MATH3511/6111: Scientific Computing

06. Floating Point Arithmetic

Lindon Roberts

Semester 1, 2022

Office: Hanna Neumann Building #145, Room 4.87
Email: lindon.roberts@anu.edu.au
Based on lecture notes written by S. Roberts, L. Stals, Q. Jin, M. Hegland, K. Duru.

Australian National University

Numerical Errors

We have already seen that the results of numerical computations can give different results than theory. For example, the error in forward differencing should be $\mathcal{O}(h)$ as $h \to 0$:

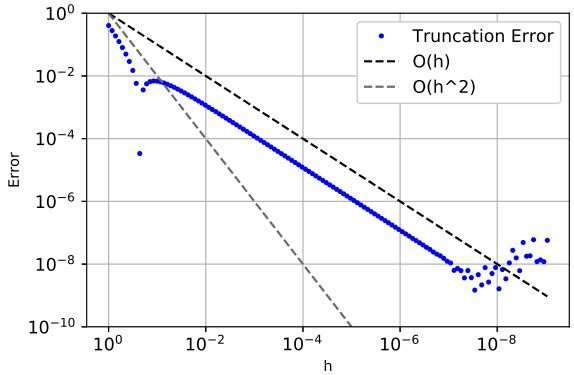

Figure 1. Error in approximating $f'(1)$ for $f(x) = x \sin x$ using forward differencing.

Computer Numbers

As of November 2021, the fastest supercomputer in the world (Fugaku at the RIKEN Center for Computational Science, Japan) has a peak performance of 442 petaflops, or about 4×10^{17} arithmetic operations per second.

The Gadi supercomputer (National Computational Infrastructure at ANU) is ranked #53, achieving 9 petaflops (9×10^{15} operations per second).

Computers are limited how many digits they can store for any number, so errors are introduced as soon as data is input into a calculation. Every arithmetic operation carries forward these errors, and may introduce new errors. If we are not careful, the results of our computations can quickly be meaningless.

To write good numerical software, it is important to know how computers store numbers and what impact arithmetic operations have on existing errors.

Number Systems

Computers do not have infinite memory, so have to decide how to represent a number using finitely many digits. There are two main ways to do this:

- In a fixed point system, every number is stored to the same number of decimal places:

$$0.012 \qquad 12.345 \qquad 1234.567$$

- In a floating point system, every number is stored in scientific notation with the same number of significant digits:

$$1.23 \times 10^{-2} \qquad 1.23 \times 10^{1} \qquad 1.23 \times 10^{3}$$

  (in this case, the number of significant digits is 3)

Modern computers store real numbers in a floating point system.

They also have a separate (exact) representation for integers up to a certain size, but they cannot be used for most scientific computing algorithms, where quantities are in $\mathbb{R}$, not $\mathbb{Z}$.

Machine Numbers

We work in base-10 (decimal system), but computers use base-2 (binary) because its hardware is based around on/off switches.

But beware that some decimal numbers, such as 0.1, has an infinite representation in binary, 0.000110011...

Therefore storing these numbers requires some kind of rounding or chopping. There are only finitely many numbers that can be stored exactly, and these are called machine numbers.

Floating Point System

Modern computers store real numbers in a floating point system. Formally, this is:

Definition

A floating point system is defined by three properties:

- β is the base (or radix) of the system (e.g. $\beta = 2$ for binary)
- t is the precision — the number of significant figures
- $[L, U]$ is the exponent range, where L and U are integers

Every number in the floating point system has the form

$$\pm \left(b_0 + \frac{b_1}{\beta} + \cdots + \frac{b_{t-1}}{\beta^{t-1}} \right) \beta^e,$$

where each $b_i \in \{0, 1, \ldots, \beta - 1\}$ and $e \in [L, U]$ are integers.

Note that integers can be stored exactly, in the usual way: $164_{10} = 128 + 32 + 4 = 10100100_2$.

Normalised Numbers

Definition

A normalised floating point system is one where $b_0 \neq 0$.

This ensures that every machine number has a unique representation. For example, in base-10, a normalised system doesn't allow representations like

$$0.123 \times 10^3 \qquad 0.008 \times 10^{-5}$$

In a binary normalised system, the first digit must be $b_0 = 1$. Since this is always true, it doesn't need to be stored in memory.

Floating Point Numbers

For a floating point number

$$\pm \underbrace{\left(b_0 + \frac{b_1}{\beta} + \cdots + \frac{b_{t-1}}{\beta^{t-1}} \right)}_{=m} \beta^e,$$

the value m is called the mantissa, e is the exponent and $\pm$ is the sign. In a normalised system, we always have $m \in [1, \beta)$.

Example

What is the binary floating point representation of 0.75 (with $\beta = 2$ and $t = 3$)?

Since $0.75 = 0.5 + 0.25$, the binary representation of 0.75 is $0.11 = 1.10 \times 2^{-1}$. With $t = 3$, the binary floating point representation is

$$+ \left(1 + \frac{1}{2} + \frac{0}{2^2} \right) 2^{-1},$$

so $b_0 = b_1 = 1$, $b_2 = 0$ and $e = -1$.

IEEE Arithmetic

Most computers use the same standard floating point system, first defined by the IEEE (Institute of Electrical and Electronics Engineers) in 1985 and last updated in 2019.

The most common format in use today (including in Python) is IEEE double precision, but other standards are defined too:

IEEE format	β	t	L	U
Half precision	2	11	-14	15
Single precision	2	24	-126	127
Double precision	2	53	-1022	1023
Quadruple precision	2	113	-16382	16383

One double precision number takes 64 bits/binary digits (8 bytes) of memory:

- 1 bit for the sign (0 for $+$, 1 for $-$)
- 11 bits for the exponent e (save the binary representation of $e + U \in \{1, \ldots, 2046\}$)
- 52 bits for the mantissa ($t = 53$ but not explicitly storing the normalised $b_0 = 1$)

IEEE Arithmetic

Consider the double precision number (sign-exponent-mantissa)

```
0 10000000011 1011100100010000000000000000000000000000000000000000000000
```

This represents:

- Sign 0 is $+$
- Exponent 10000000011 is $e + U = 2^{10} + 2^1 + 2^0 = 1027$, so $e = 4$ (since $U = 1023$)
- Mantissa 1011100100010000000000000000000000000000000000000000000000, represents $b_0 = 1$ (normalised) plus $b_1 = b_3 = b_4 = b_5 = b_8 = b_{12} = 1$, all other $b_i = 0$

Hence it represents the (decimal) number

$$+\left(1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-12}\right) \times 2^4 = 27.56640625 \,(\text{exactly})$$

Note that the exponent bits can never be all zeros or all ones (i.e. $e + U \neq 0, 2047$).

IEEE Arithmetic

The IEEE standard also defines some special 'numbers', such as ± 0 and $\pm\infty$:

```
0 00000000000 0000000000000000000000000000000000000000000000000000 = +0.0
1 00000000000 0000000000000000000000000000000000000000000000000000 = -0.0
0 11111111111 0000000000000000000000000000000000000000000000000000 = +inf
1 11111111111 0000000000000000000000000000000000000000000000000000 = -inf
0 11111111111 ************* any 52 bits not all zero ************* = NaN
```

NaN means "not a number", and is result of performing undefined operations such as $0/0$, or arithmetic involving existing NaN values, such as $\text{NaN} + 3 = \text{NaN}$. Note the convention that $\text{NaN} = \text{NaN}$ and $\text{NaN} > \text{NaN}$ and $\text{NaN} < \text{NaN}$ are all false (this can cause problems).

The value -0 acts essentially the same way as 0 for arithmetic, and has the convention that $+0 = -0$ is true. It comes about from calculations like $(-1) \times (+0)$ or from rounding very small negative numbers to zero.

The values $\pm\infty$ act as you might expect: $\infty - 1 = \infty$ and $3 > -\infty$, but $0 \times \infty = \text{NaN}$.

Overflow and Underflow

If the result of a computation is larger/smaller than the largest/smallest available (finite) value, the result is 'rounded' to $\pm\infty$. This is called overflow.

If the result is so close to zero that it can't be represented properly, the result is rounded to zero. This is called underflow.

Machine Epsilon

The smallest positive machine number ϵ such that $1 + \epsilon \neq 1$ is also a machine number is called machine epsilon.

This represents the maximum relative error introduced when you round/chop a number.

For IEEE double precision,

$$\epsilon = 2^{-52} \approx 2.22 \times 10^{-16},$$

so effectively every number is stored to approximately 15 or 16 significant figures (in its decimal representation). In Python, you can check your machine precision using NumPy:

```
import numpy as np
# What is the machine epsilon of the Python built-in "float" type?
print(np.finfo(float).eps)
```

Rounding

If the result of a calculation has more than t significant figures/bits, the result is rounded or chopped. For example, $2.316711061881 \times 10^2$ is

- Rounded to 2.316711062×10^2
- Chopped to 2.316711061×10^2

Both are considered rounding to 10 significant figures.

Is this a large or small error?

Rounding

In general, suppose we have a floating-point number

$$\sigma\left(b_0.b_1b_2\cdots b_nb_{n+1}\cdots\right)\times\beta^e,$$

(where $\sigma=\pm1$ is the sign) which is rounded to $n+1$ significant figures:

$$\sigma\left(b_0.b_1b_2\cdots b_n\right)\times\beta^e,$$

The error is

$$\sigma\left(0.b_{n+1}b_{n+2}\cdots\right)\times\beta^{e-n},$$

which is $\leq\beta^{e-n}$ in absolute value.

Is this a large or small error?

It depends on the size of the numbers involved (relative vs. absolute errors)

Relative and Absolute Errors

Definition

Suppose $\widetilde{x}$ is an approximation of x.

- The error in the approximation is

$$\widetilde{x} - x.$$

- The absolute error in the approximation is

$$|\widetilde{x} - x|.$$

- The relative error in the approximation is

$$\frac{|\widetilde{x} - x|}{|x|}.$$

Rounding Errors

Rounding errors of 1 part in 10^{16} are rarely a problem. However, larger errors, particularly those that accumulate over many calculations, can cause severe problems.

Example: Patriot Missile Failure (Gulf War, Iraq, 1991)

Missile defence system used 24 bit floats, and stored times in multiples of 0.1 seconds. With rounding, the value 0.1 (stored as a binary decimal) was rounded to 24 bits:

$$0.0001100110011001100110011001100\ldots \rightarrow 0.00011001100110011001100$$

The absolute error in this value is 9.5×10^{-8}, so this is the error in each 0.1 second increment.

The system had been online for 100 hours, so all times were calculated with error

$$(9.5 \times 10^{-8}) \times (100 \times 60 \times 60 \times 10) \approx 0.34 \text{ seconds.}$$

A missile was travelling at 1676 metres per second, so 0.34 seconds means 570 metres. The defence system didn't detect the missile properly and 28 American soldiers were killed.

Floating Point Model

Modelling

How can we mathematically analyse the effect of rounding errors?

We define a function $\text{fl} : \mathbb{R} \to \mathbb{R}$, which represents the rounding/chopping operation. That is, $\text{fl}(x)$ is the floating-point representation of (the true value) x.

Any floating-point system has a limit on the size of the relative error, so we assume:

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

for numbers within the allowable range (i.e. no overflow), and where ϵ is the machine epsilon.

- If we do chopping, $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$ for double precision.
- If we do rounding, $\epsilon = 2^{-53} \approx 1.11 \times 10^{-16}$ for double precision.

Multiplication

Suppose we want to calculate $x \times y$. Since we can only work with floating point numbers, we actually compute

$$\begin{aligned} \text{fl}(\text{fl}(x) \times \text{fl}(y)) &= \text{fl}(x(1+\delta_1) \times y(1+\delta_2), \\ &= [x(1+\delta_1)y(1+\delta_2)](1+\delta_3), \end{aligned}$$

where $|\delta_i| \leq \epsilon$. Therefore,

$$\text{fl}(\text{fl}(x) \times \text{fl}(y)) = (xy)(1+\delta_4),$$

where

$$1 + \delta_4 = (1+\delta_1)(1+\delta_2)(1+\delta_3).$$

Multiplication

Considering the largest and smallest possible values of δ_i, we get

$$(1-\epsilon)^3 \leq 1+\delta_4 \leq (1+\epsilon)^3.$$

That is

$$1-3\epsilon+3\epsilon^2-\epsilon^3 \leq 1+\delta_4 \leq 1+3\epsilon+3\epsilon^2+\epsilon^3,$$

or, approximately (since ϵ is very small)

$$|\delta_4| \leq 3\epsilon,$$

so the accumulation of our errors is (at worst) roughly additive.

Multiplication

Relative errors of size 3ϵ are usually not a problem.

However, one place which can easily cause problems is testing for equality:

```
>>> (0.1 * 0.2) == 0.02
False
>>> 0.1 * 0.2
0.020000000000000004
```

Warning

Never test if two floating-point numbers are exactly equal (`x==y`).

Instead, check that they are close (e.g. `abs(x-y) < 1e-15`).

Addition and Subtraction

Let's look at the impact of rounding errors on addition and subtraction, rounding to 10 significant figures ($\epsilon = 5 \times 10^{-10}$):

True calculation:

$$\begin{aligned} x &= 0.8888888888888888 + 0.8888888888444444 \\ &= 1.7777777777333332. \end{aligned}$$

Floating-point calculation:

$$\begin{aligned} \widetilde{x} &= \text{fl}(\text{fl}(0.8888888888888888) + \text{fl}(0.8888888888444444)) \\ &= \text{fl}(0.8888888889 + 0.8888888888) \\ &= 1.777777778. \end{aligned}$$

The relative error is very small (size similar to ϵ):

$$\frac{|\widetilde{x} - x|}{|x|} \approx 1.5 \times 10^{-10}.$$

Addition and Subtraction

Let's look at the impact of rounding errors on addition and subtraction, rounding to 10 significant figures ($\epsilon = 5 \times 10^{-10}$):

True calculation:

$$\begin{aligned} x &= 0.8888888888888888 - 0.8888888888444444 \\ &= 0.000000000444444 = 4.44444 \times 10^{-10}. \end{aligned}$$

Floating-point calculation:

$$\begin{aligned} \widetilde{x} &= \text{fl}(\text{fl}(0.8888888888888888) - \text{fl}(0.8888888888444444)) \\ &= \text{fl}(0.8888888889 - 0.8888888888) \\ &= 1.000000000 \times 10^{-10}. \end{aligned}$$

The relative error is very large, much larger than ϵ:

$$\frac{|\widetilde{x} - x|}{|x|} \approx 0.775.$$

Subtraction

In our example, floating-point subtraction introduced massive errors: we went inputs with 10 digits of accuracy to an output with zero digits of accuracy!

Let's calculate the rounding error:

$$\begin{aligned}\text{fl}(\text{fl}(x) - \text{fl}(y)) &= [x(1+\delta_1) - y(1+\delta_2)](1+\delta_3),\\ &= \left[(x-y)\left(1 + \frac{x\delta_1 - y\delta_2}{x-y}\right)\right](1+\delta_3),\\ &= (x-y)(1+\delta_4),\end{aligned}$$

where

$$|\delta_4| \leq 2\epsilon\frac{\max(|x|,|y|)}{|x-y|} + \epsilon + \mathcal{O}(\epsilon^2),$$

which can be very large if $|x-y|$ is small compared to $|x|$ and/or $|y|$.

Subtraction

Warning

Be careful subtracting two similar numbers!

For example, when doing finite differencing:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

we expect $f(x+h) \approx f(x)$ for h small. We saw the effect of rounding errors before:

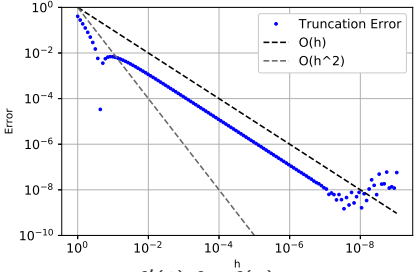

Figure 2. Error in approximating $f'(1)$ for $f(x) = x \sin x$ using forward differences.

Catastrophic Cancellation

A situation where a floating-point calculation loses almost all the accuracy in the inputs (such as subtracting similar numbers) is known as catastrophic cancellation.

Usually it is hard to know when this problem will arise, but often there are tricks to avoid it.

Catastrophic Cancellation: Example

Suppose we try to compute e^x by using the Taylor series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

This series converges for all $x \in \mathbb{R}$. In Python, we have

```
def my_exp(x):
    ans = 1.0; term = 1.0; n = 0
    while ans+term != ans:  # continue adding new terms until the answer doesn't change
        n = n+1
        term = term * (x/n)
        ans = ans+term
    return ans
```

Catastrophic Cancellation: Example

Let's compare our function to the built-in `math.exp` function for different x values:

```
 x          my_exp(x)                 math.exp(x)
 40    2.353852668370201e+17    2.353852668370200e+17
 20    4.851651954097905e+08    4.851651954097903e+08
  1    2.718281828459046e+00    2.718281828459045e+00
 -1    3.678794411714424e-01    3.678794411714423e-01
-20    6.147561828914626e-09    2.061153622438558e-09
-40    3.116951588217358e-01    4.248354255291589e-18
```

Our function is mostly accurate, but very wrong for large negative x values.

For $x > 0$, the terms in the sum are all positive. For $x < 0$, the terms alternate positive and negative (and sum to a value close to zero).

Catastrophic Cancellation: Example

For $x = -20$, the terms are

```
  n             term
  0     1.000000000000000e+00
  1    -4.000000000000000e+01
  2     8.000000000000000e+02
...
 25    -7.258620724607510e+14
 26     1.116710880708848e+15
 27    -1.654386489939034e+15
 28     2.363409271341476e+15
...
138     1.755139541306663e-16
139    -5.050761269947230e-17
140     1.443074648556351e-17
```

We can see that catastrophic cancellation can occur for $n \approx 25$.

Catastrophic Cancellation: Example

How can we avoid this problem?

We need to make sure we don't use the Taylor series for $x < 0$. What to do instead? Use the identity

$$e^{-x} = \frac{1}{e^x}.$$

```
def my_exp2(x):
    # Calculation of exp(x), avoiding catastrophic cancellation
    if x >= 0.0:
        return my_exp(x)
    else:
        return 1.0 / my_exp(-x)
```

Catastrophic Cancellation: Example

Let's compare our new function to the built-in `math.exp` function for different x values:

x	my_exp2(x)	math.exp(x)
40	2.353852668370201e+17	2.353852668370200e+17
20	4.851651954097905e+08	4.851651954097903e+08
1	2.718281828459046e+00	2.718281828459045e+00
-1	3.678794411714423e-01	3.678794411714423e-01
-20	2.061153622438557e-09	2.061153622438558e-09
-40	4.248354255291587e-18	4.248354255291589e-18

Now we get accurate results for all values of x (all within 15 significant figures).

Choice of Precision

Questions

What precision should you use?

- Double precision is standard for almost all floating-point calculations
  - Some older codes use single precision (32-bits, $\epsilon \approx 10^{-7}$)
  - Some machine learning libraries (e.g. TensorFlow, PyTorch) sometimes have single precision as default (faster speed vs. lower accuracy)
- IEEE standard has higher and lower precision types available (16-, 128-bit are the most common), available in some programming languages
- Higher precision: more accuracy, but takes more time, memory and energy
- Growing research area: mixed-precision computing — use low precision to get fast approximate solution, then switch to higher precision for accuracy
  - e.g. Baboulin et al, *Accelerating scientific computations with mixed precision algorithms*, Computer Physics Communications 180 (2009), 2526–2533.