Math 361S Lecture notes: Algorithms, floating point arithmetic and error

Jeffrey Wong

January 11, 2018

Topics covered

- Overview
  - Algorithms and pseudocode
  - Design of algorithms
  - Course themes
- Floating point arithmetic
  - Properties
  - Absolute/relative error
  - Implementation on a computer

1 Introduction

1.1 A motivating example: Riemann sums

In calculus, you have studied the definite integral

$$\int_a^b f(x)\,dx.$$

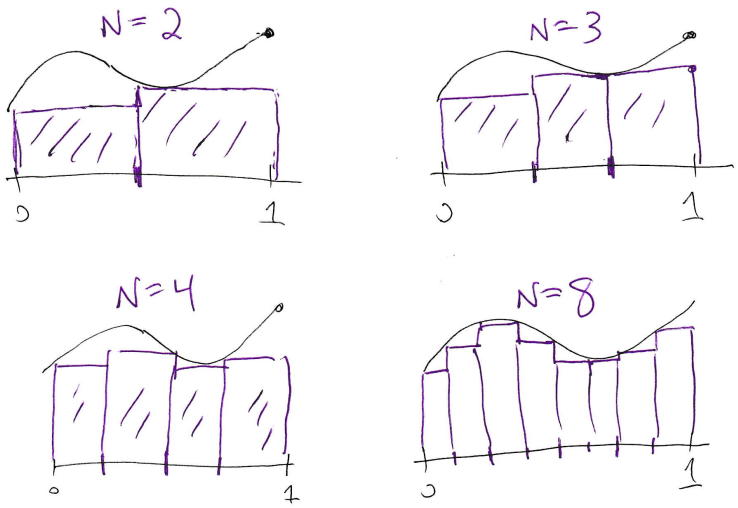
It is defined a s a limit of Riemann sums, obtained by taking finitely many points in $[a, b]$ and constructing rectangles under the curve (see Figure). More than likely, you then spent quite a bit of time establishing rules for evaluating all sorts of integrals exactly, like

$$\int_0^1 \sin^4 x\,dx = \left(\frac{3}{8}x - \frac{1}{4}\sin 2x + \frac{1}{32}\sin 4x\right)\Big|_0^1 = \cdots$$

Often, all we care about is a numerical answer - a number that is close enough to the real value. Riemann sums can provide a way to approximate. The obvious disadvantage is one

has to do a large number of tedious computations, but a computer is happy to do that for us! We can split the domain into N rectangles (say, of equal size) under the curve and obtain

$$I_n = \text{area of } n\text{-rectangle approximation} \approx \int_a^b f(x)\,dx.$$



Even with this simple scheme, there are choices to make. The height of the rectangle can be the maximum in that interval, the minimum (shown above), average and so on. In any case, we get a sequence of approximations I_n such that

$$I_n \to \int_a^b f(x)\,dx \text{ as } n \to \infty.$$

In the theoretical treatment, you may recall taking a convenient choice of Riemann sum, proving existence of the integral and moving on.

Here, we are concerned with a very different set of questions. How fast does I_n converge to the answer, and how does it depend on the function f? How should the approximations be constructed? How can the method be designed to be efficient on a computer? Answering these questions demands a new perspective.

1.2 Algorithms

An **algorithm** is a sequence of steps that takes an input and returns some output. Broadly speaking, algorithms can be considered at three levels:

$$\text{Mathematical description} \to \text{Algorithm (pseudocode)} \to \text{Implementation (code)}$$

At its most abstract, an algorithm is a sequence of mathematical steps applied to input variables (for example, the Riemann sums of the previous section). We will often work with this 'high-level' description of the algorithm, expressed in mathematical terms. Some computational details may be left vague.

Algorithm 1 Fibonacci numbers: Version 1

Input: $n \geq 2$, array F of length $n+1$
Output: F stores $F_0, \cdots F_n$

```
F[0] ← 1
F[1] ← 1
for i = 2, ··· n do
    F[i] ← F[i − 1] + F[i − 2]
end for
```

More specific is an algorithm suited for implementation on a computer. This is the level at which the computational aspects are determined and laid out, so that one could read the algorithm and blindly implement it on a computer. There should be no ambiguity at this level. Such algorithms are usually written in **pseudocode**, a precise set of steps written in a skeletal programming 'language'.

Beyond this is the implementation (the code itself). Often the code and the pseudocode are not really any different, but there can be some work to be done in translation. Here one has to worry about all the details like memory allocation/management, variable types, low-level optimization and so on (much of which is language or hardware specific). Such details are usually left out of the pseudocode unless essential.

As an example, consider the *Fibonacci numbers*, defined by the recurrence

$$F_0 = F_1 = 1, \qquad F_j = F_{j-1} + F_{j-2}, \quad j \geq 2.$$

An algorithm that generates the Fibonacci numbers up to F_n is shown in Algorithm 1 (typeset using the `algorithmcx` package). As long as it is readable and precise, the notation in pseudocode is up to you (there are a variety of styles).

An efficient algorithm takes into account minimizing waste (unneeded storage or computations). If instead, we wanted only to return the n-th Fibonacci number, the version above would do extra work because we only need F_n (the last computed value). See below:

Algorithm 2 Fibonacci numbers: Version 2

Input: $n \geq 2$
Output: y is the n-th Fibonacci number F_n

```
y ← 1        ▷ F_{i−2}
z ← 1        ▷ F_{i−1}
t ← 0        ▷ temp. variable
for i = 2, ··· n − 1 do
    t ← z
    z ← z + y
    y ← t
end for
return y
```

1.3 Good algorithms and course themes

A primary goal of numerical analysis is to produce algorithms that will give good approximations to problems. Some major considerations are as follows:

- Accuracy (*Is the output close to the exact value?*)
  - How should we measure 'error' and 'accuracy' for a given problem?
  - Given a tolerance ϵ, can the algorithm find a solution to within ϵ?
  - Can the algorithm tell us the error (how do we know the solution is close)?
- Stability/Reliability (*Is it sensitive to inputs/errors?*)
  - Do small changes in inputs lead to small changes in the solution?
  - How can we control errors that propagate through the algorithm?
  - Is the algorithm reliable (does it work as a 'black box')?
- Efficiency
  - Time efficiency: Is the algorithm fast?
  - Space efficiency: how much memory is needed?
  - How do the above scale with the size of the problem?
- Miscellaneous concerns
  - Simplicity: Is it easy to implement?
  - Is the algorithm easy to adapt/extend to new problems?
  - Specific optimziation: Can the algorithm be designed to exploit software/hardware advantages (multii-threading, GPUs, vectorization, distributed computing...)

A perfect algorithm would be one that is accurate, efficient, and works on a broad set of problems. Such algorithms are rare - most of the time, there are trade-offs involved. Studying the theory of numerical analysis will help to build intuition that will allow you to solve numerical prolems in the right way and develop algorithms. Here are a few broad themes to keep in mind:

- **Not all calculations are equal:** In pure theory, results are either true or false. A true expression is correct, however it is computed. For a numerical method, there are always errors introduced at each step. The way computations are done can have a dramatic effect on the quality of the result. In particular, there are some things that are fine when numbers are exact, and a disaster when they are not. For example, you know the roots of $ax^2 + bx + c = 0$ are

$$x = -\frac{b}{2} \pm \frac{1}{2}\sqrt{b^2 - 4ac},$$

  but this formula is not always a good way to compute numerical roots!

- **Errors propagate:** As mentioned above, error is a fact of life with numerical methods. Every variable (say, x) in an algorithm is really $x + \epsilon$ for some unknown error ϵ, and that error gets sent through the algorithm. We will therefore need to understand how to track error through an algorithm - does it grow, or shrink, and by how much? Doing so will necessitate developing the appropriate analytical tools.

- **Be precise about 'error' and 'approximation':** In using an algorithm, it is tempting to call an answer close enough. Part of numerical analysis is understanding how to quantify and bound error, which lets us make precise statements about the answer. If it is poor, is it because of rounding error? How much work do we need to do to get an answer to a certain accuracy? We can answer these questions without vague hand-waving, and defend the result with confidence.

- **When do you trust the result?** A similar point to the above. Applying an algorithm in practice requires an understanding of the method. Otherwise, you are throwing inputs into a black box and hoping the result is correct. With a background in numerical analysis, you will have ways to know when to trust an output - and how to get the method to work correctly. A method is often happy to produce an output that is completely wrong - and it is up to you to figure that out.

- **Perfect algorithms don't exist... mostly:** We have a large number of demands to make of algorithms (as noted above). For most problems, it is too much to ask for everything. For this reason, we will need to develop many approaches to solve the same problem - each of which has advantages and disadvantages. Understanding when a method is good or not requires understanding the theory in depth - how properties of functions and so on affect the algorithm.

2 Floating point numbers

2.1 Floating point arithmetic

2.1.1 Error (absolute/relative)

Suppose we have an approximation $\tilde{x}$ to an exact result x. The **absolute error** is

$$e_{abs} = |\tilde{x} - x|$$

and the **relative error** is

$$e_{rel} = \frac{|\tilde{x} - x|}{|x|}.$$

Notation: Symbols used for relative and absolute error may vary. When the quantities are needed, we will use whatever notation is convenient. Often ϵ, δ or Δ are used, but these also often have other meanings.

Note that the relative error is not defined when the exact solution is $x = 0$.

If wehave an estimate for e_{abs} and want e_{rel} but do not know the exact solution x, then it is tempting to divide by $\tilde{x}$ instead:

$$e_{rel} = \frac{e_{abs}}{|x|} \approx \frac{e_{abs}}{|\tilde{x}|}.$$

It is **not always true** that this gives a good estimate for the relative error. However, if used carefully it can still be useful.

2.1.2 Floating point numbers

To begin, it is important to understand how arithmetic is done on a computer. A non-zero **floating point number** in base b is a number x of the form

$$x = \pm(d_0 + f) \times b^e, \qquad f = \sum_{i=1}^{N} d_i b^{-i} = (0.d_1 d_2 \cdots d_N)_b \tag{1}$$

with digits $d_i \in \{0, 1, \cdots, b-1\}$ and $d_0 \neq 0$. Here f is called the **mantissa** and e is the **exponent**. The subscript b is there to clarify the base (which is usually implied). You are, of course, familiar with the base ten version, since it is the way we usually write numerical values, e.g. $c \approx 2.998 \times 10^8$ m/s. In a computer, floating point numbers are represented in binary[1], i.e. base 2. Since there are only two digits, $d_0 = 1$ so

$$x = \pm(1 + f) \times b^e, \qquad f = (0.d_1 d_2 \cdots d_N)_2. \tag{2}$$

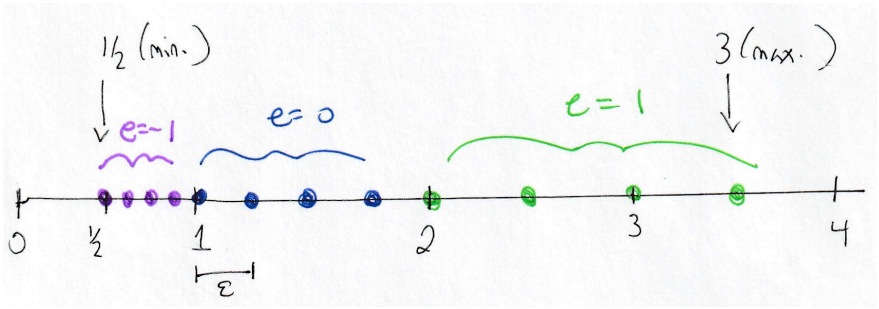
Some examples:

$$3/4 = (1 + 1/2) \times 2^{-1} = (1.1)_2 \qquad (f = (0.1)_2 \text{ and } e = -1)$$
$$35/8 = (1 + 3/32) \times 2^2 \qquad (f = (0.00011)_2 \text{ and } e = 2).$$

The set of numbers (2) with N digits past the decimal point and exponents in a finite range

$$m \leq e \leq M$$

forms a system of floating point numbers $\mathcal{F}$. This set is what we have to work with when doing computations (on a computer). Note that $\mathcal{F}$ has a finite size, and that the numbers are *not* uniformly distributed. For instance, on a number line, with $N = 2$ and $m = -1, M = 1$ the numbers look like:



[1] Unless you travel back in time to Moscow State University in the 1970s, which had a 'ternary' computer that used base 3.

The max/min values are $(1.11)_2 \times 2^1 = 3$ and $(1.00)_2 \times 2^{-1} = 1/2$.

Note that the *absolute* difference between consecutive numbers grows larger as the number grows larger, but the *relative* differences do not. The value ϵ in the diagram will be defined shortly.

2.1.3 Arithmetic, rounding, truncation

Hereafter, suppose N, m, M are fixed and we are working with the corresponding floating point system $\mathcal{F}$. We will ignore most fo the unpleasant details of arithmetic on an actual processor[2] and study a theoretical model for arithmetic in $\mathcal{F}$.

Most real numbers x are not in $\mathcal{F}$. Let us define

$$\mathrm{fl}(x) = \text{'closest' number } \tilde{x} \in \mathcal{F} \text{ to } x.$$

To make this precise, we need a notion of 'closest'. The obvious choice is to round x to the nearest $\tilde{x} \in \mathcal{F}$ by some rounding scheme[3]

The other approach is truncation or 'chopping', which throws out digits past the N-th:

$$0.d_1 d_2 d_3 \cdots d_N d_{N+1} d_{N+2} \cdots \to 0.d_1 \cdots d_N.$$

A simple model of arithmetic in $\mathcal{F}$ is to do the exact operation then round. Define

$$x \oplus y = \mathrm{fl}(x + y), \qquad x \otimes y = \mathrm{fl}(xy).$$

Addition is done by first aligning the floating point numbers to have the same exponents, then adding them together. A base ten example is easiest; take $N = 2$, $x = 1.03 \times 10^2$ and $y = 7.89 \times 10^{-1}$:

$$\begin{aligned} & 1.030 \quad \times 10^2 \\ + \ & 0.00789 \times 10^2 \\ = \ & 1.03789 \times 10^2 \end{aligned}$$

so $x \oplus y = 1.04 \times 10^2$. Notice that if y is much smaller than x then nothing will change by adding y, e.g. if $y = 7.89 \times 10^{-2}$ then

$$\begin{aligned} & 1.03 \quad \times 10^2 \\ + \ & 0.000789 \times 10^2 \\ = \ & 1.003789 \times 10^2 \end{aligned}$$

[2] some details will be included in footnotes like these if you are interested.

[3] The standard is 'round to even': round x to the closest number in $\mathcal{F}$; if x is exactly halfway between two numbers in cF, choose the one that ends in a zero ($d_N = 0$). Example: $(1.0101)_2$ and $(1.0011)_2$ with $N = 3$ both round to 1.010.

which rounds back to 1.04, so $x \otimes y = x$ even though $y \neq 0$.

Multiplication is done by adding the exponents and multiplying the mantissas If $N = 2$ and $x = 3.01 \times 10^6$ and $y = 4.56 \times 10^{15}$ then

$$xy = (3.01 \cdot 4.56) \times 10^{21} = 13.7256 \times 10^{21} \implies x \oplus y = 1.37 \times 10^{22}.$$

Note that the floating point form makes dealing with exponents easy; no re-alignment is required. The only potential concern is that the result ends up outside the range of allowed values. This is called **overflow** (too large) or **underflow** (too small). The result of an overflow is a special 'number' `Inf`, and underflow returns zero[4].

2.1.4 Machine epsilon, significant digits

Define ϵ (**machine epsilon**) to be the distance from 1 to the next-closest floating point number greater than one. For the binary floating point system $\mathcal{F}$ with a given N, the next number greater than one is

$$1.\underbrace{00\cdots0}_{N-1 \text{ zeros}}1 = 1 + 2^{-N}$$

so $\epsilon = 2^{-N}$.

Notation: There are several definitions for machine epsilon, which lead to a difference by a factor of two. You might see ϵ defined in a slightly different way, but the idea is the same. The command `eps` in MATLAB returns ϵ as defined here.

It is not too hard to show that $\epsilon/2$ is an upper bound for the relative error in representing x by a floating point number:

$$\frac{|\text{fl}(x) - x|}{|x|} \leq \epsilon = 2^{-N}. \tag{3}$$

If rounding to the nearest number, the bound can be replaced by $\epsilon/2$ (see homework).

For a `double` (default type in MATLAB/python for non-integers), $N = 52$ so the value of machine epsilon is

$$\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}.$$

Key point: ϵ tells us the typical size of an error in a floating-point arithmetic operation.

The error formula (3) says that for any real number x,

$$\text{fl}(x) = x(1 + \delta), \qquad |\delta| < \epsilon$$

[4]On a computer, enough digits are used to make sure multiplication of the mantissas is exact. In underflow, the result is partially salvaged if not too small ('gradual underflow'), returning a useful value instead of zero

where δ depends on x. This expression is useful for error analysis.

Another important concept is that of **significance**, which you should know from other sciences. The *significant digits* for a floating-point number are the digits we know for sure - the ones above the threshold of the error. When x is turned into $\text{fl}(x)$, we have N significant digits plus the last one that might be rounded:

$$\text{fl}(x) = d_0.d_1 d_2 \cdots d_{N-1} d_N$$

When an approximation has an error, digits far enough to the right are never significant, because they do not tell us anything useful. For instance if

$$x = 1.23456 \pm 0.001$$

then we really only know that $x = 1.234 \pm 0.001$. To avoid implying higher accuracy than is true, **do not report more than the significant digits** in numerical results.

2.1.5 Consequences

Order matters: Unlike exact addition, **floating point addition is not associative**. In general,

$$a \oplus (b \oplus c) \neq (a \oplus b) \oplus c$$

The order in which arithmetic is done can matter. For a simple example, consider arithmetic in $\mathcal{F}$ with a given N and $b = 2$ and let $\epsilon = 2^{-N}$ be machine epsilon. Note that $\epsilon/2$ is the smallest number $x \in \mathcal{F}$ such that $1 + x > 1$. Now let

$$\delta = \epsilon/4 = 2^{-(N+2)}.$$

Then

$$1 \oplus 2^{-(N+2)} = 1.$$

The effects can be quite dramatic when doing many arithmetic operations. Suppose

$$a_1 = a_2 = \cdots = a_{10^6} = \delta, \qquad a_0 = 1$$

and we wish to compute $\sum_{n=0}^{10^6} a_n$. Computing the sum in asecnding order,

$$1 \oplus a_1 \oplus a_2 \oplus \cdots a_{10^6} = 1$$

but, computing the sum in descending order,

$$a_{10^6} \oplus a_{10^6-1} \oplus \cdots \oplus a_1 + 1 = 10^6\delta + 1 \approx 1 + 5 \times 10^{-11}.$$

The issue is that the small contributions get rounded away when added to the large number; one has to add all the small ones together first so they can accumulate.

> **Practical note:** As a rule of thumb, it is typically a good idea to compute sums from smallest to largest to minimize such errors.

Loss of significance: We can lose significance in manipulating floating point numbers, leading to large relative errors. For example, consider (in base ten with $N = 5$) subtracting real numbers a, b with floating point representations

$$\text{fl}(a) = 1.12345, \quad \text{fl}(b) = 1.12334.$$

We get

$$a - b = 0.00011 \implies a \ominus b = 1.1 \times 10^{-5}.$$

The 5 significant digits in a and b have become two. Any further digits are irrelevant, because the are within the error in representing $\text{fl}(a)$ and $\text{fl}(b)$. The true answer could be anything from 1.05×10^{-5} to $1.1499\cdots \times 10^{-5}$ (depending on the rounding scheme):

$$a - b = (1.1 \pm 0.05) \times 10^{-5}.$$

The relative error is 5% even though the relative errors in a and b were quite small (0.05%).

When this sort of calculation arises in an algorithm, it may be necessary to find an alternate method to avoid the error. For example, to find the roots of

$$ax^2 + bx + c = 0$$

the naïve approach would be to compute

$$x = -\frac{b}{2a} \pm \frac{1}{2a}\sqrt{b^2 - 4ac}.$$

But if $b^2 \approx 4ac$ then

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

might introduce errors. However, by computing the roots in another way, we can avoid the issue entirely (see homework)!

Key point: Subtracting two equal numbers can lead to a dramatic loss of significance and very large relative errors.

2.2 Floating point numbers on a computer (optional)

Computers use the IEEE standard for floating point numbers. The representation is

$$x = (-1)^s(1+f) \times 2^{e-q}, \qquad f = (0.d_1 d_2 \cdots d_N)_2$$

where $q = 2^{M-1} - 1$ is the **bias** and e is an integer with

$$0 \le e < 2^M.$$

The exponent of the floating point number is therefore in the range

$$-2^{M-1} + 1 \le e - q \le 2^{M-1}.$$

The bias is used to make the stored value e a non-negative integer.

The number is stored as an array of bits (a string of 0s and 1s). A 'single-precision' number (a `float`) has 32 bits, $M = 8$ and $N = 23$ and a 'double-precision number' (a `double`) has $M = 11$ and $N = 52$. The convention is to list s (one bit), e (M bits) and f (N bits) in order (left to right).

For instance, 1 stored as a `double` ($M = 11$ and $N = 52$) has the form

$$\underbrace{0}_{s=0}\ \underbrace{01111111111}_{e=1023}\ \underbrace{0000\cdots0}_{f\ (52\ \text{zeros})}.$$

since the bias is $q = 2^{M-1} = 1023$ and $e - q$ is zero.

There are some special cases:

- $e = 0$ and $f = 0$ (all zeros except s): The number zero. To be precise, 'positive zero' $(+0)$ if $s = 0$ and 'negative zero' (-0) if $s = 1$, both of which are equal to zero.
- The largest value, $e = 2047$ is reserved to represent `Inf` and `NaN`.
- $e = 0$ and $f \neq 0$: the 'subnormal numbers', which are used in underflow. When a calculation gives a number smaller than the smallest allowed value, these numbers can be used; they have less significant digits.

The largest floating-point number is therefore

$$(2 - 2^{-N}) \times 2^{2^{M-1}-1} \approx 2^{2^{M-1}}$$

and the smallest (positive) 'normal' number is

$$1 \times 2^{-2^{M-1}+2}.$$

For a `double`, the values are $\approx 1.8 \times 10^{308}$ and 2.2×10^{-308}.