Chapter II

Representing Numbers

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

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

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

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

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

In particular we discuss:

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

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

II.1 Reals

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

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

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

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

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

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

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

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

II.1.1 Real numbers in binary

Integers can be written in binary as follows:

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

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

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

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

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

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

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

We will see that in binary

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

Both results can be proven using the geometric series:

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

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

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

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

II.1.2 Floating-point numbers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

II.1.3 IEEE floating-point numbers

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

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

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

$$0 \; 10000 \; 1010000000$$

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

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

hence the exponent is

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

and the number is:

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

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

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

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

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

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

$$0 \; 01101 \; 0101010101$$

II.1.4 Sub-normal and special numbers

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

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

1 00000 1100000000

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

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

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

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

1 11111 0000000000

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

1 11111 0000000001

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

II.2 Floating Point Arithmetic

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

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

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

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

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

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

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

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

where

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

Write the half-way point as:

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

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

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

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

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

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

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

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

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

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,

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

which differs by 1 bit.

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

II.2.1 Bounding errors in floating point arithmetic

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

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

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

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

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

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

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

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

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

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

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

We can use machine epsilon to determine bounds on rounding:

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

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

where the relative error *is bounded by:*

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

Proof

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

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

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

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

Define

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

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

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

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

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

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

■

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

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

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

II.2.2 Idealised floating point

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

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

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

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

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

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

We will show the *absolute error* is given by

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

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

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

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

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

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

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

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

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

We use the bounds

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

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

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

II.2.3 Divided differences floating point error bound

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

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

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

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

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

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

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

where

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

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

Proof

We have

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

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

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

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

■

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

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

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

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

Proof

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

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

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

■

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

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

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

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

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

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

II.3 Interval Arithmetic

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

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

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

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

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

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

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

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

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

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

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

The remaining two proofs are left for the problem sheet.

■

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

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

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

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

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

Noting that

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

we can extend the computation to another term:

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

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

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

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

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

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:

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

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

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

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