \neq 0$, we know that $\boldsymbol{x}_k^\top \boldsymbol{x}_k = \|\boldsymbol{x}_k\|_2^2 > 0$. But we would have

$$\boldsymbol{x}_k^\top \boldsymbol{x}_k = \sum_{\substack{i=1 \\ i \neq k}}^{n} c_i \boldsymbol{x}_k^\top \boldsymbol{x}_i = 0, \tag{6.3}$$

where we used orthogonality. This would be a contradiction, so we conclude that the vectors in an orthogonal set are linearly independent. □

► Many of the best algorithms for numerical analysis are based on the idea of orthogonality. We will see some examples this term.

An *orthonormal* set is an orthogonal set where all of the vectors have unit norm. Given an orthogonal set $S = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$, we can always construct an orthonormal set $S' = \{\boldsymbol{x}_1', \boldsymbol{x}_2', \ldots, \boldsymbol{x}_n'\}$ by normalisation, meaning

$$\boldsymbol{x}_i' = \frac{1}{\|\boldsymbol{x}_i\|} \boldsymbol{x}_i. \tag{6.4}$$

Theorem 6.2. *Let Q be an $m \times n$ matrix. The columns of Q form an orthonormal set iff $Q^\top Q = I_n$.*

If $m = n$, then such a Q is called an *orthogonal matrix*. For $m \neq n$, it is just called a *matrix with orthonormal columns*.

Proof. Let $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n$ be the columns of Q. Then

$$Q^\top Q = \begin{pmatrix} \boldsymbol{q}_1^\top \\ \boldsymbol{q}_2^\top \\ \vdots \\ \boldsymbol{q}_n^\top \end{pmatrix} \begin{pmatrix} \boldsymbol{q}_1 & \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_n \end{pmatrix} = \begin{pmatrix} \boldsymbol{q}_1^\top \boldsymbol{q}_1 & \boldsymbol{q}_1^\top \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_1^\top \boldsymbol{q}_n \\ \boldsymbol{q}_2^\top \boldsymbol{q}_1 & \boldsymbol{q}_2^\top \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_2^\top \boldsymbol{q}_n \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{q}_n^\top \boldsymbol{q}_1 & \boldsymbol{q}_n^\top \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_n^\top \boldsymbol{q}_n \end{pmatrix}. \tag{6.5}$$

So orthonormality $\boldsymbol{q}_i^\top \boldsymbol{q}_j = \delta_{ij}$ is equivalent to $Q^\top Q = I_n$, where I_n is the $n \times n$ identity matrix. □

► Note that the columns of Q are a basis for $\text{range}(Q) = \{Q\boldsymbol{x} : \boldsymbol{x} \in \mathbb{R}^n\}$.

Example → The set $S = \left\{\frac{1}{\sqrt{5}}(2,1)^\top, \frac{1}{\sqrt{5}}(1,-2)^\top\right\}$.
The two vectors in S are orthonormal, since

$$\frac{1}{\sqrt{5}}\begin{pmatrix} 2 & 1 \end{pmatrix}\frac{1}{\sqrt{5}}\begin{pmatrix} 2 \\ 1 \end{pmatrix} = 1, \quad \frac{1}{\sqrt{5}}\begin{pmatrix} 1 & -2 \end{pmatrix}\frac{1}{\sqrt{5}}\begin{pmatrix} 1 \\ -2 \end{pmatrix} = 1, \quad \frac{1}{\sqrt{5}}\begin{pmatrix} 2 & 1 \end{pmatrix}\frac{1}{\sqrt{5}}\begin{pmatrix} 1 \\ -2 \end{pmatrix} = 0.$$

Therefore S forms a basis for $\mathbb{R}^2$. If $\boldsymbol{x}$ is a vector with components x_1, x_2 in the standard basis $\{(1,0)^\top, (0,1)^\top\}$, then the components of $\boldsymbol{x}$ in the basis given by S are

$$\begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} & -\frac{2}{\sqrt{5}} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

► Inner products are preserved under multiplication by orthogonal matrices, since $(Q\boldsymbol{x})^\top Q\boldsymbol{y} = \boldsymbol{x}^\top (Q^\top Q)\boldsymbol{y} = \boldsymbol{x}^\top \boldsymbol{y}$. This means that angles between vectors and the lengths of vectors are preserved. Multiplication by an orthogonal matrix corresponds to a rigid rotation (if $\det(Q) = 1$) or a reflection (if $\det(Q) = -1$).

6.2 Discrete least squares

The *discrete least squares* problem: find $\boldsymbol{x}$ that minimizes the ℓ_2-norm of the residual, $\|A\boldsymbol{x} - \boldsymbol{b}\|_2$.

Example → Polynomial data fitting.
An overdetermined system arises if we try to fit a polynomial

$$p_n(x) = c_0 + c_1 x + \ldots + c_n x^n$$

to a function $f(x)$ at $m + 1 > n + 1$ nodes $x_0, \ldots, x_m$. In the natural basis, this leads to a rectangular system

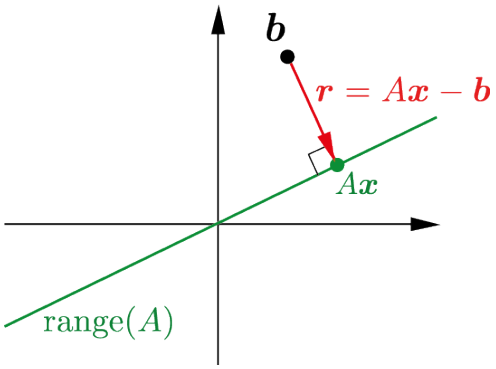
$$\begin{pmatrix} 1 & x_0 & \cdots & x_0^n \\ 1 & x_1 & \cdots & x_1^n \\ 1 & x_2 & \cdots & x_2^n \\ \vdots & \vdots & & \vdots \\ 1 & x_m & \cdots & x_m^n \end{pmatrix}\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_m) \end{pmatrix}.$$

We can't find coefficients c_k to match the function at all $m+1$ points. Instead, the least squares approach is to find coefficients that minimize

$$\sum_{i=1}^{m} \left| p(x_i) - f(x_i) \right|^2.$$

We will see how to do this shortly.

To solve the problem it is useful to think geometrically. The range of A, written range(A), is the set of all possible vectors $A\boldsymbol{x} \in \mathbb{R}^m$, where $\boldsymbol{x} \in \mathbb{R}^n$. This will only be a subspace of $\mathbb{R}^m$, and in particular it will not, in general, contain $\boldsymbol{b}$. We are therefore looking for $\boldsymbol{x} \in \mathbb{R}^n$ such that $A\boldsymbol{x}$ is as close as possible to $\boldsymbol{b}$ in $\mathbb{R}^m$ (as measured by the ℓ_2-norm/Euclidean distance).



The distance from $A\boldsymbol{x}$ to $\boldsymbol{b}$ is given by $\|\boldsymbol{r}\|_2 = \|A\boldsymbol{x} - \boldsymbol{b}\|_2$. Geometrically, we see that $\|\boldsymbol{r}\|_2$ will be minimized by choosing $\boldsymbol{r}$ orthogonal to $A\boldsymbol{x}$, i.e.,

$$(A\boldsymbol{x})^\top (A\boldsymbol{x} - \boldsymbol{b}) = 0 \quad \Longleftrightarrow \quad \boldsymbol{x}^\top (A^\top A\boldsymbol{x} - A^\top \boldsymbol{b}) = 0. \tag{6.6}$$

This will be satisfied if $\boldsymbol{x}$ satisfies the $n \times n$ linear system

$$A^\top A\boldsymbol{x} = A^\top \boldsymbol{b}, \tag{6.7}$$

called the *normal equations*.

Theorem 6.3. *The matrix $A^\top A$ is invertible iff the columns of A are linearly independent, in which case $A\boldsymbol{x} = \boldsymbol{b}$ has a unique least-squares solution $\boldsymbol{x} = (A^\top A)^{-1} A^\top \boldsymbol{b}$.*

Proof. If $A^\top A$ is singular (non-invertible), then $A^\top A\boldsymbol{x} = \boldsymbol{0}$ for some non-zero vector $\boldsymbol{x}$, implying that

$$\boldsymbol{x}^\top A^\top A\boldsymbol{x} = 0 \quad \Longrightarrow \|A\boldsymbol{x}\|_2^2 = 0 \quad \Longrightarrow A\boldsymbol{x} = \boldsymbol{0}. \tag{6.8}$$

This implies that A is rank-deficient (i.e. its columns are linearly dependent).

Conversely, if A is rank-deficient, then $A\boldsymbol{x} = \boldsymbol{0}$ for some $\boldsymbol{x} \neq 0$, implying $A^\top A\boldsymbol{x} = \boldsymbol{0}$ and hence that $A^\top A$ is singular. □

► The $n \times m$ matrix $(A^\top A)^{-1} A^\top$ is called the *Moore-Penrose pseudoinverse* of A. In practice, we would solve the normal equations (6.7) directly, rather than calculating the pseudoinverse itself.

Example → Polynomial data fitting.
For the matrix A in our previous example, we have $a_{ij} = x_i^j$ for $i = 0, \ldots, m$ and $j = 0, \ldots, n$.
So the normal matrix has entries

$$(A^\top A)_{ij} = \sum_{k=0}^{m} a_{ki} a_{kj} = \sum_{k=0}^{m} x_k^i x_k^j = \sum_{k=0}^{m} x_k^{i+j},$$

and the normal equations have the form

$$\sum_{j=0}^{n} c_j \sum_{k=0}^{m} x_k^{i+j} = \sum_{j=0}^{m} x_j^i f(x_j) \quad \text{for } i = 0, \ldots n.$$

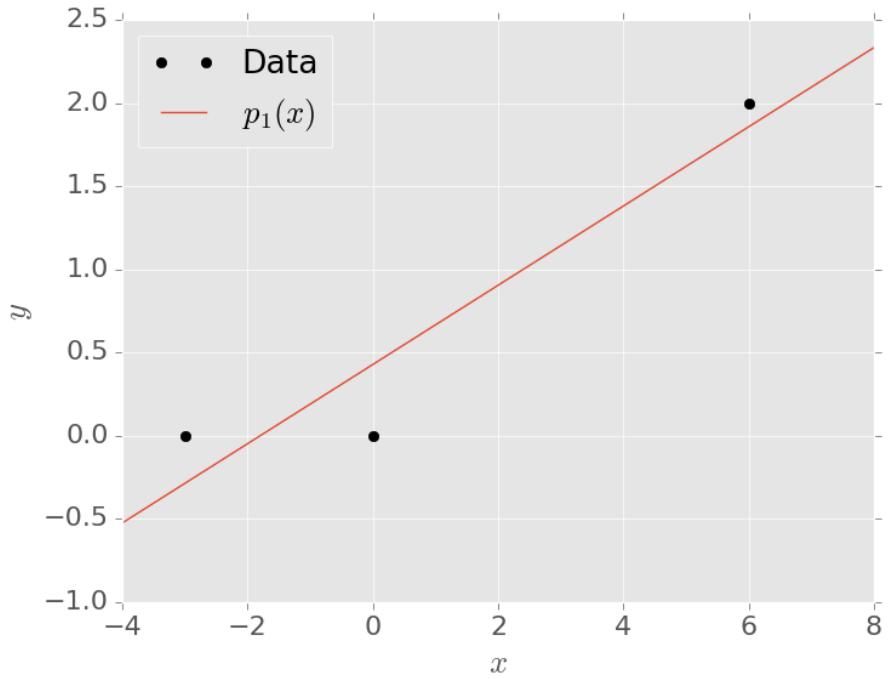
Example → Fit a least-squares straight line to the data $f(-3) = f(0) = 0$, $f(6) = 2$.
Here $n = 1$ (fitting a straight line) and $m = 2$ (3 data points), so $x_0 = -3$, $x_1 = 0$ and $x_2 = 6$.
The overdetermined system is

$$\begin{pmatrix} 1 & -3 \\ 1 & 0 \\ 1 & 6 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix},$$

and the normal equations have the form

$$\begin{pmatrix} 3 & x_0 + x_1 + x_2 \\ x_0 + x_1 + x_2 & x_0^2 + x_1^2 + x_2^2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} f(x_0) + f(x_1) + f(x_2) \\ x_0 f(x_0) + x_1 f(x_1) + x_2 f(x_2) \end{pmatrix}$$
$$\iff \quad \begin{pmatrix} 3 & 3 \\ 3 & 45 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 2 \\ 12 \end{pmatrix} \qquad \Longrightarrow \quad \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 3/7 \\ 5/21 \end{pmatrix}.$$

So the least-squares approximation by straight line is $p_1(x) = \frac{3}{7} + \frac{5}{21}x$.



6.3 QR decomposition

In practice the normal matrix $A^\top A$ can often be ill-conditioned. A better method is based on another matrix factorization.

Theorem 6.4 (QR decomposition). *Any real $m \times n$ matrix A, with $m \geq n$, can be written in the form $A = QR$, where Q is an $m \times n$ matrix with orthonormal columns and R is an upper-triangular $n \times n$ matrix.*

Proof. We will show this by construction... □

The simplest way to compute Q and R is by *Gram-Schmidt orthogonalization.*

Algorithm 6.5 (Gram-Schmidt). *Let $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n\}$ be a set of n vectors in $\mathbb{R}^m$, not necessarily orthogonal. Then we can construct an orthonormal set $\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n\}$ by*

$$\boldsymbol{p}_k = \boldsymbol{u}_k - \sum_{i=1}^{k-1} (\boldsymbol{u}_k^\top \boldsymbol{q}_i)\boldsymbol{q}_i, \qquad \boldsymbol{q}_k = \frac{\boldsymbol{p}_k}{\|\boldsymbol{p}_k\|_2} \qquad \text{for } k = 1, \ldots, n.$$

Notice that each $\boldsymbol{p}_k$ is constructed from $\boldsymbol{u}_k$ by subtracting the orthogonal projections of $\boldsymbol{u}_k$ on each of the previous $\boldsymbol{q}_i$ for $i < k$.

Example → Use the Gram-Schmidt process to construct an orthonormal basis for $W = \text{Span}\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$, where $\boldsymbol{u}_1 = (3, 6, 0)^\top$ and $\boldsymbol{u}_2 = (1, 2, 2)^\top$.
We take

$$\boldsymbol{p}_1 = \boldsymbol{u}_1 = \begin{pmatrix} 3 \\ 6 \\ 0 \end{pmatrix} \quad \Longrightarrow \quad \boldsymbol{q}_1 = \frac{\boldsymbol{p}_1}{\|\boldsymbol{p}_1\|_2} = \frac{1}{\sqrt{45}} \begin{pmatrix} 3 \\ 6 \\ 0 \end{pmatrix}.$$

Then

$$\boldsymbol{p}_2 = \boldsymbol{u}_2 - (\boldsymbol{u}_2^\top \boldsymbol{q}_1)\boldsymbol{q}_1 = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} - \frac{15}{45} \begin{pmatrix} 3 \\ 6 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix} \quad \Longrightarrow \quad \boldsymbol{q}_2 = \frac{\boldsymbol{p}_2}{\|\boldsymbol{p}_2\|_2} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

The set $\{\boldsymbol{q}_1, \boldsymbol{q}_2\}$ is orthonormal and spans W.

How do we use this to construct our QR decomposition of A? We simply apply Gram-Schmidt to the set of columns of A, $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$, which are vectors in $\mathbb{R}^m$. This produces a set of orthogonal vectors $\boldsymbol{q}_i \in \mathbb{R}^m$. Moreover, we have

$$\|\boldsymbol{p}_k\|_2 \boldsymbol{q}_k = \boldsymbol{a}_k - \sum_{i=1}^{k-1} (\boldsymbol{a}_k^\top \boldsymbol{q}_i)\, \boldsymbol{q}_i \quad \Longrightarrow \quad \boldsymbol{a}_k = \|\boldsymbol{p}_k\|_2 \boldsymbol{q}_k + \sum_{i=1}^{k-1} (\boldsymbol{a}_k^\top \boldsymbol{q}_i)\boldsymbol{q}_i. \tag{6.9}$$

Taking the inner product with $\boldsymbol{q}_k$ and using orthonormality of the $\boldsymbol{q}_i$ shows that $\|\boldsymbol{p}_k\|_2 = \boldsymbol{a}_k^\top \boldsymbol{q}_k$, so we can write the columns of A as

$$\boldsymbol{a}_k = \sum_{i=1}^{k} (\boldsymbol{a}_k^\top \boldsymbol{q}_i)\boldsymbol{q}_i. \tag{6.10}$$

In matrix form this may be written

$$A = \underbrace{\begin{pmatrix} \boldsymbol{q}_1 & \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_n \end{pmatrix}}_{Q\ (m\times n)} \underbrace{\begin{pmatrix} \boldsymbol{a}_1^\top \boldsymbol{q}_1 & \boldsymbol{a}_2^\top \boldsymbol{q}_1 & \cdots & \boldsymbol{a}_n^\top \boldsymbol{q}_1 \\ 0 & \boldsymbol{a}_2^\top \boldsymbol{q}_2 & \cdots & \boldsymbol{a}_n^\top \boldsymbol{q}_2 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \boldsymbol{a}_n^\top \boldsymbol{q}_n \end{pmatrix}}_{R\ (n\times n)} \tag{6.11}$$

► What we have done here is express each column of A in the orthonormal basis given by the columns of Q. The coefficients in the new basis are stored in R, which will be non-singular if A has full column rank.

How does this help in least squares? If $A = QR$ then

$$\begin{aligned} A^\top A\boldsymbol{x} = A^\top \boldsymbol{b} \quad &\iff \quad (QR)^\top QR\boldsymbol{x} = (QR)^\top \boldsymbol{b} && (6.12)\\ &\iff \quad R^\top (Q^\top Q) R\boldsymbol{x} = R^\top Q^\top \boldsymbol{b} && (6.13)\\ &\iff \quad R^\top \left(R\boldsymbol{x} - Q^\top \boldsymbol{b}\right) = \boldsymbol{0} && (6.14)\\ &\iff \quad R\boldsymbol{x} = Q^\top \boldsymbol{b}. && (6.15) \end{aligned}$$

In the last step we assumed that R is invertible. We see that the problem is reduced to an upper-triangular system, which may be solved by back-substitution.

Example → Use QR decomposition to find our earlier least-squares straight line, where

$$\begin{pmatrix} 1 & -3 \\ 1 & 0 \\ 1 & 6 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix}.$$

The columns of A are $\boldsymbol{a}_1 = (1,1,1)^\top$ and $\boldsymbol{a}_2 = (-3,0,6)^\top$. So applying Gram-Schmidt gives

$$\boldsymbol{p}_1 = \boldsymbol{a}_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad \Longrightarrow \boldsymbol{q}_1 = \frac{\boldsymbol{p}_1}{\|\boldsymbol{p}_1\|_2} = \frac{1}{\sqrt{3}} \begin{pmatrix} 1 \\ 1 \\ 1, \end{pmatrix}$$

and

$$\boldsymbol{p}_2 = \boldsymbol{a}_2 - (\boldsymbol{a}_2^\top \boldsymbol{q}_1)\boldsymbol{q}_1 = \begin{pmatrix} -3 \\ 0 \\ 6 \end{pmatrix} - \sqrt{3}\frac{1}{\sqrt{3}} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -4 \\ -1 \\ 5 \end{pmatrix} \quad \Longrightarrow \boldsymbol{q}_2 = \frac{\boldsymbol{p}_2}{\|\boldsymbol{p}_2\|_2} = \frac{1}{\sqrt{42}} \begin{pmatrix} -4 \\ -1 \\ 5 \end{pmatrix}.$$

Therefore $A = QR$ with

$$Q = \begin{pmatrix} 1/\sqrt{3} & -4/\sqrt{42} \\ 1/\sqrt{3} & -1/\sqrt{42} \\ 1/\sqrt{3} & 5/\sqrt{42} \end{pmatrix}, \quad R = \begin{pmatrix} \boldsymbol{a}_1^\top \boldsymbol{q}_1 & \boldsymbol{a}_2^\top \boldsymbol{q}_1 \\ 0 & \boldsymbol{a}_2^\top \boldsymbol{q}_2 \end{pmatrix} = \begin{pmatrix} \sqrt{3} & \sqrt{3} \\ 0 & \sqrt{42} \end{pmatrix}.$$

The normal equations may then be written

$$R\boldsymbol{x} = Q^\top \boldsymbol{b} \quad \Longrightarrow \begin{pmatrix} \sqrt{3} & \sqrt{3} \\ 0 & \sqrt{42} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 2/\sqrt{3} \\ 10/\sqrt{42} \end{pmatrix} \quad \Longrightarrow \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 3/7 \\ 5/21 \end{pmatrix},$$

which agrees with our earlier solution.

► In practice, Gram-Schmidt orthogonalization is not very numerically stable. Alternative methods of computing the QR decomposition such as the Householder algorithm are preferred (but beyond the scope of this course).

6.4 Continuous least squares

We have seen how to approximate a function $f(x)$ by a polynomial $p_n(x)$, by minimising the sum of squares of errors at $m > n$ nodes. In this section, we will see how to find an approximating polynomial that minimizes the error over all $x \in [a, b]$.

► Think of taking $m \to \infty$, so that our matrix A is infinitely tall. Happily, since p_n still has finitely many coefficients, we will end up with an $n \times n$ system of normal equations to solve.

Let f, g belong to the vector space $C[a, b]$ of continuous real-valued functions on $[a, b]$. We can define an *inner product* by

$$(f, g) = \int_a^b f(x)g(x)w(x)\,\mathrm{d}x \tag{6.16}$$

for any choice of *weight function* $w(x)$ that is positive, continuous, and integrable on (a, b).

► The purpose of the weight function will be to assign varying degrees of importance to errors on different portions of the interval.

Since (6.16) is an inner product, we can define a norm satisfying (N1), (N2), (N3) by

$$\|f\| := \sqrt{(f, f)} = \left(\int_a^b |f(x)|^2 w(x)\,\mathrm{d}x\right)^{1/2}. \tag{6.17}$$

Example → Inner product.
For the weight function $w(x) = 1$ on the interval $[0, 1]$, we have, for example,

$$(1, x) = \int_0^1 x\,\mathrm{d}x = \tfrac{1}{2}, \qquad \|x\| = \sqrt{(x, x)} = \left(\int_0^1 x^2\,\mathrm{d}x\right)^{1/2} = \tfrac{1}{\sqrt{3}}.$$

The *continuous least squares* problem is to find the polynomial $p_n \in \mathcal{P}_n$ that minimizes $\|p_n - f\|$, in a given inner product. The analogue of the normal equations is the following.

Theorem 6.6 (Continuous least squares). *Given $f \in C[a, b]$, the polynomial $p_n \in \mathcal{P}_n$ minimizes $\|q_n - f\|$ among all $q_n \in \mathcal{P}_n$ if and only if*

$$(p_n - f,\ q_n) = 0 \quad \textit{for all } q_n \in \mathcal{P}_n.$$

► Notice the analogy with discrete least squares: we are again setting the "error" orthogonal to the space of "possible functions" $\mathcal{P}_n$.

Proof. If $(p_n - f,\, q_n) = 0$ for all $q_n \in \mathcal{P}_n$, then for any $q_n \neq p_n$ we have

$$\|q_n - f\|^2 = \|(q_n - p_n) + (p_n - f)\|^2 = \|q_n - p_n\|^2 + \|p_n - f\|^2 > \|p_n - f\|^2\,, \tag{6.18}$$

i.e., p_n minimizes $\|q_n - f\|$.

Conversely, suppose $(p_n - f, q_n) \neq 0$ for some $q_n \in \mathcal{P}_n$, and consider

$$\|(p_n - f) + \lambda q_n\|^2 = \|p_n - f\|^2 + 2\lambda\,(p_n - f, q_n) + \lambda^2\,\|q_n\|^2\,. \tag{6.19}$$

If we choose $\lambda = -(p_n - f, q_n)/\|q_n\|^2$ then we see that

$$\|(p_n + \lambda q_n) - f\|^2 = \|p_n - f\|^2 - \frac{(p_n - f, q_n)^2}{\|q_n\|^2} < \|p_n - f\|^2\,, \tag{6.20}$$

showing that p_n does not minimize $\|q_n - f\|$. □

► The theorem holds more generally for best approximations in any subspace of an inner product space. It is an important result in the subject of Approximation Theory.

Example → Find the least squares polynomial $p_1(x) = c_0 + c_1 x$ that approximates $f(x) = \sin(\pi x)$ on the interval $[0, 1]$ with weight function $w(x) = 1$.
We can use Theorem 6.6 to find c_0 and c_1 by requiring orthogonality for both functions in the basis $\{1, x\}$ for $\mathcal{P}_1$. This will guarantee that $p_1 - f$ is orthogonal to every polynomial in $\mathcal{P}_1$. We get a system of two linearly independent equations

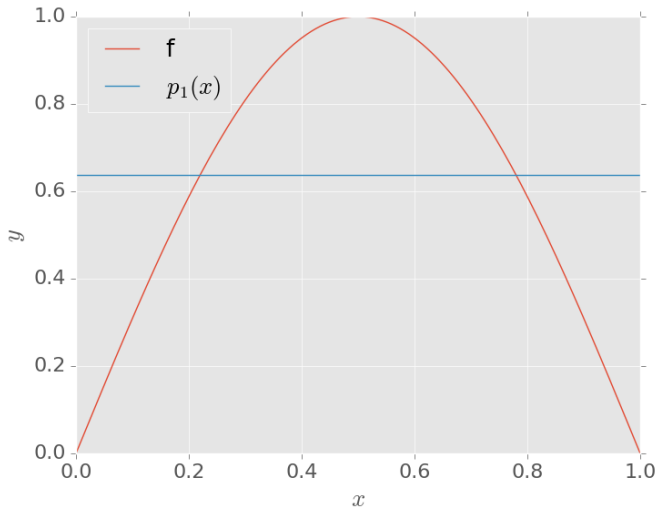
$$\begin{cases} (p_1 - f, 1) &= 0 \\ (p_1 - f, x) &= 0 \end{cases} \iff \begin{cases} (p_1, 1) = (f, 1) \\ (p_1, x) = (f, x) \end{cases} \iff \begin{cases} \int_0^1 (c_0 + c_1 x)\,\mathrm{d}x = \int_0^1 \sin(\pi x)\,\mathrm{d}x \\ \int_0^1 (c_0 + c_1 x)x\,\mathrm{d}x = \int_0^1 \sin(\pi x)x\,\mathrm{d}x \end{cases}$$

which may be written as the linear system

$$\begin{pmatrix} \int_0^1 \mathrm{d}x & \int_0^1 x\,\mathrm{d}x \\ \int_0^1 x\,\mathrm{d}x & \int_0^1 x^2\,\mathrm{d}x \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} \int_0^1 \sin(\pi x)\,\mathrm{d}x \\ \int_0^1 x \sin(\pi x)\,\mathrm{d}x \end{pmatrix}$$

These are analogous to the normal equations for the discrete polynomial approximation, with sums over x_k replaced by integrals. Evaluating the integrals, we have

$$\begin{pmatrix} 1 & 1/2 \\ 1/2 & 1/3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 2/\pi \\ 1/\pi \end{pmatrix} \implies \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 2/\pi \\ 0 \end{pmatrix} \implies p_1(x) = \tfrac{2}{\pi}.$$



6.5 Orthogonal polynomials

Just as with discrete least squares, we can make our life easier by working in an orthonormal basis for $\mathcal{P}_n$.

A family of *orthogonal polynomials* associated with the inner product (6.16) is a set $\{\phi_0, \phi_1, \phi_2, \ldots\}$ where each ϕ_k is a polynomial of degree exactly k and the polynomials satisfy the orthogonality condition

$$(\phi_j, \phi_k) = 0, \quad k \neq j. \tag{6.21}$$

▶ This condition implies that each ϕ_k is orthogonal to all polynomials of degree less than k.

The condition (6.21) determines the family uniquely up to normalisation, since multiplying each ϕ_k by a constant factor does not change their orthogonality. There are three common choices of normalisation:

1. Require each ϕ_k to be *monic* (leading coefficient 1).
2. Require *orthonormality*, $(\phi_j, \phi_k) = \delta_{jk}$.
3. Require $\phi_k(1) = 1$ for all k.

▶ The final one is the standard normalisation for Chebyshev and Legendre polynomials.

As in Theorem 6.1, a set of orthogonal polynomials $\{\phi_0, \phi_1, \ldots, \phi_n\}$ will form a basis for $\mathcal{P}_n$. Since this is a basis, the least-squares solution $p_n \in \mathcal{P}_n$ may be written

$$p_n(x) = c_0\phi_0(x) + c_1\phi_1(x) + \ldots + c_n\phi_n(x) \tag{6.22}$$

where $c_0, \ldots, c_n$ are the unknown coefficients to be found. Then according to Theorem 6.6 we can find these coefficients by requiring

$$(p_n - f, \phi_k) = 0 \quad \text{for } k = 0, \ldots, n, \tag{6.23}$$

$$\iff c_0(\phi_0, \phi_k) + c_1(\phi_1, \phi_k) + \ldots c_n(\phi_n, \phi_k) = (f, \phi_k) \quad \text{for } k = 0, \ldots, n, \tag{6.24}$$

$$\iff c_k = \frac{(f, \phi_k)}{(\phi_k, \phi_k)} \quad \text{for } k = 0, \ldots, n. \tag{6.25}$$

So compared to the natural basis, the number of integrals required is greatly reduced (at least once you have the orthogonal polynomials).

We can construct an orthogonal basis using the same Gram-Schmidt algorithm as in the discrete case. For simplicity, we will construct a set of monic orthogonal polynomials. Start with the monic polynomial of degree 0,

$$\phi_0(x) = 1. \tag{6.26}$$

Then construct $\phi_1(x)$ from x by subtracting the orthogonal projection of x on ϕ_0, giving

$$\phi_1(x) = x - \frac{(x\phi_0, \phi_0)}{(\phi_0, \phi_0)}\phi_0(x) = x - \frac{(x, 1)}{(1, 1)}. \tag{6.27}$$

In general, given the orthogonal set $\{\phi_0, \phi_1, \ldots, \phi_k\}$, we construct $\phi_{k+1}(x)$ by starting with $x\phi_k(x)$ and subtracting its orthogonal projections on $\phi_0, \phi_1, \ldots, \phi_k$. Thus

$$\phi_{k+1}(x) = x\phi_k(x) - \frac{(x\phi_k, \phi_0)}{(\phi_0, \phi_0)}\phi_0(x) - \frac{(x\phi_k, \phi_1)}{(\phi_1, \phi_1)}\phi_1(x) - \ldots - \frac{(x\phi_k, \phi_{k-1})}{(\phi_{k-1}, \phi_{k-1})}\phi_{k-1}(x) - \frac{(x\phi_k, \phi_k)}{(\phi_k, \phi_k)}\phi_k(x). \tag{6.28}$$

Now all of these projections except for the last two vanish, since, e.g., $(x\phi_k, \phi_0) = (\phi_k, x\phi_0) = 0$ using the fact that ϕ_k is orthogonal to all polynomials of lower degree. The penultimate term may be simplified similarly since $(x\phi_k, \phi_{k-1}) = (\phi_k, x\phi_{k-1}) = (\phi_k, \phi_k)$ So we get:

Theorem 6.7 (Three-term recurrence). *The set of monic orthogonal polynomials under the inner product* (6.16) *satisfy the recurrence relation*

$$\phi_0(x) = 1, \qquad \phi_1(x) = x - \frac{(x,1)}{(1,1)},$$
$$\phi_{k+1}(x) = x\phi_k(x) - \frac{(x\phi_k, \phi_k)}{(\phi_k, \phi_k)}\phi_k(x) - \frac{(\phi_k, \phi_k)}{(\phi_{k-1}, \phi_{k-1})}\phi_{k-1}(x) \quad \text{for } k \geq 1.$$

Example → Legendre polynomials.
These are generated by the inner product with $w(x) \equiv 1$ on the interval $(-1, 1)$. Starting with $\phi_0(x) = 1$, we find that

$$\phi_1(x) = x - \frac{\int_{-1}^1 x\,\mathrm{d}x}{\int_{-1}^1 \mathrm{d}x} = x, \qquad \phi_2(x) = x^2 - \frac{\int_{-1}^1 x^3\,\mathrm{d}x}{\int_{-1}^1 x^2\,\mathrm{d}x}x - \frac{\int_{-1}^1 x^2\,\mathrm{d}x}{\int_{-1}^1 \mathrm{d}x} = x^2 - \tfrac{1}{3}, \quad \dots$$

► Traditionally, the Legendre polynomials are then normalised so that $\phi_k(1) = 1$ for all k. In that case, the recurrence relation reduces to

$$\phi_{k+1}(x) = \frac{(2k+1)x\phi_k(x) - k\phi_{k-1}(x)}{k+1}.$$

Example → Use a basis of orthogonal polynomials to find the least squares polynomial $p_1 = c_0 + c_1 x$ that approximates $f(x) = \sin(\pi x)$ on the interval $[0, 1]$ with weight function $w(x) = 1$. Starting with $\phi_0(x) = 1$, we compute

$$\phi_1(x) = x - \frac{\int_0^1 x\,\mathrm{d}x}{\int_0^1 \mathrm{d}x} = x - \tfrac{1}{2}.$$

Then the coefficients are given by

$$c_0 = \frac{(f, \phi_0)}{(\phi_0, \phi_0)} = \frac{\int_0^1 \sin(\pi x)\,\mathrm{d}x}{\int_0^1 \mathrm{d}x} = \tfrac{2}{\pi},$$
$$c_1 = \frac{(f, \phi_1)}{(\phi_1, \phi_1)} = \frac{\int_0^1 (x - \frac{1}{2})\sin(\pi x)\,\mathrm{d}x}{\int_0^1 (x - \frac{1}{2})^2\,\mathrm{d}x} = 0,$$

so we recover our earlier approximation $p_1(x) = \frac{2}{\pi}$.

7 Numerical integration

How do we calculate integrals numerically?

The definite integral

$$I(f) := \int_a^b f(x)\,\mathrm{d}x \tag{7.1}$$

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

$$I_n(f) := \sum_{k=0}^{n} \sigma_k f(x_k), \tag{7.2}$$

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

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

Example → The trapezium rule

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

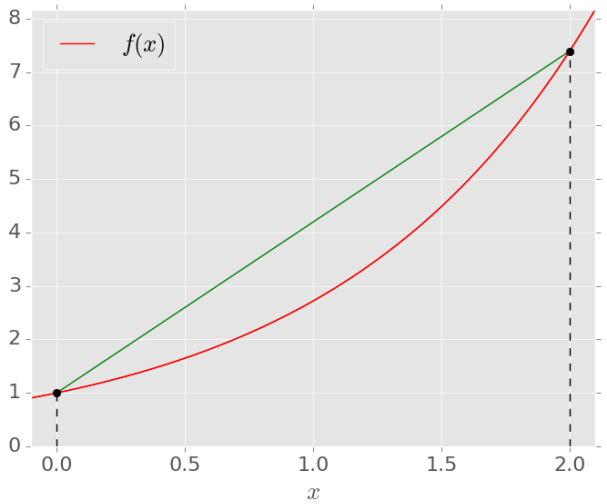
This is the quadrature formula (7.2) with $x_0 = a$, $x_1 = b$, $\sigma_0 = \sigma_1 = \frac{1}{2}(b-a)$.
For example, with $a = 0$, $b = 2$, $f(x) = \mathrm{e}^x$, we get

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

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:



7.1 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 Theorem 2.6.

Let $x_0, \dots, x_n \in [a, b]$, 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

$$p_n(x) = \sum_{k=0}^{n} f(x_k)\ell_k(x), \quad \text{where} \quad \ell_k(x) = \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x - x_j}{x_k - x_j}. \tag{7.3}$$

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

$$I_n(f) := \int_a^b \sum_{k=0}^{n} f(x_k)\ell_k(x)\,\mathrm{d}x = \sum_{k=0}^{n} f(x_k) \int_a^b \ell_k(x)\,\mathrm{d}x. \tag{7.4}$$

In other words,

$$I_n(f) := \sum_{k=0}^{n} \sigma_k f(x_k), \quad \text{where} \quad \sigma_k = \int_a^b \ell_k(x)\,\mathrm{d}x. \tag{7.5}$$

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*.

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

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

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

and

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

So

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

Theorem 7.1. *Let f be continuous on $[a, b]$ with $n+1$ continuous derivatives on (a, b). Then the Newton-Cotes formula (7.5) satisfies the error bound*

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

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

$$\left|I(f) - I_n(f)\right| = \left|\int_a^b f(x)\,\mathrm{d}x - \int_a^b p_n(x)\,\mathrm{d}x\right| = \left|\int_a^b \left[f(x) - p_n(x)\right]\,\mathrm{d}x\right| \tag{7.6}$$

$$\leq \int_a^b \left|f(x) - p_n(x)\right|\,\mathrm{d}x. \tag{7.7}$$

Now recall Theorem 2.6, which says that, for each $x \in [a, b]$, we can write

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n) \tag{7.8}$$

for some $\xi \in (a, b)$. The theorem simply follows by inserting this into inequality (7.7). □

Example → Trapezium rule.
Let $M_2 = \max_{\xi\in[a,b]} |f''(\xi)|$. Here Theorem 7.1 reduces to

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

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

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

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

► Theorem 7.1 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 (see later).

► 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.

7.2 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

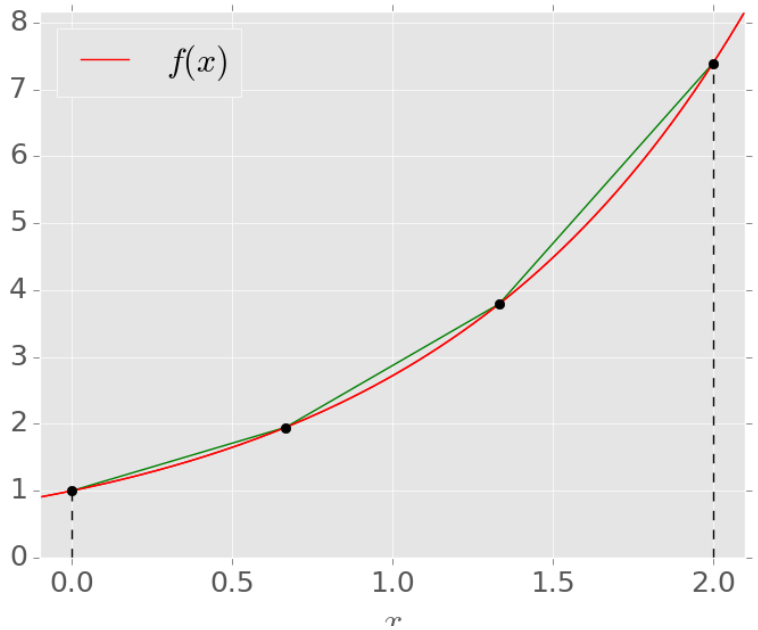
$$h := \frac{b-a}{m}, \tag{7.9}$$

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

Example → 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 $f(x) = e^x$ 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., $O(h^2)$.

To show this theoretically, we can apply Theorem 7.1 in each subinterval. In $[x_{i-1}, x_i]$ we have

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

Note that

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

So overall

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

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 $O(h^2)$.

7.3 Exactness

From Theorem 7.1, 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 → 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 → 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:

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

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 (Theorem 7.1) 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 (next section).

7.4 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 → 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:

$$\begin{aligned}
G_1(1) = I(1) \quad &\Longrightarrow \quad \sigma_0 + \sigma_1 = \int_{-1}^{1} \mathrm{d}x = 2, \\
G_1(x) = I(x) \quad &\Longrightarrow \quad \sigma_0 x_0 + \sigma_1 x_1 = \int_{-1}^{1} x \, \mathrm{d}x = 0, \\
G_1(x^2) = I(x^2) \quad &\Longrightarrow \quad \sigma_0 x_0^2 + \sigma_1 x_1^2 = \int_{-1}^{1} x^2 \, \mathrm{d}x = \tfrac{2}{3}, \\
G_1(x^3) = I(x^3) \quad &\Longrightarrow \quad \sigma_0 x_0^3 + \sigma_1 x_1^3 = \int_{-1}^{1} x^3 \, \mathrm{d}x = 0.
\end{aligned}$$

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.

Lemma 7.2. *If $\{\phi_0, \phi_1, \ldots, \phi_n\}$ is a set of orthogonal polynomials on $[a, b]$ under the inner product (6.16) 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

$$q_j(x) = (x - x_1)(x - x_2) \cdots (x - x_j). \tag{7.10}$$

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

$$(q_j, \phi_k) = 0 \quad \Longrightarrow \quad \int_a^b q_j(x)\phi_k(x)w(x) \, \mathrm{d}x = 0. \tag{7.11}$$

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

$$\int_a^b q_j(x)\phi_k(x)w(x)\,\mathrm{d}x \neq 0, \tag{7.12}$$

which is a contradiction. □

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

$$I_w(f) = \int_a^b f(x)w(x)\,\mathrm{d}x. \tag{7.13}$$

Theorem 7.3 (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 (7.13) *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 → 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 6, 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

$$\sigma_0 = \int_{-1}^{1} \ell_0(x)\,\mathrm{d}x = \int_{-1}^{1} \frac{x - \frac{1}{\sqrt{3}}}{-\frac{2}{\sqrt{3}}}\,\mathrm{d}x = -\frac{\sqrt{3}}{2}\left[\tfrac{1}{2}x^2 - \tfrac{1}{\sqrt{3}}x\right]_{-1}^{1} = 1,$$

$$\sigma_1 = \int_{-1}^{1} \ell_1(x)\,\mathrm{d}x = \int_{-1}^{1} \frac{x + \frac{1}{\sqrt{3}}}{\frac{2}{\sqrt{3}}}\,\mathrm{d}x = \frac{\sqrt{3}}{2}\left[\tfrac{1}{2}x^2 + \tfrac{1}{\sqrt{3}}x\right]_{-1}^{1} = 1,$$

as before.

► 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 → Gaussian quadrature for $\int_0^1 \cos(x)x^{-\frac{1}{2}}\,\mathrm{d}x$, with $n = 0$.
This is a Fresnel integral, with exact value $1.80905\ldots$ 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

$$\phi_1(x) = x - \frac{\int_0^1 x\,\mathrm{d}x}{\int_0^1 \mathrm{d}x} = x - \tfrac{1}{2} \qquad \Longrightarrow \qquad x_0 = \tfrac{1}{2}.$$

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 $w(x) = x^{-\frac{1}{2}}$.* This time we get

$$\phi_1(x) = x - \frac{\int_0^1 x^{\frac{1}{2}}\,\mathrm{d}x}{\int_0^1 x^{-\frac{1}{2}}\,\mathrm{d}x} = x - \frac{\frac{2}{3}}{2} \qquad \Longrightarrow \qquad x_0 = \tfrac{1}{3}.$$

The corresponding weight is $\sigma_0 = \int_0^1 x^{-\frac{1}{2}}\,\mathrm{d}x = 2$, so the new estimate is the more accurate

$$G_{0,w}\Big(\cos(x)\Big) = 2\cos\left(\tfrac{1}{3}\right) = 1.8899\ldots$$

Proof of Theorem 7.3. 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 Theorem 7.1, 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

$$p_{2n+1}(x) = \phi_{n+1}(x)q_n(x) + r_n(x) \quad \text{for some} \quad q_n, r_n \in \mathcal{P}_n. \tag{7.14}$$

Then

$$G_{n,w}(p_{2n+1}) = \sum_{k=0}^{n} \sigma_k p_{2n+1}(x_k) = \sum_{k=0}^{n} \sigma_k \Big[\phi_{n+1}(x_k)q_n(x_k) + r_n(x_k)\Big] = \sum_{k=0}^{n} \sigma_k r_n(x_k) = I_w(r_n), \tag{7.15}$$

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

$$I_w(\phi_{n+1}q_n) = \int_a^b \phi_{n+1}(x)q_n(x)w(x)\,\mathrm{d}x = 0 \tag{7.16}$$

and hence

$$G_{n,w}(p_{2n+1}) = I_w(r_n) + 0 = I_w(r_n) + I_w(\phi_{n+1}q_n) = I_w(\phi_{n+1}q_n + r_n) = I_w(p_{2n+1}). \tag{7.17}$$

□

▶ 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 7.4 (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

$$m\int_a^b g(x)\,\mathrm{d}x \leq \int_a^b f(x)g(x)\,\mathrm{d}x \leq M\int_a^b g(x)\,\mathrm{d}x. \tag{7.18}$$

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

$$m \leq \frac{1}{I}\int_a^b f(x)g(x)\,\mathrm{d}x \leq M. \tag{7.19}$$

By the Intermediate Value Theorem (Theorem 4.1), $f(x)$ attains every value between m and M somewhere in (a,b), so in particular there exists $\xi \in (a,b)$ with

$$f(\xi) = \frac{1}{I}\int_a^b f(x)g(x)\,\mathrm{d}x. \tag{7.20}$$

□

Theorem 7.5 (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 by Theorem 7.3. 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

$$p_{2n+1}(x_k) = f(x_k) \quad \text{and} \quad p'_{2n+1}(x_k) = f'(x_k) \quad \text{for } k = 0, \ldots, n. \tag{7.21}$$

In addition (see problem sheet), there exists $\lambda \in (a,b)$, depending on x, such that

$$f(x) - p_{2n+1}(x) = \frac{f^{(2n+2)}(\lambda)}{(2n+2)!}\prod_{i=0}^n (x - x_i)^2. \tag{7.22}$$

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

$$\int_a^b f(x)w(x)\,\mathrm{d}x - \int_a^b p_{2n+1}(x)w(x)\,\mathrm{d}x = \int_a^b \frac{f^{(2n+2)}(\lambda)}{(2n+2)!}\phi_{n+1}^2(x)w(x)\,\mathrm{d}x. \tag{7.23}$$

Now we know that $G_{n,w}$ must be exact for p_{2n+1}, so

$$\int_a^b p_{2n+1}(x)w(x)\,\mathrm{d}x = G_{n,w}(p_{2n+1}) = \sum_{k=0}^{n} \sigma_k p_{2n+1}(x_k) = \sum_{k=0}^{n} \sigma_k f(x_k) = G_{n,w}(f). \tag{7.24}$$

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

$$I_w(f) - G_{n,w}(f) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!}\int_a^b \phi_{n+1}^2(x)w(x)\;\mathrm{d}x \tag{7.25}$$

for some $\xi \in (a, b)$ that does not depend on x. □