MATH50003 Numerical Analysis

Sheehan Olver

March 8, 2025

Contents

Chapter I

Calculus on a Computer

In this first chapter we explore the basics of mathematical computing and numerical analysis. In particular we investigate the following mathematical problems which can not in general be solved exactly:

1. Integration. General integrals have no closed form expressions. Can we use a computer to approximate the values of definite integrals?

2. Differentiation. Differentiating a formula as in calculus is usually algorithmic, however, it is often needed to compute derivatives without access to an underlying formula, eg, a function defined only in code. Can we use a computer to approximate derivatives? A very important application is in Machine Learning, where there is a need to compute gradients to determine the "right" weights in a neural network.

3. Root finding. There is no general formula for finding roots (zeros) of arbitrary functions, or even polynomials that are of degree 5 (quintics) or higher. Can we compute roots of general functions using a computer?

In this chapter we discuss:

1. I.1 Rectangular rule: we review the rectangular rule for integration and deduce the *converge rate* of the approximation. In the lab/problem sheet we investigate its implementation as well as extensions to the Trapezium rule.

2. I.2 Divided differences: we investigate approximating derivatives by a divided difference and again deduce the convergence rates. In the lab/problem sheet we extend the approach to the central differences formula and computing second derivatives. We also observe a mystery: the approximations may have significant errors in practice, and there is a limit to the accuracy.

3. I.3 Dual numbers: we introduce the algebraic notion of a *dual number* which allows the implemention of *forward-mode automatic differentiation*, a high accuracy alternative to divided differences for computing derivatives.

4. I.4 Newton's method: Newton's method is a basic approach for computing roots/zeros of a function. We use dual numbers to implement this algorithm.

I.1 Rectangular rule

One possible definition for an integral is the limit of a Riemann sum, for example:

$$\int_a^b f(x)\mathrm{d}x = \lim_{n\to\infty} h\sum_{j=1}^{n} f(x_j)$$

where $x_j = a + jh$ are evenly spaced points dividing up the interval $[a, b]$, that is with the *step size* $h = (b-a)/n$. This suggests an algorithm known as the *(right-sided) rectangular rule* for approximating an integral: choose n large so that

$$\int_a^b f(x)\mathrm{d}x \approx h\sum_{j=1}^{n} f(x_j).$$

In the lab we explore practical implementation of this approximation, and observe that the error in approximation is bounded by C/n for some constant C. This can be expressed using "Big-O" notation:

$$\int_a^b f(x)\mathrm{d}x = h\sum_{j=1}^{n} f(x_j) + O(1/n).$$

In these notes we consider the "Analysis" part of "Numerical Analysis": we want to *prove* the convergence rate of the approximation, including finding an explicit expression for the constant C.

To tackle this question we consider the error incurred on a single panel (x_{j-1}, x_j), then sum up the errors on rectangles.

Now for a secret. There are only so many tools available in analysis (especially at this stage of your career), and one can make a safe bet that the right tool in any analysis proof is either (1) integration-by-parts, (2) geometric series or (3) Taylor series. In this case we use (1):

Lemma 1 ((Right-sided) Rectangular Rule error on one panel)**.** *Assuming f is differentiable on $[a, b]$ and its derivative is integrable we have*

$$\int_a^b f(x)\mathrm{d}x = (b-a)f(b) + \delta$$

where $|\delta| \le M(b-a)^2$ for $M = \sup_{a\le x\le b} |f'(x)|$.

Proof We write

$$\begin{aligned}\int_a^b f(x)\mathrm{d}x &= \int_a^b (x-a)'f(x)\mathrm{d}x = [(x-a)f(x)]_a^b - \int_a^b (x-a)f'(x)\mathrm{d}x \\ &= (b-a)f(b) + \underbrace{\left(-\int_a^b (x-a)f'(x)\mathrm{d}x\right)}_{\delta}.\end{aligned}$$

Recall that we can bound the absolute value of an integral by the supremum of the integrand times the width of the integration interval:

$$\left|\int_a^b g(x)\mathrm{d}x\right| \le (b-a)\sup_{a\le x\le b} |g(x)|.$$

The lemma thus follows since

$$\begin{aligned}\left|\int_a^b (x-a)f'(x)\mathrm{d}x\right| &\leq (b-a)\sup_{a\leq x\leq b}|(x-a)f'(x)|\\ &\leq (b-a)\sup_{a\leq x\leq b}|x-a|\sup_{a\leq x\leq b}|f'(x)|\\ &\leq M(b-a)^2.\end{aligned}$$

■

Now summing up the errors in each panel gives us the error of using the Rectangular rule:

Theorem 1 (Rectangular Rule error)**.** *Assuming f is differentiable on $[a,b]$ and its derivative is integrable we have*

$$\int_a^b f(x)\mathrm{d}x = h\sum_{j=1}^n f(x_j) + \delta$$

where $|\delta| \leq M(b-a)h$ for $M = \sup_{a\leq x\leq b}|f'(x)|$, $h=(b-a)/n$ and $x_j = a+jh$.

Proof We split the integral into a sum of smaller integrals:

$$\int_a^b f(x)\mathrm{d}x = \sum_{j=1}^n \int_{x_{j-1}}^{x_j} f(x)\mathrm{d}x = \sum_{j=1}^n [(x_j - x_{j-1})f(x_j) + \delta_j] = h\sum_{j=1}^n f(x_j) + \underbrace{\sum_{j=1}^n \delta_j}_{\delta}$$

where δ_j, the error on each panel as in the preceding lemma, satisfies

$$|\delta_j| \leq (x_j - x_{j-1})^2 \sup_{x_{j-1}\leq x\leq x_j}|f'(x)| \leq Mh^2.$$

Thus using the triangular inequality we have

$$|\delta| = \left|\sum_{j=1}^n \delta_j\right| \leq \sum_{j=1}^n |\delta_j| \leq Mnh^2 = M(b-a)h.$$

■

Note a consequence of this lemma is that the approximation converges as $n\to\infty$ (i.e. $h\to 0$). In the labs and problem sheets we will consider the left-sided rule:

$$\int_a^b f(x)\mathrm{d}x \approx h\sum_{j=0}^{n-1} f(x_j).$$

We also consider the *Trapezium rule*. Here we approximate an integral by an affine function:

$$\int_a^b f(x)\mathrm{d}x \approx \int_a^b \frac{(b-x)f(a)+(x-a)f(b)}{b-a}\mathrm{d}x = \frac{b-a}{2}\left[f(a)+f(b)\right].$$

Subdividing an interval $a = x_0 < x_1 < \ldots < x_n = b$ and applying this approximation separately on each subinterval $[x_{j-1}, x_j]$, where $h=(b-a)/n$ and $x_j = a+jh$, leads to the approximation

$$\int_a^b f(x)\mathrm{d}x \approx \frac{h}{2}f(a) + h\sum_{j=1}^{n-1} f(x_j) + \frac{h}{2}f(b)$$

We shall see both experimentally and provably that this approximation converges faster than the rectangular rule.

I.2 Divided Differences

Given a function, how can we approximate its derivative at a point? We consider an intuitive approach to this problem using *(Right-sided) Divided Differences*:

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

Note by the definition of the derivative we know that this approximation will converge to the true derivative as $h \to 0$. But in numerical approimxations we also need to consider the rate of convergence.

Now in the previous section I mentioned there are three basic tools in analysis: (1) integration-by-parts, (2) geometric series or (3) Taylor series. In this case we use (3):

Proposition 1 (divided differences error)**.** *Suppose that f is twice-differentiable on the interval $[x, x+h]$. The error in approximating the derivative using divided differences is*

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \delta$$

where $|\delta| \leq Mh/2$ for $M = \sup_{x \leq t \leq x+h} |f''(t)|$.

Proof Follows immediately from Taylor's theorem: recall that

$$f(x+h) = f(x) + f'(x)h + \frac{f''(t)}{2}h^2$$

for some $t \in [x, x+h]$. Rearranging we get

$$f'(x) = \frac{f(x+h) - f(x)}{2} + \underbrace{\left(-\frac{f''(t)}{2h^2}\right)}_{\delta}.$$

We then bound:

$$|\delta| \leq \left|\frac{f''(t)}{2}h\right| \leq \frac{Mh}{2}.$$

■

Unlike the rectangular rule, the computational cost of computing the divided difference is independent of h! We only need to evaluate a function f twice and do a single division. Here we are assuming that the computational cost of evaluating f is independent of the point of evaluation. Later we will investigate the details of how computers work with numbers via floating point, and confirm that this is a sensible assumption.

So why not just set h ridiculously small? In the lab we explore this question and observe that there are significant errors introduced in the numerical realisation of this algorithm. We will return to the question of understanding these errors after learning floating point numbers.

There are alternative versions of divided differences. Left-side divided differences evaluates to the left of the point where we wish to know the derivative:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

and central differences:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

We can further arrive at an approximation to the second derivative by composing a left- and right-sided finite difference:

$$f''(x) \approx \frac{f'(x+h) - f'(x)}{h} \approx \frac{\frac{f(x+h)-f(x)}{h} - \frac{f(x)-f(x-h)}{h}}{h} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

In the lab we investigate the convergence rate of these approximations (in particular, that central differences is more accurate than standard divided differences) and observe that they too suffer from unexplained (for now) loss of accuracy as $h \to 0$. In the problem sheet we prove the theoretical convergence rate, which is never realised because of these errors.

I.3 Dual Numbers

In this section we introduce a mathematically beautiful alternative to divided differences for computing derivatives: *dual numbers*. These are a commutative ring that *exactly* compute derivatives, which when implemented on a computer gives very high-accuracy approximations to derivatives. They underpin forward-mode automatic differentation. Automatic differentiation is a basic tool in Machine Learning for computing gradients necessary for training neural networks.

Definition 1 (Dual numbers)**.** Dual numbers $\mathbb{D}$ are a commutative ring (over $\mathbb{R}$) generated by 1 and ϵ such that $\epsilon^2 = 0$, that is,

$$\mathbb{D} := \{a + b\epsilon \quad : \quad a, b \in \mathbb{R}, \quad \epsilon^2 = 0\}.$$

This is very much analoguous to complex numbers, which are a field generated by 1 and i such that $\mathrm{i}^2 = -1$, that is,

$$\mathbb{C} := \{a + b\mathrm{i} \quad : \quad a, b \in \mathbb{R}, \quad \mathrm{i}^2 = -1\}.$$

Compare multiplication of each number type which falls out of the rules of the generators:

$$\begin{aligned}(a + b\mathrm{i})(c + d\mathrm{i}) &= ac + (bc + ad)\mathrm{i} + bd\mathrm{i}^2 = ac - bd + (bc + ad)\mathrm{i}, \\ (a + b\epsilon)(c + d\epsilon) &= ac + (bc + ad)\epsilon + bd\epsilon^2 = ac + (bc + ad)\epsilon.\end{aligned}$$

And just as we view $\mathbb{R} \subset \mathbb{C}$ by equating $a \in \mathbb{R}$ with $a + 0\mathrm{i} \in \mathbb{C}$, we can view $\mathbb{R} \subset \mathbb{D}$ by equating $a \in \mathbb{R}$ with $a + 0\epsilon \in \mathbb{D}$.

Conceptually, dual numbers can be thought of as introducing an infinitesimally small ϵ, where ϵ^2 is so small it is treated as zero. This is the intuitive reason they allow for differentiation of functions. But we do not need to appeal to this calculus-like interpretation, instead, their construction and relationship to differentiation can be accomplished using purely algebraic reasoning.

I.3.1 Differentiating polynomials

Polynomials evaluated on dual numbers are well-defined as they depend only on the operations $+$ and $*$. From the formula for multiplication of dual numbers we deduce that evaluating a polynomial at a dual number $a + b\epsilon$ tells us the derivative of the polynomial at a:

Theorem 2 (polynomials on dual numbers)**.** *Suppose p is a polynomial. Then*

$$p(a+b\epsilon) = p(a) + bp'(a)\epsilon$$

Proof

First consider $p(x) = x^n$ for $n \geq 0$. The cases $n = 0$ and $n = 1$ are immediate. For $n > 1$ we have by induction:

$$(a+b\epsilon)^n = (a+b\epsilon)(a+b\epsilon)^{n-1} = (a+b\epsilon)(a^{n-1} + (n-1)ba^{n-2}\epsilon) = a^n + bna^{n-1}\epsilon.$$

For a more general polynomial

$$p(x) = \sum_{k=0}^{n} c_k x^k$$

the result follows from linearity:

$$p(a+b\varepsilon) = \sum_{k=0}^{n} c_k(a+b\epsilon)^k = c_0 + \sum_{k=1}^{n} c_k(a^k + kba^{k-1}\epsilon) = \sum_{k=0}^{n} c_k a^k + b\sum_{k=1}^{n} c_k k a^{k-1}\epsilon = p(a) + bp'(a)\epsilon.$$

■

Example 1 (differentiating polynomial)**.** Consider computing $p'(2)$ where

$$p(x) = (x-1)(x-2) + x^2.$$

We can use dual numbers to differentiate, avoiding expanding in monomials or applying rules of differentiating:

$$p(2+\epsilon) = (1+\epsilon)\epsilon + (2+\epsilon)^2 = \epsilon + 4 + 4\epsilon = 4 + \underbrace{5}_{p'(2)}\epsilon.$$

I.3.2 Differentiating other functions

We can extend real-valued differentiable functions to dual numbers in a similar manner. First, consider a standard function with a Taylor series (e.g. cos, sin, exp, etc.)

$$f(x) = \sum_{k=0}^{\infty} f_k x^k$$

so that a is inside the radius of convergence. This leads naturally to a definition on dual numbers:

$$\begin{aligned} f(a+b\epsilon) &= \sum_{k=0}^{\infty} f_k(a+b\epsilon)^k = f_0 + \sum_{k=1}^{\infty} f_k(a^k + ka^{k-1}b\epsilon) = \sum_{k=0}^{\infty} f_k a^k + \sum_{k=1}^{\infty} f_k k a^{k-1} b\epsilon \\ &= f(a) + bf'(a)\epsilon. \end{aligned}$$

More generally, given a differentiable function (which may not have a Taylor series) we can extend it to dual numbers:

Definition 2 (dual extension)**.** Suppose a real-valued function $f : \Omega \to \mathbb{R}$ is differentiable in $\Omega \subset \mathbb{R}$. We can construct the *dual extension* $\underline{f} : \Omega + \epsilon\mathbb{R} \to \mathbb{D}$ by defining

$$\underline{f}(a+b\epsilon) := f(a) + bf'(a)\epsilon.$$

By viewing $\mathbb{R} \subset \mathbb{D}$, it is natural to reuse the notation f for the dual extension, hence when there's no chance of confusion we will identify $f(a+b\epsilon) \equiv \underline{f}(a+b\epsilon)$.

Thus, for basic functions we have natural extensions:

$$\begin{aligned}
\exp(a+b\epsilon) &:= \exp(a) + b\exp(a)\epsilon && (a,b\in\mathbb{R})\\
\sin(a+b\epsilon) &:= \sin(a) + b\cos(a)\epsilon && (a,b\in\mathbb{R})\\
\cos(a+b\epsilon) &:= \cos(a) - b\sin(a)\epsilon && (a,b\in\mathbb{R})\\
\log(a+b\epsilon) &:= \log(a) + \frac{b}{a}\epsilon && (a\in(0,\infty), b\in\mathbb{R})\\
\sqrt{a+b\epsilon} &:= \sqrt{a} + \frac{b}{2\sqrt{a}}\epsilon && (a\in(0,\infty), b\in\mathbb{R})\\
|a+b\epsilon| &:= |a| + b\,\mathrm{sign}a\,\epsilon && (a\in\mathbb{R}\backslash\{0\}, b\in\mathbb{R})
\end{aligned}$$

provided the function is differentiable at a. Note the last example does not have a convergent Taylor series (at 0) but we can still extend it where it is differentiable.

Going further, we can add, multiply, and compose such dual-extensions. And the beauty is these automatically satisfy the right properties to be dual-extensions themselves, thus allowing for differentiation of complicated functions built from basic differentiable building blocks.

The following lemma shows that addition and multiplication in some sense "commute" with the dual-extension, hence we can recover the product rule from dual number multiplication:

Lemma 2 (addition/multiplication)**.** *Suppose $f, g : \Omega \to \mathbb{R}$ are differentiable for $\Omega \subset \mathbb{R}$ and $c \in \mathbb{R}$. Then for $a \in \Omega$ and $b \in \mathbb{R}$ we have*

$$\begin{aligned}
\underline{f+g}(a+b\epsilon) &= \underline{f}(a+b\epsilon) + \underline{g}(a+b\epsilon)\\
\underline{cf}(a+b\epsilon) &= c\underline{f}(a+b\epsilon)\\
\underline{fg}(a+b\epsilon) &= \underline{f}(a+b\epsilon)\underline{g}(a+b\epsilon)
\end{aligned}$$

Proof The first two are immediate due to linearity:

$$\begin{aligned}
\underline{(f+g)}(a+b\epsilon) &= (f+g)(a) + b(f+g)'(a)\epsilon\\
&= (f(a)+bf'(a)\epsilon) + (g(a)+bg'(a)\epsilon) = \underline{f}(a+b\epsilon) + \underline{g}(a+b\epsilon),\\
\underline{cf}(a+b\epsilon) &= (cf)(a) + b(cf)'(a)\epsilon = c(f(a)+bf'(a)\epsilon) = c\underline{f}(a+b\epsilon).
\end{aligned}$$

The last property essentially captures the product rule of differentiation:

$$\begin{aligned}
\underline{fg}(a+b\epsilon) &= f(a)g(a) + b(f(a)g'(a) + f'(a)g'(a))\epsilon\\
&= (f(a)+bf'(a)\epsilon)(g(a)+bg'(a)\epsilon) = \underline{f}(a+b\epsilon)\underline{g}(a+b\epsilon).
\end{aligned}$$

■

Furthermore composition recovers the chain rule:

Lemma 3 (composition)**.** *Suppose $f : \Gamma \to \mathbb{R}$ and $g : \Omega \to \Gamma$ are differentiable in $\Omega, \Gamma \subset \mathbb{R}$. Then*

$$\underline{(f\circ g)}(a+b\epsilon) = \underline{f}(\underline{g}(a+b\epsilon))$$

Proof Again it falls out of the properties of dual numbers:

$$\underline{(f\circ g)}(a+b\epsilon) = f(g(a)) + bg'(a)f'(g(a))\epsilon = \underline{f}(g(a)+bg'(a)\epsilon) = \underline{f}(\underline{g}(a+b\epsilon))$$

■

A simple corollary is that any function defined in terms of addition, multiplication, composition, etc. of basic functions with dual-extensions will be differentiable via dual numbers. In this following example we see a practical realisation of this, where we differentiate a function by just evaluating it on dual numbers, implicitly, using the dual-extension for the basic build blocks:

Example 2 (differentiating non-polynomial)**.** Consider differentiating $f(x) = \exp(x^2 + \cos x)$ at the point $a = 1$, where we automatically use the dual-extension of exp and cos. We can differentiate f by simply evaluating on the duals:

$$f(1+\epsilon) = \exp(1 + 2\epsilon + \cos 1 - \sin 1\epsilon) = \exp(1+\cos 1) + \exp(1+\cos 1)(2 - \sin 1)\epsilon.$$

Therefore we deduce that

$$f'(1) = \exp(1+\cos 1)(2-\sin 1).$$

I.4 Newton's method

In school you may recall learning Newton's method: a way of approximating zeros/roots to a function by using a local approximation by an affine function. That is, approximate a function $f(x)$ locally around an initial guess x_0 by its first order Taylor series:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

and then find the root of the right-hand side which is

$$f(x_0) + f'(x_0)(x - x_0) = 0 \Leftrightarrow x = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

We can then repeat using this root as the new initial guess. In other words we have a sequence of *hopefully* more accurate approximations:

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

Thus *if* we can compute derivatives, we can (sometimes) compute roots. The lab will explore using dual numbers to accomplish this task. This is in some sense a baby version of how Machine Learning algorithms train neural networks; but where Newton uses derivatives (or in higher-dimensions, gradients) to find roots of functions Machine Learning uses gradients to roughly minimise functions that represent the error between a neural network and training data.

In terms of analysis, we can guarantee convergence provided our initial guess is accurate enough. The first step is the bound the error of an iteration in terms of the previous error:

Theorem 3 (Newton error)**.** *Suppose f is twice-differentiable in a neighbourhood B of r such that $f(r) = 0$, and f' does not vanish in B. Denote the error of the k-th Newton iteration as $\varepsilon_k := r - x_k$. If $x_k \in B$ then*

$$|\varepsilon_{k+1}| \leq M|\varepsilon_k|^2$$

where

$$M := \frac{1}{2}\sup_{x\in B}|f''(x)|\sup_{x\in B}\left|\frac{1}{f'(x)}\right|.$$

Proof Using Taylor's theorem we find that

$$0 = f(r) = f(x_k + \varepsilon_k) = f(x_k) + f'(x_k)\varepsilon_k + \frac{f''(t)}{2}\varepsilon_k^2.$$

for some $t \in B$ between r and x_k. Rearranging this we get an expression for $f(x_k)$ that tells us that

$$\varepsilon_{k+1} = r - \underbrace{x_{k+1}}_{x_k - f(x_k)/f'(x_k)} = \varepsilon_k + \frac{f(x_k)}{f'(x_k)} = -\frac{f''(t)}{2f'(x_k)}\varepsilon_k^2.$$

Taking absolute values of each side gives the result.

■

Hidden in this result is a guarantee of convergence provided x_0 is sufficiently close to r.

Corollary 1 (Newton convergence)**.** *If $x_0 \in B$ is sufficiently close to r then $x_k \to r$.*

Proof

Suppose $x_k \in B$ satisfies $|\varepsilon_k| = |r - x_k| \leq M^{-1}$. Then

$$|\varepsilon_{k+1}| \leq M|\varepsilon_k|^2 \leq |\varepsilon_k|,$$

hence $x_{k+1} \in B$. Thus from induction if x_0 satisfies the condition $|\varepsilon_0| < M^{-1}$ condition then $x_k \in B$ for all k and satisfies $|\varepsilon_k| \leq M^{-1}$. Thus we find (for large enough k)

$$|\varepsilon_k| \leq M|\varepsilon_{k-1}|^2 \leq M^3|\varepsilon_{k-2}|^4 \leq M^7|\varepsilon_{k-3}|^8 \leq \ldots \leq M^{2^k-1}|\varepsilon_0|^{2^k} = \frac{1}{M}(M|\varepsilon_0|)^{2^k}.$$

Provided x_0 satisfies the strict inequality $|\varepsilon_0| < M^{-1}$ this will go to zero as $k \to \infty$.

■

Chapter II

Representing Numbers

In this chapter we aim to answer the question: when can we rely on computations done on a computer? Why are some computations (differentiation via divided differences), extremely inaccurate whilst others (integration via rectangular rule) accurate up to about 16 digits? In order to address these questions we need to dig deeper and understand at a basic level what a computer is actually doing when manipulating numbers.

Before we begin it is important to have a basic model of how a computer works. Our simplified model of a computer will consist of a Central Processing Unit (CPU)—the brains of the computer—and Memory—where data is stored. Inside the CPU there are registers, where data is temporarily stored after being loaded from memory, manipulated by the CPU, then stored back to memory. Memory is a sequence of bits: `1`s and `0`s, essentially "on/off" switches, and memory is *finite*. Finally, if one has a p-bit CPU (eg a 32-bit or 64-bit CPU), each register consists of exactly p-bits. Most likely $p = 64$ on your machine.

Thus representing numbers on a computer must overcome three fundamental limitations:

1. CPUs can only manipulate data p-bits at a time.
2. Memory is finite (in particular at most 2^p bytes).
3. There is no such thing as an "error": if anything goes wrong in the computation we must use some of the p-bits to indicate this.

This is clearly problematic: there are an infinite number of integers and an uncountable number of reals! Each of which we need to store in precisely p-bits. Moreover, some operations are simply undefined, like division by 0. This chapter discusses the solution used to this problem, alongside the mathematical analysis that is needed to understand the implications, in particular, that computations have *error*.

In particular we discuss:

1. II.1 Reals: real numbers are approximated by floating point numbers, which are a computers version of scientific notation.
2. II.2 Floating Point Arithmetic: arithmetic with floating point numbers is exact up-to-rounding, which introduces small-but-understandable errors in the computations. We explain how these errors can be analysed mathematically to get rigorous bounds.

3. II.3 Interval Arithmetic: rounding can be controlled in order to implement *interval arithmetic*, a way to compute rigorous bounds for computations. In the lab, we use this to compute up to 15 digits of $\mathrm{e} \equiv \exp 1$ rigorously with precise bounds on the error.

II.1 Reals

In this chapter, we introduce the IEEE Standard for Floating-Point Arithmetic. There are multiplies ways of representing real numbers on a computer, as well as the precise behaviour of operations such as addition, multiplication, etc. One can use

1. Fixed-point arithmetic: essentially representing a real number as an integer where a decimal point is inserted at a fixed position. This turns out to be impractical in most applications, e.g., due to loss of relative accuracy for small numbers.

2. Floating-point arithmetic: essentially scientific notation where an exponent is stored alongside a fixed number of digits. This is what is used in practice.

3. Level-index arithmetic: stores numbers as iterated exponents. This is the most beautiful mathematically but unfortunately is not as useful for most applications and is not implemented in hardware.

Before the 1980s each processor had potentially a different representation for floating-point numbers, as well as different behaviour for operations. IEEE introduced in 1985 standardised this across processors so that algorithms would produce consistent and reliable results.

This chapter may seem very low level for a mathematics course but there are two important reasons to understand the behaviour of floating-point numbers in details:

1. Floating-point arithmetic is very precisely defined, and can even be used in rigorous computations as we shall see in the labs. But it is not exact and its important to understand how errors in computations can accumulate.

2. Failure to understand floating-point arithmetic can cause catastrophic issues in practice, with the extreme example being the explosion of the Ariane 5 rocket.

II.1.1 Real numbers in binary

Integers can be written in binary as follows:

Definition 3 (binary format)**.** For $B_0, \ldots, B_p \in \{0, 1\}$ denote an integer in *binary format* by:

$$\pm(B_p \ldots B_1 B_0)_2 := \pm \sum_{k=0}^{p} B_k 2^k$$

Reals can also be presented in binary format, that is, a sequence of `0`s and `1`s alongside a decimal point:

Definition 4 (real binary format)**.** For $b_1, b_2, \ldots \in \{0, 1\}$, Denote a non-negative real number in *binary format* by:

$$(B_p \ldots B_0.b_1b_2b_3\ldots)_2 := (B_p \ldots B_0)_2 + \sum_{k=1}^{\infty} \frac{b_k}{2^k}.$$

Example 3 (rational in binary)**.** Consider the number `1/3`. In decimal recall that:

$$1/3 = 0.3333\ldots = \sum_{k=1}^{\infty} \frac{3}{10^k}$$

We will see that in binary

$$1/3 = (0.010101\ldots)_2 = \sum_{k=1}^{\infty} \frac{1}{2^{2k}}$$

Both results can be proven using the geometric series:

$$\sum_{k=0}^{\infty} z^k = \frac{1}{1-z}$$

provided $|z| < 1$. That is, with $z = \frac{1}{4}$ we verify the binary expansion:

$$\sum_{k=1}^{\infty} \frac{1}{4^k} = \frac{1}{1-1/4} - 1 = \frac{1}{3}$$

A similar argument with $z = 1/10$ shows the decimal case.

II.1.2 Floating-point numbers

Floating-point numbers are a subset of real numbers that are representable using a fixed number of bits.

Definition 5 (floating-point numbers)**.** Given integers σ (the *exponential shift*), Q (the number of *exponent bits*) and S (the *precision*), define the set of *Floating-point numbers* by dividing into *normal*, *sub-normal*, and *special number* subsets:

$$F_{\sigma,Q,S} := F^{\text{normal}}_{\sigma,Q,S} \cup F^{\text{sub}}_{\sigma,Q,S} \cup F^{\text{special}}.$$

The *normal numbers* $F^{\text{normal}}_{\sigma,Q,S} \subset \mathbb{R}$ are

$$F^{\text{normal}}_{\sigma,Q,S} := \{\pm 2^{q-\sigma} \times (1.b_1b_2b_3\ldots b_S)_2 : 1 \le q < 2^Q - 1\}.$$

The *sub-normal numbers* $F^{\text{sub}}_{\sigma,Q,S} \subset \mathbb{R}$ are

$$F^{\text{sub}}_{\sigma,Q,S} := \{\pm 2^{1-\sigma} \times (0.b_1b_2b_3\ldots b_S)_2\}.$$

The *special numbers* $F^{\text{special}} \not\subset \mathbb{R}$ are

$$F^{\text{special}} := \{\infty, -\infty, \text{NaN}\}$$

where NaN is a special symbol representing "not a number", essentially an error flag.

Note this set of real numbers has no nice *algebraic structure*: it is not closed under addition, subtraction, etc. On the other hand, we can control errors effectively hence it is extremely useful for analysis.

Floating-point numbers are stored in $1 + Q + S$ total number of bits, in the format

$$s \; q_{Q-1} \dots q_0 \; b_1 \dots b_S$$

The first bit (s) is the *sign bit*: 0 means positive and 1 means negative. The bits $q_{Q-1} \dots q_0$ are the *exponent bits*: they are the binary digits of the unsigned integer q:

$$q = (q_{Q-1} \dots q_0)_2.$$

Finally, the bits $b_1 \dots b_S$ are the *significand bits*. If $1 \leq q < 2^Q - 1$ then the bits represent the normal number

$$x = \pm 2^{q-\sigma} \times (1.b_1 b_2 b_3 \dots b_S)_2.$$

If $q = 0$ (i.e. all bits are 0) then the bits represent the sub-normal number

$$x = \pm 2^{1-\sigma} \times (0.b_1 b_2 b_3 \dots b_S)_2.$$

If $q = 2^Q - 1$ (i.e. all bits are 1) then the bits represent a special number. If all sigificand bits are 0 then it represents $\pm\infty$. Otherwise if any significand bit is 1 then it represents `NaN`.

II.1.3 IEEE floating-point numbers

Definition 6 (IEEE floating-point numbers)**.** IEEE has 3 standard floating-point formats: 16-bit (half precision), 32-bit (single precision) and 64-bit (double precision) defined by (you *do not* need to memorise these):

$$F_{16} := F_{15,5,10}$$
$$F_{32} := F_{127,8,23}$$
$$F_{64} := F_{1023,11,52}$$

Example 4 (interpreting 16-bits as a float)**.** Consider the number with bits

$$0 \; 10000 \; 1010000000$$

assuming it is a half-precision float (F_{16}). Since the sign bit is `0` it is positive. The exponent bits encode

$$q = (10000)_2 = 2^4$$

hence the exponent is

$$q - \sigma = 2^4 - 15 = 1$$

and the number is:

$$2^1 (1.1010000000)_2 = 2(1 + 1/2 + 1/8) = 3 + 1/4 = 3.25.$$

Example 5 (rational to 16-bits)**.** How is the number $1/3$ stored in F_{16}? Recall that

$$1/3 = (0.010101\dots)_2 = 2^{-2}(1.0101\dots)_2 = 2^{13-15}(1.0101\dots)_2$$

and since $13 = (1101)_2$ the exponent bits are `01101`. For the significand we round the last bit to the nearest element of F_{16}, (the exact rule for rounding is explained in detail later), so we have

$$1.0101010101010101010101\dots \approx 1.0101010101 \in F_{16}$$

and the significand bits are `0101010101`. Thus the stored bits for $1/3$ are:

$$0 \; 01101 \; 0101010101$$

II.1.4 Sub-normal and special numbers

For sub-normal numbers, the simplest example is zero, which has $q = 0$ and all significand bits zero: `0 00000 0000000000`. Unlike integers, we also have a negative zero, which has bits: `1 00000 0000000000`. This is treated as identical to positive `0` (except for degenerate operations as explained in the lab).

Example 6 (subnormal in 16-bits)**.** Consider the number with bits

1 00000 1100000000

assuming it is a half-prevision float (F_{16}). Since all exponent bits are zero it is sub-normal. Since the sign bit is `1` it is negative. Hence this number is:

$$-2^{1-\sigma}(0.1100000000)_2 = -2^{-14}(2^{-1} + 2^{-2}) = -3 \times 2^{-16}$$

The special numbers extend the real line by adding $\pm\infty$ but also a notion of "not-a-number" NaN. Whenever the bits of q of a floating-point number are all 1 then they represent an element of F^{special}. If all $b_k = 0$, then the number represents either $\pm\infty$. All other special floating-point numbers represent NaN.

Example 7 (special in 16-bits)**.** The number with bits

1 11111 0000000000

has all exponent bits equal to 1, and significand bits 0 and sign bit 1, hence represents $-\infty$. On the other hand, the number with bits

1 11111 0000000001

has all exponent bits equal to 1 but does not have all significand bits equal to 0, hence is one of many representations for NaN.

II.2 Floating Point Arithmetic

Arithmetic operations on floating-point numbers are *exact up to rounding*. There are three basic rounding strategies: round up/down/nearest. Mathematically we introduce a function to capture the notion of rounding:

Definition 7 (rounding)**.** The function $\text{fl}^{\text{up}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ rounds a real number up to the nearest floating-point number that is greater or equal:

$$\text{fl}^{\text{up}}_{\sigma,Q,S}(x) := \min\{y \in F_{\sigma,Q,S} : y \geq x\}.$$

The function $\text{fl}^{\text{down}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ rounds a real number down to the nearest floating-point number that is less or equal:

$$\text{fl}^{\text{down}}_{\sigma,Q,S}(x) := \max\{y \in F_{\sigma,Q,S} : y \leq x\}.$$

The function $\text{fl}^{\text{nearest}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ denotes the function that rounds a real number to the nearest floating-point number. In case of a tie, it returns the floating-point number whose least significant bit is equal to zero. We use the notation fl when σ, Q, S and the rounding mode are implied by context, with $\text{fl}^{\text{nearest}}$ being the default rounding mode.

In more detail on the behaviour of nearest mode, if a positive number x is between two normal floats $x_- \le x \le x_+$ we can write its expansion as

$$x = 2^{q-\sigma}(1.b_1b_2\ldots b_S b_{S+1}\ldots)_2$$

where

$$x_- := \mathrm{fl}^{\mathrm{down}}(x) = 2^{q-\sigma}(1.b_1b_2\ldots b_S)_2$$
$$x_+ := \mathrm{fl}^{\mathrm{up}}(x) = x_- + 2^{q-\sigma-S}$$

Write the half-way point as:

$$x_{\mathrm{h}} := \frac{x_+ + x_-}{2} = x_- + 2^{q-\sigma-S-1} = 2^{q-\sigma}(1.b_1b_2\ldots b_S 1)_2$$

If $x_- \le x < x_{\mathrm{h}}$ then $\mathrm{fl}(x) = x_-$ and if $x_{\mathrm{h}} < x \le x_+$ then $\mathrm{fl}(x) = x_+$. If $x = x_{\mathrm{h}}$ then it is exactly half-way between x_- and x_+. The rule is if $b_S = 0$ then $\mathrm{fl}(x) = x_-$ and otherwise $\mathrm{fl}(x) = x_+$.

In IEEE arithmetic, the arithmetic operations `+`, `-`, `*`, `/` are defined by the property that they are exact up to rounding. Mathematically we denote these operations as $\oplus, \ominus, \otimes, \oslash : F_{\sigma,Q,S} \times F_{\sigma,Q,S} \to F_{\sigma,Q,S}$ as follows:

$$x \oplus y := \mathrm{fl}(x + y)$$
$$x \ominus y := \mathrm{fl}(x - y)$$
$$x \otimes y := \mathrm{fl}(x * y)$$
$$x \oslash y := \mathrm{fl}(x/y)$$

Note also that `^` and `sqrt` are similarly exact up to rounding. Also, note that when we convert a Julia command with constants specified by decimal expansions we first round the constants to floats, e.g., `1.1 + 0.1` is actually reduced to

$$\mathrm{fl}(1.1) \oplus \mathrm{fl}(0.1)$$

This includes the case where the constants are integers (which are normally exactly floats but may be rounded if extremely large).

Example 8 (decimal is not exact)**.** On a computer `1.1+0.1` is close to but not exactly the same thing as `1.2`. This is because $\mathrm{fl}(1.1) \ne 1 + 1/10$ and $\mathrm{fl}(0.1) \ne 1/10$ since their expansion in *binary* is not finite. For F_{16} we have:

$$\mathrm{fl}(1.1) = \mathrm{fl}((1.0001100110011\ldots)_2) = (1.0001100110)_2$$
$$\mathrm{fl}(0.1) = \mathrm{fl}(2^{-4}(1.1001100110011\ldots)_2) = 2^{-4} * (1.1001100110)_2 = (0.00011001100110)_2$$

Thus when we add them we get

$$\mathrm{fl}(1.1) + \mathrm{fl}(0.1) = (1.00110011000 11)_2$$

where the red digits indicate those beyond the 10 significant digits representable in F_{16}. In this case we round down and get

$$\mathrm{fl}(1.1) \oplus \mathrm{fl}(0.1) = (1.0011001100)_2$$

On the other hand,

$$\mathrm{fl}(1.2) = \mathrm{fl}((1.0011001100 11001100\ldots)_2) = (1.0011001101)_2$$

which differs by 1 bit.

WARNING (non-associative) These operations are not associative! E.g. $(x \oplus y) \oplus z$ is not necessarily equal to $x \oplus (y \oplus z)$. Commutativity is preserved, at least.

II.2.1 Bounding errors in floating point arithmetic

When dealing with normal numbers there are some important constants that we will use to bound errors.

Definition 8 (machine epsilon/smallest positive normal number/largest normal number)**.** *Machine epsilon* is denoted

$$\epsilon_{\mathrm{m},S} := 2^{-S}.$$

When S is implied by context we use the notation ϵ_{m}. The *smallest positive normal number* is $q = 1$ and b_k all zero:

$$\min |F^{\mathrm{normal}}_{\sigma,Q,S}| = 2^{1-\sigma}$$

where $|A| := \{|x| : x \in A\}$. The *largest (positive) normal number* is

$$\max F^{\mathrm{normal}}_{\sigma,Q,S} = 2^{2^Q-2-\sigma}(1.11\ldots)_2 = 2^{2^Q-2-\sigma}(2-\epsilon_{\mathrm{m}})$$

We can bound the error of basic arithmetic operations in terms of machine epsilon, provided a real number is close to a normal number:

Definition 9 (normalised range)**.** The *normalised range* $\mathcal{N}_{\sigma,Q,S} \subset \mathbb{R}$ is the subset of real numbers that lies between the smallest and largest normal floating-point number:

$$\mathcal{N}_{\sigma,Q,S} := \{x : \min |F^{\mathrm{normal}}_{\sigma,Q,S}| \leq |x| \leq \max F^{\mathrm{normal}}_{\sigma,Q,S}\}$$

When σ, Q, S are implied by context we use the notation $\mathcal{N}$.

We can use machine epsilon to determine bounds on rounding:

Proposition 2 (round bound)**.** *If $x \in \mathcal{N}$ then*

$$\mathrm{fl}^{\mathrm{mode}}(x) = x(1+\delta^{\mathrm{mode}}_x)$$

where the relative error *is bounded by:*

$$|\delta^{\mathrm{nearest}}_x| \leq \frac{\epsilon_{\mathrm{m}}}{2}$$
$$|\delta^{\mathrm{up/down}}_x| < \epsilon_{\mathrm{m}}.$$

Proof

We will show this result for the nearest rounding mode. Note first that

$$\mathrm{fl}(-x) = -\mathrm{fl}(x)$$

and hence it suffices to prove the result for positive x. Write

$$x = 2^{q-\sigma}(1.b_1b_2\ldots b_S b_{S+1}\ldots)_2.$$

Define

$$x_- := \mathrm{fl}^{\mathrm{down}}(x) = 2^{q-\sigma}(1.b_1b_2\ldots b_S)_2$$
$$x_+ := \mathrm{fl}^{\mathrm{up}}(x) = x_- + 2^{q-\sigma-S}$$
$$x_{\mathrm{h}} := \frac{x_+ + x_-}{2} = x_- + 2^{q-\sigma-S-1} = 2^{q-\sigma}(1.b_1b_2\ldots b_S 1)_2$$

so that $x_- \leq x \leq x_+$. We consider two cases separately.

(**Round Down**) First consider the case where x is such that we round down: $\mathrm{fl}(x) = x_-$. Since $2^{q-\sigma} \leq x_- \leq x \leq x_\mathrm{h}$ we have

$$|\delta_x| = \frac{x - x_-}{x} \leq \frac{x_\mathrm{h} - x_-}{x_-} \leq \frac{2^{q-\sigma-S-1}}{2^{q-\sigma}} = 2^{-S-1} = \frac{\epsilon_\mathrm{m}}{2}.$$

(**Round Up**) If $\mathrm{fl}(x) = x_+$ then $2^{q-\sigma} \leq x_- < x_\mathrm{h} \leq x \leq x_+$ and hence

$$|\delta_x| = \frac{x_+ - x}{x} \leq \frac{x_+ - x_\mathrm{h}}{x_-} \leq \frac{2^{q-\sigma-S-1}}{2^{q-\sigma}} = 2^{-S-1} = \frac{\epsilon_\mathrm{m}}{2}.$$

■

This immediately implies relative error bounds on all IEEE arithmetic operations, e.g., if $x + y \in \mathcal{N}$ then we have

$$x \oplus y = (x + y)(1 + \delta_1)$$

where (assuming the default nearest rounding) $|\delta_1| \leq \frac{\epsilon_\mathrm{m}}{2}$.

II.2.2 Idealised floating point

With a complicated formula it is mathematically inelegant to work with normalised ranges: one cannot guarantee apriori that a computation always results in a normal float. Extending the bounds to subnormal numbers is tedious, rarely relevant, and beyond the scope of this module. Thus to avoid this issue we will work with an alternative mathematical model:

Definition 10 (idealised floating point)**.** An idealised mathematical model of floating point numbers for which the only subnormal number is zero can be defined as:

$$F_{\infty,S} := \{\pm 2^q \times (1.b_1 b_2 b_3 \ldots b_S)_2 : q \in \mathbb{Z}\} \cup \{0\}$$

Note that $F^{\mathrm{normal}}_{\sigma,Q,S} \subset F_{\infty,S}$ for all $\sigma, Q \in \mathbb{N}$. The definition of rounding $\mathrm{fl}^{mode}_{\infty,S} : \mathbb{R} \to F_{\infty,S}$ naturally extend to $F_{\infty,S}$ and hence we can consider bounds for floating point operations such as $\oplus$, $\ominus$, etc. And in this model the round bound is valid for all real numbers (including $x = 0$).

Example 9 (bounding a simple computation)**.** We show how to bound the error in computing $(1.1 + 1.2) * 1.3 = 2.99$ and we may assume idealised floating-point arithmetic $F_{\infty,S}$. First note that `1.1` on a computer is in fact $\mathrm{fl}(1.1)$, and we will always assume nearest rounding unless otherwise stated. Thus this computation becomes

$$(\mathrm{fl}(1.1) \oplus \mathrm{fl}(1.2)) \otimes \mathrm{fl}(1.3)$$

We will show the *absolute error* is given by

$$(\mathrm{fl}(1.1) \oplus \mathrm{fl}(1.2)) \otimes \mathrm{fl}(1.3) = 2.99 + \delta$$

where $|\delta| \leq 23\epsilon_\mathrm{m}$. First we find

$$\begin{aligned} \mathrm{fl}(1.1) \oplus \mathrm{fl}(1.2) &= (1.1(1 + \delta_1) + 1.2(1 + \delta_2))(1 + \delta_3) \\ &= 2.3 + \underbrace{1.1\delta_1 + 1.2\delta_2 + 2.3\delta_3 + 1.1\delta_1\delta_3 + 1.2\delta_2\delta_3}_{\varepsilon_1}. \end{aligned}$$

While $\delta_1\delta_3$ and $\delta_2\delta_3$ are absolutely tiny in practice we will bound them rather naïvely by eg.

$$|\delta_1\delta_3| \leq \epsilon_{\mathrm{m}}^2/4 \leq \epsilon_{\mathrm{m}}/4.$$

Further we round up constants to integers in the bounds for simplicity. We thus have the bound

$$|\varepsilon_1| \leq (2+2+3+1+1)\frac{\epsilon_{\mathrm{m}}}{2} \leq 5\epsilon_{\mathrm{m}}.$$

Writing $\mathrm{fl}(1.3) = 1.3(1+\delta_4)$ and also incorporating an error from the rounding in $\otimes$ we arrive at

$$\begin{aligned}(\mathrm{fl}(1.1) \oplus \mathrm{fl}(1.2)) \otimes \mathrm{fl}(1.3) &= (2.3+\varepsilon_1)1.3(1+\delta_4)(1+\delta_5) \\ &= 2.99 + \underbrace{1.3(\varepsilon_1 + 2.3\delta_4 + 2.3\delta_5 + \varepsilon_1\delta_4 + \varepsilon_1\delta_5 + 2.3\delta_4\delta_5 + \varepsilon_1\delta_4\delta_5)}_{\delta}\end{aligned}$$

We use the bounds

$$\begin{aligned}|\varepsilon_1\delta_4|, |\varepsilon_1\delta_5| &\leq 5\epsilon_{\mathrm{m}}^2/2 \leq 5\epsilon_{\mathrm{m}}/2, \\ |\delta_4\delta_5| &\leq \epsilon_{\mathrm{m}}^2/4 \leq \epsilon_{\mathrm{m}}/4, \\ |\varepsilon_1\delta_4\delta_5| &\leq 5\epsilon_{\mathrm{m}}^3/4 \leq 5\epsilon_{\mathrm{m}}/4.\end{aligned}$$

Thus the *absolute error* is bounded (bounding 1.3 by 3/2) by

$$|\delta| \leq (3/2)(5+3/2+3/2+5/2+5/2+3/4+5/4)\epsilon_{\mathrm{m}} \leq 23\epsilon_{\mathrm{m}}.$$

II.2.3 Divided differences floating point error bound

We can use the bound on floating point arithmetic to deduce a bound on divided differences that captures the phenomena we observed where the error of divided differences became large as $h \to 0$. We assume that the function we are attempting to differentiate is computed using floating point arithmetic in a way that has a small absolute error.

Theorem 4 (divided difference error bound)**.** *Assume we are working in idealised floating-point arithmetic $F_{\infty,S}$. Let f be twice-differentiable in a neighbourhood of $x \in F_{\infty,S}$ and assume that*

$$f(x) = f^{\mathrm{FP}}(x) + \delta_x^f$$

where $f^{\mathrm{FP}} : F_{S,\infty} \to F_{S,\infty}$ has uniform absolute accuracy in that neighbourhood, that is:

$$|\delta_x^f| \leq c\epsilon_{\mathrm{m}}$$

for a fixed constant $c \geq 0$. The divided difference approximation partially implemented with floating point satisfies

$$\frac{f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x)}{h} = f'(x) + \delta_{x,h}^{\mathrm{FD}}$$

where

$$|\delta_{x,h}^{\mathrm{FD}}| \leq \frac{|f'(x)|}{2}\epsilon_{\mathrm{m}} + Mh + \frac{4c\epsilon_{\mathrm{m}}}{h}$$

for $M = \sup_{x \leq t \leq x+h} |f''(t)|$.

Proof

We have

$$\begin{aligned}(f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x))/h &= \frac{f(x+h) - \delta^f_{x+h} - f(x) + \delta^f_x}{h}(1+\delta_1) \\ &= \frac{f(x+h)-f(x)}{h}(1+\delta_1) + \frac{\delta^f_x - \delta^f_{x+h}}{h}(1+\delta_1)\end{aligned}$$

where $|\delta_1| \leq \epsilon_{\mathrm{m}}/2$. Applying Taylor's theorem we get

$$(f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x))/h = f'(x) + \underbrace{f'(x)\delta_1 + \frac{f''(t)}{2}h(1+\delta_1) + \frac{\delta^f_x - \delta^f_{x+h}}{h}(1+\delta_1)}_{\delta^{\mathrm{FD}}_{x,h}}$$

The bound then follows, using the very pessimistic bound $|1+\delta_1| \leq 2$.

■

The previous theorem neglected some errors due to rounding, which was done for simplicity. This is justified under fairly general restrictions:

Corollary 2 (divided differences in practice)**.** *We have*

$$(f^{\mathrm{FP}}(x \oplus h) \ominus f^{\mathrm{FP}}(x)) \oslash h = \frac{f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x)}{h}$$

whenever $h = 2^{j-n}$ *for* $0 \leq n \leq S$ *and the last binary place of* $x \in F_{\infty,S}$ *is zero, that is* $x = \pm 2^j (1.b_1 \dots b_{S-1}0)_2$.

Proof

We first confirm $x \oplus h = x + h$. If $b_S = 0$ the worst possible case is that we increase the exponent by one as we are just adding 1 to one of the digits $b_1, \dots, b_S$. This would cause us to lose the last digit. But if that is zero no error is incurred when we round.

Now write $y := (f^{\mathrm{FP}}(x \oplus h) \ominus f^{\mathrm{FP}}(x)) = \pm 2^\nu (1.c_1 \dots c_S)_2 \in F_{\infty,S}$. We have

$$y/h = \pm 2^{\nu+n-j}(1.c_1 \dots c_S)_2 \in F_{\infty,S} \Rightarrow y/h = y \oslash h.$$

■

The three-terms of this bound tell us a story: the first term is a fixed (small) error, the second term tends to zero as $h \to 0$, while the last term grows like ϵ_{m}/h as $h \to 0$. Thus we observe convergence while the second term dominates, until the last term takes over. Of course, a bad upper bound is not the same as a proof that something grows, but it is a good indication of what happens *in general* and suffices to choose h so that these errors are balanced (and thus minimised). Since in general we do not have access to the constants c and M we employ the following heuristic to balance the two sources of errors:

Heuristic (divided difference with floating-point step) Choose h proportional to $\sqrt{\epsilon_{\mathrm{m}}}$ in divided differences so that Mh and $\frac{4c\epsilon_{\mathrm{m}}}{h}$ are (roughly) the same magnitude.

In the case of double precision $\sqrt{\epsilon_{\mathrm{m}}} \approx 1.5 \times 10^{-8}$, which is close to when the observed error begins to increase in the examples we saw before.

Remark While divided differences is of debatable utility for computing derivatives, it is extremely effective in building methods for solving differential equations, as we shall see

later. It is also very useful as a "sanity check" if one wants something to compare with other numerical methods for differentiation.

Remark It is also possible to deduce an error bound for the rectangular rule showing that the error caused by round-off is on the order of $n\epsilon_{\mathrm{m}}$, that is it does in fact grow but the error without round-off which was bounded by M/n will be substantially greater for all reasonable values of n.

II.3 Interval Arithmetic

It is possible to use rounding modes (up/down) to do rigorous computation to compute bounds on the error in, for example, the digits of e. To do this we will use set/interval arithmetic. For sets $X, Y \subseteq \mathbb{R}$, the set arithmetic operations are defined as

$$\begin{aligned} X + Y &:= \{x + y : x \in X, y \in Y\}, \\ XY &:= \{xy : x \in X, y \in Y\}, \\ X/Y &:= \{x/y : x \in X, y \in Y\} \end{aligned}$$

We will use floating point arithmetic to construct approximate set operations $\oplus$, $\otimes$ so that

$$\begin{aligned} X + Y &\subseteq X \oplus Y, \\ XY &\subseteq X \otimes Y, \\ X/Y &\subseteq X \oslash Y \end{aligned}$$

thereby a complicated algorithm can be run on sets and the true result is guaranteed to be a subset of the output.

When our sets are intervals we can deduce simple formulas for basic arithmetic operations. For simplicity we only consider the case where all values are positive.

Proposition 3 (interval bounds)**.** *For intervals $X = [a, b]$ and $Y = [c, d]$ satisfying $0 < a \leq b$ and $0 < c \leq d$, and $n > 0$, we have:*

$$\begin{aligned} X + Y &= [a + c, b + d] \\ X/n &= [a/n, b/n] \\ XY &= [ac, bd] \end{aligned}$$

Proof We first show $X + Y \subseteq [a + c, b + d]$. If $z \in X + Y$ then $z = x + y$ such that $a \leq x \leq b$ and $c \leq y \leq d$ and therefore $a + c \leq z \leq c + d$ and $z \in [a + c, b + d]$. Equality follows from convexity. First note that $a + c, b + d \in X + Y$. Any point $z \in [a + b, c + d]$ can be written as a convex combination of the two endpoints: there exists $0 \leq t \leq 1$ such that

$$z = (1 - t)(a + c) + t(b + d) = \underbrace{(1 - t)a + tb}_{x} + \underbrace{(1 - t)c + td}_{y}$$

Because intervals are convex we have $x \in X$ and $y \in Y$ and hence $z \in X + Y$.

The remaining two proofs are left for the problem sheet.

■

We want to implement floating point variants of these operations that are guaranteed to contain the true set arithmetic operations. We do so as follows:

Definition 11 (floating point interval arithmetic)**.** For intervals $A = [a, b]$ and $B = [c, d]$ satisfying $0 < a \le b$ and $0 < c \le d$, and $n > 0$, define:

$$\begin{aligned}
[a, b] \oplus [c, d] &:= [\mathrm{fl}^{\mathrm{down}}(a + c), \mathrm{fl}^{\mathrm{up}}(b + d)] \\
[a, b] \ominus [c, d] &:= [\mathrm{fl}^{\mathrm{down}}(a - d), \mathrm{fl}^{\mathrm{up}}(b - c)] \\
[a, b] \oslash n &:= [\mathrm{fl}^{\mathrm{down}}(a/n), \mathrm{fl}^{\mathrm{up}}(b/n)] \\
[a, b] \otimes [c, d] &:= [\mathrm{fl}^{\mathrm{down}}(ac), \mathrm{fl}^{\mathrm{up}}(bd)]
\end{aligned}$$

Example 10 (small sum)**.** consider evaluating the first few terms in the Taylor series of the exponential at $x = 1$ using interval arithmetic with half-precision F_{16} arithmetic. The first three terms are exact since all numbers involved are exactly floats, in particular if we evaluate $1 + x + x^2/2$ with $x = 1$ we get

$$1 + 1 + 1/2 \in 1 \oplus [1, 1] \oplus ([1, 1] \otimes [1, 1]) \oslash 2 = [5/2, 5/2]$$

Noting that

$$1/6 = (1/3)/2 = 2^{-3}(1.01010101\ldots)_2$$

we can extend the computation to another term:

$$\begin{aligned}
1 + 1 + 1/2 + 1/6 &\in [5/2, 5/2] \oplus ([1, 1] \oslash 6) \\
&= [2(1.01)_2, 2(1.01)_2] \oplus 2^{-3}[(1.0101010101)_2, (1.0101010110)_2] \\
&= [\mathrm{fl}^{\mathrm{down}}(2(1.0101010101\mathbf{0101})_2), \mathrm{fl}^{\mathrm{up}}(2(1.0101010101\mathbf{011})_2)] \\
&= [2(1.0101010101)_2, 2(1.0101010110)_2] \\
&= [2.666015625, 2.66796875]
\end{aligned}$$

Example 11 (exponential with intervals)**.** Consider computing $\exp(x)$ for $0 \le x \le 1$ from the Taylor series approximation:

$$\exp(x) = \sum_{k=0}^{n} \frac{x^k}{k!} + \underbrace{\exp(t)\frac{x^{n+1}}{(n+1)!}}_{\delta_{x,n}}$$

where we can bound the error by (using the fact that $\mathrm{e} = 2.718\ldots \le 3$)

$$|\delta_{x,n}| \le \frac{\exp(1)}{(n+1)!} \le \frac{3}{(n+1)!}.$$

Put another way: $\delta_{x,n} \in \left[-\frac{3}{(n+1)!}, \frac{3}{(n+1)!}\right]$. We can use this to adjust the bounds derived from interval arithmetic for the interval arithmetic expression:

$$\exp(X) \subseteq \left(\bigoplus_{k=0}^{n} X \oslash k \oslash k!\right) \oplus \left[\mathrm{fl}^{\mathrm{down}}\left(-\frac{3}{(n+1)!}\right), \mathrm{fl}^{\mathrm{up}}\left(\frac{3}{(n+1)!}\right)\right]$$

For example, with $n = 3$ we have $|\delta_{1,2}| \le 3/4! = 1/2^3$. Thus we can prove that:

$$\begin{aligned}
\mathrm{e} &= 1 + 1 + 1/2 + 1/6 + \delta_x \in [2(1.0101010101)_2, 2(1.0101010110)_2] \oplus [-1/2^3, 1/2^3] \\
&= [2(1.0100010101)_2, 2(1.0110010110)_2] = [2.541015625, 2.79296875]
\end{aligned}$$

In the lab we get many more digits by using a computer to compute the bounds.

Chapter III

Numerical Linear Algebra

Many problems in mathematics are linear: for example, polynomial regression and differential equations. Numerical methods for such applications invariably result in (finite-dimensional) linear systems that must be solved numerically on a computer: the dimensions of the problems are often in the 1000s, millions, or even billions. One would certainly not want to tackle that with Gaussian elimination by hand! In this chapter we discuss algorithms, and in particular matrix factorisations, that are computed using floating point operations. We also introduce some basic applications.

In particular we discuss:

1. III.1 Structured Matrices: we discuss special structured matrices such as triangular and tridiagonal.
2. III.2 LU and PLU Factorisations: we see that Gaussian elimination can be recast as computing a factorisation of a square matrix as a product of a lower and upper triangular matrix, potentially with a permutation matrix corresponding to the case where row pivoting is required.
3. III.3 Cholesky Factorisation: In the special case where the matrix is symmetric positive definite the LU factorisation has a special form. Hidden in this is an algorithm to prove positive definiteness.
4. III.4 Orthogonal Matrices: we discuss different types of orthogonal matrices, which will be used to simplify rectangular least squares problems.
5. III.5 QR Factorisation: we introduce an algorithm to compute a factorisation of a rectangular matrix as a product of an orthogonal and upper triangular matrix, thereby solving least squares problems.

Here we are constructing underlying computational tools that are important in applications, such as solving differential equations and data regression, which we discuss later.

III.1 Structured Matrices

We have seen how algebraic operations (`+`, `-`, `*`, `/`) are defined exactly in terms of rounding ($\oplus$, $\ominus$, $\otimes$, $\oslash$) for floating point numbers. Now we see how this allows us to do (approximate) linear algebra operations on matrices.

A matrix can be stored in different formats, in particular it is important for large scale simulations that we take advantage of *sparsity*: if we know a matrix has entries that are guaranteed to be zero we can implement faster algorithms. We shall see that this comes up naturally in numerical methods for solving differential equations.

In particular, we will discuss some basic types of structure in matrices:

1. *Dense*: This can be considered unstructured, where we need to store all entries in a vector or matrix. Matrix-vector multiplication reduces directly to standard algebraic operations. Solving linear systems with dense matrices will be discussed later.

2. *Triangular*: If a matrix is upper or lower triangular, multiplication requires roughly half the number of operations. Crucially, we can apply the inverse of a triangular matrix using forward- or back-substitution.

3. *Banded*: If a matrix is zero apart from entries a fixed distance from the diagonal it is called banded and matrix-vector multiplication has a lower *complexity*: the number of operations scales linearly with the dimension (instead of quadratically). We discuss three cases: diagonal, tridiagonal and bidiagonal matrices.

Remark For those who took the first half of the module, there was an important emphasis on working with *linear operators* rather than *matrices*. That is, there was an emphasis on basis-independent mathematical techniques, which is critical for extension of results to infinite-dimensional spaces (which might not have a complete basis). However, in terms of practical computation we need to work with some representation of an operator and the most natural is a matrix. And indeed we will see in the next section how infinite-dimensional differential equations can be solved by reduction to finite-dimensional matrices. (Restricting attention to matrices is also important as some of the students have not taken the first half of the module.)

III.1.1 Dense matrices

A basic operation is matrix-vector multiplication. For a field $\mathbb{F}$ (typically $\mathbb{R}$ or $\mathbb{C}$, or this can be relaxed to be a ring), consider a matrix and vector whose entries are in $\mathbb{F}$:

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}_1 | \cdots | \boldsymbol{a}_n \end{bmatrix} \in \mathbb{F}^{m\times n}, \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{F}^n.$$

where $\boldsymbol{a}_j = A\boldsymbol{e}_j \in \mathbb{F}^m$ are the columns of A. Recall the usual definition of matrix multiplication:

$$A\boldsymbol{x} := \begin{bmatrix} \sum_{j=1}^n a_{1j}x_j \\ \vdots \\ \sum_{j=1}^n a_{mj}x_j \end{bmatrix}.$$

When we are working with floating point numbers $A \in F^{m\times n}$ we obtain an approximation:

$$A\boldsymbol{x} \approx \begin{bmatrix} \bigoplus_{j=1}^n (a_{1j} \otimes x_j) \\ \vdots \\ \bigoplus_{j=1}^n (a_{mj} \otimes x_j) \end{bmatrix}.$$

This actually encodes an algorithm for computing the entries.

This algorithm uses $O(mn)$ floating point operations (see the appendix if you are unaware of Big-O notation, here our complexities are implicitly taken to be when m or n tends to ∞): each of the m entries consists of n multiplications and $n-1$ additions, hence we have a total of $2n-1 = O(n)$ operations per row for a total of $m(2n-1) = O(mn)$ operations. For a square matrix this is $O(n^2)$ operations which we call *quadratic complexity*. In the problem sheet we see how the floating point error can be bounded in terms of norms, thus reducing the problem to a purely mathematical concept.

Sometimes there are multiple ways of implementing numerical algorithms. We have an alternative formula where we multiply by columns:

$$A\boldsymbol{x} = x_1\boldsymbol{a}_1 + \cdots + x_n\boldsymbol{a}_n.$$

The floating point formula for this is exactly the same as the previous algorithm and the number of operations is the same. Just the order of operations has changed. Suprisingly, this latter version is significantly faster.

Remark Floating point operations are sometimes called FLOPs, which are a standard measurement of speed of CPUs. However, FLOP sometimes uses an alternative definitions that combines an addition and multiplication as a single FLOP. In the lab we give an example showing that counting the precise number of operations is somewhat of a fools errand: algorithms such as the two approaches for matrix multiplication with the exact same number of operations can have wildly different speeds. We will therefore only be concerned with *complexity*; the asymptotic growth (Big-O) of operations as $n \to \infty$, in which case the difference between FLOPs and operations is immaterial.

III.1.2 Triangular matrices

The simplest sparsity case is being triangular: where all entries above or below the diagonal are zero. We consider upper and lower triangular matrices:

$$U = \begin{bmatrix} u_{11} & \cdots & u_{1n} \\ & \ddots & \vdots \\ & & u_{nn} \end{bmatrix}, \qquad L = \begin{bmatrix} \ell_{11} & & \\ \vdots & \ddots & \\ \ell_{n1} & \cdots & \ell_{nn} \end{bmatrix}.$$

Matrix multiplication can be modified to take advantage of the zero pattern of the matrix. Eg., if $L \in \mathbb{F}^{n\times n}$ is lower triangular we have:

$$L\boldsymbol{x} = \begin{bmatrix} \ell_{1,1}x_1 \\ \sum_{j=1}^{2} \ell_{2j}x_j \\ \vdots \\ \sum_{j=1}^{n} \ell_{nj}x_j \end{bmatrix}.$$

When implemented in floating point this uses roughly half the number of multiplications: $1 + 2 + \ldots + n = n(n+1)/2$ multiplications. (It is also about twice as fast in practice.) The complexity is still quadratic: $O(n^2)$ operations.

Triangularity allows us to also invert systems using forward- or back-substitution. In particular if $\boldsymbol{x}$ solves $L\boldsymbol{x} = \boldsymbol{b}$ then we have:

$$x_k = \frac{b_k - \sum_{j=1}^{k-1} \ell_{kj}x_j}{\ell_{kk}}$$

Thus we can compute $x_1, x_2, \ldots, x_n$ in sequence.

III.1.3 Banded matrices

A *banded matrix* is zero off a prescribed number of diagonals. We call the number of (potentially) non-zero diagonals the *bandwidths*:

Definition 12 (bandwidths)**.** A matrix A has *lower-bandwidth* l if $a_{kj} = 0$ for all $k-j > l$ and *upper-bandwidth* u if $a_{kj} = 0$ for all $j - k > u$. We say that it has *strictly lower-bandwidth* l if it has lower-bandwidth l and there exists a j such that $a_{j+l,j} \neq 0$. We say that it has *strictly upper-bandwidth* u if it has upper-bandwidth u and there exists a k such that $a_{k,k+u} \neq 0$.

A square banded matrix has the sparsity pattern:

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1,u+1} & & & \\ \vdots & a_{22} & \ddots & a_{2,u+2} & & \\ a_{1+l,1} & \ddots & \ddots & \ddots & \ddots & \\ & a_{2+l,2} & \ddots & \ddots & \ddots & a_{n-u,n} \\ & & \ddots & \ddots & \ddots & \vdots \\ & & & a_{n,n-l} & \cdots & a_{nn} \end{bmatrix}$$

A banded matrix has better complexity for matrix multiplication and solving linear systems: we can multiply square banded matrices in linear complexity: $O(n)$ operations. We consider two cases in particular (in addition to diagonal): bidiagonal and tridiagonal.

Definition 13 (Bidiagonal)**.** If a square matrix has bandwidths $(l, u) = (1, 0)$ it is *lower-bidiagonal* and if it has bandwidths $(l, u) = (0, 1)$ it is *upper-bidiagonal*.

For example, if

$$L = \begin{bmatrix} \ell_{11} & & & \\ \ell_{21} & \ell_{22} & & \\ & \ddots & \ddots & \\ & & \ell_{n,n-1} & \ell_{nn} \end{bmatrix}$$

then lower-bidiagonal multiplication becomes

$$L\boldsymbol{x} = \begin{bmatrix} \ell_{1,1}x_1 \\ \ell_{21}x_1 + \ell_{22}x_2 \\ \vdots \\ \ell_{n,n-1}x_{n-1} + \ell_{nn}x_n \end{bmatrix}.$$

This requires $O(1)$ operations per row (at most 2 multiplications and 1 addition) and hence the total is only $O(n)$ operations. A bidiagonal matrix is always triangular and we can also invert in $O(n)$ operations: if $L\boldsymbol{x} = \boldsymbol{b}$ then $x_1 = b_1/\ell_{11}$ and for $k = 2, \ldots, n$ we can compute

$$x_k = \frac{b_k - \ell_{k-1,k}x_{k-1}}{\ell_{kk}}.$$

Definition 14 (Tridiagonal)**.** If a square matrix has bandwidths $l = u = 1$ it is *tridiagonal*.

For example,

$$A = \begin{bmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ & & & a_{n,n-1} & a_{nn} \end{bmatrix}$$

is tridiagonal. Matrix multiplication is clearly $O(n)$ operations: each row has $O(1)$ non-zeros and there are n rows. But so is solving linear systems, which we shall see later.

III.2 LU and PLU factorisations

One of the most fundamental problems in linear algebra is solving linear systems. For a field $\mathbb{F}$ (for us either $\mathbb{R}$ or $\mathbb{C}$), given invertible matrix $A \in \mathbb{F}^{n\times n}$ and vector $\boldsymbol{b} \in \mathbb{F}^n$, find $\boldsymbol{x} \in \mathbb{F}^n$ such that

$$A\boldsymbol{x} = \boldsymbol{b}.$$

This can of course be done via Gaussian elimination, uing row swaps (or *pivoting*) if a zero is encountered on the diagonal, which can be viewed as an algorithm that can be implemented on a computer. However, a basic observation makes the practical implementation more straightforward and easier to apply to multiple right-hand sides, and connects with fundamental aspects in matrix analysis.

In particular, Gaussian elimination is equivalent to computing an *LU factorisation*:

$$A = LU$$

where L is lower triangular and U is upper triangular. Thus if we compute L and U we can deduce

$$\boldsymbol{x} = A^{-1}\boldsymbol{b} = U^{-1}L^{-1}\boldsymbol{b}$$

where $\boldsymbol{c} = L^{-1}\boldsymbol{b}$ can be computed using forward-substitution and $U^{-1}\boldsymbol{c}$ using back-substitution.

On the other hand, Gaussian elemination with pivoting (row-swapping) is equivalent to a *PLU factorisation*:

$$A = P^\top LU$$

where P is a permutation matrix (see appendix). Thus if we can compute P, L and U we can deduce

$$\boldsymbol{x} = A^{-1}\boldsymbol{b} = U^{-1}L^{-1}P\boldsymbol{b}$$

where multiplication by P is a simple swap of entries of $\boldsymbol{b}$ and L and U are again invertible via forward- and back-substitution.

III.2.1 Outer products

In what follows we will use outer products extensively:

Definition 15 (outer product)**.** Given $\boldsymbol{x} \in \mathbb{F}^m$ and $\boldsymbol{y} \in \mathbb{F}^n$ the *outer product* is:

$$\boldsymbol{x}\boldsymbol{y}^\top := [\boldsymbol{x}y_1|\cdots|\boldsymbol{x}y_n] = \begin{bmatrix} x_1y_1 & \cdots & x_1y_n \\ \vdots & \ddots & \vdots \\ x_my_1 & \cdots & x_my_n \end{bmatrix} \in \mathbb{F}^{m\times n}.$$

Note this is equivalent to matrix-matrix multiplication if we view $\boldsymbol{x}$ as a $m \times 1$ matrix and $\boldsymbol{y}^\top$ as a $1 \times n$ matrix.

Proposition 4 (rank-1)**.** *A matrix $A \in \mathbb{F}^{m\times n}$ has rank 1 if and only if there exists $\boldsymbol{x} \in \mathbb{F}^m$ and $\boldsymbol{y} \in \mathbb{F}^n$ such that*

$$A = \boldsymbol{x}\boldsymbol{y}^\top.$$

Proof If $A = \boldsymbol{x}\boldsymbol{y}^\top$ then all columns are multiples of $\boldsymbol{x}$, that is the column span has dimension 1. On the other hand, if A has rank-1 then its columns span a one-dimensional subspace: there exists $\boldsymbol{x} \in \mathbb{F}^m$

$$\operatorname{span}(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) = \{c\boldsymbol{x} : c \in \mathbb{F}\}.$$

Thus there exist $y_k \in \mathbb{F}$ such that $\boldsymbol{a}_k = y_k\boldsymbol{x}$ and we have

$$A = \boldsymbol{x} \underbrace{\begin{bmatrix} y_1 & \cdots & y_n \end{bmatrix}}_{\boldsymbol{y}^\top}.$$

■

III.2.2 LU factorisation

Gaussian elimination can be interpreted as an LU factorisation. Write a matrix $A \in \mathbb{F}^{n\times n}$ as follows:

$$A = \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ \boldsymbol{v} & K \end{bmatrix}$$

where $\alpha = a_{11}$, $\boldsymbol{v} = A[2:n, 1]$ and $\boldsymbol{w} = A[1, 2:n]$ (that is, $\boldsymbol{v} \in \mathbb{F}^{n-1}$ is a vector whose entries are the 2nd through last row of the first column of A whilst $\boldsymbol{w} \in \mathbb{F}^{n-1}$ is a vector containing the 2nd through last column of the first row of A). Gaussian elimination consists of taking the first row, dividing by α and subtracting from all other rows. That is equivalent to multiplying by a lower triangular matrix:

$$\underbrace{\begin{bmatrix} 1 & \\ -\boldsymbol{v}/\alpha & I \end{bmatrix}}_{L_1^{-1}} A = \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & K - \boldsymbol{v}\boldsymbol{w}^\top/\alpha \end{bmatrix}$$

where $A_2 := K - \boldsymbol{v}\boldsymbol{w}^\top/\alpha$ happens to be a rank-1 perturbation of K. We can write this another way:

$$A = \underbrace{\begin{bmatrix} 1 & \\ \boldsymbol{v}/\alpha & I \end{bmatrix}}_{L_1} \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & A_2 \end{bmatrix}$$

Now assume we continue this process and manage to deduce an LU factorisation $A_2 = \tilde{L}\tilde{U}$. Then

$$A = L_1 \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & \tilde{L}\tilde{U} \end{bmatrix} = \underbrace{L_1 \begin{bmatrix} 1 & \\ & \tilde{L} \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & \tilde{U} \end{bmatrix}}_{U}$$

Note we can multiply through to find

$$L = \begin{bmatrix} 1 & \\ \boldsymbol{v}/\alpha & \tilde{L} \end{bmatrix}.$$

Noting that if $A \in \mathbb{F}^{1\times 1}$ then it has a trivial LU factorisation we can use the above construction to proceed recursively until we arrive at the trivial case.

Rather than a recursive definition, we can view the above as an inductive procedure:

$$\begin{aligned} A &= L_1 \begin{bmatrix} \alpha_1 & \boldsymbol{w}_1^\top \\ & A_2 \end{bmatrix} = L_1 \begin{bmatrix} \alpha_1 & & \boldsymbol{w}_1^\top \\ & L_2 & \begin{bmatrix} \alpha_2 & \boldsymbol{w}_2^\top \\ & A_3 \end{bmatrix} \end{bmatrix} \\ &= L_1 \begin{bmatrix} 1 & \\ & L_2 \end{bmatrix} \begin{bmatrix} \alpha_1 & & \boldsymbol{w}_1^\top \\ & \begin{bmatrix} \alpha_2 & \boldsymbol{w}_2^\top \\ & L_3 \end{bmatrix} & \begin{bmatrix} \alpha_3 & \boldsymbol{w}_3^\top \\ & A_4 \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 1 & & \\ \boldsymbol{v}_1/\alpha_1 & \begin{bmatrix} 1 & \\ \boldsymbol{v}_2/\alpha_2 & \begin{bmatrix} 1 & \\ \boldsymbol{v}_3/\alpha_3 & \ddots \end{bmatrix} \end{bmatrix} \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \alpha_1 & & \boldsymbol{w}_1^\top \\ & \begin{bmatrix} \alpha_2 & \boldsymbol{w}_2^\top \\ & \begin{bmatrix} \alpha_3 & \boldsymbol{w}_3^\top \\ & \ddots \end{bmatrix} \end{bmatrix} \end{bmatrix}}_{U}. \end{aligned}$$

We can see this procedure clearer in the following example.

Example 12 (LU by-hand)**.** Consider the matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 1 & 4 & 9 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & & \\ 2 & 1 & \\ 1 & & 1 \end{bmatrix}}_{L_1} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 6 \\ 0 & 3 & 8 \end{bmatrix}$$

In more detail, for $\alpha_1 := a_{11} = 1$, $\boldsymbol{v}_1 := A[2:3,1] = [2,1]^\top$, $\boldsymbol{w}_1 = A[1,2:3] = [1,1]^\top$ and

$$K_1 := A[2:3,2:3] = \begin{bmatrix} 4 & 8 \\ 4 & 9 \end{bmatrix}$$

we have

$$A_2 := K_1 - \boldsymbol{v}_1 \boldsymbol{w}_1^\top/\alpha_1 = \begin{bmatrix} 4 & 8 \\ 4 & 9 \end{bmatrix} - \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 6 \\ 3 & 8 \end{bmatrix}.$$

We then repeat the process and determine (with $\alpha_2 := A_2[1,1] = 2$, $\boldsymbol{v}_2 := A_2[2:2,1] = [3]$, $\boldsymbol{w}_2 := A_2[1,2:2] = [6]$ and $K_2 := A_2[2:2,2:2] = [8]$):

$$A_2 = \begin{bmatrix} 2 & 6 \\ 3 & 8 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \\ 3/2 & 1 \end{bmatrix}}_{L_2} \begin{bmatrix} 2 & 6 \\ & -1 \end{bmatrix}$$

The last "matrix" is 1 x 1 so we get the trivial decomposition:

$$A_3 := K_2 - \boldsymbol{v}_2 \boldsymbol{w}_2^\top/\alpha_2 = [-1] = \underbrace{[1]}_{L_3}[-1]$$

Putting everything together and placing the j-th column of L_j inside the j-th column of L we have

$$A = \underbrace{\begin{bmatrix} 1 & & \\ 2 & 1 & \\ 1 & 3/2 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & 1 & 1 \\ & 2 & 6 \\ & & -1 \end{bmatrix}}_{U}$$

III.2.3 PLU factorisation

We learned in first year linear algebra that if a diagonal entry is zero when doing Gaussian elimination one has to *row pivot*. For stability, in implementation one may wish to pivot even if the diagonal entry is nonzero: swap the largest in magnitude entry for the entry on the diagonal turns out to be significantly more stable than standard LU.

This is equivalent to a PLU decomposition. Here we use a *permutation matrix*, whose action on a vector permutes its entries, as discussed in the appendix. That is, consider a permutation which we identify with a vector $\sigma = [\sigma_1, \dots, \sigma_n]$ containing the integers $1, \dots, n$ exactly once. The permutation operator represents the action of permuting the entries in a vector:

$$P_\sigma(\boldsymbol{v}) := \boldsymbol{v}[\sigma] = \begin{bmatrix} v_{\sigma_1} \\ \vdots \\ v_{\sigma_n} \end{bmatrix}$$

This is a linear operator, and hence we can identify it with a *permutation matrix* $P_\sigma \in \mathbb{R}^{n\times n}$ (more precisely the entries of P_σ are either 1 or 0). Importantly, products of permutation matrices are also permutation matrices and permutation matrices are orthogonal, that is, $P_\sigma^{-1} = P_\sigma^\top$.

Theorem 5 (PLU)**.** *A matrix $A \in \mathbb{C}^{n\times n}$ is invertible if and only if it has a PLU decomposition:*

$$A = P^\top LU$$

where the diagonal of L are all equal to 1 and the diagonal of U are all non-zero, and P is a permutation matrix.

Proof

If we have a PLU decomposition of this form then L and U are invertible and hence the inverse is simply $A^{-1} = U^{-1}L^{-1}P$. Hence we consider the orther direction.

If $A \in \mathbb{C}^{1\times 1}$ we trivially have an LU decomposition $A = [1] * [a_{11}]$ as all 1×1 matrices are triangular. We now proceed by induction: assume all invertible matrices of lower dimension have a PLU factorisation. As A is invertible not all entries in the first column are zero. Therefore there exists a permutation P_1 so that $\alpha := (P_1 A)[1,1] \neq 0$. Hence we write

$$P_1 A = \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ \boldsymbol{v} & K \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \\ \boldsymbol{v}/\alpha & I \end{bmatrix}}_{L_1} \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & K - \boldsymbol{v}\boldsymbol{w}^\top/\alpha \end{bmatrix}$$

We deduce that $A_2 := K - \boldsymbol{v}\boldsymbol{w}^\top/\alpha$ is invertible because A and L_1 are invertible (Exercise).

By assumption we can write $A_2 = \tilde{P}^\top \tilde{L}\tilde{U}$. Thus we have:

$$\begin{aligned} \underbrace{\begin{bmatrix} 1 & \\ & \tilde{P} \end{bmatrix} P_1}_{P} A &= \begin{bmatrix} 1 & \\ & \tilde{P} \end{bmatrix} L_1 \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & A_2 \end{bmatrix} = \begin{bmatrix} 1 & \\ & \tilde{P} \end{bmatrix} L_1 \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & \tilde{P}^\top \tilde{L}\tilde{U} \end{bmatrix} \\ &= \begin{bmatrix} 1 & \\ \tilde{P}\boldsymbol{v}/\alpha & \tilde{P} \end{bmatrix} \begin{bmatrix} 1 & \\ & \tilde{P}^\top \tilde{L} \end{bmatrix} \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & \tilde{U} \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 1 & \\ \tilde{P}\boldsymbol{v}/\alpha & \tilde{L} \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & \tilde{U} \end{bmatrix}}_{U}. \end{aligned}$$

■

For stability one uses the permutation that always puts the largest in magnitude entry in the top row, eg., by a simple swap with the row corresponding to the diagonal. One could try to justify this by considering floating point rounding, but actually there is no guaranteed this will produce accurate results and indeed in the lab we given an example of a 'bad matrix′ where large errors are still produced. However, it is observed in practice that the probability of encountering a 'bad matrix′ is extremely small. The biggest open problem in numerical linear algebra is proving this observation rigorously.

Again, the above recursive proof encodes an inductive procedure, which we see in the following example.

Example 13 (PLU by-hand)**.** Consider the matrix

$$A = \begin{bmatrix} 0 & 2 & 1 \\ 2 & 6 & 2 \\ 1 & -1 & 5 \end{bmatrix}$$

The largest entry in the first column is 2 in the second row, hence we swap these rows then factor:

$$\underbrace{\begin{bmatrix} 0 & 1 & \\ 1 & 0 & \\ & & 1 \end{bmatrix}}_{P_1} A = \begin{bmatrix} 2 & 6 & 2 \\ 0 & 2 & 1 \\ 1 & -1 & 5 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & & \\ 0 & 1 & \\ 1/2 & 0 & 1 \end{bmatrix}}_{L_1} \begin{bmatrix} 2 & 6 & 2 \\ 0 & 2 & 1 \\ 0 & -4 & 4 \end{bmatrix}$$

Even though

$$A_2 := \begin{bmatrix} 2 & 1 \\ -4 & 4 \end{bmatrix}$$

is non-singular, we still permute the largest entry to the diagonal (this is helpful on a computer for stability). So we permute again to get:

$$\underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}}_{P_2} A_2 = \begin{bmatrix} -4 & 4 \\ 2 & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \\ -1/2 & 1 \end{bmatrix}}_{L_2} = \underbrace{\begin{bmatrix} -4 & 4 \\ & 3 \end{bmatrix}}_{U_2}$$

Putting it together we have

$$\begin{aligned} A &= P_1^\top L_1 \begin{bmatrix} \alpha_1 & \boldsymbol{w}_1^\top \\ & A_2 \end{bmatrix} = P_1^\top L_1 \begin{bmatrix} \alpha_1 & \boldsymbol{w}_1^\top \\ & P_2^\top L_2 U_2 \end{bmatrix} \\ &= P_1^\top \begin{bmatrix} 1 & \\ \boldsymbol{v}_1/\alpha_1 & I \end{bmatrix} \begin{bmatrix} 1 & \\ & P_2^\top L_2 \end{bmatrix} \begin{bmatrix} \alpha_1 & \boldsymbol{w}_1^\top \\ & U_2 \end{bmatrix} = P_1^\top \begin{bmatrix} 1 & \\ & P_2^\top \end{bmatrix} \begin{bmatrix} 1 & \\ P_2\boldsymbol{v}_1/\alpha_1 & L_2 \end{bmatrix} \begin{bmatrix} \alpha_1 & \boldsymbol{w}_1^\top \\ & U_2 \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}}_{P^\top} \underbrace{\begin{bmatrix} 1 & & \\ 1/2 & 1 & \\ 0 & -1/2 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 2 & 6 & 2 \\ & -4 & 4 \\ & & 3 \end{bmatrix}}_{U}. \end{aligned}$$

III.3 Cholesky factorisation

In the special case where A is a real square *symmetric positive definite* (SPD, that is $A \in \mathbb{R}^{n\times n}$ such that $A^\top = A$ and $\boldsymbol{x}^\top A\boldsymbol{x} > 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{x} \neq 0$) matrix the LU factorisation has a

special form called the *Cholesky factorisation*:

$$A = LL^\top,$$

i.e., $U = L^\top$, but now L does not necessarily have 1 on the diagonal. This provides an algorithmic way to *prove* that a matrix is symmetric positive definite, and is roughly twice as fast as the LU factorisation to compute.

Definition 16 (positive definite)**.** A square matrix $A \in \mathbb{R}^{n\times n}$ is *positive definite* if for all $\boldsymbol{x} \in \mathbb{R}^n, x \neq 0$ we have

$$\boldsymbol{x}^\top A\boldsymbol{x} > 0$$

First we establish some basic properties of positive definite matrices:

Proposition 5 (conjugating positive definite)**.** *If $A \in \mathbb{R}^{n\times n}$ is positive definite and $V \in \mathbb{R}^{n\times n}$ is non-singular then*

$$V^\top AV$$

is positive definite.

Proof

For all $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{x} \neq 0$, define $\boldsymbol{y} = V\boldsymbol{x} \neq 0$ (since V is non-singular). Thus we have

$$\boldsymbol{x}^\top V^\top AV\boldsymbol{x} = \boldsymbol{y}^\top A\boldsymbol{y} > 0.$$

■

Proposition 6 (diag positivity)**.** *If $A \in \mathbb{R}^{n\times n}$ is positive definite then its diagonal entries are positive: $a_{kk} > 0$.*

Proof

$$a_{kk} = \boldsymbol{e}_k^\top A\boldsymbol{e}_k > 0.$$

■

Lemma 4 (subslice positive definite)**.** *If $A \in \mathbb{R}^{n\times n}$ is positive definite then $A[2:n, 2:n] \in \mathbb{R}^{(n-1)\times(n-1)}$ is also positive definite.*

Proof For all $\boldsymbol{x} \in \mathbb{R}^{n-1}$, define $\boldsymbol{y} := [0, \boldsymbol{x}]$. Then we have

$$\boldsymbol{x}^\top A[2:n, 2:n]\boldsymbol{x} = \boldsymbol{y}^\top A\boldsymbol{y} > 0.$$

■

Here is the key result:

Theorem 6 (Cholesky and SPD)**.** *A matrix A is symmetric positive definite if and only if it has a Cholesky factorisation*

$$A = LL^\top$$

where L is lower triangular with positive diagonal entries.

Proof If A has a Cholesky factorisation it is symmetric ($A^\top = (LL^\top)^\top = A$) and for $\boldsymbol{x} \neq 0$ we have

$$\boldsymbol{x}^\top A\boldsymbol{x} = (L^\top \boldsymbol{x})^\top L^\top \boldsymbol{x} = \|L^\top \boldsymbol{x}\|^2 > 0$$

where we use the fact that L is non-singular.

For the other direction we will prove it by induction, with the 1×1 case being trivial. Assume all lower dimensional symmetric positive definite matrices have Cholesky decompositions. Modifying the LU factorisation slightly we write

$$A = \begin{bmatrix} \alpha & \boldsymbol{v}^\top \\ \boldsymbol{v} & K \end{bmatrix} = \underbrace{\begin{bmatrix} \sqrt{\alpha} & \\ \frac{\boldsymbol{v}}{\sqrt{\alpha}} & I \end{bmatrix}}_{L_1} \begin{bmatrix} 1 & \\ & K - \frac{\boldsymbol{v}\boldsymbol{v}^\top}{\alpha} \end{bmatrix} \underbrace{\begin{bmatrix} \sqrt{\alpha} & \frac{\boldsymbol{v}^\top}{\sqrt{\alpha}} \\ & I \end{bmatrix}}_{L_1^\top}.$$

Note that $A_2 := K - \frac{\boldsymbol{v}\boldsymbol{v}^\top}{\alpha}$ is a subslice of $L_1^{-1} A L_1^{-\top}$, hence by combining the previous propositions is itself SPD. Thus we can write

$$A_2 = K - \frac{\boldsymbol{v}\boldsymbol{v}^\top}{\alpha} = L_2 L_2^\top$$

and hence $A = LL^\top$ for

$$L = L_1 \begin{bmatrix} 1 & \\ & L_2 \end{bmatrix} = \begin{bmatrix} \sqrt{\alpha} & \\ \frac{\boldsymbol{v}}{\sqrt{\alpha}} & L_2 \end{bmatrix}.$$

■

Example 14 (Cholesky by hand)**.** Consider the matrix

$$A = \begin{bmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix}$$

Then $\alpha_1 = 2$, $\boldsymbol{v}_1 = [1, 1, 1]$, and

$$A_2 = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} - \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 3 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 3 \end{bmatrix}.$$

Continuing, we have $\alpha_2 = 3/2$, $\boldsymbol{v}_2 = [1/2, 1/2]$, and

$$A_3 = \frac{1}{2} \left(\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} - \frac{1}{3} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} \right) = \frac{1}{3} \begin{bmatrix} 4 & 1 \\ 1 & 4 \end{bmatrix}$$

Next, $\alpha_3 = 4/3$, $\boldsymbol{v}_3 = [1/3]$, and

$$A_4 = [4/3 - 3/4 * (1/3)^2] = [5/4]$$

i.e. $\alpha_4 = 5/4$.

Thus we get

$$L = \begin{bmatrix} \sqrt{\alpha_1} & & & \\ \frac{\boldsymbol{v}_1[1]}{\sqrt{\alpha_1}} & \sqrt{\alpha_2} & & \\ \frac{\boldsymbol{v}_1[2]}{\sqrt{\alpha_1}} & \frac{\boldsymbol{v}_2[1]}{\sqrt{\alpha_2}} & \sqrt{\alpha_3} & \\ \frac{\boldsymbol{v}_1[3]}{\sqrt{\alpha_1}} & \frac{\boldsymbol{v}_2[2]}{\sqrt{\alpha_2}} & \frac{\boldsymbol{v}_3[1]}{\sqrt{\alpha_3}} & \sqrt{\alpha_4} \end{bmatrix} = \begin{bmatrix} \sqrt{2} & & & \\ \frac{1}{\sqrt{2}} & \sqrt{\frac{3}{2}} & & \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & \frac{2}{\sqrt{3}} & \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{12}} & \frac{\sqrt{5}}{2} \end{bmatrix}$$

III.4 Orthogonal and Unitary Matrices

PLU factorisations are an effective scheme for inverting systems, however, we saw in the lab that for very special matrices it can fail to be accurate. In the next two sections we introduce an alternative approach that is guaranteed to be stable: factorise a matrix as

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is a *right-triangular matrix*, which for square matrices is another name for upper-triangular.

This factorisation is valid for rectangular matrices $A \in \mathbb{C}^{m\times n}$, where now *right-triangular* is a rectangular version of upper-triangular. For rectangular systems we can no longer solve linear systems of the form $A\boldsymbol{x} = \boldsymbol{b}$ (unless $\boldsymbol{b}$ lies in the column span of A) but instead we want to solve $A\boldsymbol{x} \approx \boldsymbol{b}$, where $\boldsymbol{x} \in \mathbb{C}^n$ and $\boldsymbol{b} \in \mathbb{C}^m$. More precisely, we can use a QR factorisation to solve *least squares* problems, find $\boldsymbol{x}$ that minimises the 2-norm:

$$\|A\boldsymbol{x} - \boldsymbol{b}\|$$

Before we discuss the computation of a QR factorisation and its role in solving least-squares problems, we introduce orthogonal and unitary matrices. In particular we will discuss reflections and rotations, which can be used to represent more general orthogonal matrices.

Definition 17 (orthogonal/unitary matrix)**.** A square real matrix is *orthogonal* if its inverse is its transpose:

$$O(n) = \{Q \in \mathbb{R}^{n\times n} : Q^\top Q = I\}$$

A square complex matrix is *unitary* if its inverse is its adjoint:

$$U(n) = \{Q \in \mathbb{C}^{n\times n} : Q^\star Q = I\}.$$

Here the adjoint is the same as the conjugate-transpose: $Q^\star := \bar{Q}^\top$.

Note that $O(n) \subset U(n)$ as for real matrices $Q^\star = Q^\top$. Because in either case $Q^{-1} = Q^\star$ we also have $QQ^\star = I$ (which for real matrices is $QQ^\top = I$). These matrices are particularly important for numerical linear algebra for a number of reasons (we'll explore these properties in the problem sheets):

1. They are norm-preserving: for any vector $\boldsymbol{x} \in \mathbb{C}^n$ and $Q \in U(n)$ we have $\|Q\boldsymbol{x}\| = \|\boldsymbol{x}\|$ where $\|\boldsymbol{x}\|^2 := \sum_{k=1}^n x_k^2$ (i.e. the 2-norm).
2. All eigenvalues have absolute value equal to 1.
3. For $Q \in O(n)$, $\det Q = \pm 1$.
4. They are trivially invertible (just take the adjoint).
5. They are generally "stable": errors due to rounding when multiplying a vector by Q are controlled.
6. They are *normal matrices*: they commute with their adjoint ($QQ^\star = QQ^\star$).
7. Both $O(n)$ and $U(n)$ are groups, in particular, they are closed under multiplication.

On a computer there are multiple ways of representing orthogonal/unitary matrices. The obvious way is to store entries as a dense matrix, however, this is very inefficient. In the appendices we have seen permutation matrices, which are a special type of orthogonal matrices where we can store only the order the entries are permuted as a vector.

More generally, we will use the group structure: represent general orthogonal/unitary matrices as products of simpler elements of the group. In partular we will use two building blocks:

1. *Rotations*: Rotations are equivalent to special orthogonal matrices $SO(2)$ and correspond to rotations in 2D.
2. *Reflections*: Reflections are elements of $U(n)$ that are defined in terms of a single unit vector $\boldsymbol{v} \in \mathbb{C}^n$ which is reflected.

We remark a related concept to orthogonal/unitary matrices are rectangular matrices with orthonormal columns, e.g.

$$U = [\boldsymbol{u}_1| \cdots |\boldsymbol{u}_n] \in \mathbb{C}^{m\times n}$$

where $m \geq n$ such that $U^\star U = I_n$ (the $n \times n$ identity matrix). In the case where $m > n$ we must have $UU^\star \neq I_m$ as the rank of U is $n < m$.

III.4.1 Rotations

We begin with a general definition:

Definition 18 (Special Orthogonal and Rotations)**.** *Special Orthogonal Matrices* are

$$SO(n) := \{Q \in O(n) | \det Q = 1\}$$

And (simple) *rotations* are $SO(2)$.

In what follows we use the following for writing the angle of a vector:

Definition 19 (two-arg arctan)**.** The two-argument arctan function gives the angle θ through the point $[a, b]^\top$, i.e.,

$$\sqrt{a^2 + b^2} \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}.$$

It can be defined in terms of the standard arctan as follows:

$$\operatorname{atan}(b, a) := \begin{cases} \operatorname{atan}\frac{b}{a} & a > 0 \\ \operatorname{atan}\frac{b}{a} + \pi & a < 0 \text{ and } b > 0 \\ \operatorname{atan}\frac{b}{a} - \pi & a < 0 \text{ and } b < 0 \\ \pi/2 & a = 0 \text{ and } b > 0 \\ -\pi/2 & a = 0 \text{ and } b < 0 \end{cases}$$

We show $SO(2)$ are exactly equivalent to standard rotations:

Proposition 7 (simple rotation)**.** *A 2×2* rotation matrix *through angle θ is*

$$Q_\theta := \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

We have $Q \in SO(2)$ if and only if $Q = Q_\theta$ for some $\theta \in \mathbb{R}$.

Proof

First assume Q_θ is of that form and write $c = \cos\theta$ and $s = \sin\theta$. Then we have

$$Q_\theta^\top Q_\theta = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = \begin{pmatrix} c^2 + s^2 & 0 \\ 0 & c^2 + s^2 \end{pmatrix} = I$$

and $\det Q_\theta = c^2 + s^2 = 1$ hence $Q_\theta \in SO(2)$.

Now suppose $Q = [\boldsymbol{q}_1, \boldsymbol{q}_2] \in SO(2)$ where we know its columns have norm 1, i.e. $\|\boldsymbol{q}_k\| = 1$, and are orthogonal. Write $\boldsymbol{q}_1 = [c, s]$ where we know $c = \cos\theta$ and $s = \sin\theta$ for $\theta = \operatorname{atan}(s, c)$. Since $\boldsymbol{q}_1 \cdot \boldsymbol{q}_2 = 0$ we can deduce $\boldsymbol{q}_2 = \pm[-s, c]$. The sign is positive as $\det Q = \pm(c^2 + s^2) = \pm 1$.

■

We can rotate an arbitrary vector in $\mathbb{R}^2$ to the unit axis using rotations, which are useful in linear algebra decompositions. Interestingly it only requires basic algebraic functions (no trigonometric functions):

Proposition 8 (rotation of a vector)**.** *The matrix*

$$Q = \frac{1}{\sqrt{a^2 + b^2}} \begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

is a rotation matrix ($Q \in SO(2)$) satisfying

$$Q \begin{bmatrix} a \\ b \end{bmatrix} = \sqrt{a^2 + b^2} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Proof

The last equation is trivial so the only question is that it is a rotation matrix. This follows immediately:

$$Q^\top Q = \frac{1}{a^2 + b^2} \begin{bmatrix} a^2 + b^2 & 0 \\ 0 & a^2 + b^2 \end{bmatrix} = I$$

and $\det Q = 1$.

■

Example 15 (rotating a vector)**.** Consider the vector

$$\boldsymbol{x} = \begin{bmatrix} -1 \\ -\sqrt{3} \end{bmatrix}.$$

We can use the proposition above to deduce the rotation matrix that rotates this vector to the positive real axis is:

$$\frac{1}{\sqrt{1+3}} \begin{bmatrix} -1 & -\sqrt{3} \\ \sqrt{3} & -1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -1 & -\sqrt{3} \\ \sqrt{3} & -1 \end{bmatrix}.$$

Alternatively, we could determine the matrix by computing the angle of the vector via:

$$\theta = \operatorname{atan}(-\sqrt{3}, -1) = \operatorname{atan}(\sqrt{3}) - \pi = -\frac{2\pi}{3}.$$

We thus compute:

$$Q_{-\theta} = \begin{bmatrix} \cos(2\pi/3) & -\sin(2\pi/3) \\ \sin(2\pi/3) & \cos(2\pi/3) \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -1 & -\sqrt{3} \\ \sqrt{3} & -1 \end{bmatrix}.$$

More generally, we can consider rotations that operate on two entries of a vector at a time. This will be explored in the problem sheet/lab.

III.4.2 Reflections

In addition to rotations, another type of orthogonal/unitary matrix are reflections. These are specified by a single vector which is reflected, with everything orthogonal to the vector left fixed.

Definition 20 (reflection matrix)**.** Given a unit vector $\boldsymbol{v} \in \mathbb{C}^n$ (satisfying $\|\boldsymbol{v}\| = 1$), define the corresponding *reflection matrix* as:

$$Q_{\boldsymbol{v}} := I - 2\boldsymbol{v}\boldsymbol{v}^\star$$

These are indeed reflections in the direction of $\boldsymbol{v}$. We can show this as follows:

Proposition 9 (Householder properties)**.** *$Q_{\boldsymbol{v}}$ satisfies:*

1. *Symmetry: $Q_{\boldsymbol{v}} = Q_{\boldsymbol{v}}^\star$*
2. *Orthogonality: $Q_{\boldsymbol{v}} \in U(n)$*
3. *The vector $\boldsymbol{v}$ is an eigenvector of $Q_{\boldsymbol{v}}$ with eigenvalue -1*
4. *For the dimension $n-1$ space $W := \{\boldsymbol{w} : \boldsymbol{w}^\star\boldsymbol{v} = 0\}$, all vectors $\boldsymbol{w} \in W$ satisfy $Q_{\boldsymbol{v}}\boldsymbol{w} = \boldsymbol{w}$.*
5. *Not a rotation: $\det Q_{\boldsymbol{v}} = -1$*

Proof

Property 1 follows immediately. Property 2 follows from

$$Q_{\boldsymbol{v}}^\star Q_{\boldsymbol{v}} = Q_{\boldsymbol{v}}^2 = I - 4\boldsymbol{v}\boldsymbol{v}^\star + 4\boldsymbol{v}\boldsymbol{v}^\star\boldsymbol{v}\boldsymbol{v}^\star = I.$$

Property 3 follows since

$$Q_{\boldsymbol{v}}\boldsymbol{v} = \boldsymbol{v} - 2\boldsymbol{v}(\boldsymbol{v}^\star\boldsymbol{v}) = -\boldsymbol{v}.$$

Property 4 follows from:

$$Q_{\boldsymbol{v}}\boldsymbol{w} = \boldsymbol{w} - 2\boldsymbol{v}(\boldsymbol{w}^\star\boldsymbol{v}) = \boldsymbol{w}$$

Property 5 then follows: Property 4 tells us that 1 is an eigenvalue with multiplicity $n-1$. Since -1 is an eigenvalue with multiplicity 1, the determinant, which is product of the eigenvalues, is -1.

■

Example 16 (reflection through 2-vector)**.** Consider reflection through $\boldsymbol{x} = [1, 2]^\top$. We first need to normalise $\boldsymbol{x}$:

$$\boldsymbol{v} = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|} = \begin{bmatrix} \frac{1}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} \end{bmatrix}$$

The reflection matrix is:

$$Q_{\boldsymbol{v}} = I - 2\boldsymbol{v}\boldsymbol{v}^\top = \begin{bmatrix} 1 & \\ & 1 \end{bmatrix} - \frac{2}{5}\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} = \frac{1}{5}\begin{bmatrix} 3 & -4 \\ -4 & -3 \end{bmatrix}$$

Indeed it is symmetric, and orthogonal. It sends $\boldsymbol{x}$ to $-\boldsymbol{x}$:

$$Q_{\boldsymbol{v}}\boldsymbol{x} = \frac{1}{5}\begin{bmatrix} 3-8 \\ -4-6 \end{bmatrix} = -\boldsymbol{x}$$

Any vector orthogonal to $\boldsymbol{x}$, like $\boldsymbol{y} = [-2, 1]^\top$, is left fixed:

$$Q_{\boldsymbol{v}}\boldsymbol{y} = \frac{1}{5}\begin{bmatrix} -6-4 \\ 8-3 \end{bmatrix} = \boldsymbol{y}$$

Note that *building* the matrix $Q_{\boldsymbol{v}}$ will be expensive ($O(n^2)$ operations), but we can *apply* $Q_{\boldsymbol{v}}$ to a vector in $O(n)$ operations using the expression:

$$Q_{\boldsymbol{v}}\boldsymbol{x} = \boldsymbol{x} - 2\boldsymbol{v}(\boldsymbol{v}^\star\boldsymbol{x}) = \boldsymbol{x} - 2\boldsymbol{v}(\boldsymbol{v}\cdot\boldsymbol{x}).$$

Householder reflections

Just as rotations can be used to rotate vectors to be aligned with coordinate axes, so can reflections, but in this case it works for vectors in $\mathbb{C}^n$, not just $\mathbb{R}^2$. We begin with the real case:

Definition 21 (Householder reflection, real case)**.** For a given vector $\boldsymbol{x} \in \mathbb{R}^n$, define the Householder reflection

$$Q_{\boldsymbol{x}}^{\pm,\mathrm{H}} := Q_{\boldsymbol{w}}$$

for $\boldsymbol{y} = \mp\|\boldsymbol{x}\|\boldsymbol{e}_1 + \boldsymbol{x}$ and $\boldsymbol{w} = \frac{\boldsymbol{y}}{\|\boldsymbol{y}\|}$. The default choice in sign is:

$$Q_{\boldsymbol{x}}^{\mathrm{H}} := Q_{\boldsymbol{x}}^{-\mathrm{sign}(x_1),\mathrm{H}}.$$

Lemma 5 (Householder reflection maps to axis)**.** *For $\boldsymbol{x} \in \mathbb{R}^n$,*

$$Q_{\boldsymbol{x}}^{\pm,\mathrm{H}}\boldsymbol{x} = \pm\|\boldsymbol{x}\|\boldsymbol{e}_1$$

Proof Note that

$$\|\boldsymbol{y}\|^2 = 2\|\boldsymbol{x}\|^2 \mp 2\|\boldsymbol{x}\|x_1,$$
$$\boldsymbol{y}^\top\boldsymbol{x} = \|\boldsymbol{x}\|^2 \mp \|\boldsymbol{x}\|x_1$$

where $x_1 = \boldsymbol{e}_1^\top\boldsymbol{x}$. Therefore:

$$Q_{\boldsymbol{x}}^{\pm,\mathrm{H}}\boldsymbol{x} = (I - 2\boldsymbol{w}\boldsymbol{w}^\top)\boldsymbol{x} = \boldsymbol{x} - 2\frac{\boldsymbol{y}\|\boldsymbol{x}\|}{\|\boldsymbol{y}\|^2}(\|\boldsymbol{x}\| \mp x_1) = \boldsymbol{x} - \boldsymbol{y} = \pm\|\boldsymbol{x}\|\boldsymbol{e}_1.$$

■

Remark Why do we choose the the opposite sign of x_1 for the default reflection? For stability, but we won't discuss this in more detail.

We can extend this definition for complex vectors. In this case the choice of the sign is delicate and so we only generalise the default choice using a complex-analogue of the sign fuunction.

Definition 22 (Householder reflection, complex case)**.** For a given vector $\boldsymbol{x} \in \mathbb{C}^n$, define the Householder reflection as

$$Q^{\mathrm{H}}_{\boldsymbol{x}} := Q_{\boldsymbol{w}}$$

for $\boldsymbol{y} = \mathrm{csign}(x_1)\|\boldsymbol{x}\|\boldsymbol{e}_1 + \boldsymbol{x}$ and $\boldsymbol{w} = \frac{\boldsymbol{y}}{\|\boldsymbol{y}\|}$, for $\mathrm{csign}(z) = \mathrm{e}^{\mathrm{i}\arg z}$.

Lemma 6 (Householder reflection maps to axis, complex case)**.** *For $\boldsymbol{x} \in \mathbb{C}^n$,*

$$Q^{\mathrm{H}}_{\boldsymbol{x}}\boldsymbol{x} = -\mathrm{csign}(x_1)\|\boldsymbol{x}\|\boldsymbol{e}_1$$

Proof Denote $\alpha := \mathrm{csign}(x_1)$. Note that $\bar{\alpha}x_1 = \mathrm{e}^{-\mathrm{i}\arg x_1}x_1 = |x_1|$. Now we have

$$\begin{aligned}\|\boldsymbol{y}\|^2 &= (\alpha\|\boldsymbol{x}\|\boldsymbol{e}_1 + \boldsymbol{x})^\star(\alpha\|\boldsymbol{x}\|\boldsymbol{e}_1 + \boldsymbol{x}) = |\alpha|\|\boldsymbol{x}\|^2 + \|\boldsymbol{x}\|\alpha\bar{x}_1 + \bar{\alpha}x_1\|\boldsymbol{x}\| + \|\boldsymbol{x}\|^2 \\ &= 2\|\boldsymbol{x}\|^2 + 2|x_1|\|\boldsymbol{x}\| \\ \boldsymbol{y}^\star\boldsymbol{x} &= \bar{\alpha}x_1\|\boldsymbol{x}\| + \|\boldsymbol{x}\|^2 = \|\boldsymbol{x}\|^2 + |x_1|\|\boldsymbol{x}\|\end{aligned}$$

Therefore:

$$Q^{\mathrm{H}}_{\boldsymbol{x}}\boldsymbol{x} = (I - 2\boldsymbol{w}\boldsymbol{w}^\star)\boldsymbol{x} = \boldsymbol{x} - 2\frac{\boldsymbol{y}}{\|\boldsymbol{y}\|^2}(\|\boldsymbol{x}\|^2 + |x_1|\|\boldsymbol{x}\|) = \boldsymbol{x} - \boldsymbol{y} = -\alpha\|\boldsymbol{x}\|\boldsymbol{e}_1.$$

■

III.5 QR Factorisation

Let $A \in \mathbb{C}^{m\times n}$ be a rectangular or square matrix such that $m \geq n$ (i.e. more rows then columns). In this section we consider two closely related factorisations:

Definition 23 (QR factorisation)**.** The *QR factorisation* is

$$A = QR = \underbrace{\begin{bmatrix}\boldsymbol{q}_1|\cdots|\boldsymbol{q}_m\end{bmatrix}}_{Q\in U(m)} \underbrace{\begin{bmatrix}\times & \cdots & \times \\ & \ddots & \vdots \\ & & \times \\ & & 0 \\ & & \vdots \\ & & 0\end{bmatrix}}_{R\in\mathbb{C}^{m\times n}}$$

where Q is unitary (i.e., $Q \in U(m)$, satisfying $Q^\star Q = I$, with columns $\boldsymbol{q}_j \in \mathbb{C}^m$) and R is *right triangular*, which means it is only nonzero on or to the right of the diagonal ($r_{kj} = 0$ if $k > j$).

Definition 24 (Reduced QR factorisation)**.** The *reduced QR factorisation*

$$A = \hat{Q}\hat{R} = \underbrace{\begin{bmatrix}\boldsymbol{q}_1|\cdots|\boldsymbol{q}_n\end{bmatrix}}_{\hat{Q}\in\mathbb{C}^{m\times n}} \underbrace{\begin{bmatrix}\times & \cdots & \times \\ & \ddots & \vdots \\ & & \times\end{bmatrix}}_{\hat{R}\in\mathbb{C}^{n\times n}}$$

where $\hat{Q}$ has orthonormal columns ($\hat{Q}^\star\hat{Q} = I$, $\boldsymbol{q}_j \in \mathbb{C}^m$) and $\hat{R}$ is upper triangular.

Note for a square matrix the reduced QR factorisation is equivalent to the QR factorisation, in which case R is *upper triangular*. The importance of these factorisation for square matrices is that their component pieces are easy to invert:

$$A = QR \qquad \Rightarrow \qquad A^{-1}\boldsymbol{b} = R^{-1}Q^\top \boldsymbol{b}$$

and we saw previously that triangular and orthogonal matrices are easy to invert when applied to a vector $\boldsymbol{b}$.

For rectangular matrices we will see that the QR factorisation leads to efficient solutions to the *least squares problem*: find $\boldsymbol{x}$ that minimizes the 2-norm $\|A\boldsymbol{x} - \boldsymbol{b}\|$. Note in the rectangular case the QR factorisation contains within it the reduced QR factorisation:

$$A = QR = \begin{bmatrix}\hat{Q}|\boldsymbol{q}_{n+1}|\cdots|\boldsymbol{q}_m\end{bmatrix}\begin{bmatrix}\hat{R}\\ \mathbf{0}_{m-n\times n}\end{bmatrix} = \hat{Q}\hat{R}.$$

In this section we discuss the following:

1. Reduced QR and Gram–Schmidt: We discuss computation of the Reduced QR factorisation using Gram–Schmidt.
2. Householder reflections and QR: We discuss computing the QR factorisation using Householder reflections. This is a more accurate approach for computing QR factorisations.
3. QR and least squares: We discuss the QR factorisation and its usage in solving least squares problems.

III.5.1 Reduced QR and Gram–Schmidt

How do we compute the QR factorisation? We begin with a method you may have seen before in another guise. Write

$$A = \begin{bmatrix}\boldsymbol{a}_1|\cdots|\boldsymbol{a}_n\end{bmatrix}$$

where $\boldsymbol{a}_k \in \mathbb{C}^m$ and assume they are linearly independent (A has full column rank).

Proposition 10 (Column spaces match)**.** *Suppose $A = \hat{Q}\hat{R}$ where $\hat{Q} = [\boldsymbol{q}_1|\ldots|\boldsymbol{q}_n]$ has orthonormal columns and $\hat{R}$ is upper-triangular, and A has full rank. Then the first j columns of $\hat{Q}$ span the same space as the first j columns of A:*

$$span(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_j) = span(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_j).$$

Proof

Because A has full rank we know $\hat{R}$ is invertible, i.e. its diagonal entries do not vanish: $r_{jj} \neq 0$. If $\boldsymbol{v} \in \text{span}(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_j)$ we have for $\boldsymbol{c} \in \mathbb{C}^j$

$$\boldsymbol{v} = \begin{bmatrix}\boldsymbol{a}_1|\cdots|\boldsymbol{a}_j\end{bmatrix}\boldsymbol{c} = \begin{bmatrix}\boldsymbol{q}_1|\cdots|\boldsymbol{q}_j\end{bmatrix}\hat{R}[1:j,1:j]\boldsymbol{c} \in \text{span}(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_j)$$

while if $\boldsymbol{w} \in \text{span}(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_j)$ we have for $\boldsymbol{d} \in \mathbb{R}^j$

$$\boldsymbol{w} = \begin{bmatrix}\boldsymbol{q}_1|\cdots|\boldsymbol{q}_j\end{bmatrix}\boldsymbol{d} = \begin{bmatrix}\boldsymbol{a}_1|\cdots|\boldsymbol{a}_j\end{bmatrix}\hat{R}[1:j,1:j]^{-1}\boldsymbol{d} \in \text{span}(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_j).$$

■

It is possible to find $\hat{Q}$ and $\hat{R}$ using the *Gram–Schmidt algorithm*. We construct it column-by-column. For $j = 1, 2, \ldots, n$ define

$$\begin{aligned} \boldsymbol{v}_j &:= \boldsymbol{a}_j - \sum_{k=1}^{j-1} \underbrace{\boldsymbol{q}_k^\star \boldsymbol{a}_j}_{r_{kj}} \boldsymbol{q}_k, \\ r_{jj} &:= \|\boldsymbol{v}_j\|, \\ \boldsymbol{q}_j &:= \frac{\boldsymbol{v}_j}{r_{jj}}. \end{aligned}$$

Theorem (Gram–Schmidt and reduced QR) Define $\boldsymbol{q}_j$ and r_{kj} as above (with $r_{kj} = 0$ if $k > j$). Then a reduced QR factorisation is given by:

$$A = \underbrace{\begin{bmatrix} \boldsymbol{q}_1 | \cdots | \boldsymbol{q}_n \end{bmatrix}}_{\hat{Q} \in \mathbb{C}^{m \times n}} \underbrace{\begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{bmatrix}}_{\hat{R} \in \mathbb{C}^{n \times n}}$$

Proof

We first show that $\hat{Q}$ has orthonormal columns. Assume that $\boldsymbol{q}_\ell^\star \boldsymbol{q}_k = \delta_{\ell k}$ for $k, \ell < j$. For $\ell < j$ we then have

$$\boldsymbol{q}_\ell^\star \boldsymbol{v}_j = \boldsymbol{q}_\ell^\star \boldsymbol{a}_j - \sum_{k=1}^{j-1} \boldsymbol{q}_\ell^\star \boldsymbol{q}_k \boldsymbol{q}_k^\star \boldsymbol{a}_j = 0$$

hence $\boldsymbol{q}_\ell^\star \boldsymbol{q}_j = 0$ and indeed $\hat{Q}$ has orthonormal columns. Further: from the definition of $\boldsymbol{v}_j$ we find

$$\boldsymbol{a}_j = \boldsymbol{v}_j + \sum_{k=1}^{j-1} r_{kj} \boldsymbol{q}_k = \sum_{k=1}^{j} r_{kj} \boldsymbol{q}_k = \hat{Q} \hat{R} \boldsymbol{e}_j$$

■

III.5.2 Householder reflections and QR

As an alternative, we will consider using Householder reflections to introduce zeros below the diagonal. Thus, if Gram–Schmidt is a process of *triangular orthogonalisation* (using triangular matrices to orthogonalise), Householder reflections is a process of *orthogonal triangularisation* (using orthogonal matrices to triangularise).

Consider multiplication by the Householder reflection corresponding to the first column, that is, for

$$Q_1 := Q_{\boldsymbol{a}_1}^{\mathrm{H}},$$

consider

$$Q_1 A = \begin{bmatrix} \times & \times & \cdots & \times \\ & \times & \cdots & \times \\ & \vdots & \ddots & \vdots \\ & \times & \cdots & \times \end{bmatrix} = \begin{bmatrix} \alpha_1 & \boldsymbol{w}_1^\top \\ & A_2 \end{bmatrix}$$

where

$$\alpha_1 := -\mathrm{csign}(a_{11}) \|\boldsymbol{a}_1\|, \boldsymbol{w}_1 = (Q_1 A)[1, 2:n] \qquad \text{and} \qquad A_2 = (Q_1 A)[2:m, 2:n],$$

where as before $\text{csign}(z) := \mathrm{e}^{\mathrm{i}\arg z}$. That is, we have made the first column triangular. In terms of an algorithm, we then introduce zeros into the first column of A_2, leaving an A_3, and so-on. But we can wrap this iterative algorithm into a simple proof by induction, reminisicent of our proofs for the PLU and Cholesky factorisations:

Theorem 7 (QR)**.** *Every matrix $A \in \mathbb{C}^{m\times n}$ has a QR factorisation:*

$$A = QR$$

where $Q \in U(m)$ and $R \in \mathbb{C}^{m\times n}$ is right triangular.

Proof

First assume $m \geq n$. If $A = [\boldsymbol{a}_1] \in \mathbb{C}^{m\times 1}$ then we have for the Householder reflection $Q_1 = Q^{\mathrm{H}}_{\boldsymbol{a}_1}$

$$Q_1 A = \alpha \boldsymbol{e}_1$$

which is right triangular, where $\alpha = -\text{csign}(a_{11})\|\boldsymbol{a}_1\|$. In other words

$$A = \underbrace{Q_1}_{Q}\underbrace{\alpha \boldsymbol{e}_1}_{R}.$$

For $n > 1$, assume every matrix with less columns than n has a QR factorisation. For $A = [\boldsymbol{a}_1|\dots|\boldsymbol{a}_n] \in \mathbb{C}^{m\times n}$, let $Q_1 = Q^{\mathrm{H}}_{\boldsymbol{a}_1}$ so that

$$Q_1 A = \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & A_2 \end{bmatrix}.$$

By assumption $A_2 = Q_2 R_2$. Thus we have (recalling that $Q_1^{-1} = Q_1^\star = Q_1$):

$$\begin{aligned} A &= Q_1 \begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & Q_2 R_2 \end{bmatrix} \\ &= \underbrace{Q_1 \begin{bmatrix} 1 & \\ & Q_2 \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} \alpha & \boldsymbol{w}^\top \\ & R_2 \end{bmatrix}}_{R}. \end{aligned}$$

If $m < n$, i.e., A has more columns then rows, write

$$A = \begin{bmatrix} \tilde{A} & B \end{bmatrix}$$

where $\tilde{A} \in \mathbb{C}^{m\times m}$. From above we know we can write $\tilde{A} = Q\tilde{R}$. We thus have

$$A = Q \underbrace{\begin{bmatrix} \tilde{R} & Q^\star B \end{bmatrix}}_{R}$$

where R is right triangular.

■

Example 17 (QR by hand)**.** We will now do an example by hand. Consider finding the QR factorisation where the diagonal of R is positive for the 4×3 matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

For the first column, since the entry $a_{11} > 0$ on a computer we would want to choose the Householder reflection that makes this negative, but in this case we want R to have a positive diagonal (partly because the numbers involved become very complicated otherwise!). So instead we choose the "wrong" sign and leave it positive. Since $\|\boldsymbol{a}_1\| = 2$ we have

$$\boldsymbol{y}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix} - \begin{bmatrix} 2 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \Rightarrow \boldsymbol{w}_1 = \frac{\boldsymbol{y}_1}{\|\boldsymbol{y}_1\|} = \frac{1}{2}\begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}.$$

Hence

$$Q_1 := I - \frac{1}{2}\begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}\begin{bmatrix} -1 & -1 & -1 & -1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

so that

$$Q_1 A = \begin{bmatrix} 2 & 1 & 0 \\ & 0 & 0 \\ & -1 & -1 \\ & 0 & -1 \end{bmatrix}$$

For the second column we have a zero entry so on a computer we can either send it to positive or negative sign, but in this case we are told to make it positive. Thus we have

$$\boldsymbol{y}_2 := [0, -1, 0] - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix} \Rightarrow \boldsymbol{w}_2 = \frac{\boldsymbol{y}_2}{\|\boldsymbol{y}_2\|} = \frac{1}{\sqrt{2}}\begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix}$$

Thus we have

$$Q_2 := I - \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix}\begin{bmatrix} -1 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

so that

$$\tilde{Q}_2 Q_1 A = \begin{bmatrix} 2 & 1 & 0 \\ & 1 & 1 \\ & & 0 \\ & & -1 \end{bmatrix}$$

The final vector is

$$\boldsymbol{y}_3 := \begin{bmatrix} 0 \\ -1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \Rightarrow \boldsymbol{w}_3 = -\frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Hence

$$Q_3 := I - \begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$$

so that

$$\tilde{Q}_3 \tilde{Q}_2 Q_1 A = \begin{bmatrix} 2 & 1 & 0 \\ & 1 & 1 \\ & & 1 \\ & & 0 \end{bmatrix} =: R$$

and

$$Q := Q_1 \tilde{Q}_2 \tilde{Q}_3 = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & 1 & -1 & -1 \end{bmatrix}.$$

III.5.3 QR and least squares

We consider rectangular matrices with more rows than columns. Given $A \in \mathbb{C}^{m\times n}$ and $\boldsymbol{b} \in \mathbb{C}^m$, a least squares problem consists of finding a vector $\boldsymbol{x} \in \mathbb{C}^n$ that minimises the 2-norm: $\|A\boldsymbol{x} - \boldsymbol{b}\|$. There is a lot of theory around least squares, however, we focus on a simple computational aspect: we can solve least squares problems using the QR factorisation.

Theorem 8 (least squares via QR)**.** *Suppose $A \in \mathbb{C}^{m\times n}$ with $m \geq n$ has full rank and a QR factorisation $A = QR$ (which includes within it a reduced QR factorisation $A = \hat{Q}\hat{R}$). The vector*

$$\boldsymbol{x} = \hat{R}^{-1}\hat{Q}^\star \boldsymbol{b}$$

minimises $\|A\boldsymbol{x} - \boldsymbol{b}\|$.

Proof

The norm-preserving property ($\|Q\boldsymbol{x}\| = \|\boldsymbol{x}\|$) of unitary matrices tells us

$$\|A\boldsymbol{x} - \boldsymbol{b}\| = \|QR\boldsymbol{x} - \boldsymbol{b}\| = \|Q(R\boldsymbol{x} - Q^\star\boldsymbol{b})\| = \|R\boldsymbol{x} - Q^\star\boldsymbol{b}\| = \left\| \begin{bmatrix} \hat{R} \\ \mathbf{0}_{m-n\times n} \end{bmatrix} \boldsymbol{x} - \begin{bmatrix} \hat{Q}^\star \\ \boldsymbol{q}_{n+1}^\star \\ \vdots \\ \boldsymbol{q}_m^\star \end{bmatrix} \boldsymbol{b} \right\|$$

Now note that the rows $k > n$ are independent of $\boldsymbol{x}$ and are a fixed contribution. Thus to minimise this norm it suffices to drop them and minimise:

$$\|\hat{R}\boldsymbol{x} - \hat{Q}^\star\boldsymbol{b}\|$$

This norm is minimised if it is zero. Provided the column rank of A is full, $\hat{R}$ will be invertible.

Chapter IV

Linear Algebra Applications

Numerical linear algebra underlies many numerical methods in applications, from simulating fluids, to understanding data and neural networks. Here we briefly investigate some applications, allowing us to go beyond our preliminary numerical algorithms from Chapter I, giving methods for approximating functions, a more powerful numerical method for computing integrals, and the ability to solve ordinary differential equations.

1. IV.1 Polynomial Interpolation and Regression: Often in data science one needs to approximate data by a polynomial. We discuss polynomial interpolation and see how it can be used to compute integrals. We also discuss regression, where more data is used than the degree of the polynomial, leading to a robust approach produced by solving a rectangular least squares problem.

2. IV.2 Differential Equations: Divided differences can can be used to *discretise* differential equations. That is, we can recast differential equation as (approximate) solutions to linear systems. We investigate using this approach on some simple linear initial and boundary value problems.

IV.1 Polynomial Interpolation and Regression

Polynomial interpolation is the process of finding a polynomial that equals data at a precise set of points. In this section we see how an interpolant can be constructed by either solving a linear system involving the Vandermonde matrix, or directly in terms of the Lagrange basis for polynomials. We also investigate an application of polynomial interpolation to computing integrals, giving an alternative to the rectangular and triangular rules from the first chapter. In the lab we see that this leads to much more accurate computation. We also see in the lab that polynomial interpolation has issues, particular with an evenly spaced grid or with a monomial basis. Overcoming this will motivate orthogonal polynomials later in the module.

A more robust scheme that overcomes some of the issues with naive polynomial interpolation is *polynomial regression*, where we use more data than the degrees of freedom in the polynomial. We can determine such a polynomial by solving a *least squares peroblem*: instead of insisting that the polynomial matches data exactly, we find the polynomial whose samples at the points are as close as possible to the data, as measured in the 2-norm.

IV.1.1 Polynomial interpolation

Our prelimary goal is given a set of points and data at those points, usually samples of a function $f_j = f(x_j)$, find a polynomial that interpolates the data at the points:

Definition 25 (interpolatory polynomial)**.** Given *distinct* points $\boldsymbol{x} = [x_1, \dots, x_n]^\top \in \mathbb{C}^n$ and *data* $\boldsymbol{f} = [f_1, \dots, f_n]^\top \in \mathbb{C}^n$, a degree $n-1$ *interpolatory polynomial* $p(x)$ satisfies

$$p(x_j) = f_j$$

The easiest way to solve this problem is to invert the Vandermonde system:

Definition 26 (Vandermonde)**.** The *Vandermonde matrix* associated with $\boldsymbol{x} \in \mathbb{C}^m$ is the matrix

$$V_{\boldsymbol{x},n} := \begin{bmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & \cdots & x_m^{n-1} \end{bmatrix} \in \mathbb{C}^{m \times n}.$$

When it is clear from context we omit the subscripts $\boldsymbol{x}, n$.

Writing the coefficients of a polynomial

$$p(x) = \sum_{k=0}^{n-1} c_k x^k$$

as a vector $\boldsymbol{c} = [c_0, \dots, c_{n-1}]^\top \in \mathbb{C}^n$, we note that V encodes the linear map from coefficients to values at a grid, that is,

$$V\boldsymbol{c} = \begin{bmatrix} c_0 + c_1 x_1 + \cdots + c_{n-1} x_1^{n-1} \\ \vdots \\ c_0 + c_1 x_m + \cdots + c_{n-1} x_m^{n-1} \end{bmatrix} = \begin{bmatrix} p(x_1) \\ \vdots \\ p(x_m) \end{bmatrix}.$$

In the square case (where $m = n$), the coefficients of an interpolatory polynomial are given by $\boldsymbol{c} = V^{-1}\boldsymbol{f}$, so that

$$\begin{bmatrix} p(x_1) \\ \vdots \\ p(x_n) \end{bmatrix} = V\boldsymbol{c} = VV^{-1}\boldsymbol{f} = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}.$$

This inversion is justified by the following:

Proposition 11 (interpolatory polynomial uniqueness)**.** *Interpolatory polynomials are unique and therefore square Vandermonde matrices are invertible.*

Proof Suppose p and $\tilde{p}$ are both interpolatory polynomials of the same function. Then $p(x) - \tilde{p}(x)$ vanishes at n distinct points x_j. By the fundamental theorem of algebra it must be zero, i.e., $p = \tilde{p}$.

For the second part, if $V\boldsymbol{c} = 0$ for $\boldsymbol{c} = [c_0, \dots, c_{n-1}]^\top \in \mathbb{C}^n$ then for $q(x) = c_0 + \cdots + c_{n-1}x^{n-1}$ we have

$$q(x_j) = \boldsymbol{e}_j^\top V\boldsymbol{c} = 0$$

hence q vanishes at n distinct points and is therefore 0, i.e., $\boldsymbol{c} = 0$.

■

We can invert square Vandermonde matrix numerically in $O(n^3)$ operations using the PLU factorisation. But it turns out we can also construct the interpolatory polynomial directly, and evaluate the polynomial in only $O(n^2)$ operations. We will use the following polynomials which equal 1 at one grid point and zero at the others:

Definition 27 (Lagrange basis polynomial)**.** The *Lagrange basis polynomial* is defined as

$$\ell_k(x) := \prod_{j\neq k} \frac{x - x_j}{x_k - x_j} = \frac{(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_1)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)}$$

Plugging in the grid points verifies that $\ell_k(x_j) = \delta_{kj}$.

We can use the Lagrange basis to directly construct the interpolatory polynomial:

Theorem 9 (Lagrange interpolation)**.** *The unique interpolation polynomial is:*

$$p(x) = f_1\ell_1(x) + \cdots + f_n\ell_n(x)$$

Proof It follows from inspection:

$$p(x_j) = \sum_{k=1}^{n} f_k\ell_k(x_j) = f_j.$$

■

Example 18 (interpolating an exponential)**.** We can interpolate $\exp(x)$ at the points $0, 1, 2$. That is, our data is $\boldsymbol{f} = [1, \mathrm{e}, \mathrm{e}^2]^\top$ and the interpolatory polynomial is

$$\begin{aligned} p(x) &= \ell_1(x) + \mathrm{e}\ell_2(x) + \mathrm{e}^2\ell_3(x) = \frac{(x-1)(x-2)}{(-1)(-2)} + \mathrm{e}\frac{x(x-2)}{(-1)} + \mathrm{e}^2\frac{x(x-1)}{2} \\ &= (1/2 - \mathrm{e} + \mathrm{e}^2/2)x^2 + (-3/2 + 2\mathrm{e} - \mathrm{e}^2/2)x + 1 \end{aligned}$$

Remark Interpolating at evenly spaced points is a really *bad* idea as it is inherently ill-conditioned. The lab explores this issue experimentally. Another serious issue is that monomials are a horrible basis for interpolation. This is intuitive: when n is large x^n is basically zero near the origin and hence x_j^n numerically lose linear independence, that is, on a computer they appear to be linearly dependent (up to rounding errors). Use alternative sets of points and bases entirely overcomes this issue.

IV.1.2 Interpolatory quadrature rules

Interpolation leads naturally to quadrature rules where one integrates the interpolatory polynomial exactly. This can be viewed as an extension of one-panel Rectangular Rules (which are degree 0 interpolants at a single point) and Trapezium Rules (which are degree 1 interpolants at two points). Using the Lagrange basis for interpolation we can write general interpolatory quadrature rules as a simple weighted sum:

Definition 28 (interpolatory quadrature rule)**.** Given a set of points $\boldsymbol{x} = [x_1, \ldots, x_n]^\top$ the interpolatory quadrature rule is:

$$\Sigma_n^{w,\boldsymbol{x}}[f] := \sum_{j=1}^{n} w_j f(x_j)$$

where

$$w_j := \int_a^b \ell_j(x)w(x)\mathrm{d}x.$$

The convergence of such a scheme is explored in the lab. But an important feature is that it is exact for all low degree polynomials:

Proposition 12 (interpolatory quadrature is exact for polynomials)**.** *Interpolatory quadrature is exact for all degree $n-1$ polynomials p:*

$$\int_a^b p(x)w(x)\mathrm{d}x = \Sigma_n^{w,\boldsymbol{x}}[p]$$

Proof The result follows since, by uniqueness of interpolatory polynomial, if p is a polynomial then

$$p(x) = \sum_{j=1}^n p(x_j)\ell_j(x)$$

Hence

$$\int_a^b p(x)w(x)\mathrm{d}x = \sum_{j=1}^n p(x_j)\int_a^b \ell_j(x)w(x)\mathrm{d}x = \Sigma_n^{w,\boldsymbol{x}}[p].$$

■

Example 19 (3-point interpolatory quadrature)**.** We find the interpolatory quadrature rule for $w(x) = 1$ on $[0,1]$ with points $[x_1, x_2, x_3] = [0, 1/4, 1]$. We have:

$$\begin{aligned} w_1 &= \int_0^1 w(x)\ell_1(x)\mathrm{d}x = \int_0^1 \frac{(x-1/4)(x-1)}{(-1/4)(-1)}\mathrm{d}x = -1/6 \\ w_2 &= \int_0^1 w(x)\ell_2(x)\mathrm{d}x = \int_0^1 \frac{x(x-1)}{(1/4)(-3/4)}\mathrm{d}x = 8/9 \\ w_3 &= \int_0^1 w(x)\ell_3(x)\mathrm{d}x = \int_0^1 \frac{x(x-1/4)}{3/4}\mathrm{d}x = 5/18 \end{aligned}$$

That is we have

$$\Sigma_n^{w,\boldsymbol{x}}[f] = -\frac{f(0)}{6} + \frac{8f(1/4)}{9} + \frac{5f(1)}{18}.$$

This is indeed exact for polynomials up to degree 2 (and no more):

$$\Sigma_n^{w,\boldsymbol{x}}[1] = 1, \Sigma_n^{w,\boldsymbol{x}}[x] = 1/2, \Sigma_n^{w,\boldsymbol{x}}[x^2] = 1/3, \Sigma_n^{w,\boldsymbol{x}}[x^3] = 7/24 \neq 1/4.$$

IV.1.3 Polynomial regression

In many settings interpolation is not an accurate or appropriate tool. Data is often on an evenly spaced grid in which case (as seen in the lab) interpolation breaks down catastrophically. Or the data is noisy and one ends up over resolving: approximating the noise rather than the signal. A simple solution is *polynomial regression*: use more sample points than the degrees of freedom in the polynomial. The special case of an affine polynomial is called *linear regression.*

More precisely, for $\boldsymbol{x} \in \mathbb{C}^m$ and for $n < m$ we want to find a degree $n-1$ polynomial

$$p(x) = \sum_{k=0}^{n-1} c_k x^k$$

such that

$$\begin{bmatrix} p(x_1) \\ \vdots \\ p(x_m) \end{bmatrix} \approx \underbrace{\begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix}}_{\boldsymbol{f}}.$$

Mapping between coefficients $\boldsymbol{c} \in \mathbb{C}^n$ to polynomial values on a grid can be accomplished via rectangular Vandermonde matrices. In particular, our goal is to choose $\boldsymbol{c} \in \mathbb{C}^n$ so that

$$V\boldsymbol{c} = \begin{bmatrix} p(x_1) \\ \vdots \\ p(x_m) \end{bmatrix} \approx \boldsymbol{f}.$$

We do so by solving the *least squares* system: given $V \in \mathbb{C}^{m \times n}$ and $\boldsymbol{f} \in \mathbb{C}^m$ we want to find $\boldsymbol{c} \in \mathbb{C}^n$ such that

$$\|V\boldsymbol{c} - \boldsymbol{f}\|$$

is minimal. Note interpolation is a special case where this norm is precisely zero (which is indeed minimal), but in general this norm may be rather large. We will discuss the numerical solution of least squares problems in the next few sections.

Remark Using regression instead of interpolation can overcome the issues with evenly spaced grids. However, the monomial basis is still very problematic.

IV.2 Differential Equations via Finite Differences

Linear algebra is a powerful tool for solving linear equations, including ∞-dimensional ones like differential equations. In this section we discuss *finite differences*: an algorithmic way of reducing ODEs to linear systems by replacing derivatives with divided difference approximations.

We will focus on the following differential equations. Indefinite integration for $a \leq x \leq b$ can be viewed as solving a very simple first-order linear ODE: given a constant $c \in \mathbb{F}$ (where $\mathbb{F}$ is either $\mathbb{R}$ or $\mathbb{C}$) and a function $f : [a, b] \to \mathbb{F}$, find a differentiable function $u : [a, b] \to \mathbb{F}$ such that

$$\begin{aligned} u(a) &= c, \\ u'(x) &= f(x). \end{aligned}$$

We will then allow for more complicated first order linear ODEs with variable coefficients: given a constant c and functions $f, \omega : [a, b] \to \mathbb{F}$ find u such that

$$\begin{aligned} u(a) &= c, \\ u'(x) - \omega(x)u(x) &= f(x). \end{aligned}$$

For second-order differential equations you may have seen *initial value problems* where the value and derivative at an initial point $x = a$ are provided. Instead, we will consider *boundary value problems* where the value at the left and right endpoints are imposed. In particular we will consider the Poisson equation with *Dirichlet conditions* (i.e. conditions on the left and

right of an interval): given constants c, d and a function f find a twice-differentiable function u such that

$$\begin{aligned} u(a) &= c, \\ u''(x) &= f(x), \\ u(b) &= d \end{aligned}$$

In higher dimensions, the Poisson equation (and other *elliptic* partial differential equations) typically have boundary conditions imposed on the boundary of a complicated geometry, and give the temperature equilibrium of a plate where the temperature is held fixed. The techniques we discuss for our simple 1D model problem extend to these more challenging settings.

Briefly, the basic idea of finite differences is a systematic way of reducing a differential equation to a linear system. For each problem we will do the following steps:

1. Discretise the interval $[a, b]$ by a grid of points $x_0, \ldots, x_n$ and write the ODE on each grid point.
2. Replace true derivatives of the solution u with approximate derivatives in terms of values on the grid $u(x_j)$ via the divided difference formula.
3. In the formula replace unknown values of $u(x_j)$ at the grid by new unknowns u_j.
4. Deduce from this a linear system that can be solved to compute u_j so that (hopefully) $u_j \approx u(x_j)$.

Remark There is a rich theory proving convergence of finite difference approximations to the true solution of ODEs but this is beyond the scope of this module, instead, we just learn the recipe. In the lab we determine convergence rates experimentally.

IV.2.1 Indefinite integration

We begin with the simplest differential equation on an interval $[a, b]$:

$$\begin{aligned} u(a) &= c, \\ u'(x) &= f(x) \end{aligned}$$

As in integration we will use an evenly spaced grid $a = x_0 < x_1 < \ldots < x_n = b$ defined by $x_j := a + hj$ where $h := (b - a)/n$. The solution is of course $u(x) = c + \int_a^x f(x)\mathrm{d}x$ and we could use Rectangular or Trapezium rules to obtain approximations to $u(x_j)$ for each j, however, we shall take another approach that will generalise to other differential equations.

Consider a divided difference approximation like right-sided divided differences:

$$u'(x) \approx \frac{u(x + h) - u(x)}{h}.$$

When applied to a grid point $x_j \in \{x_0, \ldots, x_{n-1}\}$ this becomes:

$$u'(x_j) \approx \frac{u(x_j + h) - u(x_j)}{h} = \frac{u(x_{j+1}) - u(x_j)}{h}$$

Note that x_n is not permitted since that would depend on $u(x_{n+1})$, but $x_{n+1} > b$ and we have only assumed f is defined on $[a, b]$. We use this approximation as follows:

(1) Write the ODE and initial conditions on the grid. Since right-sided divided differences will depend on x_j and x_{j+1} we stop at x_{n-1} to avoid going past our grid:

$$\begin{bmatrix} u(x_0) \\ u'(x_0) \\ u'(x_1) \\ \vdots \\ u'(x_{n-1}) \end{bmatrix} = \underbrace{\begin{bmatrix} c \\ f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{n-1}) \end{bmatrix}}_{\boldsymbol{b}}$$

(2) Replace derivatives with divided differences, changing equality to an approximation:

$$\begin{bmatrix} u(x_0) \\ (u(x_1) - u(x_0))/h \\ (u(x_2) - u(x_1))/h \\ \vdots \\ (u(x_n) - u(x_{n-1})/h \end{bmatrix} \approx \boldsymbol{b}$$

(3) We do not know $u(x_j)$ hence we replace it with other unknowns u_j, but where the approximation is turned back into an equality: we want to find $u_0, \ldots, u_n$ such that

$$\begin{bmatrix} u_0 \\ (u_1 - u_0)/h \\ (u_2 - u_1)/h \\ \vdots \\ (u_n - u_{n-1})/h \end{bmatrix} = \boldsymbol{b}$$

(4) This can be rewritten as a lower bidiagonal linear system:

$$\underbrace{\begin{bmatrix} 1 & & & \\ -1/h & 1/h & & \\ & \ddots & \ddots & \\ & & -1/h & 1/h \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{bmatrix}}_{\boldsymbol{u}} = \boldsymbol{b}$$

We can determine $\boldsymbol{u}$ by solving $L\boldsymbol{u} = \boldsymbol{b}$ using forward-substitution.

As mentioned before, it is possible to prove that as $n \to \infty$ we have for all j that $u_j \to u(x_j)$ (uniformly). But we will leave this to be shown experimentally in the lab.

IV.2.2 Forward Euler

We can extend this to more general first-order linear differential equations with a variable coefficient:

$$u(a) = c$$
$$u'(x) + \omega(x)u(x) = f(x)$$

The steps proceed very similar to before:

(1) Write the ODE and initial conditions on the grid, avoiding x_n so that we don't go past the endpoint:

$$\begin{bmatrix} u(x_0) \\ u'(x_0)+\omega(x_0)u(x_0) \\ u'(x_1)+\omega(x_1)u(x_1) \\ \vdots \\ u'(x_{n-1})+\omega(x_{n-1})u(x_{n-1}) \end{bmatrix} = \underbrace{\begin{bmatrix} c \\ f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{n-1}) \end{bmatrix}}_{\boldsymbol{b}}$$

(2) Replace derivatives with divided differences:

$$\begin{bmatrix} u(x_0) \\ (u(x_1)-u(x_0))/h+\omega(x_0)u(x_0) \\ (u(x_2)-u(x_1))/h+\omega(x_1)u(x_1) \\ \vdots \\ (u(x_n)-u(x_{n-1}))/h+\omega(x_{n-1})u(x_{n-1}) \end{bmatrix} \approx \boldsymbol{b}$$

(3) Replace $u(x_j)$ by its approximation u_j but now with the system being an equality:

$$\begin{bmatrix} u_0 \\ (u_1-u_0)/h+\omega(x_0)u_0 \\ (u_2-u_1)/h+\omega(x_1)u_1 \\ \vdots \\ (u_n-u_{n-1})/h+\omega(x_{n-1})u_{n-1} \end{bmatrix} = \boldsymbol{b}$$

(4) This is equivalent to the linear system:

$$\underbrace{\begin{bmatrix} 1 & & & \\ \omega(x_0)-1/h & 1/h & & \\ & \ddots & \ddots & \\ & & \omega(x_{n-1})-1/h & 1/h \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{bmatrix}}_{\boldsymbol{u}} = \boldsymbol{b}$$

We can solve $L\boldsymbol{u} = \boldsymbol{b}$ using forward-substitution so that $u_j \approx u(x_j)$.

IV.2.3 Poisson equation

Consider the Poisson equation with Dirichlet conditions (a two-point boundary value problem): given constants c, d and a function $f : [a, b] \to \mathbb{R}$ find a twice differentiable function $u : [a, b] \to \mathbb{R}$ satisfying

$$\begin{aligned} u(a) &= c, \\ u''(x) &= f(x), \\ u(b) &= d \end{aligned}$$

We shall adapt the procedure using the second-order divided difference approximation from the first probem sheet:

$$u''(x) \approx \frac{u(x-h)-2u(x)+u(x+h)}{h^2}$$

When applied to a grid point $x_1, \dots, x_{n-1}$ this becomes:

$$u'(x_j) \approx \frac{u(x_j - h) - 2u(x_j) + u(x_j + h)}{h^2} = \frac{u(x_{j-1}) - 2u(x_j) + u(x_{j+1})}{h^2}$$

Note that x_0 and x_n are not permitted since that would go past the endpoints of the interval. We use this approximation as follows:

(1) Write the ODE and boundary conditions on the grid (putting the left condition on the top and right condition on the bottom):

$$\begin{bmatrix} u(x_0) \\ u''(x_1) \\ u''(x_2) \\ \vdots \\ u''(x_{n-1}) \\ u(x_n) \end{bmatrix} = \underbrace{\begin{bmatrix} c \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{n-1}) \\ d \end{bmatrix}}_{\boldsymbol{b}}$$

(2) Replace derivatives with divided differences:

$$\begin{bmatrix} u(x_0) \\ \frac{u(x_0)-2u(x_1)+u(x_2)}{h^2} \\ \frac{u(x_1)-2u(x_2)+u(x_3)}{h^2} \\ \vdots \\ \frac{u(x_{n-2})-2u(x_{n-1})+u(x_n)}{h^2} \\ u(x_n) \end{bmatrix} \approx \boldsymbol{b}$$

(3) Replace $u(x_j)$ by its approximation u_j: we want to find $u_0, \dots, n_n$ so that

$$\begin{bmatrix} u_0 \\ \frac{u_0-2u_1+u_2}{h^2} \\ \frac{u_1-2u_2+u_3}{h^2} \\ \vdots \\ \frac{u_{n-2}-2u_{n-1}+u_n}{h^2} \\ u_n \end{bmatrix} = \boldsymbol{b}$$

(4) This is equivalent to a tridiagonal linear system:

$$\underbrace{\begin{bmatrix} 1 & & & & & \\ 1/h^2 & -2/h^2 & 1/h^2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1/h^2 & -2/h^2 & 1/h^2 & \\ & & & & 1 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{bmatrix}}_{\boldsymbol{u}} = \boldsymbol{b}$$

This can be solved in $O(n)$ complexity using a banded LU or QR factorisation.

Chapter V

Numerical Fourier series

So far, we have seen intuitive numerical methods for computing derivatives, integrals, and solving differential equations, primarily based on representing functions by their values at a grid of points. But by using more sophisticated mathematical tools, we can achieve much more accurate and reliable numerical methods. In particular, we can effectively use Fourier series for computing very accurately with periodic functions, and orthogonal polynomials for non-periodic functions that are smooth within an interval. Here we introduce these fundamental tools and explore applications to quadrature (computing integrals) where they produce incredibly accurate approximations, ones that converge exponentially (or faster) for analytic functions.

1. IV.1 Fourier Expansions: we discuss Fourier series and their usage in approximating periodic functions, using the Trapezium rule to compute the Fourier coefficients.

2. IV.2 Discrete Fourier Transform: The Trapezium rule approximation can be recast as a unitary matrix, known as the Discrete Fourier Transform (DFT). This is used to prove interpolation properties.

V.1 Fourier Expansions

Fourier series are a powerful tool in wide areas of mathematics, including solving partial differential equations, signal processing, and elsewhere. They are also very useful in computational methods, particularly for problems that have periodicity. Periodicity arises naturally when solving problems in radial coordinates, or when approximating a problem on the real line by a periodic problem with a large period. Fourier series are also related to orthogonal polynomials, which can be used for non-periodic problems.

The most fundamental basis is (complex) Fourier: we have $\mathrm{e}^{\mathrm{i}k\theta}$ are orthogonal with respect to the inner product

$$\langle f, g\rangle := \frac{1}{2\pi}\int_0^{2\pi} \bar{f}(\theta)g(\theta)\mathrm{d}\theta,$$

where we conjugate the first argument to be consistent with the vector inner product $\boldsymbol{x}^\star\boldsymbol{y}$. We will use the notation $\mathbb{T} := [0, 2\pi)$ (typically this has the topology of a circle attached but we do not need to worry about that here). We can (typically) expand functions in this basis:

Definition 29 (Fourier)**.** A function f has a Fourier expansion if

$$f(\theta) = \sum_{k=-\infty}^{\infty} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta}$$

where

$$\hat{f}_k := \langle \mathrm{e}^{\mathrm{i}k\theta}, f \rangle = \frac{1}{2\pi} \int_0^{2\pi} \mathrm{e}^{-\mathrm{i}k\theta} f(\theta) \mathrm{d}\theta$$

A basic observation is if a Fourier expansion has no negative terms it is equivalent to a Taylor series in disguise:

Definition 30 (Fourier-Taylor)**.** A function f has a Fourier–Taylor expansion if

$$f(\theta) = \sum_{k=0}^{\infty} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta} = \sum_{k=0}^{\infty} \hat{f}_k z^k$$

where $\hat{f}_k := \langle \mathrm{e}^{\mathrm{i}k\theta}, f \rangle$, and $z = \mathrm{e}^{\mathrm{i}\theta}$.

In numerical analysis we try to build on the analogy with linear algebra as much as possible. Therefore we can write this this as:

$$f(\theta) = \underbrace{[1|\mathrm{e}^{\mathrm{i}\theta}|\mathrm{e}^{2\mathrm{i}\theta}|\cdots]}_{T(\theta)} \underbrace{\begin{bmatrix} \hat{f}_0 \\ \hat{f}_1 \\ \hat{f}_2 \\ \vdots \end{bmatrix}}_{\hat{\boldsymbol{f}}}.$$

Essentially, expansions in bases are viewed as a way of turning *functions* into (infinite) *vectors*. And (differential) *operators* into *matrices*.

V.1.1 Convergence of Fourier series

In analysis one typically works with continuous functions and relates results to continuity. In numerical analysis we inherently have to work with *vectors*, so it is more natural to focus on the case where the *Fourier coefficients* $\hat{f}_k$ are *absolutely convergent*:

Definition 31 (absolute convergent)**.** We write $\hat{\boldsymbol{f}} \in \ell^1$ if it is absolutely convergent, or in otherwords, the 1-norm of $\hat{\boldsymbol{f}}$ is bounded:

$$\|\hat{\boldsymbol{f}}\|_1 := \sum_{k=-\infty}^{\infty} |\hat{f}_k| < \infty.$$

We first state a basic results (whose proof is beyond the scope of this module):

Theorem 10 (Fourier series equivalence)**.** *If $f, g : \mathbb{T} \to \mathbb{C}$ are periodic and continuous and $\hat{f}_k = \hat{g}_k$ for all $k \in \mathbb{Z}$ then $f = g$.*

Proof See Körner 2022 (Theorem 2.4). ■

This allows us to prove the following:

Theorem 11 (Absolute converging Fourier series)**.** *If* $\hat{\boldsymbol{f}} \in \ell^1$ *then*

$$f(\theta) = \sum_{k=-\infty}^{\infty} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta},$$

which converges uniformly.

Proof

Note that

$$g_n(\theta) := \sum_{k=-n}^{n} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta}$$

is uniformly-absolutely convergent as $n \to \infty$, that is,

$$\sum_{k=-n}^{n} |\hat{f}_k \mathrm{e}^{\mathrm{i}k\theta}| = \sum_{k=-n}^{n} |\hat{f}_k| \to \|\hat{\boldsymbol{f}}\|_1.$$

This guarantees that $g_n(\theta)$ converges uniformly to a continuous function $g(\theta)$. We have for $n > k$, that the k-th Fourier coefficient of $g_n(\theta)$ equals $\hat{f}_k$. Thus, by the properties of uniform convergence,

$$\hat{f}_k = \lim_{n\to\infty} \hat{f}_k = \lim_{n\to\infty} \frac{1}{2\pi} \int_0^{2\pi} \mathrm{e}^{-\mathrm{i}k\theta} g_n(\theta) \mathrm{d}\theta = \frac{1}{2\pi} \int_0^{2\pi} \mathrm{e}^{-\mathrm{i}k\theta} \lim_{n\to\infty} g_n(\theta) \mathrm{d}\theta = \hat{g}_k.$$

Since f and g are continuous and share the same Fourier coefficients, they are equal.

■

When does a function have absolutely convergent Fourier coefficients? We can deduce it from periodic differentiability of the function:

Lemma 7 (differentiability and absolutely convergence)**.** *If* $f : \mathbb{T} \to \mathbb{C}$ *and* f' *are periodic and* f'' *is uniformly bounded, then* $\hat{\boldsymbol{f}} \in \ell^1$*.*

Proof Integrate by parts twice using the fact that $f(0) = f(2\pi)$, $f'(0) = f'(2\pi)$:

$$\begin{aligned} 2\pi \hat{f}_k &= \int_0^{2\pi} f(\theta) \mathrm{e}^{-\mathrm{i}k\theta} \mathrm{d}\theta = [f(\theta) \frac{\mathrm{e}^{-\mathrm{i}k\theta}}{-\mathrm{i}k}]_0^{2\pi} + \frac{1}{\mathrm{i}k} \int_0^{2\pi} f'(\theta) \mathrm{e}^{-\mathrm{i}k\theta} \mathrm{d}\theta \\ &= [f'(\theta) \frac{\mathrm{e}^{-\mathrm{i}k\theta}}{(-\mathrm{i}k)^2}]_0^{2\pi} - \frac{1}{k^2} \int_0^{2\pi} f''(\theta) \mathrm{e}^{-\mathrm{i}k\theta} \mathrm{d}\theta \\ &= -\frac{1}{k^2} \int_0^{2\pi} f''(\theta) \mathrm{e}^{-\mathrm{i}k\theta} \mathrm{d}\theta. \end{aligned}$$

Thus uniform boundedness of f'' guarantees $|\hat{f}_k| \le M|k|^{-2}$ for some M, and we have

$$\sum_{k=-\infty}^{\infty} |\hat{f}_k| \le |\hat{f}_0| + 2M \sum_{k=1}^{\infty} |k|^{-2} < \infty$$

using the dominant convergence test.

■

This condition can be weakened to Lipschitz continuity but the proof is beyond the scope of this module. Of more practical importance is the other direction: the more times differentiable a function the faster the coefficients decay, and thence the faster Fourier expansions converge. In fact, if a function is smooth and 2π-periodic its Fourier coefficients decay faster than algebraically: they decay like $O(k^{-\lambda})$ for any λ. This will be explored in the problem sheet.

V.1.2 Trapezium rule and discrete Fourier coefficients

Definition 32 (Periodic Trapezium Rule)**.** Let $\theta_j = 2\pi j/n$ for $j = 0, 1, \ldots, n$ denote $n+1$ evenly spaced points over $[0, 2\pi]$. Recall that the *Trapezium rule* over $[0, 2\pi]$ is the approximation:

$$\int_0^{2\pi} f(\theta)\mathrm{d}\theta \approx \frac{2\pi}{n}\left[\frac{f(0)}{2} + \sum_{j=1}^{n-1} f(\theta_j) + \frac{f(2\pi)}{2}\right]$$

But if f is periodic we have $f(0) = f(2\pi)$ and we get the *periodic Trapezium rule*:

$$\frac{1}{2\pi}\int_0^{2\pi} f(\theta)\mathrm{d}\theta \approx \underbrace{\frac{1}{n}\sum_{j=0}^{n-1} f(\theta_j)}_{\Sigma_n[f]}$$

We know that $\mathrm{e}^{\mathrm{i}k\theta}$ are orthogonal with respect to the continuous inner product. The following says that this property is maintained (up to "aliasing") when we replace the continuous integral with a trapezium rule approximation:

Lemma 8 (Discrete orthogonality)**.** *We have:*

$$\sum_{j=0}^{n-1} \mathrm{e}^{\mathrm{i}k\theta_j} = \begin{cases} n & k = \ldots, -2n, -n, 0, n, 2n, \ldots \\ 0 & \textit{otherwise} \end{cases}$$

In other words,

$$\Sigma_n[\mathrm{e}^{\mathrm{i}(k-\ell)\theta}] = \begin{cases} 1 & k-\ell = \ldots, -2n, -n, 0, n, 2n, \ldots \\ 0 & \textit{otherwise} \end{cases}.$$

Proof

Consider $\omega := \mathrm{e}^{\mathrm{i}\theta_1} = \mathrm{e}^{\frac{2\pi\mathrm{i}}{n}}$. This is an n-th root of unity: $\omega^n = 1$. Note that $\mathrm{e}^{\mathrm{i}\theta_j} = \mathrm{e}^{\frac{2\pi\mathrm{i}j}{n}} = \omega^j$.

(Case 1: $k = pn$ for an integer p) We have

$$\sum_{j=0}^{n-1} \mathrm{e}^{\mathrm{i}k\theta_j} = \sum_{j=0}^{n-1} \omega^{kj} = \sum_{j=0}^{n-1} (\omega^{pn})^j = \sum_{j=0}^{n-1} 1 = n$$

(Case 2: $k \neq pn$ for an integer p) Recall that (via a telescoping sum argument)

$$\sum_{j=0}^{n-1} z^j = \frac{z^n - 1}{z - 1}.$$

Then we have

$$\sum_{j=0}^{n-1} \mathrm{e}^{\mathrm{i}k\theta_j} = \sum_{j=0}^{n-1} (\omega^k)^j = \frac{\omega^{kn} - 1}{\omega^k - 1} = 0.$$

where we use the fact that k is not a multiple of n to guarantee that $\omega^k \neq 1$.

■

V.1.3 Convergence of Approximate Fourier expansions

We will now use the Trapezium rule to approximate Fourier coefficients and expansions:

Definition 33 (Discrete Fourier coefficients)**.** Define the Trapezium rule approximation to the Fourier coefficients by:

$$\hat{f}^n_k := \Sigma_n[\mathrm{e}^{-ik\theta} f(\theta)] = \frac{1}{n}\sum_{j=0}^{n-1} \mathrm{e}^{-ik\theta_j} f(\theta_j)$$

A remarkable fact is that the discete Fourier coefficients can be expressed as a sum of the true Fourier coefficients:

Theorem 12 (discrete Fourier coefficients)**.** *If $\hat{\boldsymbol{f}} \in \ell^1$ (absolutely convergent Fourier coefficients) then*

$$\hat{f}^n_k = \cdots + \hat{f}_{k-2n} + \hat{f}_{k-n} + \hat{f}_k + \hat{f}_{k+n} + \hat{f}_{k+2n} + \cdots$$

Proof

$$\begin{aligned}\hat{f}^n_k = \Sigma_n[f(\theta)\mathrm{e}^{-\mathrm{i}k\theta}] &= \sum_{\ell=-\infty}^{\infty} \hat{f}_\ell \Sigma_n[\mathrm{e}^{\mathrm{i}(\ell-k)\theta}] \\ &= \sum_{\ell=-\infty}^{\infty} \hat{f}_\ell \begin{cases} 1 & \ell - k = \ldots, -2n, -n, 0, n, 2n, \ldots \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

■

Example 20 (Taylor coefficients via Geometric series)**.** Consider the function

$$f(\theta) = \frac{2}{2 - \mathrm{e}^{\mathrm{i}\theta}}$$

Under the change of variables $z = \mathrm{e}^{\mathrm{i}\theta}$ we know for z on the unit circle this becomes (using the geometric series with $z/2$)

$$\frac{2}{2-z} = \sum_{k=0}^{\infty} \frac{z^k}{2^k}$$

i.e., $\hat{f}_k = 1/2^k$ which is absolutely summable:

$$\sum_{k=0}^{\infty} |\hat{f}_k| = f(0) = 2.$$

If we use an n point discretisation we get for $0 \le k \le n-1$ (using the geoemtric series with 2^{-n})

$$\hat{f}^n_k = \hat{f}_k + \hat{f}_{k+n} + \hat{f}_{k+n} + \cdots = \sum_{p=0}^{\infty} \frac{1}{2^{k+pn}} = \frac{2^{n-k}}{2^n - 1}$$

Note that as $n \to \infty$, we have $\hat{f}^n_k \to \hat{f}_k$.

Note that there is redundancy:

Corollary 3 (aliasing)**.** *For all $p \in \mathbb{Z}$, $\hat{f}^n_k = \hat{f}^n_{k+pn}$.*

Proof Follows immediately:

$$\hat{f}^n_{k+pn} = \sum_{j=-\infty}^{\infty} \hat{f}_{k+(p+j)n} = \sum_{j=-\infty}^{\infty} \hat{f}_{k+jn} = \hat{f}^n_k.$$

■

In other words if we know $\hat{f}^n_0, \ldots, \hat{f}^n_{n-1}$, we know $\hat{f}^n_k$ for all k via a permutation, for example if $n = 2m + 1$ we have

$$\begin{bmatrix} \hat{f}^n_{-m} \\ \vdots \\ \hat{f}^n_{-1} \\ \hat{f}^n_0 \\ \vdots \\ \hat{f}^n_m \end{bmatrix} = \underbrace{\begin{bmatrix} & & & 1 & & & \\ & & & & \ddots & & \\ & & & & & & 1 \\ 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \end{bmatrix}}_{P_\sigma} \begin{bmatrix} \hat{f}^n_0 \\ \vdots \\ \hat{f}^n_m \\ \hat{f}^n_{m+1} \\ \vdots \\ \hat{f}^n_{n-1} \end{bmatrix}$$

where σ has Cauchy notation (*Careful*: we are using 1-based indexing here):

$$\begin{pmatrix} 1 & 2 & \cdots & m & m+1 & m+2 & \cdots & n \\ m+2 & m+3 & \cdots & n & 1 & 2 & \cdots & m+1 \end{pmatrix}.$$

We can prove *convergence* whenever f has absolutely summable coefficients. We will prove the result here in the special case where the negative coefficients are zero. That is, $\hat{f}^n_0, \ldots, \hat{f}^n_{n-1}$ are approximations of the Fourier–Taylor coefficients.

Theorem 13 (Approximate Fourier-Taylor expansions converge)**.** *If* $0 = \hat{f}_{-1} = \hat{f}_{-2} = \cdots$ *and* $\hat{\boldsymbol{f}}$ *is absolutely convergent then*

$$f_n(\theta) = \sum_{k=0}^{n-1} \hat{f}^n_k \mathrm{e}^{\mathrm{i}k\theta}$$

converges uniformly to $f(\theta)$*.*

Proof

$$|f(\theta) - f_n(\theta)| = |\sum_{k=0}^{n-1} (\hat{f}_k - \hat{f}^n_k)\mathrm{e}^{\mathrm{i}k\theta} + \sum_{k=n}^{\infty} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta}| = |\sum_{k=n}^{\infty} \hat{f}_k (\mathrm{e}^{\mathrm{i}k\theta} - \mathrm{e}^{\mathrm{i}\mathrm{mod}(k,n)\theta})| \leq 2 \sum_{k=n}^{\infty} |\hat{f}_k|$$

which goes to zero as $n \to \infty$. ■

For the general case we need to choose a range of coefficients that includes roughly an equal number of negative and positive coefficients (preferring negative over positive in a tie as a convention):

$$f_n(\theta) = \sum_{k=-\lceil n/2 \rceil}^{\lfloor n/2 \rfloor} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta}$$

In the problem sheet we will prove this converges provided the coefficients are absolutely convergent.

V.2 Discrete Fourier Transform

In the previous section we explored using the trapezium rule for approximating Fourier coefficients. This is a linear map from function values to coefficients and thus can be reinterpreted as a matrix-vector product, called the the Discrete Fourier Transform. It turns out the matrix is unitary which leads to important properties including interpolation.

Remark A clever way of decomposing the DFT leads to a fast way of applying and inverting it, which is one of the most influencial algorithms of the 20th century: the Fast Fourier Transform. But this is beyond the scope of this module.

V.2.1 The Discrete Fourier transform

Definition 34 (DFT)**.** The *Discrete Fourier Transform (DFT)* is defined as:

$$Q_n := \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \mathrm{e}^{-\mathrm{i}\theta_1} & \mathrm{e}^{-\mathrm{i}\theta_2} & \cdots & \mathrm{e}^{-\mathrm{i}\theta_{n-1}} \\ 1 & \mathrm{e}^{-\mathrm{i}2\theta_1} & \mathrm{e}^{-\mathrm{i}2\theta_2} & \cdots & \mathrm{e}^{-\mathrm{i}2\theta_{n-1}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \mathrm{e}^{-\mathrm{i}(n-1)\theta_1} & \mathrm{e}^{-\mathrm{i}(n-1)\theta_2} & \cdots & \mathrm{e}^{-\mathrm{i}(n-1)\theta_{n-1}} \end{bmatrix}$$
$$= \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)^2} \end{bmatrix}$$

for the n-th root of unity $\omega = \mathrm{e}^{2\pi\mathrm{i}/n}$.

Note that

$$Q_n^\star = \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \mathrm{e}^{\mathrm{i}\theta_1} & \mathrm{e}^{\mathrm{i}2\theta_1} & \cdots & \mathrm{e}^{\mathrm{i}(n-1)\theta_1} \\ 1 & \mathrm{e}^{\mathrm{i}\theta_2} & \mathrm{e}^{\mathrm{i}2\theta_2} & \cdots & \mathrm{e}^{\mathrm{i}(n-1)\theta_2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \mathrm{e}^{\mathrm{i}\theta_{n-1}} & \mathrm{e}^{\mathrm{i}2\theta_{n-1}} & \cdots & \mathrm{e}^{\mathrm{i}(n-1)\theta_{n-1}} \end{bmatrix}$$
$$= \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{1} & \omega^{2} & \cdots & \omega^{(n-1)} \\ 1 & \omega^{2} & \omega^{4} & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix}$$

Hence we have

$$\underbrace{\begin{bmatrix} \hat{f}_0^n \\ \vdots \\ \hat{f}_{n-1}^n \end{bmatrix}}_{\hat{\boldsymbol{f}}^n} = \frac{1}{\sqrt{n}} Q_n \underbrace{\begin{bmatrix} f(\theta_0) \\ \vdots \\ f(\theta_{n-1}) \end{bmatrix}}_{\boldsymbol{f}^n}$$

The choice of normalisation constant is motivated by the following:

Proposition 1 (DFT is Unitary) $Q_n \in U(n)$, that is, $Q_n^\star Q_n = Q_n Q_n^\star = I$.

Proof

$$Q_n Q_n^\star = \begin{bmatrix} \Sigma_n[1] & \Sigma_n[\mathrm{e}^{\mathrm{i}\theta}] & \cdots & \Sigma_n[\mathrm{e}^{\mathrm{i}(n-1)\theta}] \\ \Sigma_n[\mathrm{e}^{-\mathrm{i}\theta}] & \Sigma_n[1] & \cdots & \Sigma_n[\mathrm{e}^{\mathrm{i}(n-2)\theta}] \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_n[\mathrm{e}^{-\mathrm{i}(n-1)\theta}] & \Sigma_n[\mathrm{e}^{-\mathrm{i}(n-2)\theta}] & \cdots & \Sigma_n[1] \end{bmatrix} = I$$

■

In other words, Q_n is easily inverted and we also have a map from discrete Fourier coefficients back to values:

$$\sqrt{n} Q_n^\star \hat{\boldsymbol{f}}^n = \boldsymbol{f}^n$$

Example 21 (Computing Sum)**.** Define the following infinite sum (which has no name apparently, according to Mathematica):

$$S_n(k) := \sum_{p=0}^{\infty} \frac{1}{(k+pn)!}$$

We can use the DFT to compute $S_n(0), \ldots, S_n(n-1)$. Consider

$$f(\theta) = \exp(\mathrm{e}^{\mathrm{i}\theta}) = \sum_{k=0}^{\infty} \frac{\mathrm{e}^{\mathrm{i}k\theta}}{k!}$$

where we know the Fourier coefficients from the Taylor series of e^z. The discrete Fourier coefficients satisfy for $0 \le k \le n-1$:

$$\hat{f}_k^n = \hat{f}_k + \hat{f}_{k+n} + \hat{f}_{k+2n} + \cdots = S_n(k)$$

Thus we have

$$\begin{bmatrix} S_n(0) \\ \vdots \\ S_n(n-1) \end{bmatrix} = \frac{1}{\sqrt{n}} Q_n \begin{bmatrix} \mathrm{e} \\ \exp(\mathrm{e}^{2\mathrm{i}\pi/n}) \\ \vdots \\ \exp(\mathrm{e}^{2\mathrm{i}(n-1)\pi/n}) \end{bmatrix}$$

V.2.2 Interpolation

We investigated interpolation and least squares using polynomials at evenly spaced points, observing that there were issues with stability. We now show that the DFT actually gives coefficients that interpolate using Fourier expansions. As the DFT is a unitary matrix multiplication is "stable", i.e. it preserves norms and hence we know it cannot cause the same huge blow-up we saw for polynomials. That is: whilst polynomials are bad for interpolation at evenly spaced points, trigonometric polynomials are great.

The following guarantees that our approximate Fourier series actually interpolates the data:

Corollary 4 (Interpolation)**.**

$$f_n(\theta) := \sum_{k=0}^{n-1} \hat{f}_k^n \mathrm{e}^{\mathrm{i}k\theta}$$

interpolates f at θ_j:

$$f_n(\theta_j) = f(\theta_j)$$

Proof We have

$$f_n(\theta_j) = \sum_{k=0}^{n-1} \hat{f}^n_k \mathrm{e}^{\mathrm{i}k\theta_j} = \sqrt{n}\boldsymbol{e}^\top_{j+1} Q^\star_n \hat{\boldsymbol{f}}^n = \boldsymbol{e}^\top_{j+1} Q^\star_n Q_n \boldsymbol{f}^n = f(\theta_j).$$

■

Example 22 (DFT versus Lagrange)**.** Consider interpolating $\exp z$ by a polynomial at the points $1, \mathrm{i}, -1, -\mathrm{i}$. We can use Lagrange polynomials:

$$\begin{aligned}
\ell_1(z) &= \frac{(z-\mathrm{i})(z+1)(z+\mathrm{i})}{2(1-\mathrm{i})(1+\mathrm{i})} = \frac{z^3+z^2+z+1}{4} \\
\ell_2(z) &= \frac{(z-1)(z+1)(z+\mathrm{i})}{(\mathrm{i}-1)(\mathrm{i}+1)2\mathrm{i}} = \frac{\mathrm{i}z^3-z^2-\mathrm{i}z+1}{4} \\
\ell_3(z) &= \frac{(z-1)(z-\mathrm{i})(z+\mathrm{i})}{-2(-1-\mathrm{i})(-1+\mathrm{i})} = \frac{-z^3+z^2-z+1}{4} \\
\ell_4(z) &= \frac{(z-1)(z-\mathrm{i})(z+1)}{(-\mathrm{i}-1)(-2\mathrm{i})(-\mathrm{i}+1)} = \frac{-\mathrm{i}z^3-z^2+\mathrm{i}z+1}{4}
\end{aligned}$$

So we get the interpolant:

$$\begin{aligned}
&\mathrm{e}\ell_1(z) + \mathrm{e}^{\mathrm{i}}\ell_2(z) + \mathrm{e}^{-1}\ell_3(z) + \mathrm{e}^{-\mathrm{i}}\ell_4(z) \\
&= \frac{\mathrm{e}+\mathrm{e}^{\mathrm{i}}+\mathrm{e}^{-1}+\mathrm{e}^{-\mathrm{i}}}{4} + \frac{\mathrm{e}-\mathrm{i}\mathrm{e}^{\mathrm{i}}-\mathrm{e}^{-1}+\mathrm{i}\mathrm{e}^{-\mathrm{i}}}{4}z + \frac{\mathrm{e}-\mathrm{e}^{\mathrm{i}}+\mathrm{e}^{-1}-\mathrm{i}\mathrm{e}^{-\mathrm{i}}}{4}z^2 + \frac{\mathrm{e}+\mathrm{i}\mathrm{e}^{\mathrm{i}}-\mathrm{e}^{-1}-\mathrm{i}\mathrm{e}^{-\mathrm{i}}}{4}z^3
\end{aligned}$$

Alternatively we could have deduced this directly from the DFT. In particular, we know the coefficients of the interpolating polynomial must be, for $\omega = \mathrm{i}$ and $f(\theta) = \exp(\mathrm{e}^{\mathrm{i}\theta})$,

$$\begin{bmatrix} \hat{f}^4_0 \\ \hat{f}^4_1 \\ \hat{f}^4_2 \\ \hat{f}^4_3 \end{bmatrix} = \frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \end{bmatrix}\begin{bmatrix} \mathrm{e} \\ \mathrm{e}^{\mathrm{i}} \\ \mathrm{e}^{-1} \\ \mathrm{e}^{-\mathrm{i}} \end{bmatrix} = \frac{1}{4}\begin{bmatrix} \mathrm{e}+\mathrm{e}^{\mathrm{i}}+\mathrm{e}^{-1}+\mathrm{e}^{-\mathrm{i}} \\ \mathrm{e}-\mathrm{i}\mathrm{e}^{\mathrm{i}}-\mathrm{e}^{-1}+\mathrm{i}\mathrm{e}^{-\mathrm{i}} \\ \mathrm{e}-\mathrm{e}^{\mathrm{i}}+\mathrm{e}^{-1}-\mathrm{e}^{-\mathrm{i}} \\ \mathrm{e}+\mathrm{i}\mathrm{e}^{\mathrm{i}}-\mathrm{e}^{-1}-\mathrm{i}\mathrm{e}^{-\mathrm{i}} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} \cosh 1 + \cos 1 \\ \sinh 1 + \sin 1 \\ \cosh 1 - \cos 1 \\ \sinh 1 - \sin 1 \end{bmatrix}.$$

The interpolation property also applies to the approximation

$$f_n(\theta) = \sum_{k=-\lceil n/2 \rceil}^{\lfloor n/2 \rfloor} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta}$$

for general Fourier series, which is investigated in the problem sheet.

Chapter VI

Orthogonal Polynomials

1. VI.1 Orthogonal Polynomials: For non-periodic functions we consider orthogonal polynomials, and discuss their basic properties.
2. VI.2 Classical Orthogonal Polynomials: For certain weights, orthogonal polynomials are classical and have addition structure that are useful for computations.
3. VI.3 Gaussian Quadrature: Finally, we revisit the problem of computing integrals, and see that using orthogonal polynomials we can derive much more accurate methods.

We stop at integration, but Fourier and orthogonal polynomial expansions also lead to very effective scheme for solving differential equations and many other applications.

VI.1 Orthogonal Polynomials

Fourier series are very powerful for approximating periodic functions. If periodicity is lost, however, the Fourier coefficients are no longer in ℓ^1 and uniform convergence is lost. In this chapter we introduce *Orthogonal Polynomials (OPs)* are more effective for computing with non-periodic (but still continuous/differentiable) functions. That is we consider expansions of the form

$$f(x) = \sum_{k=0}^{\infty} c_k p_k(x)$$

where $p_k(x)$ are special families of polynomials and c_k are expansion coefficients. The approximation of the coefficients $c_k \approx c_k^n$ using quadrature will be explored later.

Why not use monomials as in Taylor series? Hidden in the discussion on Fourier series was that we could effectively compute Taylor coefficients by evaluating on the unit circle in the complex plane, *only* if the radius of convergence was 1. Many functions are smooth on say $[-1, 1]$ but have non-convergent Taylor series, e.g.:

$$\frac{1}{25x^2 + 1}$$

While orthogonal polynomials span the same space as monomials, orthogonal polynomials are *much* more stable and can effectively and accurately approximate such functions. In particular, where we saw that interpolation by monomials at evenly spaced points performed

horribly in practice we can use orthogonal polynomials with specially chosen points to get reliable interpolation of functions.

In addition to numerics, OPs play a very important role in many mathematical areas including functional analysis, integrable systems, singular integral equations, complex analysis, and random matrix theory.

VI.1.1 General properties

Definition 35 (graded polynomial basis)**.** A set of polynomials $\{p_0(x), p_1(x), \ldots\}$ is *graded* if p_n is precisely degree n: i.e.,

$$p_n(x) = k_n x^n + k_n^{(1)} x^{n-1} + \cdots + k_n^{(n-1)} x + k_n^{(n)}$$

for $k_n \neq 0$.

Note that if p_n are graded then $\{p_0(x), \ldots, p_n(x)\}$ are a basis of all polynomials of degree n.

Definition 36 (Orthogonal Polynomials)**.** Given an (integrable) *weight* $w(x) > 0$ for $x \in (a, b)$, which defines a continuous inner product

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

a graded polynomial basis $\{p_0(x), p_1(x), \ldots\}$ are *orthogonal polynomials (OPs)* if

$$\langle p_n, p_m \rangle = 0$$

whenever $n \neq m$. We assume through that integrals of polynomials are finite:

$$\int_a^b x^k w(x) \mathrm{d}x < \infty.$$

Note in the above

$$h_n := \langle p_n, p_n \rangle = \|p_n\|^2 = \int_a^b p_n(x)^2 w(x) \mathrm{d}x > 0.$$

Multiplying any orthogonal polynomial by a nonzero constant necessarily is also an orthogonal polynomial. We have two standard normalisations:

Definition 37 (Orthonormal Polynomials)**.** A set of orthogonal polynomials $\{q_0(x), q_1(x), \ldots\}$ are *orthonormal* if $\|q_n\| = 1$.

Definition 38 (Monic Orthogonal Polynomials)**.** A set of orthogonal polynomials $\{\pi_0(x), \pi_1(x), \ldots\}$ are *monic* if $k_n = 1$.

Proposition 13 (existence)**.** *Given a weight $w(x)$, monic orthogonal polynomials exist.*

Proof Existence follows immediately from the Gram–Schmidt procedure. That is, define $\pi_0(x) := 1$ and

$$\pi_n(x) := x^n - \sum_{k=0}^{n-1} \frac{\langle x^n, \pi_k \rangle}{\|\pi_k\|^2} \pi_k(x).$$

Assume π_m are monic OPs for all $m < n$. Then we have

$$\langle \pi_m, \pi_n \rangle = \langle \pi_m, x^n \rangle - \sum_{k=0}^{n-1} \frac{\langle x^n, \pi_k \rangle}{\|\pi_k\|^2} \underbrace{\langle \pi_m, \pi_k \rangle}_{=0 \text{ if } m \neq k} = \langle \pi_m, x^n \rangle - \langle x^n, \pi_m \rangle = 0.$$

■

We are primarly concerned with the usage of orthogonal polynomials in approximating functions. First we observe the following:

Proposition 14 (expansion)**.** *If $r(x)$ is a degree n polynomial and $\{p_n\}$ are orthogonal then*

$$r(x) = \sum_{k=0}^{n} \frac{\langle p_k, r \rangle}{\|p_k\|^2} p_k(x).$$

Note for $\{q_n\}$ orthonormal we have

$$r(x) = \sum_{k=0}^{n} \langle q_k, r \rangle q_k(x).$$

Proof Because $\{p_0, \ldots, p_n\}$ are a basis of polynomials we can write

$$r(x) = \sum_{k=0}^{n} r_k p_k(x)$$

for constants $r_k \in \mathbb{R}$. By linearity we have

$$\langle p_m, r \rangle = \sum_{k=0}^{n} r_k \langle p_m, p_k \rangle = r_m \langle p_m, p_m \rangle$$

which implies that $r_m = \langle p_m, r \rangle / \langle p_m, p_m \rangle$. ■

Corollary 5 (zero inner product)**.** *If a degree n polynomial r satisfies*

$$0 = \langle p_0, r \rangle = \ldots = \langle p_n, r \rangle$$

then $r = 0$.

Proof If all the inner products are zero the coefficients in the expansion are all zero and r is zero. ■

Corollary 6 (uniqueness)**.** *Monic orthogonal polynomials are unique.*

Proof If $p_n(x)$ and $\pi_n(x)$ are both monic orthogonal polynomials then $r(x) = p_n(x) - \pi_n(x)$ is degree $n - 1$ but satisfies

$$\langle r, \pi_k \rangle = \langle p_n, \pi_k \rangle - \langle \pi_n, \pi_k \rangle = 0$$

for $k = 0, \ldots, n-1$. Note $\langle p_n, \pi_k \rangle = 0$ can be seen by expanding

$$\pi_k(x) = \sum_{j=0}^{k} c_j p_j(x).$$

■

OPs are uniquely defined (up to a constant) by the property that they are orthogonal to all lower degree polynomials.

Theorem 14 (orthogonal to lower degree)**.** *Given a weight $w(x)$, a polynomial*

$$p(x) = k_n x^n + O(x^{n-1})$$

with $k_n \neq 0$ satisfies

$$\langle p, f_m \rangle = 0$$

for all polynomials f_m of degree $m < n$ if and only if $p(x) = k_n \pi_n(x)$ where $\pi_n(x)$ are the monic orthogonal polynomials. Therefore an orthogonal polynomial is uniquely defined by the weight and leading order coefficient k_n.

Proof We leave this proof to the problem sheets. ■

VI.1.2 Three-term recurrence

The most *fundamental* property of orthogonal polynomials is their three-term recurrence.

Theorem 15 (3-term recurrence, 2nd form)**.** *If $\{p_n\}$ are OPs then there exist real constants $a_n b_n, c_{n-1}$ such that*

$$\begin{aligned} xp_0(x) &= a_0 p_0(x) + b_0 p_1(x) \\ xp_n(x) &= c_{n-1} p_{n-1}(x) + a_n p_n(x) + b_n p_{n+1}(x), \end{aligned}$$

where $b_n \neq 0$ and $c_{n-1} \neq 0$.

Proof The $n = 0$ case is immediate since $\{p_0, p_1\}$ are a basis of degree 1 polynomials. The $n > 0$ case follows from

$$\langle xp_n, p_k \rangle = \langle p_n, xp_k \rangle = 0$$

for $k < n - 1$ as xp_k is of degree $k + 1 < n$.

Note that

$$b_n = \frac{\langle p_{n+1}, xp_n \rangle}{\|p_{n+1}\|^2} \neq 0$$

since $xp_n = k_n x^{n+1} + O(x^n)$ is precisely degree n. Further,

$$c_{n-1} = \frac{\langle p_{n-1}, xp_n \rangle}{\|p_{n-1}\|^2} = \frac{\langle p_n, xp_{n-1} \rangle}{\|p_{n-1}\|^2} = b_{n-1} \frac{\|p_n\|^2}{\|p_{n-1}\|^2} \neq 0.$$

■

Clearly if π_n is monic then so is $x\pi_n$ which leads to the following:

Corollary 7 (monic 3-term recurrence)**.** *$\{\pi_n\}$ are monic if and only if $b_n = 1$.*

Proof

If $b_n = 1$ and $\pi_n(x) = x^n + O(x^{n-1})$ then the 3-term recurrence shows us that

$$\pi_{n+1}(x) = x\pi_n(x) - c_{n-1}\pi_{n-1}(x) - a_n \pi_n(x) = x^{n+1} + O(x^n)$$

and $\pi_{n+1}(x)$ is also monic. Similarly, if $\pi_n(x)$ is monic and $b_n \neq 1$ then $\pi_{n+1}(x)$ is not monic, which would be a contradiction. ■

Note this implies that we can define $\pi_{n+1}(x)$ in terms of π_{n-1} and π_n:

$$\pi_{n+1}(x) = x\pi_n(x) - a_n\pi_n(x) - c_{n-1}\pi_{n-1}(x)$$

where

$$a_n = \frac{\langle x\pi_n, \pi_n\rangle}{\|\pi_n\|^2} \qquad \text{and} \qquad c_{n-1} = \frac{\langle x\pi_n, \pi_{n-1}\rangle}{\|\pi_{n-1}\|^2}.$$

Example 23 (constructing OPs)**.** What are the monic OPs $\pi_0(x), \ldots, \pi_3(x)$ with respect to $w(x) = 1$ on $[0, 1]$? We can construct these using Gram–Schmidt, but exploiting the 3-term recurrence to reduce the computational cost. We have $\pi_0(x) = 1$, which we see is already normalised:

$$\|\pi_0\|^2 = \langle \pi_0, \pi_0\rangle = \int_0^1 \mathrm{d}x = 1.$$

We know from the 3-term recurrence that

$$x\pi_0(x) = a_0\pi_0(x) + \pi_1(x)$$

where

$$a_0 = \frac{\langle \pi_0, x\pi_0\rangle}{\|\pi_0\|^2} = \int_0^1 x\mathrm{d}x = 1/2.$$

Thus

$$\begin{aligned} \pi_1(x) = x\pi_0(x) - a_0\pi_0(x) = x - 1/2 \qquad &\Rightarrow \\ \|\pi_1\|^2 = \int_0^1 (x^2 - x + 1/4)\mathrm{d}x = 1/12. & \end{aligned}$$

From

$$x\pi_1(x) = c_0\pi_0(x) + a_1\pi_1(x) + \pi_2(x)$$

we have

$$\begin{aligned} c_0 &= \frac{\langle \pi_0, x\pi_1\rangle}{\|\pi_0\|^2} = \int_0^1 (x^2 - x/2)\mathrm{d}x = 1/12, \\ a_1 &= \frac{\langle \pi_1, x\pi_1\rangle}{\|\pi_1\|^2} = 12\int_0^1 (x^3 - x^2 + x/4)\mathrm{d}x = 1/2, \\ \pi_2(x) &= x\pi_1(x) - c_0 - a_1\pi_1(x) = x^2 - x + 1/6 \qquad \Rightarrow \\ \|\pi_2\|^2 &= \int_0^1 (x^4 - 2x^3 + 4x^2/3 - x/3 + 1/36)\mathrm{d}x = \frac{1}{180} \end{aligned}$$

Finally, from

$$x\pi_2(x) = c_1\pi_1(x) + a_2\pi_2(x) + \pi_3(x)$$

we have

$$\begin{aligned} c_1 &= \frac{\langle \pi_1, x\pi_2\rangle}{\|\pi_1\|^2} = 12\int_0^1 (x^4 - 3x^3/2 + 2x^2/3 - x/12)\mathrm{d}x = 1/15, \\ a_2 &= \frac{\langle \pi_2, x\pi_2\rangle}{\|\pi_2\|^2} = 180\int_0^1 (x^5 - 2x^4 + 4x^3/3 - x^2/3 + x/36)\mathrm{d}x = 1/2, \\ \pi_3(x) &= x\pi_2(x) - c_1\pi_1(x) - a_2\pi_2(x) \\ &= x^3 - x^2 + x/6 - x/15 + 1/30 - x^2/2 + x/2 - 1/12 \\ &= x^3 - 3x^2/2 + 3x/5 - 1/20 \end{aligned}$$

VI.1.3 Jacobi matrices

The three-term recurrence can also be interpreted as a matrix:

Corollary 8 (multiplication matrix)**.** *For*

$$P(x) := [p_0(x)|p_1(x)|\cdots]$$

then we have

$$xP(x) = P(x)\underbrace{\begin{bmatrix} a_0 & c_0 & & \\ b_0 & a_1 & c_1 & \\ & b_1 & a_2 & \ddots \\ & & \ddots & \ddots \end{bmatrix}}_{X}$$

More generally, for any polynomial $a(x)$ we have

$$a(x)P(x) = P(x)a(X).$$

Proof The expression follows:

$$xP(x) = [xp_0(x)|xp_1(x)|\cdots] = [a_0p_0(x) + b_0p_1(x)|c_0p_0(x) + a_1p_1(x) + b_1p_2(x)|\cdots] = P(x)X.$$

For polynomials, note that

$$x^kP(x) = x^{k-1}P(x)X = \cdots = P(x)X^k.$$

Thus if $a(x) = \sum_{k=0}^n a_kx^k$ we have by linearity

$$a(x)P(x) = \sum_{k=0}^n a_kx^kP(x) = P(x)\sum_{k=0}^n a_kX^k = P(x)a(X).$$

■

Remark If you are worried about multiplication of infinite matrices/vectors note it is well-defined by the standard definition because it is banded. It can also be defined in terms of functional analysis where one considers these as linear operators (functions of functions) between vector spaces.

For the special cases of orthonormal polynomials we have extra structure, in which case we refer to the matrix as a *Jacobi matrix*:

Corollary 9 (Jacobi matrix)**.** *The multiplication matrix of a family of orthogonal polynomials $p_n(x)$ is symmetric,*

$$X = X^\top = \begin{bmatrix} a_0 & b_0 & & \\ b_0 & a_1 & b_1 & \\ & b_1 & a_2 & \ddots \\ & & \ddots & \ddots \end{bmatrix},$$

if and only if $p_n(x)$ is up-to-sign a fixed constant scaling of orthonormal: for $q_n(x) := \pi_n(x)/\|\pi_n\|$ we have for a fixed $\alpha \in \mathbb{R}$ and $s_n \in \{-1, 1\}$

$$p_n(x) = \alpha s_n q_n(x).$$

Proof First, assume $p_n(x)$ has the specified form. Noting that $\|q_n\|^2 = 1$ and thence $\|p_n\|^2 = \alpha^2$, if $p_n(x) = \alpha s_n q_n(x)$ we have

$$b_n = \frac{\langle xp_n, p_{n+1}\rangle}{\|p_{n+1}\|^2} = s_n s_{n+1}\langle xq_n, q_{n+1}\rangle = s_n s_{n+1}\langle q_n, xq_{n+1}\rangle = \frac{\langle p_n, xp_{n+1}\rangle}{\|p_n\|^2} = c_n$$

and therefore X is symmetric.

Conversely, suppose $X = X^\top$, i.e., $b_n = c_n$ and write the corresponding orthogonal polynomials as $p_n(x) = \alpha_n q_n(x)$. We have

$$\begin{aligned} b_n &= \frac{\langle xp_n, p_{n+1}\rangle}{\|p_{n+1}\|^2} = \frac{\alpha_n}{\alpha_{n+1}}\langle xq_n, q_{n+1}\rangle = \frac{\alpha_n}{\alpha_{n+1}}\langle q_n, xq_{n+1}\rangle = \frac{\alpha_n^2}{\alpha_{n+1}^2}\frac{\langle p_n, xp_{n+1}\rangle}{\|p_n\|^2} \\ &= \frac{\alpha_n^2}{\alpha_{n+1}^2}c_n = \frac{\alpha_n^2}{\alpha_{n+1}^2}b_n. \end{aligned}$$

Hence $\alpha_n^2 = \alpha_{n+1}^2$ which implies that $\alpha_{n+1} = \pm\alpha_n$. By induction the result follows, where $\alpha := \alpha_0$. ■

Remark Every compactly supported integrable weight generates a family of orthonormal polynomials, which in turn generates a Jacobi matrix. There is a "Spectral Theorem for Jacobi matrices" that says one can go the other way: every tridiagonal symmetric matrix with bounded entries is a Jacobi matrix for some integrable weight with compact support. This is an example of what Barry Simon calls a "Gem of spectral theory".

Example 24 (uniform weight orthonormal polynomials)**.** Consider computing orthonormal polynomials with respect to $w(x) = 1$ on $[0, 1]$. Above we constructed the monic OPs $\pi_0(x), \ldots, \pi_3(x)$ so we can deduce the orthonormal polynomials by dividing by their norm, but there is another way: writing $q_n(x) = k_n\pi_n(x)$, find the normalisation k_n that turns the 3-term recurrence into a symmetric matrix. We can write the 3-term recurrence coefficients for monic OPs as a multiplication matrix:

$$x[\pi_0(x)|\pi_1(x)|\cdots] = [\pi_0(x)|\pi_1(x)|\cdots]\underbrace{\begin{bmatrix} 1/2 & 1/12 & & & \\ 1 & 1/2 & 1/15 & & \\ & 1 & 1/2 & \ddots & \\ & & 1 & \ddots & \ddots \\ & & & \ddots & \end{bmatrix}}_{X}$$

The previous theorem says that if we rescale the polynomials so that the resulting Jacobi matrix is symmetric than they are by necessity the orthonormal polynomials. In particular, consider writing:

$$[q_0(x)|q_1(x)|\cdots] = [\pi_0(x)|\pi_1(x)|\cdots]\underbrace{\begin{bmatrix} k_0 & & & \\ & k_1 & & \\ & & k_2 & \\ & & & \ddots \end{bmatrix}}_{K}$$

where we want to find the normalisation constants. Since $\|\pi_0\| = 1$ we know $k_0 = 1$. We have

$$x[q_0(x)|q_1(x)|\cdots] = [\pi_0(x)|\pi_1(x)|\cdots]XK = [q_0(x)|q_1(x)|\cdots]\underbrace{K^{-1}XK}_{J}$$

where

$$J = \begin{bmatrix} a_0 & c_0k_1 & & & \\ k_1^{-1} & a_1 & c_1k_2/k_1 & & \\ & k_1/k_2 & a_2 & c_2k_3/k_2 & \\ & & \ddots & \ddots & \ddots \end{bmatrix}.$$

Thus for this to be symmetric we need

$$c_0k_1 = k_1^{-1}, c_1k_2/k_1 = k_2^{-1}, c_2k_3/k_2 = k_3^{-1}, \dots$$

Note that

$$c_2 = \frac{\langle \pi_2, x\pi_3 \rangle}{\|\pi_2\|^2} = 180 \int_0^1 (x^6 - 5x^5/2 + 34x^4/15 - 9x^3/10 + 3x^2/20 - x/120)\mathrm{d}x = 9/140.$$

Thus we have (noting that the k_n are all positive which simplifies the square-roots):

$$\begin{aligned} k_1 &= 1/\sqrt{c_0} = 2\sqrt{3}, \\ k_2 &= k_1/\sqrt{c_1} = 6\sqrt{5}, \\ k_3 &= k_2/\sqrt{c_2} = 20\sqrt{7}. \end{aligned}$$

Thus we have

$$\begin{aligned} q_0(x) &= \pi_0(x) = 1, \\ q_1(x) &= 2\sqrt{3}\pi_1(x) = \sqrt{3}(2x-1), \\ q_2(x) &= 6\sqrt{5}\pi_2(x) = \sqrt{5}(6x^2 - 6x + 1), \\ q_3(x) &= 20\sqrt{7}\pi_3(x) = \sqrt{7}(20x^3 - 30x^2 + 12x - 1), \end{aligned}$$

which have the Jacobi matrix

$$J = \begin{bmatrix} 1/2 & 1/(2\sqrt{3}) & & & & \\ 1/(2\sqrt{3}) & 1/2 & 1/\sqrt{15} & & & \\ & 1/\sqrt{15} & 1/2 & 3/(2\sqrt{35}) & & \\ & & 3/(2\sqrt{35}) & 1/2 & \ddots & \\ & & & & \ddots & \ddots \end{bmatrix}.$$

Example 25 (expansion via Jacobi matrix)**.** What are the expansion coefficients of $x^3 - x + 1$ in $\{q_n\}$? We could deduce this by computing the inner products though its actually simpler to use the multiplication matrix. In particular if we write

$$Q(x) := [q_0(x)|q_1(x)|q_2(x)|\cdots]$$

Then we have (note: $q_0(x) \equiv 1$ only because the weight integrates to 1) $1 = Q(x)\boldsymbol{e}_1$. This tells us that:

$$x = xQ(x)\boldsymbol{e}_1 = Q(x)X\boldsymbol{e}_1 = Q(x)\begin{bmatrix} 1/2 \\ 1/(2\sqrt{3}) \\ 0 \\ \vdots \end{bmatrix}.$$

Continuing we have

$$x^2 = Q(x)X\begin{bmatrix}1/2\\ 1/(2\sqrt{3})\\ 0\\ \vdots\end{bmatrix} = Q(x)\begin{bmatrix}1/3\\ 1/(2\sqrt{3})\\ 1/(6\sqrt{5})\\ 0\\ \vdots\end{bmatrix}$$

$$x^3 = Q(x)X\begin{bmatrix}1/3\\ 1/(2\sqrt{3})\\ 1/(6\sqrt{5})\\ 0\\ \vdots\end{bmatrix} = Q(x)\begin{bmatrix}1/4\\ \frac{3\sqrt{3}}{20}\\ \frac{1}{4\sqrt{5}}\\ \frac{1}{20\sqrt{7}}\\ 0\\ \vdots\end{bmatrix}$$

Thus by linearity we find that

$$\begin{aligned}x^3 - x + 1 &= Q(x)\begin{bmatrix}3/4\\ -1/(20\sqrt{3})\\ \frac{1}{4\sqrt{5}}\\ \frac{1}{20\sqrt{7}}\\ 0\\ \vdots\end{bmatrix}\\ &= \frac{3}{4}q_0(x) - \frac{1}{20\sqrt{3}}q_1(x) + \frac{1}{4\sqrt{5}}q_2(x) + \frac{1}{20\sqrt{7}}q_3(x).\end{aligned}$$

VI.2 Classical Orthogonal Polynomials

Classical orthogonal polynomials are special families of orthogonal polynomials with a number of beautiful properties, for example (1) their derivatives are also OPs and (2) they are eigenfunctions of simple differential operators. As stated above orthogonal polynomials are uniquely defined by the weight $w(x)$ and the constant k_n and hence we can define the classical OPs by specifying their weights and normalisation constants.

The classical orthogonal polynomials are:

1. Chebyshev polynomials (1st kind) $T_n(x)$: $w(x) = 1/\sqrt{1-x^2}$ on $[-1,1]$.
2. Chebyshev polynomials (2nd kind) $U_n(x)$: $\sqrt{1-x^2}$ on $[-1,1]$.
3. Legendre polynomials $P_n(x)$: $w(x) = 1$ on $[-1,1]$.
4. Ultrapsherical polynomials (my fav!): $C_n^{(\lambda)}(x)$: $w(x) = (1-x^2)^{\lambda-1/2}$ on $[-1,1]$, $\lambda \neq 0$, $\lambda > -1/2$.
5. Jacobi polynomials: $P_n^{(a,b)}(x)$: $w(x) = (1-x)^a(1+x)^b$ on $[-1,1]$, $a, b > -1$.
6. Laguerre polynomials: $L_n(x)$: $w(x) = \exp(-x)$ on $[0,\infty)$.
7. Hermite polynomials $H_n(x)$: $w(x) = \exp(-x^2)$ on $(-\infty,\infty)$.

In the notes we will discuss:

1. Chebyshev polynomials: These are closely linked to Fourier series and are one of the most powerful tools in numerics.
2. Legendre polynomials: These have no simple closed-form expression but can be defined in terms of a Rodriguez formula, a feature that applies to all other classical families.

VI.2.1 Chebyshev polynomials

There are four families of Chebyshev polynomials but we will consider the first two:

Definition 39 (Chebyshev polynomials, 1st kind)**.** $T_n(x)$ are orthogonal with respect to $1/\sqrt{1-x^2}$ and satisfy:

$$T_0(x) = 1,$$
$$T_n(x) = 2^{n-1}x^n + O(x^{n-1})$$

Definition 40 (Chebyshev polynomials, 2nd kind)**.** $U_n(x)$ are orthogonal with respect to $\sqrt{1-x^2}$ and satisfy:

$$U_n(x) = 2^n x^n + O(x^{n-1})$$

A beautiful fact is that Chebyshev polynomials are really trigonometric polynomials in disguise:

Theorem 16 (Chebyshev T are cos)**.** *For* $-1 \le x \le 1$

$$T_n(x) = \cos n \operatorname{acos} x.$$

In other words

$$T_n(\cos\theta) = \cos n\theta.$$

Proof

We need to show that $p_n(x) := \cos n \operatorname{acos} x$ are

1. graded polynomials
2. orthogonal w.r.t. $1/\sqrt{1-x^2}$ on $[-1,1]$, and
3. have the right normalisation constant $k_n = 2^{n-1}$ for $n = 2, \ldots$.

Property (2) follows under a change of variables:

$$\int_{-1}^{1} \frac{p_n(x)p_m(x)}{\sqrt{1-x^2}}\mathrm{d}x = \int_0^{\pi} \frac{\cos(n\theta)\cos(m\theta)}{\sqrt{1-\cos^2\theta}}\sin\theta\mathrm{d}\theta = \int_0^{\pi} \cos(n\theta)\cos(m\theta)\mathrm{d}x = 0$$

if $n \neq m$.

To see that they are graded we use the fact that

$$xp_n(x) = \cos\theta\cos n\theta = \frac{\cos(n-1)\theta + \cos(n+1)\theta}{2} = \frac{p_{n-1}(x) + p_{n+1}(x)}{2}.$$

In other words $p_{n+1}(x) = 2xp_n(x) - p_{n-1}(x)$. Since each time we multiply by $2x$ and $p_0(x) = 1$ we have by induction

$$p_{n+1}(x) = 2x(2^{n-1}x^n + O(x^{n-1})) + O(x^{n-1})) = 2^n x^{n+1} + O(x^n)$$

which completes the proof.

■

Recall that the 3-term recurrence is an important property of a family of orthogonal polynomials. We can deduce from the relationship with cosines the following:

Corollary 10 (Chebyshev 3-term recurrence)**.**

$$\begin{aligned} xT_0(x) &= T_1(x) \\ xT_n(x) &= \frac{T_{n-1}(x) + T_{n+1}(x)}{2} \end{aligned}$$

Proof This is rewriting the expression we used to show that $p_n(x)$ are graded in the previous proof. ■

Chebyshev polynomials are particularly powerful as their expansions are cosine series in disguise: for

$$f(x) = \sum_{k=0}^{\infty} \check{f}_k T_k(x)$$

we have

$$f(\cos\theta) = \sum_{k=0}^{\infty} \check{f}_k \cos k\theta.$$

Thus the coefficients can be recovered fast using FFT-based techniques.

We will also see the following:

Theorem 17 (Chebyshev U are sin)**.** *For $x = \cos\theta$,*

$$U_n(x) = \frac{\sin(n+1)\theta}{\sin\theta}$$

which satisfy:

$$\begin{aligned} xU_0(x) &= U_1(x)/2 \\ xU_n(x) &= \frac{U_{n-1}(x)}{2} + \frac{U_{n+1}(x)}{2}. \end{aligned}$$

Proof Shown in the problem sheet. ■

VI.2.2 Legendre polynomials

Definition 41 (Legendre)**.** Legendre polynomials $P_n(x)$ are orthogonal polynomials with respect to $w(x) = 1$ on $[-1, 1]$, with

$$k_n = \frac{1}{2^n}\binom{2n}{n} = \frac{(2n)!}{2^n(n!)^2}$$

The reason for this complicated normalisation constant is both historical and that it leads to simpler formulae for recurrence relationships.

Classical orthogonal polynomials have *Rodriguez formulae*, defining orthogonal polynomials as high order derivatives of simple functions. In this case we have:

Lemma 9 (Legendre Rodriguez formula)**.**

$$P_n(x) = \frac{1}{(-2)^n n!}\frac{\mathrm{d}^n}{\mathrm{d}x^n}(1-x^2)^n$$

Proof We need to verify:

1. graded polynomials
2. orthogonal to all lower degree polynomials on $[-1,1]$, and
3. have the right normalisation constant $k_n = \frac{1}{2^n}\binom{2n}{n}$.

(1) follows since its a degree n polynomial (the n-th derivative of a degree $2n$ polynomial). (2) follows by integration by parts. Note that $(1-x^2)^n$ and its first $n-1$ derivatives vanish at ± 1. If r_m is a degree $m < n$ polynomial we have:

$$\begin{aligned}\int_{-1}^{1}\frac{\mathrm{d}^n}{\mathrm{d}x^n}(1-x^2)^n r_m(x)\mathrm{d}x &= -\int_{-1}^{1}\frac{\mathrm{d}^{n-1}}{\mathrm{d}x^{n-1}}(1-x^2)^n r_m'(x)\mathrm{d}x \\ &= \cdots = (-)^n\int_{-1}^{1}(1-x^2)^n r_m^{(n)}(x)\mathrm{d}x = 0.\end{aligned}$$

(3) follows since:

$$\begin{aligned}\frac{\mathrm{d}^n}{\mathrm{d}x^n}[(-)^n x^{2n} + O(x^{2n-1})] &= (-)^n 2n\frac{\mathrm{d}^{n-1}}{\mathrm{d}x^{n-1}}x^{2n-1} + O(x^{2n-1})] \\ &= (-)^n 2n(2n-1)\frac{\mathrm{d}^{n-2}}{\mathrm{d}x^{n-2}}x^{2n-2} + O(x^{2n-2})] = \cdots \\ &= (-)^n 2n(2n-1)\cdots(n+1)x^n + O(x^{n-1}) \\ &= (-)^n\frac{(2n)!}{n!}x^n + O(x^{n-1})\end{aligned}$$

■

This allows us to determine the coefficients $k_n^{(\lambda)}$ which are useful in proofs. In particular we will use $k_n^{(2)}$:

Lemma 10 (Legendre monomial coefficients)**.**

$$\begin{aligned}P_0(x) &= 1 \\ P_1(x) &= x \\ P_n(x) &= \underbrace{\frac{(2n)!}{2^n(n!)^2}}_{k_n}x^n - \underbrace{\frac{(2n-2)!}{2^n(n-2)!(n-1)!}}_{k_n^{(2)}}x^{n-2} + O(x^{n-4}).\end{aligned}$$

Here the $O(x^{n-4})$ is as $x \to \infty$, which implies that the term is a polynomial of degree $\leq n-4$. For $n = 2, 3$ the $O(x^{n-4})$ term is therefore precisely zero.

Proof

The $n = 0$ and 1 case are immediate. For the other case we expand $(1 - x^2)^n$ to get:

$$\begin{aligned}(-)^n \frac{\mathrm{d}^n}{\mathrm{d}x^n}(1-x^2)^n &= \frac{\mathrm{d}^n}{\mathrm{d}x^n}[x^{2n} - nx^{2n-2} + O(x^{2n-4})] \\ &= (2n)\cdots(2n-n+1)x^n - n(2n-2)\cdots(2n-2-n+1)x^{n-2} + O(x^{n-4}) \\ &= \frac{(2n)!}{n!}x^n - \frac{n(2n-2)!}{(n-2)!}x^{n-2} + O(x^{n-4})\end{aligned}$$

Multiplying through by $\frac{1}{2^n(n!)}$ completes the derivation.

■

Theorem 18 (Legendre 3-term recurrence)**.**

$$\begin{aligned}xP_0(x) &= P_1(x) \\ (2n+1)xP_n(x) &= nP_{n-1}(x) + (n+1)P_{n+1}(x)\end{aligned}$$

Proof The $n = 0$ case is immediate (since $w(x) = w(-x)$ $a_n = 0$, from the problem sheet). For the other cases we match terms:

$$\begin{aligned}(2n+1)xP_n(x) - nP_{n-1}(x) - (n+1)P_{n+1}(x) \\ = [(2n+1)k_n - (n+1)k_{n+1}]x^{n+1} \\ + [(2n+1)k_n^{(2)} - nk_{n-1} - (n+1)k_{n+1}^{(2)}]x^{n-1} + O(x^{n-3})\end{aligned}$$

Using the expressions for k_n and $k_n^{(2)}$ above we have (leaving the manipulations as an exercise):

$$\begin{aligned}(2n+1)k_n - (n+1)k_{n+1} &= \frac{(2n+1)!}{2^n(n!)^2} - (n+1)\frac{(2n+2)!}{2^{n+1}((n+1)!)^2} = 0 \\ (2n+1)k_n^{(2)} - nk_{n-1} - (n+1)k_{n+1}^{(2)} &= -(2n+1)\frac{(2n-2)!}{2^n(n-2)!(n-1)!} - n\frac{(2n-2)!}{2^{n-1}((n-1)!)^2} \\ &\quad + (n+1)\frac{(2n)!}{2^{n+1}(n-1)!n!} \\ &= 0\end{aligned}$$

Thus

$$(2n+1)xP_n(x) - nP_{n-1}(x) - (n+1)P_{n+1}(x) = O(x^{n-3})$$

But as it is orthogonal to $P_k(x)$ for $0 \le k \le n-3$ it must be zero. ■

Appendix A

Asymptotics and Computational Cost

We introduce Big-O, little-o and asymptotic notation and see how they can be used to describe computational cost.

A.1 Asymptotics as $n \to \infty$

Big-O, little-o, and "asymptotic to" are used to describe behaviour of functions at infinity.

Definition 42 (Big-O)**.**

$$f(n) = O(\phi(n)) \qquad (\text{as } n \to \infty)$$

means $\left|\frac{f(n)}{\phi(n)}\right|$ is bounded for sufficiently large n. That is, there exist constants C and N_0 such that, for all $n \geq N_0$, $|\frac{f(n)}{\phi(n)}| \leq C$.

Definition 43 (little-O)**.**

$$f(n) = o(\phi(n)) \qquad (\text{as } n \to \infty)$$

means $\lim_{n\to\infty} \frac{f(n)}{\phi(n)} = 0$.

Definition 44 (asymptotic to)**.**

$$f(n) \sim \phi(n) \qquad (\text{as } n \to \infty)$$

means $\lim_{n\to\infty} \frac{f(n)}{\phi(n)} = 1$.

Example 26 (asymptotics with n)**.** 1.

$$\frac{\cos n}{n^2 - 1} = O(n^{-2})$$

as

$$\left|\frac{\frac{\cos n}{n^2-1}}{n^{-2}}\right| \leq \left|\frac{n^2}{n^2-1}\right| \leq 2$$

for $n \geq N_0 = 2$.

2.
$$\log n = o(n)$$
as $\lim_{n\to\infty} \frac{\log n}{n} = 0$.

3.
$$n^2 + 1 \sim n^2$$
as $\frac{n^2+1}{n^2} \to 1$.

Note we sometimes write $f(O(\phi(n)))$ for a function of the form $f(g(n))$ such that $g(n) = O(\phi(n))$.

We have some simple algebraic rules:

Proposition 15 (Big-O rules)**.**

$$O(\phi(n))O(\psi(n)) = O(\phi(n)\psi(n)) \qquad (as\ n \to \infty)$$
$$O(\phi(n)) + O(\psi(n)) = O(|\phi(n)| + |\psi(n)|) \qquad (as\ n \to \infty).$$

Proof See any standard book on asymptotics, eg F.W.J. Olver, Asymptotics and Special Functions. ■

A.2 Asymptotics as $x \to x_0$

We also have Big-O, little-o and "asymptotic to" at a point:

Definition 45 (Big-O)**.**
$$f(x) = O(\phi(x)) \qquad (\text{as } x \to x_0)$$
means $|\frac{f(x)}{\phi(x)}|$ is bounded in a neighbourhood of x_0. That is, there exist constants C and r such that, for all $0 \le |x - x_0| \le r$, $|\frac{f(x)}{\phi(x)}| \le C$.

Definition 46 (little-O)**.**
$$f(x) = o(\phi(x)) \qquad (\text{as } x \to x_0)$$
means $\lim_{x\to x_0} \frac{f(x)}{\phi(x)} = 0$.

Definition 47 (asymptotic to)**.**

$$f(x) \sim \phi(x) \qquad (\text{as } x \to x_0)$$

means $\lim_{x\to x_0} \frac{f(x)}{\phi(x)} = 1$.

Example 27 (asymptotics with x)**.**

$$\exp x = 1 + x + O(x^2) \qquad \text{as } x \to 0$$

since $\exp x = 1 + x + \frac{\exp t}{2}x^2$ for some $t \in [0, x]$ and

$$\left|\frac{\frac{\exp t}{2}x^2}{x^2}\right| \le \frac{3}{2}$$

provided $x \le 1$.

A.3 Computational cost

We will use Big-O notation to describe the computational cost of algorithms. Consider the following simple sum

$$\sum_{k=1}^{n} x_k^2$$

which we might implement as:

```
function sumsq(x)
    n = length(x)
    ret = 0.0
    for k = 1:n
        ret = ret + x[k]^2
    end
    ret
end
```

```
sumsq (generic function with 1 method)
```

Each step of this algorithm consists of one memory look-up (`z = x[k]`), one multiplication (`w = z*z`) and one addition (`ret = ret + w`). We will ignore the memory look-up in the following discussion. The number of CPU operations per step is therefore 2 (the addition and multiplication). Thus the total number of CPU operations is $2n$. But the constant 2 here is misleading: we didn't count the memory look-up, thus it is more sensible to just talk about the asymptotic complexity, that is, the *computational cost* is $O(n)$.

Now consider a double sum like:

$$\sum_{k=1}^{n} \sum_{j=1}^{k} x_j^2$$

which we might implement as:

```
function sumsq2(x)
    n = length(x)
    ret = 0.0
    for k = 1:n
        for j = 1:k
            ret = ret + x[j]^2
        end
    end
    ret
end
```

```
sumsq2 (generic function with 1 method)
```

Now the inner loop is $O(1)$ operations (we don't try to count the precise number), which we do k times for $O(k)$ operations as $k \to \infty$. The outer loop therefore takes

$$\sum_{k=1}^{n} O(k) = O\left(\sum_{k=1}^{n} k\right) = O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

operations.

Appendix B

Integers

B.1 Integers

In this section we discuss the following:

1. Unsigned integers: how computers represent non-negative integers using only p-bits, via modular arithmetic.
2. Signed integers: how negative integers are handled using the Two's-complement format.

Mathematically, CPUs only act on p-bits at a time, with 2^p possible sequences. That is, essentially all functions f are either of the form $f : \mathbb{Z}_{2^p} \to \mathbb{Z}_{2^p}$ or $f : \mathbb{Z}_{2^p} \times \mathbb{Z}_{2^p} \to \mathbb{Z}_{2^p}$, where we use the following notation:

Definition 48 (finite integers)**.** Denote the set of the first m non-negative integers as $\mathbb{Z}_m := \{0, 1, \ldots, m-1\}$.

To translate between integers and bits we will need to write integers in binary format. That is, as sequence of `0`s and `1`s:

Definition 49 (binary format)**.** For $B_0, \ldots, B_p \in \{0, 1\}$ denote an integer in *binary format* by:

$$\pm(B_p \ldots B_1 B_0)_2 := \pm \sum_{k=0}^{p} B_k 2^k$$

Example 28 (integers in binary)**.** A simple integer example is $5 = 2^2 + 2^0 = (101)_2$. On the other hand, we write $-5 = -(101)_2$. Another example is $258 = 2^8 + 2 = (100000010)_2$.

B.1.1 Unsigned Integers

Computers represent integers by a finite number of p-bits, with 2^p possible combinations of 0s and 1s. Denote these p-bits as $B_{p-1} \ldots B_1 B_0$ where $B_k \in \{0, 1\}$. For *unsigned integers* (non-negative integers) these bits dictate the first p binary digits: $(B_{p-1} \ldots B_1 B_0)_2$. Integers represented with p-bits on a computer are interpreted as representing elements of $\mathbb{Z}_{2^p}$ and

integer arithmetic on a computer is equivalent to arithmetic modulo 2^p. We denote modular arithmetic with $m = 2^p$ as follows:

$$x \oplus_m y := (x + y) \pmod m$$
$$x \ominus_m y := (x - y) \pmod m$$
$$x \otimes_m y := (x * y) \pmod m$$

When m is implied by context we just write $\oplus, \ominus, \otimes$. Note that the (mod m) function simply drops all bits except for the first p-bits when writing a number in binary.

Example 29 (arithmetic with 8-bit unsigned integers)**.** If the result of an operation lies between 0 and $m = 2^8 = 256$ then airthmetic works exactly like standard integer arithmetic. For example,

$$17 \oplus_{256} 3 = 20 \pmod{256} = 20$$
$$17 \ominus_{256} 3 = 14 \pmod{256} = 14$$

Example 30 (overflow with 8-bit unsigned integers)**.** If we go beyond the range the result "wraps around". For example, with true integers we have

$$255 + 1 = (11111111)_2 + (00000001)_2 = (100000000)_2 = 256$$

However, the result is impossible to store in just 8-bits! So as mentioned instead it treats the integers as elements of $\mathbb{Z}_{256}$ by dropping any extra digits:

$$255 \oplus_{256} 1 = 255 + 1 \pmod{256} = (100000000)_2 \pmod{256} = 0.$$

On the other hand, if we go below 0 we wrap around from above:

$$3 \ominus_{256} 5 = -2 \pmod{256} = 254 = (11111110)_2$$

Example 31 (multiplication of 8-bit unsigned integers)**.** Multiplication works similarly: for example,

$$254 \otimes_{256} 2 = 254 * 2 \pmod{256} = (11111110)_2 * 2 \pmod{256} = (111111100)_2 \pmod{256} = 252.$$

Note that multiplication by 2 is the same as shifting the binary digits left by one, just as multiplication by 10 shifts base-10 digits left by 1.

B.1.2 Signed integer

Signed integers use the Two's complemement convention. The convention is if the first bit is 1 then the number is negative: in this case if the bits had represented the unsigned integer $2^p - y$ then the represent the signed integer $-y$. Thus for $p = 8$ we are interpreting 2^7 through $2^8 - 1$ as negative numbers. More precisely:

Definition 50 (signed integers)**.** Denote the finite signed integers as

$$\mathbb{Z}^{\mathrm{s}}_{2^p} := \{-2^{p-1}, \ldots, -1, 0, 1, \ldots, 2^{p-1} - 1\}.$$

Definition 51 (Shifted mod)**.** Define for $y = x \pmod{2^p}$

$$x \pmod[\mathrm{s}]{2^p} := \begin{cases} y & 0 \le y \le 2^{p-1} - 1 \\ y - 2^p & 2^{p-1} \le y \le 2^p - 1 \end{cases}$$

Note that if $R_p(x) = x \pmod^{\mathrm{s}} 2^p$ then it can be viewed as a map $R_p : \mathbb{Z} \to \mathbb{Z}^{\mathrm{s}}_{2^p}$ or a one-to-one map $R_p : \mathbb{Z}_{2^p} \to \mathbb{Z}^{\mathrm{s}}_{2^p}$ whose inverse is $R_p^{-1}(x) = x \pmod{2^p}$. It can also be viewed as the identity map on signed integers $R_p : \mathbb{Z}^{\mathrm{s}}_{2^p} \to \mathbb{Z}^{\mathrm{s}}_{2^p}$, that is, $R_p(x) = x$ if $x \in \mathbb{Z}^{\mathrm{s}}_{2^p}$.

Arithmetic works precisely the same for signed and unsigned integers up to the mapping R_p, e.g. we have for $m = 2^p$

$$\begin{aligned} x \oplus^{\mathrm{s}}_m y &:= (x + y) \pmod^{\mathrm{s}} m \\ x \ominus^{\mathrm{s}}_m y &:= (x - y) \pmod^{\mathrm{s}} m \\ x \otimes^{\mathrm{s}}_m y &:= (x * y) \pmod^{\mathrm{s}} m \end{aligned}$$

Example 32 (addition of 8-bit signed integers)**.** Consider `(-1) + 1` in 8-bit arithmetic:

$$-1 \oplus^{\mathrm{s}}_{256} 1 = -1 + 1 \pmod^{\mathrm{s}} 256 = 0$$

On the bit level this computation is exactly the same as unsigned integers. We represent the number -1 using the same bits as the unsigned integer $2^8 - 1 = 255$, that is using the bits `11111111` (i.e., we store it equivalently to $(11111111)_2 = 255$) and the number 1 is stored using the bits `00000001`. When we add this with true integer arithmetic we have

$$\begin{aligned} &(011111111)_2 + \\ &(000000001)_2 \ = \\ &(100000000)_2 \end{aligned}$$

Modular arithmetic drops the leading 1 and we are left with all zeros.

Example 33 (signed overflow with 8-bit signed integers)**.** If we go above $2^{p-1}-1 = 2^7-1 = 127$ we have perhaps unexpected results:

$$127 \oplus^{\mathrm{s}}_{256} 1 = 128 \pmod^{\mathrm{s}} 256 = 128 - 256 = -128.$$

Again on the bit level this computation is exactly the same as unsigned integers. We represent the number 127 using the bits `01111111` and the number 1 is stored using the bits `00000001`. When we add this with true integer arithmetic we have

$$\begin{aligned} &(01111111)_2 + \\ &(00000001)_2 \ = \\ &(10000000)_2 \end{aligned}$$

Because the first bit is `1` we interpret this as a negative number using the formula:

$$(10000000)_2 \pmod^{\mathrm{s}} 256 = 128 (\mathrm{mod}^{\mathrm{s}}\ 256) = -128.$$

Example 34 (multiplication of 8-bit signed integers)**.** Consider computation of `(-2) * 2`:

$$(-2) \otimes^{\mathrm{s}}_{2^p} 2 = -4 \pmod^{\mathrm{s}} 2^p = -4$$

On the bit level, the bits of -2 (which is one less than -1) are `11111110`. Multiplying by 2 is like multiplying by 10 in base-10, that is, we shift the bits. Hence in true arithmetic we have

$$\begin{aligned} &(011111110)_2 * 2 = \\ &(111111100)_2 \end{aligned}$$

We drop the leading 1 due to modular arithmetic. We still have a leading 1 hence the number is viewed as negative. In particular we have

$$\begin{aligned} (111111100)_2 \pmod^{\mathrm{s}} 256 \ &= (11111100)_2 \pmod^{\mathrm{s}} 256 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 \pmod^{\mathrm{s}} 256 \\ &= 252 \pmod^{\mathrm{s}} 256 = -4. \end{aligned}$$

B.1.3 Hexadecimal format

In coding it is often convenient to use base-16 as it is a power of 2 but uses less characters than binary. The digits used are 0 through 9 followed by $a = 10$, $b = 11$, $c = 12$, $d = 13$, $e = 14$, and $f = 15$.

Example 35 (Hexadecimal number)**.** We can interpret a number in format as follows:

$$(a5f2)_{16} = a * 16^3 + 5 * 16^2 + f * 16 + 2 = 10 * 16^3 + 5 * 16^2 + 15 * 16 + 2 = 42,482$$

We will see in the labs that unsigned integers are displayed in base-16.

Appendix C

Permutation Matrices

Permutation matrices are matrices that represent the action of permuting the entries of a vector, that is, matrix representations of the symmetric group S_n, acting on $\mathbb{R}^n$. Recall every $\sigma \in S_n$ is a bijection between $\{1, 2, \ldots, n\}$ and itself. We can write a permutation σ in *Cauchy notation*:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ \sigma_1 & \sigma_2 & \sigma_3 & \cdots & \sigma_n \end{pmatrix}$$

where $\{\sigma_1, \ldots, \sigma_n\} = \{1, 2, \ldots, n\}$ (that is, each integer appears precisely once). We denote the *inverse permutation* by σ^{-1}, which can be constructed by swapping the rows of the Cauchy notation and reordering.

We can encode a permutation in vector $\sigma = [\sigma_1, \ldots, \sigma_n]$. This induces an action on a vector (using indexing notation)

$$\boldsymbol{v}[\sigma] = \begin{bmatrix} v_{\sigma_1} \\ \vdots \\ v_{\sigma_n} \end{bmatrix}$$

Example 36 (permutation of a vector)**.** Consider the permutation σ given by

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 2 & 5 & 3 \end{pmatrix}$$

We can apply it to a vector:

```
using LinearAlgebra
σ = [1, 4, 2, 5, 3]
v = [6, 7, 8, 9, 10]
v[σ] # we permutate entries of v

5-element Vector{Int64}:
  6
  9
  7
 10
  8
```

Its inverse permutation σ^{-1} has Cauchy notation coming from swapping the rows of the Cauchy notation of σ and sorting:

$$\begin{pmatrix} 1 & 4 & 2 & 5 & 3 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \to \begin{pmatrix} 1 & 2 & 4 & 3 & 5 \\ 1 & 3 & 2 & 5 & 4 \end{pmatrix}$$

Note that the operator

$$P_\sigma(\boldsymbol{v}) = \boldsymbol{v}[\sigma]$$

is linear in $\boldsymbol{v}$, therefore, we can identify it with a matrix whose action is:

$$P_\sigma \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} v_{\sigma_1} \\ \vdots \\ v_{\sigma_n} \end{bmatrix}.$$

The entries of this matrix are

$$P_\sigma[k, j] = \boldsymbol{e}_k^\top P_\sigma \boldsymbol{e}_j = \boldsymbol{e}_k^\top \boldsymbol{e}_{\sigma_j^{-1}} = \delta_{k,\sigma_j^{-1}} = \delta_{\sigma_k,j}$$

where $\delta_{k,j}$ is the *Kronecker delta*:

$$\delta_{k,j} := \begin{cases} 1 & k = j \\ 0 & \text{otherwise} \end{cases}.$$

This construction motivates the following definition:

Definition 52 (permutation matrix)**.** $P \in \mathbb{R}^{n\times n}$ is a permutation matrix if it is equal to the identity matrix with its rows permuted.

Proposition 16 (permutation matrix inverse)**.** *Let P_σ be a permutation matrix corresponding to the permutation σ. Then*

$$P_\sigma^\top = P_{\sigma^{-1}} = P_\sigma^{-1}$$

That is, P_σ is orthogonal*:*

$$P_\sigma^\top P_\sigma = P_\sigma P_\sigma^\top = I.$$

Proof

We prove orthogonality via:

$$\boldsymbol{e}_k^\top P_\sigma^\top P_\sigma \boldsymbol{e}_j = (P_\sigma \boldsymbol{e}_k)^\top P_\sigma \boldsymbol{e}_j = \boldsymbol{e}_{\sigma_k^{-1}}^\top \boldsymbol{e}_{\sigma_j^{-1}} = \delta_{k,j}$$

This shows $P_\sigma^\top P_\sigma = I$ and hence $P_\sigma^{-1} = P_\sigma^\top$.

■