MATH50003 Numerical Analysis

Sheehan Olver

January 11, 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

$$\left|\int_a^b (x-a)f'(x)\mathrm{d}x\right| \le (b-a)\sup_{a\le x\le b}|(x-a)f'(x)| \le (b-a)\sup_{a\le x\le b}|(x-a)f'(x)|$$
$$\le (b-a)\sup_{a\le x\le b}|x-a|\sup_{a\le x\le b}|f'(x)|$$
$$\le M(b-a)^2.$$

■

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 $|\delta| \le M(b-a)h$ for $M = \sup_{a\le x\le b}|f'(x)|$, $h = (b-a)/n$ 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

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

Thus using the triangular inequality we have

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

■

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 the 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 (signed integers)**.** Denote the

$$\mathbb{Z}_m := \{0, 1, \ldots, m-1\}$$

The limitations this imposes on representing integers is substantial. If we have an implementation of $+$, which we shall denote $\oplus_m$, how can we possibly represent $m+1$ in this implementation when the result is above the largest possible integer?

The solution that is used is straightforward: *modular arithmetic*. E.g., we have

$$(m-1) \oplus_m 1 = m \pmod m = 0.$$

In this section we discuss the implications of this approach and how it works with negative numbers.

We will 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 $258 = 2^8 + 2 = (1000000010)_2$.

II.1.1 Unsigned Integers

Computers represent integers by a finite number of p bits, with 2^p possible combinations of 0s and 1s. For *unsigned integers* (non-negative integers) these bits dictate the first p binary digits: $(B_{p-1} \dots B_1 B_0)_2$.

Integers on a computer follow modular arithmetic: Integers represented with p-bits on a computer actually represent 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 $\oplus, \ominus, \otimes$.

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,

$$\begin{aligned} 17 \oplus_{256} 3 &= 20 \pmod{256} = 20 \\ 17 \ominus_{256} 3 &= 14 \pmod{256} = 14 \end{aligned}$$

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 $\mathbb{Z}_{256}$:

$$255 \oplus_{256} 1 = 255 + 1 \pmod{256} = (00000000)_2 \pmod{256} = 0 \pmod{256}$$

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,

$$254 \otimes_{256} 2 = 254 * 2 \pmod{256} = 252 \pmod{256} = (11111100)_2 \pmod{256}$$

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: the number $2^p - y$ is interpreted as $-y$. Thus for $p = 8$ we are interpreting 2^7 through $2^8 - 1$ as negative numbers. More precisely:

Definition ($\mathbb{Z}^{\mathrm{s}}_{2^p}$, signed integers)

$$\mathbb{Z}^{\mathrm{s}}_{2^p} := \{-2^{p-1}, \dots, -1, 0, 1, \dots, 2^{p-1} - 1\}$$

■

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

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

Note that if $R_p(x) = x \pmod[s]{2^p}$ then it can be viewed as a map $R_p : \mathbb{Z} \to \mathbb{Z}^s_{2^p}$ or a one-to-one map $R_p : \mathbb{Z}_{2^p} \to \mathbb{Z}^s_{2^p}$ whose inverse is $R_p^{-1}(x) = x(\mod 2^p)$. It can also be viewed as the identity map on signed integers $R_p : \mathbb{Z}^s_{2^p} \to \mathbb{Z}^s_{2^p}$, that is, $R_p(x) = x$ if $x \in \mathbb{Z}^s_{2^p}$.

Arithmetic works precisely the same for signed and unsigned integers up to the mapping R_p, e.g. we have

$$x \oplus^s_{2^p} y := x + y \pmod[s]{2^p}$$

Example 7 (addition of 8-bit signed integers)**.** Consider `(-1) + 1` in 8-bit arithmetic. The number -1 has the same bits as $2^8 - 1 = 255$. Thus this is equivalent to the previous question and we get the correct result of `0`. In other words:

$$-1 \oplus^s_{256} 1 = -1 + 1 \pmod[s]{256} = 0$$

But on the bit level this computation is exactly the same as unsigned integers. We represent the number -1 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

$$\begin{aligned} &(011111111)_2 + \\ &(000000001)_2 \ = \\ &(100000000)_2 \end{aligned}$$

The 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:

$$127 \oplus^s_{256} 1 = 128 \pmod[s]{256} = 128 - 256 = -128.$$

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:

$$(10000000)_2 \pmod[s]{256} = 128 \pmod[s]{256} = -128.$$

Example (multiplication of 8-bit signed integers) Consider `(-2) * 2`. -2 has the same bits as $2^{256} - 2 = 254$ and -4 has the same bits as $2^{256} - 4 = 252$, and hence from the previous example we get the correct result of `-4`. In other words:

$$(-2) \otimes^s_{2^p} 2 = -4 \pmod[s]{2^p} = -4$$

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

$$(011111110)_2 * 2 = (111111100)_2$$

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

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

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 6 (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 9 (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 7 (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

$$sq_{Q-1} \dots q_0 b_1 \dots b_S$$

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, discussed later.

II.2.3 IEEE floating-point numbers

Definition 8 (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 10 (a real number in 16-bits)**.** 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 $2^4 - Q = 16 - 15 = 1$ Hence this number is:

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

Example 11 (rational in 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
```

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 special numbers).

Example 12 (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 13 (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.