CSC338 Winter 2022

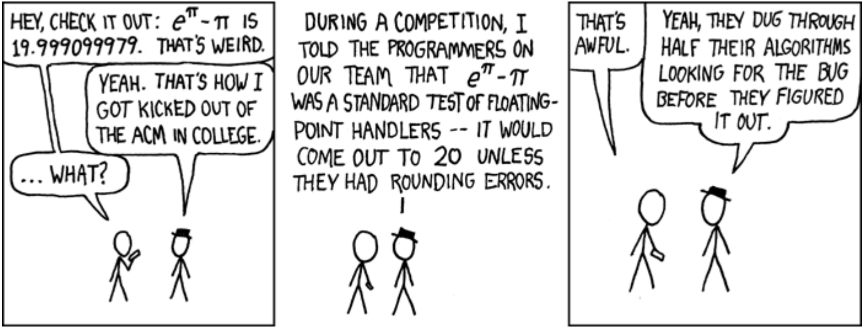
Week 2 - Floating Point Numbers

Ilir Dema

University of Toronto

Jan 21, 2022

Compute $e^{\pi} - \pi$



`https://xkcd.com/217/`

Scientific Notation

Motivation

- Often, we need to represent very small or very large numbers
- How much energy does a photon carry? Well, we just need to multiply its frequency with **Planck's constant** $6.62607004 \times 10^{-34}$
  - Alternatively, 0.0000000000000000000000000662607004
  - So, lots of wasted space to keep zeros, compared to more compressed notation above, known as **scientific notation**.
- NASA uses 16 digits of π in its Space Integrated Global Positioning System/Inertial Navigation System (SIGI) (Source: `https://www.scitecpov.com/blog/pi-mania`)

Floating-Point Numbers

Definition

- Floating-point number system is the set of numbers defined by the tuple $\mathcal{F}(\beta, p, L, U)$ where

β	base, integer ≥ 1
p	precision, integer ≥ 1
$[L, U]$	exponent range, integers $L \leq U$

- A number $x \in \mathcal{F}$ if an only if

  $$x = \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_{p-1}}{\beta^{p-1}}\right)\beta^E$$

  $$0 \leq d_i \leq \beta - 1, i = 0, \ldots, p-1, \quad L \leq E \leq U$$

- $d_0 d_1 \ldots d_{p-1}$ is called *mantissa*
- $d_1 \ldots d_{p-1}$ is called *fractional part*
- E is called *exponent*
- If $\mathcal{F}$ is a fixed floating-point number system, its elements are called *machine numbers*.

Typical Floating-Point Systems

Most computers use binary ($\beta = 2$) arithmetic

Parameters for typical floating-point systems shown below

system	β	p	L	U
IEEE SP	2	24	-126	127
IEEE DP	2	53	-1022	1023
Cray	2	48	-16383	16384
HP calculator	10	12	-499	499
IBM mainframe	16	6	-64	63

IEEE standard floating-point systems almost universally adopted for personal computers and workstations

Example

$\mathcal{F}(10,3,-5,5)$

- $x = -20.5 \in \mathcal{F}$, beacuse $x = -(2 + \frac{0}{10} + \frac{5}{10^2})10^1$
  - mantissa: 205
  - fractional part: 05
  - exponent: 1
- $0.012 = 1.2 \times 10^{-2} \in \mathcal{F}$, however $2.012 \notin \mathcal{F}$ (Why?)
- Notice $0.012 = 0.12 \times 10^{-1}$.
  - Same number may have more than one floating point representation.

Normalized floating-point numbers

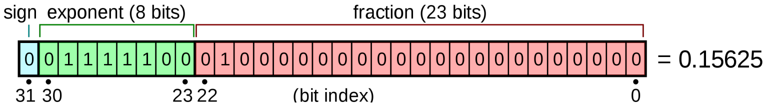
Definition

The floating point system $\mathcal{F}(\beta, p, L, U)$ is called normalized if $d_0 > 0$ for all machine numbers of the form $x = \pm\left(\sum_{k=0}^{p-1} \frac{d_k}{\beta^k}\right)\beta^E$.

Normalized: IEEE-754: $\mathcal{F}(2, 24, -127, 126)$

The 32 bits for single precision are divided as follows: 1 bit for the sign, 8 for the exponent, 23 for the fraction.
(Where is the full mantissa?)

Here is an example (credit - Wikipedia):



The IEEE 754 Floating Point Single Precision Standard (32 bits)

Normalized numbers

- *sign* bit: 0 means positive, 1 means negative
- *exponent*: `eeeeeeee` is the 8 bits of the exponent
  - written in excess 127, that is,
  - real exponent = eeeeeeee-127
  - for normalized numbers:

    `0[00000000]00000000000000000000000 = 0`

    `0[11111111]00000000000000000000000 = ∞`

    previous slide: `01111100-1111111=-11=`$(3)_{10}$
- *mantissa*: `1.`mmmmmmmmmmmmmmmmmmmmmmm
  - 24 digits in 23 bits
  - prev. slide example: `1.01000000000000000000000`

The IEEE 754 Floating Point Single Precision Standard (32 bits)

Special values

- UFL: 0[00000001]00000000000000000000000 $= 2^{-126}$
- OFL: 0[11111110]11111111111111111111111 $= 2^{127}$
- Narrowing the gap between 0 and UFL, we have *denormalized numbers*:

  `0[00000000]mmmmmmmmmmmmmmmmmmmmmmm`

  where not all digits `m` can equal zero

  exponent switches to excess 126

  so the smallest denormalized number is

  0[00000000]00000000000000000000001 $= 2^{-149}$
- NaN (Not a Number), not all `m` can equal `0`:

  `0[11111111]mmmmmmmmmmmmmmmmmmmmmmm`

The IEEE 754 Floating Point Double Precision Standard (64 bits)

Normalized numbers

- *sign* bit: 0 means positive, 1 means negative
- *exponent*: `eeeeeeeeeee` is the 11 bits of the exponent
  - written in excess 1023, that is,
  - real exponent = eeeeeeeeeee-1023
- *mantissa*: `1.m...m` (53 digits in 52 bits)

Representing real numbers using floating point

Floating-point numbers

The representation operator *fl*

- It should be clear we are unable to represent irrational numbers precisely (i.e. $\sqrt{2}, \pi, e$) (Why?)
- Also, many rational numbers have infinite decimal representations (i.e. $1/3$).
- For a real number $x \in \mathbb{R}$ and a floating-point system $\mathcal{F}$ we would like $fl(x)$ to be the element of $\mathcal{F}$ that minimizes $|x - y|, \quad \forall y \in \mathcal{F}$.
- Compactness of $\mathcal{F}$ implies $fl(x)$ exists. In order to make it unique, we will impose a rounding condition.
- For IEEE-754 the rounding rule used is know as *round to even*.

Round to even

- IEEE754 standard allows us to store 23 bits of our number within the mantissa.
- At the 23rd bit, we must round to nearest even.
- When rounding, take a look at what the next three bits (after the 23rd) would have been.
- Cases to consider:
  1. If the next (24th) bit is a 0, then you round down (do nothing)
  2. If the next bit is a 1, followed by either a 10, 01, or 11, you round up
  3. If the next three digits are "100" this is a tie (we are midway between two representable numbers). In this case:
     A. If the last number in the mantissa (23rd bit) is a 1, then round up adding 1 to the mantissa's 0 least significant digit.
     B. If the last number in the mantissa (23rd bit) is a 0, then do nothing

Absolute Representation Error

Definition and consequences

- By definition, absolute representation error is $fl(x) - x$.
- If we chose a represenation by chopping extra digits, $fl(x) - x$ has the opposite sign of x, bounded in the interval $[-\beta^{1-p}, 0]$ (Exercise: prove this!)
- If we use the round to even rule, the error is bounded in $[-\beta^{-p}, \beta^{-p}]$
- If we apply chopping, generally the errors during computations tend to accumulate.
- As opposed to that, rounding gives better error propagation results.

ε_{mach}

DEFINITION

Smallest positive machine number such that $fl(1+\varepsilon) > 1$ is called *epsilon of the machine* and denoted ε_{mach}.

- **Exercise:** $\varepsilon_{mach} = \beta^{1-p}$ if rounding is done by chopping
- **Exercise:** $\varepsilon_{mach} = \frac{1}{2}\beta^{1-p}$ using rounding to the nearest
- **Exercise:** For any nonzero real number x within the normalized range

$$\left|\frac{fl(x)-x}{x}\right| \leq \frac{1}{2}\beta^{1-p}$$

. You may assume rounding to the nearest.
- Generally

$$0 < UFL < \varepsilon_{mach} < OFL$$

Relative Representation Error

Definition and consequences

- By definition, for any nonzero x, relative representation error is $\delta = \frac{fl(x)-x}{x}$.
- Rearraning, $fl(x) = (1+\delta)x$.
- The triangle inequality implies $|fl(x)| = |(1+\delta)x| \leq (1+|\delta|)|x|$. Since we aim for the best representation possible, it follows $|\delta| \leq \varepsilon_{mach}$.

Properties of floating-point numbers

$\mathcal{F}(\beta, p, L, U)$

- $\mathcal{F}$ is a finite set. In particular, a normalized floating-point system contains $2(\beta-1)\beta^{p-1}(U-L+1)+1$ elements (Why?).
- Largest positive number in $\mathcal{F}$ is $(\sum_{k=0}^{p-1}\frac{\beta-1}{\beta^k})\beta^U = \beta^{U+1}(1-\frac{1}{\beta^p})$ (Why?) [Such number is called *overflow*]
- Smallest positive number in $\mathcal{F}$ is β^L. [Such number is called *underflow*]

Arithmetic operations on floating-point values

$\mathcal{F}(\beta, p, L, U)$

- Usual operations (+,-,*,/) are not well defined on $\mathcal{F}$.
- Addition: $\beta^{U+1}(1 - \frac{1}{\beta^p}) + 1 \notin \mathcal{F}$
- Division: $\beta^L/\beta \notin \mathcal{F}$
- Subtraction and multiplication: exercise
- However, for two elements $x, y \in \mathcal{F}$ we can compute their regular sum, product, etc. Then using the *fl* operator on the result we can obtain a number if $\mathcal{F}$ which is the best approximation of the result.
- This idea can be extended for any two real number as follows: for any two $x, y \in \mathbb{R}$ and any operator *op*, it is desirable to have:

$$fl(x)\, flop\, fl(y) = fl(x\, op\, y).$$

Propagated error

Assume variables $x_1, \ldots, x_n \in \mathcal{F}(\beta, p, L, U)$

- Consider the statement $y = f(x_1, \ldots, x_n)$. Assuming $x_1, \ldots, x_n$ are floating point numbers and y is within representation range, $fl(y) = (1+\delta) f(x_1, \ldots, x_n)$.
- Now assume $x_1, \ldots, x_n$ are produced from prior computations. Then, in order to carry out computation of f, we apply $f((1+\delta_1)x_1, \ldots, (1+\delta_n)x_n)$.
- The relative error into computing f is called *propagated error*.
- In the next slides, we will compute the propagated error for basic arithmetic operations and for functions of one variable. The ideas laid here certanly may apply to functions of many variables as well.

Floating point addition

$fl(x) +_{fl} fl(y) = fl(x+y), x, y$ have same sign

- As crazy as it sounds, this definition can be made to work, with a few caveats. (Note the plus operator suffixed with "fl" indicates floating point addition.)
- The idea is as follows:
    - Step 1, is to rewrite both $fl(x)$ and $fl(y)$ using as exponent the maximal exponent of both exponents.
    - Step 2, is to add mantissas, round using the prescribed round rule, and normalize (if working with normalized numbers)
- Example:

```
  1.23450x10^12  = 1.23450x10^12
 +4.50000x10^10  = 0.04500x10^12
--------------------------
                 = 1.27950x10^12
```

Propagated Error Analysis

From before, if x, y are already in $\mathcal{F}$, the relative error producedby addition is δ, for $fl(x+y) = (x+y)(1+\delta)$.

As a result, $fl(x+y) = (x+y)(1+\delta) = x(1+\delta) + y(1+\delta)$.

That means, letting $\hat{x} = x(1+\delta), \hat{y} = y(1+\delta)$ we see result of floating point addition is exact for the operands x, y perturbed with a relative backward error $|\delta|$.

If two terms x, y originate from a prior computation, are both nonnegative, with at least one of them strictly positive, so $fl(x) = x(1+\delta_1), fl(y) = x(1+\delta_2)$, the propagated relative error is

$$\frac{x(1+\delta_1) + x(1+\delta_2) - (x+y)}{x+y}$$

$$= \frac{x}{x+y}\delta_1 + \frac{y}{x+y}\delta_2 \leq \delta_1 + \delta_2$$

Unexpcted consequences of the floating point operations

Some (normal arithmetic) divergent series may converge

Consider the harmonic series

$$\sum_{n=1}^{\infty} \frac{1}{n}.$$

Exercise: for any $M > 0$, find N, such that $\sum_{n=1}^{N} \frac{1}{n} > M$. This shows harmonic series diverge.
Now assume $a_n = \frac{1}{n} \in \mathcal{F}$. Let m be the first integer such that $\frac{1}{m} < \varepsilon_{mach}$. It is not hard to show for all $n > m$, the equality $\sum_{k=1}^{n} a_k = \sum_{k=1}^{n+1} a_k$ holds.
This implies **the harmonic series, given its general term is computed as a floating-point number, converges!**
Exercise: Sum this series in single precision IEEE-754.

Some familiar algebraic laws no longer hold (but some do)

```
In [27]: eps = 1
         e = 1
         while 1+e>1:
             eps = e
             e=e/10
         print(eps)

         1e-15
```

Associative Property no longer holds

```
In [48]: a = 0.1*eps
         print(((((((((((1+a)+a)+a)+a)+a)+a)+a)+a)+a)+a))
         print((1+(a+(a+(a+(a+(a+(a+(a+(a+(a+a)))))))))))

         1.0
         1.000000000000001
```

Note, commutative property for the addition of two numbers holds. (Why?)

Rearranging terms in an expression can have serious consequences

In [49]:
```
print(1+a+a+a+a+a+a+a+a+a+a)
print(a+a+a+a+a+a+a+a+a+a+1)
```

```
1.0
1.000000000000001
```

In [4]:
```
from decimal import *
getcontext().prec=6
a,b = Decimal(1.2345e12), Decimal(1.25000e2)
print(a+b-a, a-a+b)
```

```
0 125
```

ERROR ANALYSIS FOR THE SUM OF n NUMBERS

Let $S = a_1 + a_2 + \cdots + a_n$. Then

$$fl(a_1 + a_2) = (a_1 + a_2)(1 + \delta_2)$$

$$fl(a_3 + S_2) = (a_3 + S_2)(1 + \delta_3)$$

$$\ldots$$

$$fl(a_n + S_{n-1}) = (a_n + S_{n-1})(1 + \delta_n)$$

Therefore, denoting ΔS the absolute error,

$$\Delta S = S_n - S$$

$$= (\delta_2 + \cdots + \delta_n)a_1 + (\delta_2 + \cdots + \delta_n)a_2 + (\delta_3 + \cdots + \delta_n)a_3 + \cdots + a_n\delta_n$$

It follows the bound for the relative error is $|\Delta S/S| \leq (n-1)\varepsilon_{mach}$. Also the error can be minimized if we multiply the larger sum of δ_k, that is the coefficient to the first two factors, with smallest term. It follows we minimize the error if the terms of the sum S are sorted in increasing order.

Floating point subtraction

The definition and implementation of subtraction is similar to the addition. However subtraction comes out with a serious problem.

A simple example

Let $0 < \varepsilon < \varepsilon_{mach}$.

$$(1+\varepsilon)-(1-\varepsilon) = 0 \neq 2\varepsilon$$

Please note, both $1+\varepsilon$ and $1-\varepsilon$ values are close to one another. This fenomenon is known as *catastrophic cancellation*. Its effects can be mitigated by modifying the computational method in use.

Exercise: Do the propagated error analysis for subtraction.

Catastrofic cancellation

Example - quadratic formula

In [10]:
```
from math import sqrt
from decimal import *

getcontext().prec = 4

def solve_quad(a, b, c):
    a, b, c = Decimal(a), Decimal(b), Decimal(c)
    two, four = Decimal(2.0), Decimal(4.0)
    d = Decimal(sqrt(b*b-four*a*c))
    return (-b+d)/two/a, (-b-d)/two/a

a, b, c = 0.05010, -98.78, 5.015
#
# Correct roots are 1971.605916, 0.05077069387
#
r1, r2 = solve_quad(a, b, c)
print("Naively computed roots are: {}, {}".format(r1, r2))
```

```
Naively computed roots are: 1972, 0.07519
```

In [11]:
```
#
# Compare:
#
x1, x2 = 0.05077069387, 0.07519
print(a*x1*x1+b*x1+c, a*x2*x2+b*x2+c)
```

```
4.5553694150157753e-10 -2.4119849578413906
```

CATASTROFIC CANCELLATION

IMPROVED QUADRATIC FORMULA

Clearly, $-b$ and d have similar values. Let's take on this case and avoid subtraction:

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

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

The formula for the other root does not change.

Exercise: Write a Python function that computes the roots of quadratic equation avoiding catastrophic cancellation.

Floating point multiplication

Overflow

Although addition can result in overflow, multiplication is more prone to it. As an example, consider computation of Euclidean distance of a point from the origin:

$$d(x,y) = \sqrt{x^2 + y^2}$$

A straight forward implementation of it would suffer from unncessary overflow. Consider instead, letting $m = \max\{x, y\}$:

$$d(x,y) = m\sqrt{(\frac{x}{m})^2 + (\frac{y}{m})^2}$$

Since both $\frac{x}{m}, \frac{y}{m}$ are ≤ 1 we avoid the overflow as much as possible. Also note one of the terms is precisely 1. Exploit this to do an error analsys for this formula.

Floating point multiplication

Definition

Let the floating point representation for x, y be respectively $fl(x), fl(y)$. The floating point product of x and y is the floating point representation of the product $fl(x)fl(y)$. Then $fl(fl(x)fl(y)) = fl(x)fl(y)(1+\delta)$ where $|\delta| \leq \varepsilon_{mach}$.

Relative Error Analysis

$$r(xy) = \frac{fl(fl(x)fl(y)) - xy}{xy} = \frac{fl(x)fl(y)(1+\delta) - xy}{xy}$$

$$= \frac{fl(x)fl(y) - xy}{xy} + \frac{fl(x)fl(y)}{xy}\delta$$

Since $fl(x)fl(y) \approx xy$, we may conclude

$$r(xy) = \frac{fl(x)fl(y) - xy}{xy} + \delta$$

Floating point multiplication

Propagated error

Let relative representation errors for x, y be $r(x) = \frac{fl(x)-x}{x}, r(y) = \frac{fl(y)-y}{y}$ and denote propagated error $p(xy)$. Compute:

$$p(xy) = \frac{fl(x)fl(y) - xy}{xy} = \frac{(xr(x)+x)(yr(y)+y) - xy}{xy}$$

After expanding and cancelling, we find

$$p(xy) = r(x) + r(y) + r(x)r(y)$$

Ignoring the $r(x)r(y)$ (what is its bound ?) we conclude

$$p(xy) = r(x) + r(y)$$

Floating point division

Exercise

We leave the error analysis for the floating point division as an exercise. In particular, you need to show that the propagated error for $\frac{x}{y}$ is $r(x) - r(y)$ where $r(x), r(y)$ are the floating point relative errors of representations of x, y.

Error analysis for the computation of a smooth function $f(x)$

Propagated error

Using Taylor's expansion,

$$fl(f(x)) = f(fl(x)) = f(x(1+\delta)) = f(x) + x\delta f'(\xi)$$

where $x \leq \xi \leq (1+\delta)x$. The relative error of computing f is bounded by

$$\varepsilon_{mach}\left|\frac{xf'(x)}{f(x)}\right|$$

Exercises

1. Evaluate the relative error of computing $f(x) = \frac{1}{x}$.
2. Assume we need to compute the list of numbers $x_i = a + ih$, where a, h are fixed, and i varies from 1 to n. Which one would you prefer to use given the computations are done in floating-point arithmetic?

(I)
```
x=[a+i*h for i in range(1,n+1)]
```

(II)
```
x = []
val = a
i = 0
while i < n:
    val = val + h
    x.append(val)
```

Exercises

1. Evaluate the relative error of computing dot product of two vectors: $\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$. Now suppose the coordinates are given is single precision and the answer is also expected to be in single precision. Give a solution with total relative error ε_{mach}.

References

Michael T. Heath
Scientific Computing (Revised Second Edition)
SIAM