Math 361S Lecture notes:
Introduction and computer arithmetic

Updated Jan. 12, 2019

Overview

- **Relevant book sections:** 1.1-1.2 and Chapter 2; Moler 1.1, 1.7. The textbook contains further details on floating point arithmetic.
- Introduction
  - Course themes
  - Desirable properties of numerical methods
  - Algorithms and pseudocode
  - Absolute/relative error
- Floating point arithmetic
  - Floating point number systems
  - Implementation on a computer
  - Rounding errors and consequences
  - Cancellation

1 Introduction

The field of **numerical analysis**, broadly speaking, is concerned with obtaining approximations to the solutions of mathematical problems. The primary goal is to derive methods for 'numerically' solving problems - using a computer to do the calculations[1] - and to understand the relevant properties. The field is situated somewhere between pure analysis and computer science, and draws from both. Some examples in the umbrella of numerical analysis:

- Theory (mostly math)
  - Convergence (limits of sequences that approach the true solution)

[1] The same was true centuries ago - in the 1700s, advances in astronomy demanded precise calculations, which led to clever tools like Napier's tables of logarithms. In that time, a 'computer' was an actual person tasked with doing the computations. Thankfully, that is no longer the case.

- Finite-dimensional spaces for approximation
- Discrete analogues of continuous processes

- Applied (somewhere in between)
  - Derivation of (practical) numerical methods
  - Intuition for interpreting results, measuring error
  - Adapting/generalizing methods to get desired properties
- Implementation (mostly computer science)
  - Translating methods to actual code
  - Efficient implementation
  - Packaging algorithms for general use

The applied side of numerical methods is often called **scientific computing** to distinguish it from the theory, but really all aspects are intertwined. In this course, we focus more on the first two aspects and address the last one in less depth. Hopefully, you will be convinced by the end that an understanding of the underlying mathematics is valuable, even when one is concerned with practical results.

1.1 Algorithms

A numerical **algorithm** is a sequence of steps that takes an input and returns some output that (approximately) solves some mathematical problem. Such algorithms can be considered at three levels:

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

On one end, the code has complete computational detail (variables, memory, control structure etc.), and on the other end, the method is described in abstract terms - as a mathematical process. We will often work with this'high-level' description of the algorithm and leave out computational details.

In between the purely mathematical and the code is an algorithm in pseudocode (. At this level, the computational aspects are determined and laid out, so that the pseudocode could be more or less directly translated to code. At its most precise, pseudocode is specific enough that any two users who implement it should get code that does the same thing.

Example (algorithms and pseudocode): The *Fibonacci numbers* are defined by

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

This is the 'mathematical description' - it tells us exactly how to generate the numbers. But it does not specify how it should be done, and there are decisions to be made!

Algorithm 1 takes an integer $N > 0$ and generates all the Fibonacci numbers up to F_N (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). Because the problem is trivial, the algorithm is just (1) written as a for loop.

Now suppose we only need to generate the n-th number, and not the whole sequence. An efficient algorithm to do so should minimize waste (unneeded storage or computations). Algorithm 2 uses only 3 variables instead of a whole array by re-using them - a useful trick that we will use often.

For the code, there are only a few things to worry about - for instance, in Algorithm 1, the length $N + 1$ array should be initialized at the start so it does not have to grow in the for loop. In both, we may wish to throw an error if N is so large that the result would overflow[2], a practical concern that does not have much to do with the algorithm.

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
```

Algorithm 2 Fibonacci numbers: Version 2

Input: $N \geq 2$
Output: 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
```

1.2 Good algorithms

Some desirable properties for numerical methods are...

- 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)?
- 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?
- Stability/Reliability
  - Do small changes in inputs lead to small changes in the solution?
  - How can we control errors that propagate through the algorithm?
  - How much human attention is required? (Ideally, one should be able to feed it inputs, get an output and not have to worry about what happens inside.)
- Other concerns
  - Can the method handle a wide range of inputs? (How general is it?)
  - Is it simple or convenient to implement?

An algorithm that drastically fails at any of the three is probably useless. An ideal algorithm is one that is accurate, efficient, and reliable. Such algorithms are rare - most of the time, there are trade-offs involved. For most problems, there are a variety of methods with different properties - and numerical analysis provides us with the intuition to choose which is the right one to use.

2 Error and floating point numbers

2.1 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.

Relative error is not defined when $x = 0$. The two notions of error can be quite different, and we will find that both are useful in different contexts. If

$$\tilde{x} = 1.6 \times 10^{-6}, \qquad x = 1.57 \times 10^{-6}$$

then

$$e_{abs} = 3 \times 10^{-8}, \qquad e_{rel} = 0.019.$$

The absolute difference is quite small, but there is an error of about 2% in the approximation.

As another example, from Taylor's theorem, we have that

$$\sin x = x - \frac{1}{6}x^3 + \cdots$$

from which it follows that $\sin x \approx x$ if x is small. When $x = 10^{-3}$,

$$e_{abs} = |\sin x - x| \approx 1.7 \times 10^{-10}, \quad e_{rel} = \frac{|\sin x - x|}{|x|} \approx 1.7 \times 10^{-7}.$$

By both measures, the error is quite small, so $\sin x \approx x$ is a good approximation.

Remark: If we have 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.

2.2 Floating point arithmetic

To begin, we must understand how arithmetic is done on a computer. A non-zero **floating point number** with t digits in base b is a number x of the form

$$x = \pm d_0.d_1d_2\cdots d_{t-1} \times b^e \ = \ \pm b^e \sum_{k=0}^{t-1} d_k b^{-k} \tag{2}$$

with digits $d_k \in \{0, 1, \cdots, b-1\}$ and $d_0 \neq 0$. The part past the decimal is called the **mantissa** and e is the **exponent**. The $b = 10$ is obviously familiar since it is just scientific notation for numbers, e.g. $3142 = 3.142 \times 10^3$. In a computer, floating point numbers are in binary[3], i.e. $b = 2$.

Given $b \geq 2$, a value of t and integers L, U, define

$$\mathcal{F} = \{0\} \cup \{x \text{ of the form (2) with } L \leq e \leq U\}.$$

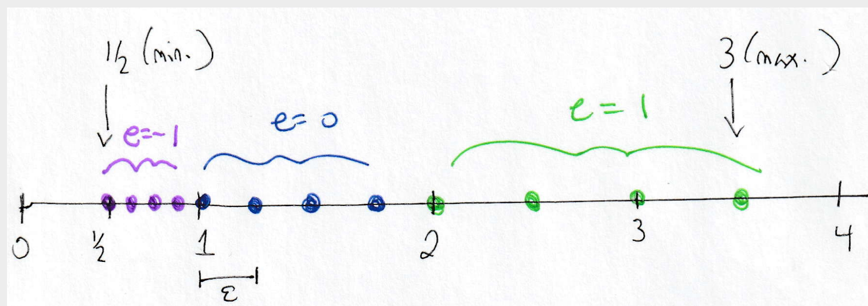
This is a space of 'floating point numbers' with t digits in a fixed range of exponents. The set $\mathcal{F}$ is finite, which allows it to be represented by a fixed-size block of memory in a computer. The two standards that are implemented in hardware are

- Single-precision (`float`): 32 bits; $b = 2, t = 53, L = -126, U = 127$.
- Double-precision (`double`): 64 bits; $b = 2, t = 24, L = -1022, U = 1023$..

Double-precision is the default in modern systems (e.g. in python, Matlab, etc.), and you should use it too (it is rare for single-precision to be needed!).

Example (a small set of floating point numbers):

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 positive numbers look like:



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

The *absolute* difference between consecutive numbers grows larger as the number grows larger, but the *relative* differences do not. The value ϵ is to be defined.

[3]The Setun computer at Moscow State University, developed around 1960, used base 3.

2.3 Arithmetic and rounding

Hereafter, fix some b and t and consider the floating point system $\mathcal{F}$, ignoring the min/max constraints on the exponent e. The implementation details of arithmetic are rather involved, so we will consider only the essentials.

Most real numbers x are not in $\mathcal{F}$. For $x \in \mathbb{R}$, define

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

By closest, we mean that x is rounded either up or down to the nearest floating point number.[4].

For example if $N = 3$ and the base is $b = 10$ then

$$\text{fl}(10\pi) = 3.14159\cdots \times 10^2 \to \begin{cases} 3.142 \times 10^2 & \text{rounding} \\ 3.141 \times 10^2 & \text{chopping} \end{cases}.$$

If $N = 3$ and $b = 2$ then

$$\text{fl}(\pi) = 1.1001001\cdots \times 2^1 \to \begin{cases} (1.101)_2 \times 2^1 & \text{rounding} \\ (1.100)_2 \times 2^1 & \text{chopping.} \end{cases}$$

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

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

Addition is done by first aligning the floating point numbers to have the same exponents, then adding them together. For example, suppose $t = 3$, $x = 1.03 \times 10^2$ and $y = 7.89 \times 10^{-1}$:

$$\begin{aligned} &1.030 \quad\;\; \times 10^2 \\ +\ &0.00789 \times 10^2 \\ =\ &1.03789 \times 10^2 \\ \implies x \oplus y &= 1.04 \times 10^2. \end{aligned}$$

If $|y| \ll |x|$ then nothing will change by adding y, e.g. if $y = 7.89 \times 10^{-2}$ then

$$\begin{aligned} &1.030 \quad\;\;\; \times 10^2 \\ +\ &0.000789 \times 10^2 \\ =\ &1.033789 \times 10^2 \\ \implies x \oplus y &= 1.03 \times 10^2 = x. \end{aligned}$$

[4] There also needs to be a tiebreak rule. The standard is 'round to even': round x to the closest number in $\mathcal{F}$; if x is exactly halfway between two numbers in $\mathcal{F}$, choose the one that ends in a zero ($d_{t-1} = 0$). Example: 1.0101 and 1.0011 with $b = 2$ and $t = 4$ both round to 1.010.

Multiplication is done by adding the exponents and multiplying the mantissas. For instance, with $t = 3$ and $x = 3.01 \times 10^6$ and $y = 4.56 \times 10^{15}$,

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

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.

For a double, $L = 1023$ so the largest number is

$$1.\underbrace{11\cdots 1}_{52\text{ digits}} \times 2^{1023} \approx 2^{1024} \approx 1.8 \times 10^{308}$$

so, for example, 10^{400} will just evaluate to `Inf`. Division by zero or other undefined operations will return another special number, `NaN`, defined so that if $x =$ `NaN` then all arithmetic operations with x also return `NaN`.

2.3.1 Relative error in floating point arithmetic, significance

Observe that for $\mathcal{F}$ with some b and t, the next number after 1 is

$$1.\underbrace{00\cdots 0}_{t-2\text{ zeros}}1 = 1 + b^{-(t-1)}.$$

For each value of e, the spacing between consecutive numbers is:

$$1.\underbrace{00\cdots 0}_{t-2\text{ zeros}}1 = 1 + b^{-(t-1)}.$$

If we round x to the nearest value in $\mathcal{F}$, then it follows that the absolute error in $\text{fl}(x)$ is bounded by half this spacing, so

$$|x - \text{fl}(x)| = \frac{1}{2} b^{-(t-1)} b^e.$$

But

$$x = d_0.d_1 \cdots d_{t-1} \times b^e \geq 1.0 \cdots 0 \times b^e = b^e.$$

since $d_0 \neq 0$, so the relative error has a simpler bound:

$$\frac{|x - \text{fl}(x)|}{|x|} = \frac{1}{2} b^{-(t-1)}.$$

It is also true that arithmetic is implemented in such a way that the relative error is bounded by η. To be precise, if $x, y \in \mathcal{F}$ then

$$x \oplus y = (x + y)(1 + \epsilon), \qquad |\epsilon| < \eta$$

and

$$x \otimes y = xy(1+\delta), \qquad |\delta| < \epsilon$$

where ϵ is the relative error. The value of ϵ depends on x and y, and in practice, it can be though of as a random number between $-\eta$ and η. This means that often, rounding errors will cancel out, but we usually assume the worst.

Rounding unit: The value $\eta = b^{1-t}/2$ is called the **rounding unit** or **machine epsilon** for the floating point system. This value is a **bound on the relative error in representing x by a floating point number and arithmetic operations.**

For a double, the value is $2^{-53} \approx 1.1 \times 10^{-16}$. Sometimes, machine epsilon is defined as twice this value (e.g. in Matlab's `eps` command).

2.4 Consequences (why does this matter?)

Order is important: Floating point arithmetic is not associative! For a simple example, consider arithmetic in $\mathcal{F}$ with a given t and $b = 2$ and let $\eta = 2^{-t}$ be machine epsilon. It is straightforward to show

$$\text{fl}(1+x) = 1 \text{ if } 0 < x < \eta.$$

This has some striking consequences:

$$(\frac{\eta}{2} \oplus 1) \ominus 1 = 0, \qquad \frac{\eta}{2} \oplus (1 \ominus 1) = \frac{\eta}{2} \oplus 0 = \frac{\eta}{2}.$$

For a more dramatic example, consider

$$a_1 = a_2 = \cdots = a_{10^6} = \frac{\eta}{2}, \qquad a_0 = 1$$

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

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

The small values never accumulate and get (wrongly) ignored. But in descending order,

$$a_{10^6} \oplus a_{10^6-1} \oplus \cdots \oplus a_1 + 1 = 10^6 \frac{\eta}{2} + 1 \approx 1 + 5 \times 10^{-11}.$$

More generally, if $|x| \ll |y|$ then some digits of x will get ignored when computed $x + y$, leading to a large absolute error.

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

Cancellation is dangerous: Subtracting two nearly equal numbers can lead to large relative errors. For example, consider (in base ten with $t = 6$) subtracting real numbers a, b with floating point representations

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

We get

$$a - b = 0.00011 \implies a \ominus b = 1.1 \times 10^{-4}.$$

But the errors in a and b are on the order of $\frac{1}{2}10^{-5}$, so the error in $a - b$ true answer could at least be in the range 1.05×10^{-4} to $1.1499 \cdots \times 10^{-4}$ (possibly worse!), i.e.

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

which is a relative error of 5% (quite large!). We have gone from a relative error of $\eta = 5 \times 10^{-5}$ to an error of 0.05 - an increase of 3 orders of magnitude!

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)!

Even worse, when we take a difference and then divide by a small number, the absolute error becomes large. Such quotients appear often, and require care to compute. For example, the derivative of a function $f(x)$ is

$$f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

so it itempting to take small values of h and compute the slope $(f(x + h) - f(x))/h$. But $f(x + h) \approx f(x)$ and h is small, so this may introduce large errors (we'll deal with this in detail when studying numerical differentiation).

Key point: Subtracting two equal numbers can lead to a dramatic increase in relative errors. Dividing by a small number can then lead to a large absolute error; computing $(a - b)/(c - d)$ is a concern when $a \approx b$ and $c \approx d$.

2.5 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

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

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 approximately 1.8×10^{308} and 2.2×10^{-308}.