MATH50003 Numerical Analysis

Sheehan Olver

February 1, 2024

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 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 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:

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

for some $x \leq t \leq x+h$, by bounding:

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

■

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 converge 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 $\epsilon^2 = 0$. Dual numbers are typically written as $a + b\epsilon$ where a and b are real.

This is very much analoguous to complex numbers, which are a field generated by 1 and i such that $\mathrm{i}^2 = -1$. Compare multiplication of each number type:

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

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 $a + 0\epsilon \in \mathbb{D}$.

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 we can extend it to dual numbers:

Definition 2 (dual extension)**.** Suppose a real-valued function f is differentiable at a. If

$$f(a+b\epsilon) = f(a) + bf'(a)\epsilon$$

then we say that it is a *dual extension at* a.

Thus, for basic functions we have natural extensions:

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

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 functions:

Lemma 2 (product and chain rule)**.** *If f is a dual extension at $g(a)$ and g is a dual extension at a, then $q(x) := f(g(x))$ is a dual extension at a. If f and g are dual extensions at a then $r(x) := f(x)g(x)$ is also dual extensions at a. In other words:*

$$\begin{aligned} q(a+b\epsilon) &= q(a) + bq'(a)\epsilon \\ r(a+b\epsilon) &= r(a) + br'(a)\epsilon \end{aligned}$$

Proof For q it follows immediately:

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

For r we have

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

A simple corollary is that any function defined in terms of addition, multiplication, composition, etc. of functions that are dual with differentiation will be differentiable via dual numbers.

Example 2 (differentiating non-polynomial)**.** Consider differentiating $f(x) = \exp(x^2 + \mathrm{e}^x)$ at the point $a = 1$ by evaluating on the duals:

$$f(1+\epsilon) = \exp(1 + 2\epsilon + \mathrm{e} + \mathrm{e}\epsilon) = \exp(1+\mathrm{e}) + \exp(1+\mathrm{e})(2+\mathrm{e})\epsilon.$$

Therefore we deduce that

$$f'(1) = \exp(1+\mathrm{e})(2+\mathrm{e}).$$

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:

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

The convergence theory of Newton's method is rich and beautiful but outside the scope of this module. But provided f is smooth, if x_0 is sufficiently close to a root this iteration will converge.

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.

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 Integers: unsigned (non-negative) and signed integers are representable using exactly p-bits by using modular arithmetic in all operations.

2. II.2 Reals: real numbers are approximated by floating point numbers, which are a computers version of scientific notation.

3. II.3 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.

4. II.4 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 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 3 (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 4 (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 3 (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 $258 = 2^8 + 2 = (100000010)_2$.

II.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:

$$\begin{aligned} x \oplus_m y &:= (x + y) \pmod m \\ x \ominus_m y &:= (x - y) \pmod m \\ x \otimes_m y &:= (x * y) \pmod m \end{aligned}$$

When m is implied by context we just write $\oplus, \ominus, \otimes$. Note that the $(\mathrm{mod}\ m)$ function simply drops all bits except for the first p-bits when writing a number in binary.

Example 4 (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 5 (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 6 (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.

II.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 5 (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 6 (Shifted mod)**.** Define for $y = x \pmod{2^p}$

$$x \pmod[\mathrm{s}]{2^p} := \begin{cases} y & 0 \leq y \leq 2^{p-1} - 1 \\ y - 2^p & 2^{p-1} \leq y \leq 2^p - 1 \end{cases}$$

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 $m = 2^p$

$$x \oplus^{\mathrm{s}}_m y := (x + y) \pmod[\mathrm{s}]{m}$$
$$x \ominus^{\mathrm{s}}_m y := (x - y) \pmod[\mathrm{s}]{m}$$
$$x \otimes^{\mathrm{s}}_m y := (x * y) \pmod[\mathrm{s}]{m}$$

Example 7 (addition of 8-bit signed integers)**.** Consider `(-1) + 1` in 8-bit arithmetic:

$$-1 \oplus^{\mathrm{s}}_{256} 1 = -1 + 1 \ (\mathrm{mod}^{\mathrm{s}}\ 256) = 0$$

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 8 (signed overflow with 8-bit signed integers)**.** If we go above $2^{p-1}-1 = 2^7-1 = 127$ we have perhaps unexpected results:

$$127 \oplus^{\mathrm{s}}_{256} 1 = 128 \ (\mathrm{mod}^{\mathrm{s}}\ 256) = 128 - 256 = -128.$$

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:

$$(10000000)_2 \ (\mathrm{mod}^{\mathrm{s}}\ 256) = 128(\mathrm{mod}^{\mathrm{s}}\ 256) = -128.$$

Example 9 (multiplication of 8-bit signed integers)**.** Consider computation of `(-2) * 2`:

$$(-2) \otimes^{\mathrm{s}}_{2^p} 2 = -4 \ (\mathrm{mod}^{\mathrm{s}}\ 2^p) = -4$$

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

$$\begin{aligned}(111111100)_2 \ (\mathrm{mod}^{\mathrm{s}}\ 256) \ &= (11111100)_2 \ (\mathrm{mod}^{\mathrm{s}}\ 256) = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 \ (\mathrm{mod}^{\mathrm{s}}\ 256)\\ &= 252 \ (\mathrm{mod}^{\mathrm{s}}\ 256) = -4.\end{aligned}$$

II.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 10 (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.

II.2 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 was a means to standardise 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.2.1 Real numbers in binary

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

Definition 7 (real binary format)**.** For $b_1, b_2, \ldots \in \{0, 1\}$, Denote a non-negative real number in *binary format* by:

$$(B_p \ldots B_0.b_1 b_2 b_3 \ldots)_2 := (B_p \ldots B_0)_2 + \sum_{k=1}^{\infty} \frac{b_k}{2^k}.$$

Example 11 (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.2.2 Floating-point numbers

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

Definition 8 (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

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

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

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

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

$$x = \pm 2^{q-\sigma} \times (1.b_1b_2b_3\ldots b_S)_2.$$

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

$$x = \pm 2^{1-\sigma} \times (0.b_1b_2b_3\ldots b_S)_2.$$

If $q = 2^Q - 1$ (i.e. all bits are 1) then the bits represent a special number, discussed later.

II.2.3 IEEE floating-point numbers

Definition 9 (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 12 (interpreting 16-bits as a float)**.** Consider the number with bits

0 10000 1010000000

assuming it is a half-prevision float (F_{16}). Since the sign bit is `0` it is positive. The exponent is $2^4 - \sigma = 16 - 15 = 1$ Hence this number is:

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

Example 13 (rational to 16-bits)**.** How is the number $1/3$ stored in F_{16}? Recall that

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

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

$$1.01010101010101010101010101\ldots \approx 1.0101010101 \in F_{16}$$

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

0 01101 0101010101

II.2.4 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 14 (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 15 (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.3 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 10 (rounding)**.** $\mathrm{fl}^{\mathrm{up}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ denotes the function that rounds a real number up to the nearest floating-point number that is greater or equal. $\mathrm{fl}^{\mathrm{down}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ denotes the function that rounds a real number down to the nearest floating-point number that is greater or equal. $\mathrm{fl}^{\mathrm{nearest}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ 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 $\mathrm{fl}^{\mathrm{nearest}}$ 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

$$x = 2^{q-\sigma}(1.b_1 b_2 \ldots b_S b_{S+1} \ldots)_2$$

where

$$\begin{aligned} x_- &:= \mathrm{fl}^{\mathrm{down}}(x) = 2^{q-\sigma}(1.b_1 b_2 \ldots b_S)_2 \\ x_+ &:= \mathrm{fl}^{\mathrm{up}}(x) = x_- + 2^{q-S} \end{aligned}$$

Write the half-way point as:

$$x_{\mathrm{h}} := \frac{x_+ + x_-}{2} = x_- + 2^{q-S-1} = 2^{q-\sigma}(1.b_1 b_2 \ldots b_S 1)_2$$

If $x_- \le x < x_{\mathrm{h}}$ then $\mathrm{fl}(x) = x_-$ and if $x_{\mathrm{h}} < x \le x_+$ then $\mathrm{fl}(x) = x_{\mathrm{h}}$. 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 $\oplus, \ominus, \otimes, \oslash : F_{\sigma,Q,S} \otimes F_{\sigma,Q,S} \to F_{\sigma,Q,S}$ as follows:

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

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 16 (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 $\mathrm{fl}(1.1) \neq 1 + 1/10$ and $\mathrm{fl}(0.1) \neq 1/10$ 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

$$\mathrm{fl}(1.1) + \mathrm{fl}(0.1) = (1.0011001100011)_2$$

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,

$$\mathrm{fl}(1.2) = \mathrm{fl}((1.0011001100110011001100\ldots)_2) = (1.0011001101)_2$$

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.3.1 Bounding errors in floating point arithmetic

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

Definition 11 (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 12 (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:

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

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*

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

where the relative error *is bounded by:*

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

Proof

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

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

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

$$x = 2^{q-\sigma}(1.b_1 b_2 \ldots b_S b_{S+1} \ldots)_2.$$

Define

$$\begin{aligned} x_- &:= \text{fl}^{\text{down}}(x) = 2^{q-\sigma}(1.b_1 b_2 \ldots b_S)_2 \\ x_+ &:= \text{fl}^{\text{up}}(x) = x_- + 2^{q-\sigma-S} \\ x_{\text{h}} &:= \frac{x_+ + x_-}{2} = x_- + 2^{q-\sigma-S-1} = 2^{q-\sigma}(1.b_1 b_2 \ldots b_S 1)_2 \end{aligned}$$

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: $\text{fl}(x) = x_-$. Since $2^{q-\sigma} \leq x_- \leq x \leq x_{\text{h}}$ we have

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

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

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

■

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) $|\delta_1| \leq \frac{\epsilon_{\text{m}}}{2}$.

II.3.2 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 13 (idealised floating point)**.** An idealised mathematical model of floating point numbers for which the only subnormal number is zero can be defined as:

$$F_{\infty,S} := \{\pm 2^q \times (1.b_1b_2b_3\dots b_S)_2 : q \in \mathbb{Z}\} \cup \{0\}$$

Note that $F^{\text{normal}}_{\sigma,Q,S} \subset F_{\infty,S}$ for all $\sigma, Q \in \mathbb{N}$. The definition of rounding $\text{fl}^{mode}_{\infty,S} : \mathbb{R} \to F_{\infty,S}$ 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 17 (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 $\text{fl}(1.1)$, and we will always assume nearest rounding unless otherwise stated. Thus this computation becomes

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

We will show the *absolute error* is given by

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

where $|\delta| \leq 28\epsilon_{\text{m}}$. First we find

$$\begin{aligned}\text{fl}(1.1) \oplus \text{fl}(1.2) &= (1.1(1+\delta_1) + 1.2(1+\delta_2))(1+\delta_3)\\ &= 2.3 + \underbrace{1.1\delta_1 + 1.2\delta_2 + 2.3\delta_3 + 1.1\delta_1\delta_3 + 1.2\delta_2\delta_3}_{\delta_4}.\end{aligned}$$

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

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

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

$$|\delta_4| \leq (2+2+3+1+1)\frac{\epsilon_{\text{m}}}{2} \leq 5\epsilon_{\text{m}}.$$

Writing $\text{fl}(1.3) = 1.3(1+\delta_5)$ and also incorporating an error from the rounding in $\otimes$ we arrive at

$$\begin{aligned}(\text{fl}(1.1) \oplus \text{fl}(1.2)) \otimes \text{fl}(1.3) &= (2.3+\delta_4)1.3(1+\delta_5)(1+\delta_6)\\ &= 2.99 + \underbrace{1.3(\delta_4 + 2.3\delta_5 + 2.3\delta_6 + \delta_4\delta_5 + \delta_4\delta_6 + 2.3\delta_5\delta_6 + \delta_4\delta_5\delta_6)}_{\delta}\end{aligned}$$

We use the bounds

$$\begin{aligned}|\delta_4\delta_5|, |\delta_4\delta_6| &\leq 5\epsilon_{\text{m}}^2/2 \leq 5\epsilon_{\text{m}}/2,\\ |\delta_5\delta_6| &\leq \epsilon_{\text{m}}^2/4 \leq \epsilon_{\text{m}}/4\\ |\delta_4\delta_5\delta_6| &\leq 5\epsilon_{\text{m}}^3/4 \leq \epsilon_{\text{m}}.\end{aligned}$$

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

$$|\delta| \leq 2(5 + 3/2 + 3/2 + 5/4 + 5/2 + 3/4 + 5/4)\epsilon_{\text{m}} \leq 28\epsilon_{\text{m}}$$

II.3.3 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 3 (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

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

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

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

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 (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 $h = 2^{-n}$ and $x = \pm 2^j (1.b_1 \dots b_S)_2$ and $b_S = 0$.

Proof

When $h = 2^{-n}$ we have for any normal float $y = \pm 2^j (1.b_1 \dots b_S)_2 \in F_{\infty,S}$ that

$$y/h = \pm 2^{j-n}(1.b_1 \dots b_S)_2 \in F_{\infty,S} \Rightarrow y/h = y \oslash h.$$

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.

■

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.4 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 14 (floating point interval arithmetic)**.** For intervals $A = [a, b]$ and $B = [c, d]$ satisfying $0 < a \leq b$ and $0 < c \leq d$, 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 18 (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:

$$\begin{aligned} 1 + 1 + 1/2 + 1/6 &\in [5/2, 5/2] \oplus ([1, 1] \oslash 6) \\ &= [2(1.01)_2, 2(1.01)_2] \oplus 2^{-3}[(1.0101010101)_2, (1.0101010110)_2] \\ &= [\mathrm{fl}^{\mathrm{down}}(2(1.01010101010101)_2), \mathrm{fl}^{\mathrm{up}}(2(1.01010101011)_2)] \\ &= [2(1.0101010101)_2, 2(1.0101010110)_2] \\ &= [2.666015625, 2.66796875] \end{aligned}$$

Example 19 (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:

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

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 billion. 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 Differential Equations: using divided differences we can reduce differential equations to linear systems. This motivates the investigation of numerical algorithms for solving linear systems.
3. III.3 Cholesky Factorisation: we look at computing a factorisation of a square matrix as a product of a lower and upper triangular matrix in the special case where the matrix is symmetric positive definite. Hidden in this is an algorithm to prove positive definiteness.
4. III.4 Polynomial Regression: often in data science one needs to approximate data by a polynomial. We discuss how to reduce this problem to solving a rectangular least squares problem.
5. III.5 Orthogonal Matrices: we discuss different types of orthogonal matrices, which will be used to simplify rectangular least squares problems.
6. III.6 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.

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-elimination 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): each of the m entry 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 multiple 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 widely 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:

$$U = \begin{bmatrix} u_{11} & \cdots & u_{1n} \\ & \ddots & \vdots \\ & & u_{mn} \end{bmatrix}, \qquad L = \begin{bmatrix} \ell_{11} & & \\ \vdots & \ddots & \\ \ell_{m1} & \cdots & \ell_{mn} \end{bmatrix}.$$

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:

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

When implemented in floating point this uses roughly half the number of multiplications: $n + (n-1) + \ldots + 1 = n(n+1)/2$ 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 15 (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 multiple square banded matrices in linear complexity: $O(n)$ operations. We consider two cases in particular (in addition to diagonal): bidiagonal and tridiagonal.

Definition 16 (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 17 (Tridiagonal)**.** If a square matrix has bandwidths $l = u = 1$ it is *tridiagonal*.

For example,

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & & & \\ a_{2,1} & a_{2,2} & a_{2,3} & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ & & & a_{n,n-1} & a_{n,n} \end{bmatrix}$$

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 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 differences 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:

$$u(a) = c, u'(x) = f(x)$$

We will then allow for more complicated first order linear ODEs with variable coefficients:

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

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*:

$$\begin{aligned} u(a) = c_0, u(b) = c_1, \\ u''(x) = f(x) \end{aligned}$$

In higher dimensions, the Poisson equation (and other *elliptic* partial differential equations) typically have boundary conditions and 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 $[a, b]$ by a grid of points $x_0, \ldots, x_n$ and write the ODE on each grid point.
2. Replace derivatives of the solution u with its values on a grid $u(x_j)$ by using a 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 $u_j \approx u(x_j)$.

Remark One can prove convergence of finite difference approximations but this is beyond the scope of this module.

III.2.1 Indefinite integration

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

$$u(a) = c$$
$$u'(x) = f(x)$$

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 to obtain approximations to $u(x_j)$ for each j, however, we shall take another (equivalent) 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 $x_0, \ldots, x_{n-1}$ 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 go past the interval. We use this approximation as follows:

(1) Write the ODE and initial conditions on the grid. Since right-sided 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:

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

(3) We do not know $u(x_j)$ hence we replace it with other unknowns u_j, but where the approximation becomes equality:

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

(4) This is actually 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 solve $L\boldsymbol{u} = \boldsymbol{b}$ using forward-substitution as discussed in the previous section.

III.2.2 Forward Euler

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

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

The steps proceed very similar to before:

(1) Write the ODE and initial conditions on the grid:

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

(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:

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

(4) We now get 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.

III.2.3 Poisson equation

Consider the Poisson equation with Dirichlet conditions (a two-point boundary value problem):

$$\begin{aligned} u(0) &= c \\ u''(x) &= f(x) \\ u(1) &= d \end{aligned}$$

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 $x_1, \ldots, x_{n-1}$ 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 is not permitted since that would go past the interval. We use this approximation as follows:

(1) Write the ODE and boundary conditions on the grid:

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

(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:

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

(4) We now get a tridiagonal linear system:

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

But how do we solve a tridiagonal linear system $A\boldsymbol{u} = \boldsymbol{b}$?

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 18 (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 19 (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 20 (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 20 (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 4 (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 21 (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 22 (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 23 (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 21 (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.