Floating Point Lecture Notes

Willie A.

Contents

Remark. Starred sections are optional and are provided for interested readers.

1 Introductory Remarks

1.1 A closer look at number systems

We begin by introducing some terminology used to describe number systems. Our current number system is both positional and base 10. The system is *positional* because the position of each digit determines its value. For instance, the number 333 is written with three identical symbols, or *digits*, but each digit represents a different value. The system is *base* 10 because it uses the ten digits $0, 1, 2, \ldots, 9$ to represent numbers.

Combined, the two properties give meaning to a string of digits via the *expanded form*. For instance, in the current positional base 10 number system the symbol 1457 is defined by

$$1457 \triangleq 1000 + 400 + 50 + 7 = 1 * 10^3 + 4 * 10^2 + 5 * 10^1 + 7 * 10^0.$$

Positional number systems with different bases give meaning to strings of digits in a similar fashion. In the following example, we use a subscript to denote the base in which the string of digits is defined. With a few straightforward calculations, we can express the decimal number 23_{10} in the bases 5, 3, and 2, respectively:

$$23_{10} = \begin{cases} 2 * 10^1 + 3 * 10^0 & \\ 4 * 5^1 + 3 * 5^0 & = 43_5 \\ 2 * 3^2 + 1 * 3^1 + 2 * 3^0 & = 212_3 \\ 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 & = 10111_2. \end{cases}$$

We note that the positional base 2, 3, and 10 number systems are respectively called *binary*, *ternary*, and *decimal*.

1.2 Brief historical remarks on number systems*

We provide a few remarks regarding the fascinating historical development of number systems. Dating back more than four millennia, the Babylonian number system is one of the oldest known in writing. In fact, it is the oldest known positional system. A historical exception in many respects, it was *sexagesimal* (base 60). Interestingly, the influence of the sexagesimal system persists. It can be glimpsed in the 360° of a circle and the 60 seconds of a minute.

Though 60 seems like a natural choice of base because it is a highly composite integer, it was by no means standard among civilizations. The Mayas, for example, developed a positional *vigesimal* (base 20) system early in the first millennium CE. The development was very significant because it allowed for very accurate calendar computations in the fifth century CE.

The choice of a positional system was not standard either. Historically important civilizations, like the Romans, did not develop a positional system. Roman numerals have fixed values which do not depend on their placement in the string (although there is a formal grammar which dictates proper arrangement of numerals in a string). In fact, the prowess of the Roman empire may explain the delay in the spread of the much more practical Hindu-Arabic numeral system, on which our current positional decimal system is based.

Despite the disparity in the early development of number systems, most civilizations eventually converged upon the positional decimal system. Some historians have suggested that the choice of base was anatomically motivated: it represents the number of things we can count with our fingers.

2 Floating Point System

2.1 Defining an FPS

The set of integers $\mathbb{Z}$ is infinite and the set of reals $\mathbb{R}$ is much larger. In addition, we need an infinite string of digits to represent a single real number! This fact is woven into the very fabric of $\mathbb{R}$ and is stated precisely in Theorem 2.1.

Theorem 2.1 (Decimal Expansion)**.** *For every nonnegative real number x, there exists $k \in \mathbb{N}$ and a sequence $d_n \in \{0, 1, \ldots, 9\}$ such that*

$$x = k + \sum_{n=1}^{\infty} d_n (10)^{-n}.$$

In addition, either the sequence d_n is unique, or there exists a sequence which ends in an infinite string of 0's and a sequence which ends in an infinite string of 9's.

For an accessible proof of this result, please refer to Section 2.16 of Kenneth Ross's Elementary Analysis: The Theory of Calculus (electronic resource available through the Stanford library website). We note that the choice of base 10 in Theorem 2.1 is arbitrary: analogous results holds for every integer base $\beta > 1$. (In base β, each digit d_n belongs to the set $\{0, 1, \ldots, \beta - 1\}$).

The infinite size and resolution of $\mathbb{R}$ pose great challenges for machine representation, since computers have finite memory and processing power. To circumvent the issue mathematicians, engineers, and computer scientists developed the *floating point system* (FPS), a finite approximation of $\mathbb{R}$ based on finite decimal expansions.

With Theorem 2.1 in mind, we fix a base β and integers p, m, and M, and define the FPS as the set of all combinations of the form

$$\pm \left(\sum_{n=1}^{p} d_n \beta^{-n} \right) \times \beta^e,$$

with $d_n \in \{0, 1, \ldots, \beta - 1\}$, and $m \leq e \leq M$. We note that the combinations afforded by the above sum can be alternatively expressed as

$$\pm (0.d_1 d_2 \ldots d_p)_\beta \times \beta^e. \tag{1}$$

We stress that the FPS is parametrized by (β, p, m, M). We comment on commonly used choices in Section 2.4. For the purpose of illustration, we take $\beta = 10$.

We now motivate the term *floating point*. In any FPS, the point floats, literally. This idea is perhaps best explained by example. Consider the values of the following elements of an FPS with $p = 4$:

$$\begin{aligned} 0.1234 \times 10^{-2} &= .001234 \\ 0.1234 \times 10^{1} &= 1.234 \\ 0.1234 \times 10^{4} &= 1234. \\ 0.1234 \times 10^{8} &= 12,340,000. \end{aligned}$$

Changing e moves the decimal point: it floats to where we need it! Given this term, we refer to elements of an FPS as *floats*.

2.2 Finite approximation!

In this section, we stress two fundamental differences between $\mathbb{R}$ and the FPS. These differences explain the error produced when representing reals by floats. In Section 3.1 we explore a third fundamental difference, which elucidates a subtlety in quantifying the approximation error.

First, every FPS is discrete. Every FPS has finite resolution. In particular, every float in an FPS is given by a string of p digits. The notion of "consecutive" floats is well-defined. In contrast, the reals are continuous: between any two real numbers there exist infinitely many other reals.

The specific value of the *precision* p of an FPS may vary from implentation to implementation and depends on machine hardware. We comment on the historical development and current standard choices of p in Section 2.4.

In any case, p is always finite and this in large part explains the error in approximating reals by floats. Consider for instance the rational number $1/3$ (not to mention an irrational like $\sqrt{7}$ or trannscendental like π). By Theorem 2.1, its unique decimal expansion is

$$\frac{1}{3} = \sum_{n=1}^{\infty} 3(10)^{-n},$$

or equivalently, $1/3 = 0.\bar{3}$, where the overset bar indicates that every digit in the expansion is 3. For **any** choice of p, we have

$$\sum_{n=1}^{\infty} 3(10)^{-n} - \sum_{n=1}^{p} 3(10)^{-n} = \epsilon > 0.$$

No matter the choice of p, there is there is a positive approximation error. (It is left to the reader to verify that the string of all 3's gives the best approximation, in the sense of smallest absolute difference, of $1/3$ in the base 10 FPS with precision p.) No FPS can fit all reals exactly.

Second, every FPS is finite. Every FPS has a smallest and a largest element. In particular,

$$0.1_\beta \times \beta^{-m} \leq \left| \left(\sum_{n=1}^{p-1} d_n \beta^{-n} \right) \times \beta^e \right| \leq 0.(\beta-1)(\beta-1)\cdots(\beta-1)_\beta \times \beta^M,$$

since by definition every float $\pm(0.d_1 d_2 \ldots d_p)_\beta \times \beta^e$ satisfies $m \leq e \leq M$ and $1 \leq d_n \leq \beta - 1$. In contrast, the reals are unbounded: for every real number, there exists a smaller and larger real.

The specific values of m and M may vary from implementation to implementation and depend on machine hardware. We comment on current standard choices in Section 2.4.

In any case, the bounds m and M on the *exponenent* e are finite and this completes the explanation of the error in approximating reals by floats. The finitude of m gives rise to the region of *underflow*, the space between zero and the next larger float. The finitude of M gives rise to the region of overflow, the portion of the real line to the right of the largest float.

For example, consider an FPS $\mathcal{F}$ with $\beta = 10$, $m = -4$, and $M = 8$. Planck's constant, $h \approx 6.63 \times 10^{-34} J \cdot s$, and the wavelength of blue light, $\approx 7 \times 10^{-7}$m, lie in $\mathcal{F}$'s region of underflow and they must be approximated by either 0 or 10^{-5}. Similarly, $\mathcal{F}$ cannot represent the wealth of Bill Gates, $\approx \$8.91 \times 10^{10}$, let alone Avogadro's number, $\approx 6.02 \times 10^{23}\text{mol}^{-1}$. Such large values, in $\mathcal{F}$'s region of overflow, would be approximated by 10^8.

2.3 Warning: approximation error has spelled disaster*

In practice, under- and overflow can lead to significant error. In the past, overflow has caused some veritable disasters. In June 1996, it destroyed the decade-long $7 billion European Space Agency project Ariane 5. The board investigating the unmanned spacecraft's sudden crash reported that

the loss of the rocket's guidance and altitude information resulted from a failed attempt to store a value larger than was admissible by the onboard FPS. Interested readers may find a copy of the original report at Ariane 5 Accident report.

A few years later, overflow made the world fear the dawn of January $1, 2000$. Most computers maintaining yearly records used an FPS with two digits of precision (1985, for example, was recorded as 85). The year 2000 lay beyond the capacity of such systems, since 00 represented 1900. International alarm became widespread because such systems–used in banks and governments–were expected to crash and feverish upgrade efforts ensued. Britannica Encyclopaedia estimates over $300 billion were spent eradicating what has become known as the "Y2K bug." Interested readers may find more at: Y2K bug and Interview with a Y2K software engineer.

Interested readers may find short briefings on other disasters caused by the likes of overflow at: Some disasters attributable to bad numerical computing.

2.4 Standards*

We begin this section by motivating the base 2 in the vast majority practical FPS implementations.

At the end of the day, every process in a CPU is driven by the flow of electrons. *Transistors*, tiny switches in the heart of the processor, control the passage of current in the processor circuit. Transistors can take only two states: open and closed; 1 and 0. Since all computation boils down to the flow of current along a chain of transistors, and these can only take on two states, binary seems like the natural choice.

(Due to physical limitations and quantum interference effects, current technologies only allow for reliable configuration of two-state transistors. Keep in mind that state-of-the-art CPUs are manufactured by 14nm processes and that even smartphone CPUs contain billions of transistors!)

With this in mind, it is useful to introduce the notions of bit and byte. A *bit* is a single binary digit: a single 0 or 1 in a binary string. A *byte* is an 8-bit string. You may be familiar with the notion of mega- (MB) or gigabyte (GB). Technically, $1\text{MB} = 10^6\text{B}$, and B stands for byte.

Having considered the standard choice of base, we turn to standard configurations of p, m, M in practical implementations of FPS. These standards are somewhat arbitrary in the sense that they are much more the consequence of historical development than of theoretical optimality or physical limitation. Indeed, there was much inconsistency and debate during the 1970s and early 1980s regarding the "right" configuration of the nascent FPS. Different manufacturers used different precisions and allocated different amounts of memory to the storage of floating point numbers, which led to cross-platform incompatibility and differing results to the same computation (due to the difference in approximation errors afforded by the varying FPS). Efforts to standardize FPS implementations culminated in the seminal IEEE-754 paper in 1985. The standard set was accepted internationally in 1989.

We now take a closer look at the standards defined by the IEEE-754. Recall expression 1, which states that every element of an FPS has three parts: a *sign* s, a *significand* $d_1 d_2 \cdots d_p$, and an exponent e. If we are to represent each float in a fixed number N of bits, we must decide how to allocate the N bits among its three components.

Either $s = 1$ or $s = -1$, so s can be encoded in a single bit: we place the states ± 1 in bijective correspondence with 0 and 1. Clearly p bits are needed to encode the significand. Once we fix p, we may use the remaining $N - p - 1$ bits to encode the exponent. Note that designing the FPS in this way implicitly defines the bounds m and M on the exponent as the smallest and largest integers representable in $N - p - 1$ bits.

The IEEE-754 defined the concepts of single and double precision by presenting two model FPS so desgined. The *single precision* FPS uses 32 bits to represent its elements and the *double precision* FPS uses $2 \times 32 = 64$ bits. In both cases, the sign is encoded in the first bit, the significand in the following p bits, and the exponent in the remaining bits. The single precision FPS has 23 binary digits of precision, and it uses 8 bits for the exponent. Thus, for instance, $M = 127$ in this case. The double precision system has 52 digits of precision, and it uses 11 digits for the exponent.

We conclude this section by offering a few concluding remarks. First, double precision is standard in modern computer architectures. Second, some applications demand even more than double precision. The 1985 standard has been updated to include *quadruple precision* (128 bit) FPS and even higher bit systems, which we do not discuss here. Finally, we mention that apart from *floats*, or floating point types, the IEEE-754 also defined other specialized data types. For instance, it defined the `int8`, `int32`, and `int64` types. As their names suggest, these data types can be used exclusively for the storage of integers, and they respectively provide 8, 32, and 64 bits for integer representation. Thus, for example, the `int8` type is most appropriate when only small integers are required (in the range -128 to 127).

Interested readers may verify that the FPS used by MATLAB on their machines is on par with IEEE standards by running the following code:

```
%In single precision, we have
log2(eps('single'))
%BINARY digits of precision.

%In double precision, we have
log2(eps('double'))
%BINARY digits of precision.
```

3 Properties and Subtleties of FPS

3.1 Storing a real: roundoff error

In this section we explain exactly how floats approximate real numbers and we provide bounds on the approximation error.

Roundoff error is produced when approximating a real by a float. In essence, we must choose a string of p digits to represent a number's entire decimal expansion (see Theorem 2.1). Practical implementations use either chopping or rounding to accomplish this task.

Chopping (or truncating) consists of storing only the first p digits in a number's decimal expansion. *Rounding*, as its name suggests, consists of rounding the decimal expansion to p significant digits. We denote chopped *floating point equivalent* of x by $\mathrm{fl}_{\mathrm{chop}}(x)$ and the rounded equivalent by $\mathrm{fl}_{\mathrm{round}}(x)$. For example, if $p = 7$,

$$\begin{aligned}\mathrm{fl}_{\mathrm{chop}}(\sqrt{2}) &= 0.1414213 \times 10^1\\ \mathrm{fl}_{\mathrm{round}}(\sqrt{2}) &= 0.1414214 \times 10^1,\end{aligned}$$

since $\sqrt{2} \approx 1.41421356\ldots$.

We now turn to quantifying roundoff error. For simplicity, we consider an FPS with chopping. A common measure of roundoff is absolute error, defined by

$$\epsilon_{\text{abs}} = |\text{fl}_{\text{chop}}(x) - x|.$$

In the example above we have $\epsilon_{\text{abs}} = |1.414213 - \sqrt{2}| \approx 5.62373095 \times 10^{-7}$.

Note that in general the absolute roundoff error is bounded above:

$$\begin{aligned} \epsilon_{\text{abs}} =& |\text{fl}_{\text{chop}}(x) - x| \\ =& (0.d_{p+1}d_{p+2}d_{p+3}\ldots)_\beta \times \beta^{e-p} \\ \leq& (1.0)_\beta \times \beta^{e-p} \\ =& \beta^{e-p}. \end{aligned} \tag{2}$$

Bound 2 reveals an interesting property of the FPS. Dependence on the exponent e implies that approximations of large x are more inaccurate than those of x close to zero: absolute roundoff error grows with x. This is symptomatic of a flaw of the FPS.

Floats are distributed nonuniformly: elements near to zero are more closely spaced than those at the extremes of the repersentable values. For instance, consider a base 10 FPS with 4 digits of precision and let $x = 0.1234 \times 10^e$. When $e = 1$, the distance between x and the next larger float is given by

$$0.1235 * 10^1 - 0.1234 * 10^1 = .0001 * 10^1 = .001.$$

When $e = 4$, the separation is much larger:

$$0.1235 * 10^4 - 0.1234 * 10^4 = .0001 * 10^4 = 1.$$

Interested readers should run the following commands in MATLAB to persuade themselves that the separation between consecutive floats varies even in state-of-the-art practical implementations:

```
%Illustration of varying absolute distance between consecutive floats
%(the following is done in double precision)
eps(1.0)
eps(2.0)
eps(10^2)
eps(10^6)
```

Since absolute error varies with the size of the input, we typically measure the *relative error*, which scales the absolute error to the size of the input and is defined by

$$\epsilon_{\text{rel}} = \frac{|\text{fl}_{\text{chop}}(x) - x|}{|x|}.$$

We use bound 2 and the following inequality to derive a tight uniform bound on the relative error. Note that for any x, we have

$$\begin{aligned} |x| =& (0.d_1d_2d_3\ldots)_\beta \times \beta^e \\ \geq& (.1)_\beta \times \beta^e \\ =& \beta^{e-1}. \end{aligned}$$

(The inequality holds by a small detail we glossed over: in order to obtain unique representations, we require $d_1 \neq 0$ whenever $x \neq 0$.)

Combining we results, we conclude that

$$\epsilon_{\text{rel}} = \frac{|\text{fl}_{\text{chop}}(x) - x|}{|x|} \leq \frac{\beta^{e-p}}{\beta^{e-1}} = \beta^{1-p}.$$

We mention that we can derive a similar bound for the relative error produced by the rounding mechanism using analogous reasoning. However, rounding produces a smaller relative error:

$$\epsilon_{\text{rel}} = \frac{|\text{fl}_{\text{round}}(x) - x|}{|x|} \leq \frac{1}{2}\beta^{1-p}.$$

As stated, this bound assumes β is even. We may achieve the same relative error bound for general β by rounding to the nearest float (as measured by the absolute value), instead of rounding to the nearest pth digit. The two schemes differ only when β is odd, and the distinction is quite subtle. For instance, when $\beta = 5$ and $p = 3$, rounding 0.143244_5 according to the $(p+1)$th digit $d_4 = 2$ yields the floating point equivalent 0.143_5, whereas rounding to the nearest float gives 0.144_5, since perhaps unexpectedly, 0.143244_5 is closer to 0.144_5 than to 0.143_5:

$$|0.143244_5 - 0.143_5| = 0.000244_5 > 0.000201_5 = |0.143244_5 - 0.144_5|.$$

We conclude this section by noting that the roundoff error directly depends on the hardware implementation (since the uniform bound depends on p, the number of digits of precision of the FPS implementation). In fact we refer to the bound as *machine error* or *machine epsilon* and we commonly denote it by u.

Interested readers may compute their machine epsilon by running the following commands in MATLAB:

```
%Extract machine epsilon (single and double precision)
eps('single')
eps('double')
```

Every interested reader should obtain the same values, assuming of course that their machine complies with the standards described in Section 2.4.

4 Arithmetic in FPS

4.1 Theoretical computation

In this section we discuss the main idea behind floating point or *finite precision* arithmetic. Define $\text{fl}(x)$ to be the floating point equivalent of the real number x. (We suppress the subscript on fl when the stated result is independent of fl's implementation–see Section 3.1.)

Let @ denote a binary operation on $\mathbb{R}$ (in this context, *binary* means that @ takes in two inputs from $\mathbb{R}$ and returns a single output in $\mathbb{R}$). For $x, y \in \mathbb{R}$ the floating point equivalent output is defined by

$$\text{fl}(x@y) \triangleq \text{fl}(\text{fl}(x)@\text{fl}(y)). \tag{3}$$

In words, Equation 3 suggests the following algorithmic procedure for computing the floating point equivalent output of a binary operation @:

1. `Find floating point equivalents` $\tilde{x} = \texttt{fl}(x)$ `and` $\tilde{y} = \texttt{fl}(y)$.
2. `Compute` $z = \tilde{x} @ \tilde{y}$ `in exact arithmetic.`
3. `Find floating point equivalent` $\tilde{z} = \texttt{fl}(z)$ `and output.`

For example, consider finite precision addition. In this case we replace @ in the above discussion by +. For simplicity, consider a base 10 FPS with four digits of precision and suppose $x = 0.01234431$ and $y = 98.76321$. We quickly find that $\tilde{x} = \text{fl}(x) = 0.1234 \times 10^{-1}$ and $\tilde{y} = \text{fl}(y) = 0.9876 \times 10^{2}$. Next, we perform the addition $\tilde{x} + \tilde{y}$ in exact arithmetic. Aligning $\tilde{x}$ and $\tilde{y}$ vertically with respect to the decimal point and then adding column by column yields

$$\begin{array}{r} .01234 \\ +\ 98.76 \\ \hline 98.77234 \end{array}$$

Finally, we compute $\text{fl}(98.77234) = 0.9877 \times 10^{2}$ and output it as the result.

Finite precision multiplication is carried out similarly, and it is perhaps simpler. Given $x, y \in \mathbb{R}$, we first obtain $\tilde{x} = \text{fl}(x)$ and $\tilde{y} = \text{fl}(y)$. Then we align the digits vertically and multiply as usual. Next we add the exponents (using the procedure described above). Finally, we round the product and output. With x and y as in the addition example above, we have

$$\text{fl}\big((0.1234 \times 10^{-1}) * (0.9876 \times 10^{2})\big) = \text{fl}\big(12.186984 \times 10^{-1}\big) = \begin{cases} 0.1218 \times 10^{1}, & \text{with chopping} \\ 0.1219 \times 10^{1}, & \text{with rounding.} \end{cases}$$

We conclude this section by encouraging delicate care when interpreting results from finite precision computations. Many of the well-known and loved properties of exact arithmetic are lost in translating problems to the finite precision setting. For instance, finite precision addition is **not** associative. That is,

$$\text{fl}((x + y) + z) \neq \text{fl}(x + (y + z)).$$

To illustrate this point we consider for simplicity a base 10 FPS with four digits of precision which uses chopping. Let $x = -1.234$, $y = 1.234$, $z = .0009999$. Then $\text{fl}((x + y) + z) = \text{fl}(0 + z) = 0.9999 \times 10^{-3}$. However, if we add $y+z$ first, we see that $\text{fl}(y+z) = \text{fl}(0.1234 \times 10^{1} + 0.9999 \times 10^{-3}) = \text{fl}(1.2349) = 0.1234 \times 10^{1}$, since the FPS has 4 digits of precision. Therefore,

$$\text{fl}((x + y) + z) = 0.9999 \times 10^{-3} \neq 0 = \text{fl}(x + (y + z)).$$

Since the above example seems trivial, we include two computations which the interested reader should perform MATLAB :

```
%Loss of associativity example
(-1.0 + 1.0) + 2^(-53)
(-1.0) + (1.0 + 2^(-53))
```

Although the two lines express the same mathematical operation, only one of them returns 0!

Even the MATLAB computation will seem trivial to the suspicious reader. Indeed a single computation is unlikely to cause alarm. However, most numerical algorithms typically perform millions of computations and we can run into serious problems when the error from successive computations accumulates. Section 5.2 further explores this issue.

4.2 Two's complement*

In this section we present an interesting idea critical to the development of the FPS. The concept of two's complement dissolves the need to implement subtraction separately from addition. We illustrate this idea via a couple of examples using the type `int8`, for simplicity (recall the alternate data types mentioned at the end of Section 2.4). For completeness, we restate that the `int8` type can only represent integers in the range -128 to 127.

In Section 2.4 we mentioned that a number's sign is encoded in the leading or *most significant bit* (MSB) of its representation. In the context of `int8`, this means that positive integers (with MSB $= 0$) occupy the range 0 to 127 in the representation, while negative integers (with MSB $= 1$) fill out the range 128 to 255. *Two's complement* suggests that we arrange the negative integers in reverse order: -1 takes the representation of 255, -2 that of 254, and so on, down to -128. More precisely, using two's complement, the number represented by $d_1 d_2 \dots d_8$ has the value

$$\sum_{n=2}^{8} d_n 2^n - d_1 2^7.$$

For instance, $\texttt{int8}(-1) = 11111111$, $\texttt{int8}(-2) = 11111110$, and $\texttt{int8}(-128) = 1000000$.

With this clever arrangement, we can implement subtraction via the addition mechanism we described in Section 4.1. In essence, we compute a difference via an equivalent problem: $x - y \triangleq x + (-y)$. For instance, $1 - 1$ becomes

$$\begin{array}{r} \texttt{00000001} \\ +\texttt{11111111} \\ \hline \texttt{100000000} \end{array}.$$

We obtain the correct answer, 0, by ignoring the MSB. Similarly, we compute $4 - 3$ via the addition

$$\begin{array}{r} \texttt{00000100} \\ +\ \texttt{11111101} \\ \hline \texttt{100000001} \end{array}.$$

Again ignoring the MSB gives 1, the correct answer.

4.3 Practical implementation**

In this section we (very, very) briefly describe the physical implementation of floating point arithmetic. For simplicity, we focus on binary addition.

As mentioned in Section 2.4, it all boils down to the flow of electrons in the CPU. We use electrical current to represent bits. The life of a bit in a processor is therefore governed by transistor switches, which control the flow of current in the processor circuit.

The presence of current represents 1 and its absence 0. The key to this section lies in equating 1 with the state "on" and the boolean T, and equating 0 with the state "off" and the boolean F.

We can arrange transistors to construct *logic gates*, which physically implement logical relations like $\wedge$, $\oplus$, and $\neg$. Much like the relations $\wedge$, $\oplus$, and $\neg$, the corresponding logic gates receive two input signals and combine them into a single output following a specified rule. For instance, the AND gate implements the $\wedge$ relation and it outputs current only if its two input terminals receive current. The implementation mirrors the relation $A \wedge B = T$ only if $A = B = T$. Similarly, the

XOR (exclusive OR) gate implements the $\oplus$ relation and it outputs current when exactly one of its input terminals receives current. This reflects the fact that $A \oplus B = T$ only if either $A = T \wedge B = F$ or $A = F \wedge B = T$.

We can also arrange logic gates to construct more complex circuits. In particular, we can combine a XOR gate with an AND gate to construct a *half adder*, which physically implements single bit addition.

The half adder is built as follows. Let A and B denote the two input bits. Physically, each is represented by a wire. As described above, the wire representing A carries current only if $A = 1$, and similarly for the other wire. In the half adder we connect both A and B to the input terminals of the XOR and AND gates. We say that the output S of the XOR gate is the *sum* bit and the output C of the AND gate is the carry bit (note that the sum of two single bits may occupy up to two bits). Figure 1 shows the circuit schematic.

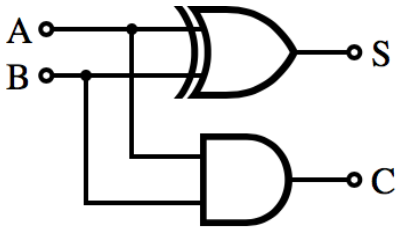

Figure 1: Half adder circuit schematic. The top symbol denotes the XOR gate. Source: Wikimedia.

Interpreting CS as a binary string yields the sum $A + B$. Let's consider all four possible cases. If $A = 0 \wedge B = 0$, then $A + B = 00$. Since the AND gate outputs current only if both inputs are "on," $C = 0$. Since the XOR gate outputs current only if exactly one input is "on," we have $S = 0$. Therefore $A + B = 00 = CS$. The cases $A = 1 \wedge B = 0$ and $A = 0 \wedge B = 1$ are clearly identical and we only consider the former. In this case $A + B = 01$. Since B is "off," $C = 0$ and since exactly one input is "on," $S = 1$. Therefore $A + B = 01 = CS$. Finally, if $A = 1 \wedge B = 1$ then $A + B = 10$. Since both inputs are "on," $C = 1$ and $S = 0$, giving $A + B = 10 = CS$.

Now recall the bit-by-bit addition procedure described in Section 4.1. The procedure suggests that we implement the addition of binary strings by repeatedly adding single bits. Our half adder is insufficient for this implementation, since it cannot incorporate bits carried over from previous calculations. The *full adder* extends the functionality of our circuit, remedying the issue. We chain together full adders to add long binary strings. The interested reader may find more at: Full Adder.

5 Why Should You Care?

5.1 Order of operations

As mentioned in Section 4.1, many of the well-known and loved properties of binary operations are lost when translating to the world of finite precision calculations. In the approximation setting, the specific order in which operations are performed becomes important–really important.

In fact this issue gave rise to the study of the stability of numerical methods–a procedure's response to small changes in its input. Consider the problem of solving a homogeneous system of linear equations with coefficient matrix A. Computers do not store the real entries a_{ij} of A, but

their floating point equivalents $\tilde{a}_{ij} = \text{fl}(a_{ij})$, as described in Section 3.1. Therefore, the input to any numerical method designed to solve $Ax = 0$ is actually $\tilde{A}$, and not A! We say that a numerical method is *stable* if it outputs a solution $\tilde{x}^*$ close to the true solution x^*. In many cases algorithm design is the key to stability. As illustrated by the examples in Section 5.2, simply reordering calculations can bring stability to a method.

We conclude this section by introducing the notion of conditioning. We say that a problem is *well-conditioned* if small changes in problem input lead to small changes in the problem output. If not, we say that the problem is *ill-conditioned*. Stable methods satisfactorily solve well-conditioned problems. Ill-conditioned problems are much harder to solve. Though by definition no strictly stable methods to solve them can be defined, such methods are the best available for approaching many ill-conditioned problems. For example, we consider the ill-conditioned determinant problem in Section 5.2.

5.2 Examples/Illustrations

The examples below illustrate the importance of performing computations in the "right" order.

We single out an important phenomenon which we encounter frequently. Consider two floats $\tilde{x}$ and $\tilde{y}$ which are equal to within their last digit. Their difference $\tilde{z} = \text{fl}(\tilde{x} - \tilde{y}) = \tilde{x} - \tilde{y}$ will have only one significant digit even though no roundoff error will be produced in the subtraction. Using $\tilde{z}$ in subsequent calculations will typically limit their result to one correct digit. There is an irreversible loss of information and we call this *catastrophic cancellation*. Catastrophic cancellation usually leads to numerical instability and we strive to avoid it by rearranging the order of operations.

Example 5.1 (Function evaluation, taken from Bradie)**.** In this example, we evaluate the function $f(x) = e^x - \cos x - x$ at values close to $x = 0$. Since e^x and $\cos x$ both tend to 1 as $x \to 0$ naive numerical methods might run into catastrophic cancellation.

Before we actually compute the values, we use tools from basic analysis to determine what we should expect. For starters, note that f is very smooth since it is infinitely differentiable. Next, note that $f(0) = 1 - 1 - 0 = 0$. Since $f'(x) = e^x + \sin x - 1 > 0$ for $x \in I = (0, \pi)$ because $e^x > 1$ and $\sin x > 0$ on I, we know that f is increasing on I. In addition, since $e^x < 1$ and $\sin x < 0$ for $x \in I' = (-\pi, 0)$ we have that $f'(x) < 0$ and therefore f is decreasing on I'. Since f is decreasing to the left of zero and increasing to its right, we conclude that $x = 0$ is the only root of f in the interval (π, pi). Finally, we observe that $f''(x) = e^x + \cos x > 0$ near zero, so that f is concave.

Figure 2 directly computes values of f on the interval $[-5, 5] \times 10^{-8}$, at 1001 uniformly spaced points in IEEE standard double precision. The overall trend in the graph is in agreement with our analysis but the fine detail (smoothness) is clearly not.

We propose an alternative calculation which will yield better results in finite precision. We first approximate f by a Maclaurin polynomial analytically, and we then evaluate the approximation polynomial numerically. Recalling the Maclaurin series for e^x and $\cos x$ yields

$$\begin{aligned} f(x) =& e^x - \cos x - x \\ =& \left(1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + O(x^5)\right) - \left(1 - \frac{x^2}{2} + \frac{x^4}{24} + O(x^6)\right) - x \\ =& x^2 + \frac{x^3}{6} + O(x^5) = x^2(1 + \frac{x}{6}) + O(x^5). \end{aligned}$$

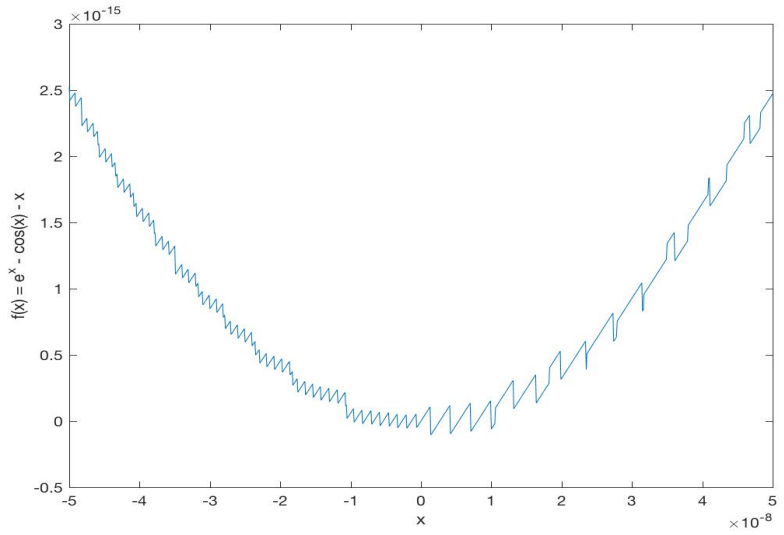

Figure 2: Graph of the function $f(x) = e^x - \cos x - x$ computed in IEEE standard double precision.

The interested reader may show that the relative error in using the last polynomial to approximate f on $[-5, 5] \times 10^{-8}$ is less than 10^{-24}. Therefore, the polynomial approximates f to machine precision in IEEE standard double precision (see Section 2.4). The plot in Figure 3 confirms this assessment.

Example 5.2 (Order of operations in function evaluations)**.** In this case we evaluate the polynomial $p_c(x) = (x-10)^9$ near $x = 10$. Recall the Binomial Theorem and note that it guarantees the equality

$$\begin{aligned} p_e(x) &= x^9 - 90x^8 + 3600x^7 - 84000x^6 + 1,260,000x^5 - 12,600,000x^4 \\ &\quad + 84,000,000x^3 - 360,000,000x^2 + 900,000,000x - 1,000,000,000 \\ &= (x-10)^9. \end{aligned}$$

Figure 4 shows a plot of the relative error $|p_e(x) - p_c(x)|/|p_e(x)|$ over the interval $[9, 11]$ computed in standard double precision by direct evaluation at 1001 evenly spaced points. We note that in exact arithmetic, the relative error should be zero on the entire interval (except at $x = 10$ where it is undefined). Note the logarithmic scale on the vertical axis, which indicates that the computation of p_e is accurate to zero digits when x is very close to 10. We note that when x is close to 10 direct evaluation of the expanded polynomial p_e exhibits particularly detrimental catastrophic cancellation, since it involves subtractions of large numbers (on the order of 10^{11}) which are very close to each other in magnitude. The evaluation of the compact form p_c does not involve such operations, so it is much more stable.

Example 5.3 (Cancellation magic*)**.** This example once again illustrates the importance of perfoming operations in the "correct" order.

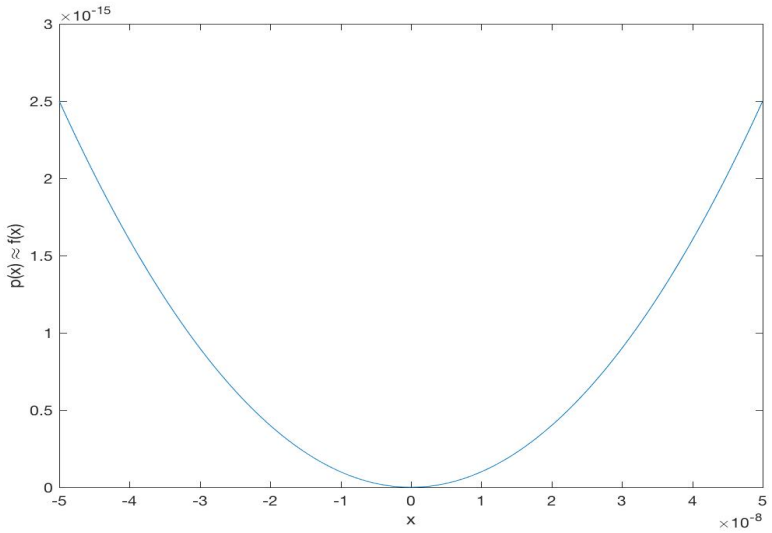

Figure 3: Graph of the approximation $f(x) \approx p(x) = x^2 + \frac{x^3}{6}$ computed in IEEE standard double precision.

Consider the following plots. Figure 5 shows a plot of the function

$$f(x) = \frac{\log(1+x)}{x} - 1$$

over the interval $J = [-8u, 8u]$ computed in double precision by direct evaluation at 1001 evenly spaced points, where u denotes standard double precision machine epsilon. Figure 6 shows a plot of the function

$$g(x) = \frac{\log(1+x)}{(1+x)-1} - 1$$

over the same interval and computed in the same way. Althought $f \equiv g$ mathematically, the plots are quite different. In particular, the values displayed in Figure 5 are on the order of 1 while those in Figure 6 are on the order of 10^{-15}!

The first few terms of the Maclaurin series of $\log(1+x)$ suggest that the plot in Figure 6 better illustrates the behavior of f near zero:

$$\frac{\log(1+x)}{x} - 1 = -\frac{x}{2} + O(x^2).$$

This example illustrates a "magical" cancellation. To evaluate f and g, we first add $1+x$. Since every $x \in J$ is small compared to 1, the addition essentially rounds x, causing much loss of precision (approximately 8 digits in double precision). The evaluation of the logarithm is then limited to at most 8 digits of precision. Since x still has all 16 digits of precision, evaluating the quotient $\log(1+x)/x$ leads to a large deviation from the theoretical value because of the difference in the precision of the numerator and the denominator. This explains the huge error in Figure 5.

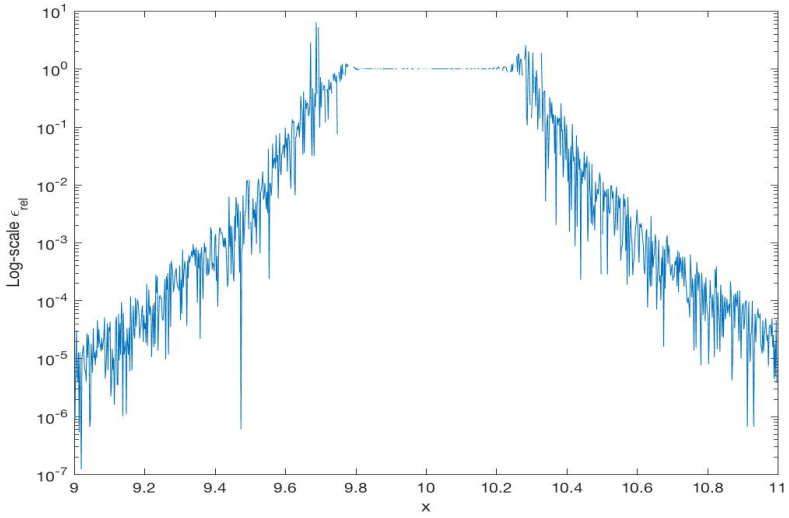

Figure 4: Graph of relative error $|p_e(x) - p_c(x)|/|p_e(x)|$ computed in standard IEEE double precision.

The evaluation of g avoids this error by reducing the precision in x via the operations $(1+x)-1$. The processed denominator has the same precision as the numerator and therefore the quotient can be accurately evaluated.

We now provide an example of a mathematically ill-conditioned problem. Stable numerical methods are even more important in this setting, since small perturbations in the problem input already lead to large theoretical changes in the solution!

Example 5.4. (Determinant problem*) Consider the problem of computing the determinant of a matrix A. Suppose the entries of A are accurate to three digits of precision. For concreteness, let

$$A = \begin{bmatrix} 100 & 15.8 & 12.9 \\ & 101 & -89.3 \\ & & .001 \end{bmatrix}.$$

We can show directly from definition that the determinant of an upper triangular matrix is given by the product of its diagonal entries (readers more advanced in linear algebra may recall that the determinant of a matrix is the product of its eigenvalues and that the eigenvalues of an upper triangular matrix lie on its diagonal). In any case, we have $\det A = 11$.

Now consider the perturbed matrix

$$A_p = \begin{bmatrix} 100 & 15.8 & 12.9 \\ & 101 & -89.3 \\ & & .002 \end{bmatrix},$$

which is identical to A except for the smallest possible perturbation of the bottom diagonal entry. Reasoning as above, we find that $\det A_p = 22$.

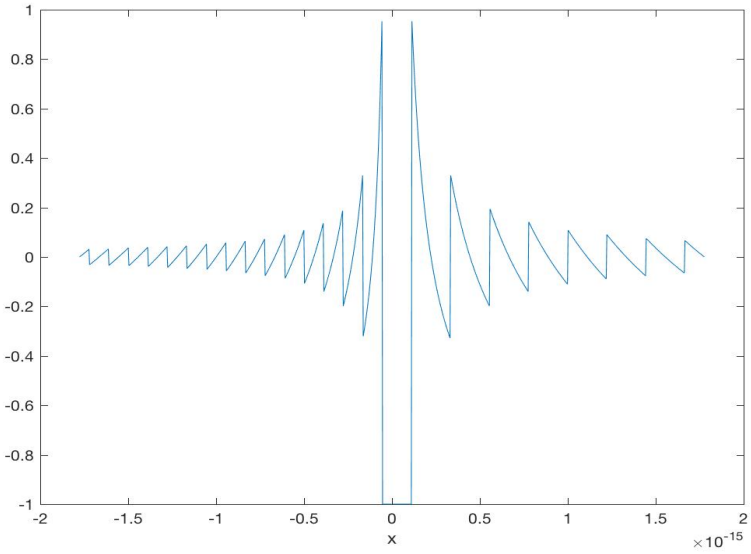

Figure 5: Graph of $f(x) = \frac{\log(1+x)}{x} - 1$ computed in IEEE standard double precision.

Note that the relative error arising from the perturbation is 100%! However, the large deviation is in this case a mathematical difficulty and not an artifice of our numerical method (evidenced by the fact that our computations are exact in each case). It should be clear that applying even a slightly unstable method to the solution of this problem may lead to wildly devianting results.

We note that matrix inversion and the related solution of systems of linear equations with nearly-singular coefficient matrix are also ill-conditioned problems.

Example 5.5. (Permuted LU) In this example we revisit LU computations performed in the last few lectures and argue for the necessity of the permuted LU algorithm. For simplicity, we consider a 2×2 example. Suppose

$$A = \begin{bmatrix} \epsilon & 1 \\ 1 & \pi \end{bmatrix},$$

with ϵ small. For concreteness, we may take $\epsilon = 10^{-14}$.

Performing naive LU decomposition via direct Gaussian elimination (with no row swaps allowed, so that each column is eliminated in turn) yields, in exact arithmetic,

$$L = \begin{bmatrix} 1 & \\ \epsilon^{-1} & 1 \end{bmatrix}, \text{ and } U = \begin{bmatrix} \epsilon & 1 \\ & \pi - \epsilon^{-1} \end{bmatrix},$$

since we eliminate the first column of A by adding to its second row $-\epsilon^{-1}$ times its first row.

Inputting L and U into MATLAB and performing the multiplication LU in standard double precision yields

$$LU = \begin{bmatrix} \epsilon & 1 \\ 1 & 3.140625 \end{bmatrix},$$

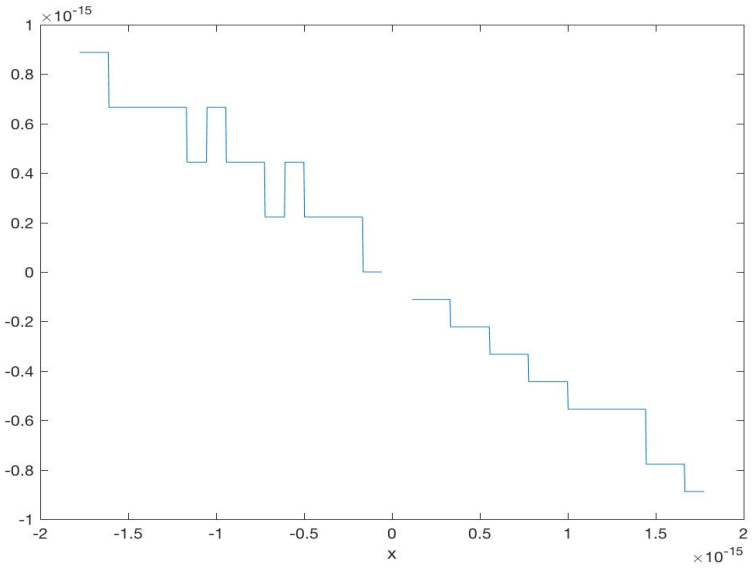

Figure 6: Graph of $g(x) = \frac{\log(1+x)}{(1+x)-1} - 1$ computed in IEEE standard double precision.

and the bottom diagonal is given in the full precision computed by MATLAB. Note that the product recovers $\pi = 3.1415923565897932\ldots$ to only three digits of precision!

The interested reader should verify our computations by running the following code:

```
%Nearly singular LU computation
%Small example
epsilon = 10^(-14);
B = [epsilon 1; 1 pi]
%Brute force LU:
L = [1. 0.; 1/epsilon 1.]
U = [epsilon 1.; 0 pi - 1/epsilon]
L*U
```

The last computation shows that a lot of precision can be lost in an LU computation even in a small toy example. Given the ubiquity of linear systems and the need for their precise solution, the development of stable LU factorizations is imperative.

Permuted LU algorithms alleviate the situation–albeit at some additional cost. In particular, either row- or column-permuted LU yield full precision recovery of A in our example.

Again, the interested reader should verify the recovery by computing the following in MATLAB (the `lu` function automatically applies a permuted LU algorithm when needed):

```
%Permuted LU
[L_perm, U_perm] = lu(B)
L_perm*U_perm
```