ENGSCI 233 - Numerical Errors
Lecture 1

Colin Simpson

Semester One 2022

Department of Engineering Science
University of Auckland

(Beamer Theme: Metropolis)

Introduction to Numerical Errors

In Software Quality, you were introduced to the concepts of software testing and handling coding errors.

In Numerical Errors, we will consider the issues of:

- Finite precision, associated with computer representation of numbers in a floating point format.
- Truncation error, associated with the simplification of exact mathematical solutions to an approximate one.

We can use *convergence testing* to determine if our numerical answer is sufficiently accurate.

Floating Point Numbers

Computer Representation of Numbers

We often require computers to represent a wide range of numbers. For example, the mass of an electron is given by $m_e \approx 9.1093897 \times 10^{-31}$ kg.

We also often require computers to represent numbers to a high level of precision. For example, π is an irrational number i.e. there are an infinite number of digits after the decimal point.

How does a computer with finite memory represent numbers with both high precision and across a wide range of values?

Floating Point Representation

Computers typically store real numbers in floating point format, with a *sign*, *significand*, *base* and *exponent*:

mentissa

$$3.1415 = \underbrace{+}_{\text{sign}} \underbrace{31415}_{\text{significand}} \times \underbrace{10}_{\text{base}}{}^{\overbrace{-4}^{\text{exponent}}}$$

This can be generalised to any real number:

$$x = \pm d_0 d_1 d_2 \dots d_{p-1} \times \beta^e$$

$$x = \pm \beta^e \sum_{i=0}^{p-1} d_i \beta^{-i}, \quad 0 \le d_i < \beta.$$

where there are p digits in the significand.

$x = + 10^{-4} \left(3 \cdot 10^{-0} + 1 \cdot 10^{-1} + 4 \cdot 10^{-2} + 1 \cdot 10^{-3} + 5 \cdot 10^{-4} \right)$

Floating Point and Computers

Computers typically store numbers in memory using a binary representation. In other words, they use a base of $\beta = 2$.

Lets consider what the following number represents, $\beta = 2$, $e = 2$, $p = 4$, $d = 1101$ and a positive sign:

$$x = +\beta^e \sum_{i=0}^{p-1} d_i \beta^{-i}, \quad 0 \leq d_i < \beta.$$

$$x = 2^2 \left(1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}\right)$$

$$x = 4 \left(1 + \frac{1}{2} + \frac{1}{8}\right)$$

$$x = 6.5$$

Representation Error

Representation Error

Consider the case of representing the number 0.1 with $\beta = 10$:

$$x = +1 \times 10^{-1}$$

With $\beta = 2$ it is more difficult. We could try the following:

$$x = +100110011001100110011001101 \times 2^{-4}$$

$$x = 0.10000000\underbrace{14901161193847656 25}_{\text{representation error}}$$

This is a number that can't be exactly represented in $\beta = 2$. There will always be some *representation error* for many numbers.

Floating Point Precision

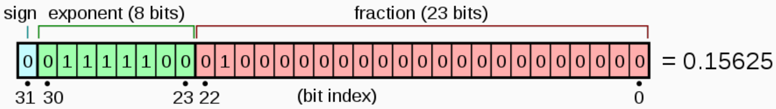
Achieving higher precision typically requires a longer significand. In Python, we can choose our precision to some extent:

name	significand bits	exponent bits	approx. decimal precision	type
half	11	5	3	`numpy.float16`
single	24	8	6	`numpy.float32`
double	53	11	15	`numpy.float64`

By default, single precision numbers are used. They achieve an overall good balance between memory and precision.

IEEE 754

IEEE 754 is a technical standard that formalises working with floating point numbers. Storing a single point precision number:



IEEE 754 provides formal definitions for higher than double precision numbers, though these are rarely used in computer programming applications.

Let's leave our consideration of floating point numbers there...

Rounding Error

Floating Point Operations

Floating point operations can result in round-off error. Consider multiplying two numbers, each with n bits in the significand:

- The exact result would have $2n$ bits in the significand.
- In floating point arithmetic, our result will typically have the same number of significand bits i.e. we must round out the trailing n bits.

Let's consider the example in error_rounding.py.

Example Explained

We were able to represent the number a exactly in both half and single precision i.e. no representation error.

The multiplication of a with itself (i.e. a^2) can be represented exactly in single precision i.e. no round-off error. In this case, the significand was large enough to accommodate the exact result.

However, in half precision the trailing bits in the significand were rounded out, giving an error in the floating point arithmetic.

Error Accumulation

Error Accumulation

As we have seen, floating point operations will generally incur both *representation error* and *rounding error*.

In calculations that require multiple consecutive floating point operations, the total error can accumulate. (Sum)

Let's consider the examples in error_accumulation.py.

Examples Explained

In our first example, the vector components were such that there was no representation error in double and single precision, but there was representation error in half precision.

In our second example, each dot product resulted in rounding error in single precision but none in double precision.

In both cases there was accumulation of error in our sum. For large vectors this led to quite substantial differences in the final results.

Errors in Subtraction

Catastrophic Cancellation

Subtraction of similar floating point numbers can sometimes result in a large relative error, called *catastrophic cancellation*.

The number being subtracted needs to share the same exponent, and therefore may be subject to considerable representation error.

Example of Catastrophic Cancellation

Consider the example of calculating:

$$x - y = 10.1 - 9.93$$

The exact answer is 0.17. Consider representing both numbers with $\beta = 10$ and $p = 3$. The first number is stored as:

$$x = 1.01 \times 10^1$$

y must be stored with the same exponent, but due to the limited number of significand bits is subject to representation error:

$$y = 0.99 \times 10^1$$

The floating point arithmetic therefore returns a result of $\underline{0.2}$ rather than 0.17 - a very large error relative to the original result.

Additional Example

Consider another example of floating point subtraction:

$$x - y = 1.00000 - 0.0000001$$

Representing these with $\beta = 10$ and $p = 6$ will result in the second number being rounded to zero:

$$x = 1.00000 \times 10^0$$
$$y = 0.00000 \times 10^0$$

Therefore, there is no effect of subtracting y from x in our floating point arithmetic.

Machine Epsilon

For a given precision, machine epsilon is the smallest number that can be subtracted from another number and have no effect.

If say we check if two floating point numbers are equal, we are comparing their difference to machine epsilon.

Let's consider the example in error_eps.py.