MECH2700 - Week 1

Floating-point numbers and errors

Travis Mitchell - t.mitchell@uq.edu.au

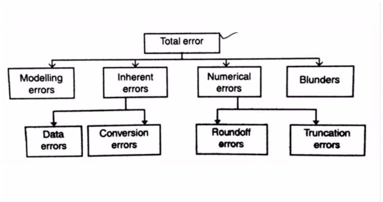
THE UNIVERSITY OF QUEENSLAND AUSTRALIA

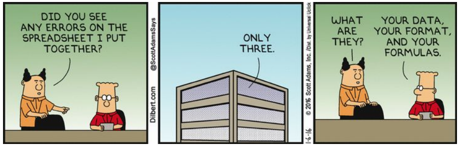
CREATE CHANGE

Lecture Overview

This lecture will cover:

- Approaches to Numerical Problem Solving
- Truncation Errors
- Round-off Errors
- Machine Representation and Arithmetic




DID YOU SEE ANY ERRORS ON THE SPREADSHEET I PUT TOGETHER?
ONLY THREE.
WHAT ARE THEY?
YOUR DATA, YOUR FORMAT, AND YOUR FORMULAS.

Introduction to Numerical Problem Solving

Steps in the problem-solving method, according to Feldman & Koffman (1996):

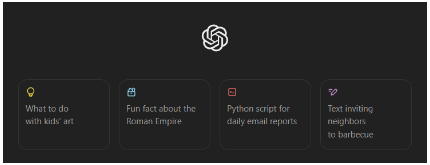
1. Gain an understanding of the specification.
2. Conduct a thorough analysis.
3. Develop a design that includes an *algorithm*.
4. Develop a test plan.
5. Implement the design.
6. Test the implementation.

Notes:

- Note that we don't start with writing code.
- The whole process will be iterative. We probably won't achieve the desired result immediately
- Programming is an exercise in design. Usually we will implement the design and see if it works. This is at the core of engineering.

Introduction to Numerical Problem Solving - Cares in the modern world

The rise of Large Language Models and ‘Artificial Intelligence’


What to do with kids' art
Fun fact about the Roman Empire
Python script for daily email reports
Text inviting neighbors to barbecue

- These tools can greatly boost a programmer’s efficiency[1]. *Not sure if the electricity costs of training the models balances out this efficiency gain.
- But you need to be VERY careful. Kabir et al.[2] from Purdue University found “52% of ChatGPT answers contain incorrect formation”.
- Important to know e.g., what algorithms exist, approaches to verify and validate implementations, how simple functions can be coupled together to solve complex problems, what makes good vs bad (or dangerous) code, . . .

[1]Under the caveat that they know what they are doing!
[2]Kabir et al. (2024) Is Stack Overflow Obsolete? An Empirical Study of the Characteristics of ChatGPT Answers to Stack Overflow Questions, https://arxiv.org/pdf/2308.02312

Introduction to Numerical Problem Solving - Approaches

Approaches to the task of devising a solution:

1. **Top-down approach**:
    - Start from the problem specification and work, step by step, toward a solution.
    - At each step, subdivide the problem into smaller chunks.
    - This approach is sometimes called *stepwise refinement*.
2. **Bottom-up approach**:
    - Build small components, or tools, with limited functionality.
    - Build bigger components from the smaller components until we have a tool that will do the desired job.

- Classical texts emphasised the top-down approach. It provides a structure to your implementation, however, practical programming seems to be a mix of the two approaches.
- As your experience increases and your collection of programming tools grows, the bottom-up approach gets easier and becomes more natural.

Introduction to Numerical Methods

We will be looking at some of the procedures or “recipes” for making specific calculations, such as approximating an integral.

- This will involve combinations of floating-point numbers, via their operators + - * /
- Both the methods and the individual operations will typically be in (hopefully small) error
- The study of the errors in numerical methods is called numerical analysis

Types of error

To get an appreciation for the approximate nature of many numerical methods, we consider two classes of error

1. **Truncation error** is related to errors in the method
2. **Round-off error** is related to the inexact nature of computer arithmetic for floating-point numbers

- For those who believe that the computer is always right, we are going to sow a few seeds of doubt...
- Spreadsheet example:

```
old = 0.1
new = 11 * old - 1
repeat down column
```

Truncation Error

Truncation error is so named because it is made by approximating an infinite series by truncating it to a finite sum

- the **Taylor Series** is a common example of an infinite series, and will be used often; go back and revise it if necessary
- applies equally to other truncated infinite sums, for example using a polynomial to compute the exponential function e^x

$$e^x \approx P_5(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!}$$

Truncation Error Example

Exponential function
The exponential function e^x is defined by the infinite series:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

If we were to evaluate this expression numerically, we truncate after n terms:

$$P_n(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

Truncation Error Example cont.

Consider the case where the polynomial is truncated as such:

$$P_5(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!}$$

The infinite sum can be expressed in terms of this polynomial:

$$e^x = P_5(x) + \sum_{n=6}^{\infty} \frac{x^n}{n!}$$

and hence $\sum_{n=6}^{\infty} \frac{x^n}{n!}$ is the truncation error of the finite sum.

Truncation Error Example — Python Code

The following code progressively calculates better estimates to $e(-0.4)$ by adding more terms of the infinite sum to our approximation.

```python
# exp_taylor.py
"""
Exponential function evaluated via Taylor Series.
"""

from math import exp

x = -0.4
n = 10
t = 1.0
series_sum = 1.0
for i in range(1,n+1):
    t = t * x / i
    series_sum = series_sum + t
    print('i={:3d} t={:12.4e} series_sum={:16.8e}'.format(i, t, series_sum))
print("-"*50)
print("                  exp({:f})={:16.8e}".format(x, exp(x)))
```

Truncation Error Example — Python Code Output

As the number of terms used increases (left column), the series sum approaches the math library calculated value (right column)

```
i=  1 t= -4.0000e-01 series_sum=  6.00000000e-01
i=  2 t=  8.0000e-02 series_sum=  6.80000000e-01
i=  3 t= -1.0667e-02 series_sum=  6.69333333e-01
i=  4 t=  1.0667e-03 series_sum=  6.70400000e-01
i=  5 t= -8.5333e-05 series_sum=  6.70314667e-01
i=  6 t=  5.6889e-06 series_sum=  6.70320356e-01
i=  7 t= -3.2508e-07 series_sum=  6.70320030e-01
i=  8 t=  1.6254e-08 series_sum=  6.70320047e-01
i=  9 t= -7.2240e-10 series_sum=  6.70320046e-01
i= 10 t=  2.8896e-11 series_sum=  6.70320046e-01
---------------------------------------------------
                 exp(-0.400000)=  6.70320046e-01
```

- As expected for an alternating series, the error is less than the magnitude of the final term.
- So it seems that we simply need to select the number of terms to retain and we can compute the exponential function to any precision that we wish, however...

Round-off Error

There is another numerical error, related to the inexact nature of computer arithmetic for floating-point numbers

- Consider the same sum for x = -30.4 with 120 terms

```python
from math import exp

x = -30.4
n = 35
t = 1.0
series_sum = 1.0
for i in range(1,n+1):
    t = t * x / i
    series_sum = series_sum + t
    print('i={:3d} t={:12.4e} series_sum={:16.8e}'.format(i, t, series_sum))
print("-"*50)
print("                exp({:f})={:16.8e}".format(x, exp(x)))
```

Round-off Error Example — Python Code Ouput

At first the output doesn't look out of the ordinary, with alternating terms as last time:

```
i=  1 t= -3.0400e+01 series_sum= -2.94000000e+01
i=  2 t=  4.6208e+02 series_sum=  4.32680000e+02
i=  3 t= -4.6824e+03 series_sum= -4.24973067e+03
i=  4 t=  3.5586e+04 series_sum=  3.13365904e+04
i=  5 t= -2.1636e+05 series_sum= -1.85028242e+05
```

And by the end of the code it appears to have converged...

```
i=110 t=  8.2258e-16 series_sum= -2.27170954e-04
i=111 t= -2.2528e-16 series_sum= -2.27170954e-04
i=112 t=  6.1148e-17 series_sum= -2.27170954e-04
i=113 t= -1.6450e-17 series_sum= -2.27170954e-04
i=114 t=  4.3868e-18 series_sum= -2.27170954e-04
i=115 t= -1.1596e-18 series_sum= -2.27170954e-04
i=116 t=  3.0391e-19 series_sum= -2.27170954e-04
i=117 t= -7.8963e-20 series_sum= -2.27170954e-04
```

Except when we check it against the math library!

```
i=118 t=  2.0343e-20 series_sum= -2.27170954e-04
i=119 t= -5.1969e-21 series_sum= -2.27170954e-04
i=120 t=  1.3165e-21 series_sum= -2.27170954e-04
--------------------------------------------------
                exp(-30.400000)=  6.27260226e-14
```

Round-off Error Example cont.

To check that it is our sum that is in error (and not `math.exp`)

- Casio calculator results: $e^{-30.4} = 6.272602\text{e}{-14}$
- check alternative calculation, both exponential terms computed using the series Python code we wrote:

$$\frac{e^{-0.4}}{e^{30}} = 6.272602\text{e}{-14}$$

To understand this error, we need to understand **floating point numbers** and **floating point arithmetic** .

Representation of Floating-Point Numbers

The general form of a floating-point number:

$$\pm \quad 0. \quad \underbrace{d_1 \; d_2 \; \dots \; d_p}_{\text{fractional part}} \quad \times \quad \underset{\substack{\uparrow \\ \text{Base}}}{B}^{e} \longleftarrow \text{integer exponent}$$

- The fractional part has digits $0 \leq d_i \leq B - 1$
- We use base 10 commonly, most computers use 2
- Integer exponent has range $E_{min} \leq e \leq E_{max}$

Representation of Floating-Point Numbers cont.

Example with Base 10, p=4

$$\begin{aligned} 27.39 &\rightarrow +0.2739 \times 10^{2} \\ -0.00124 &\rightarrow -0.1240 \times 10^{-2} \end{aligned}$$

- normalized values with $d_1 \neq 0$

Note that there are an infinite number of real values (on the number line) however there are only a finite number of floating point values in our system:

$$2 \times (B - 1) \times B^{P-1} \times (E_{max} - E_{min} + 1) + 1$$

IEEE Floating-Point Standard

There are two levels of precision in the IEEE standard:

- **single** precision: B=2, -128 $\leq e \leq$ 127, p=23

$\pm$	e	$d_1\ d_2\ \ldots\ d_p$
1 sign bit	8 bits	23 bits

- **double** precision: B=2, -1024 $\leq e \leq$ 1023, p=52

$\pm$	e	$d_1\ d_2\ \ldots\ d_p$
1 sign bit	11 bits	52 bits

Mechanics of Computer Arithmetic

Computer arithmetic is the source of the rounding error in our exponential example.

Let's look at some examples of addition and subtraction.

Computer arithmetic examples
Perform the following operations using the system:

$$\text{B}=10,\ \text{p}=3,\ -9 \leq e \leq 9$$

Example 1: $0.137 \times 10^{1} + 0.269 \times 10^{-1}$
Example 2: $0.485 \times 10^{4} - 0.482 \times 10^{4}$

Computer arithmetic examples

Example 1

$0.137 \times 10^1 + 0.269 \times 10^{-1}$

$$\begin{array}{rlll}
 & 0\,.\,1\,3\,7 & \times 10^1 & \text{align the decimal points by} \\
+ & 0\,.\,0\,0\,2\,6\,9 & \times 10^1 & \text{changing the small exponent} \\
\hline
 & 0\,.\,1\,3\,9\,6\,9 & \times 10^1 & \text{chopped to give } 0.139 \times 10^1 \text{ (p=4)} \\
+ & 0\,.\,0\,0\,0\,5 & & \\
 & 0\,.\,1\,4\,0\,1\,9 & \times 10^1 & \text{rounded to give } 0.140 \times 10^1
\end{array}$$

Note the difference in the last digit between the "chopped" and "rounded" result. This is why most floating point hardware works to an 80-bit register then stores 64 bits to main memory.

Computer arithmetic examples

Example 2

$0.485 \times 10^4 - 0.482 \times 10^4$

$$\begin{array}{rcll} & 0\,.\,4\,8\,5 & \times 10^4 & \text{no alignment this time} \\ - & 0\,.\,4\,8\,2 & \times 10^4 & \\ \hline & 0\,.\,0\,0\,3 & \times 10^4 & \text{no chopping or rounding this time} \end{array}$$

$$\begin{array}{rcll} & 0\,.\,3\,0\,0 & \times 10^2 & \text{normalize} \end{array}$$

This time, chopping/rounding gives the same result.

Note that we have only 1 significant digit in the result.

Floating point precision ε_M

Any real number $x \neq 0$ is replaced by its representative floating-point value x^* such that:

$$\frac{|x - x^*|}{|x|} \leq \frac{1}{2}\varepsilon_M$$

where $\varepsilon_M = B^{1-p}$ is the distance between 1 and the closest floating-point number different from 1.

Floating point precision ε_M in Python

Python command to return machine precision: `sys.float_info`

We might compute it ourselves:

```
eps = 1.0
while eps + 1.0 > 1.0:
    eps /= 2.0
```

Let's look at the resulting value of ε_M

```
>>> eps = 1.110223e-16
>>> eps
1.110223e-16
>>> 1.0+eps == 1.0 #test statement
True
>>> eps *= 2.0
>>> eps
2.220446e-16
>>> 1.0+eps == 1.0 #test statement
False
```

Tips for working with floating point

- Avoid exact comparisons to floating point values.
  Say we want to test if $x = 8.932$:
  - Avoid: `if x == 8.932`
  - Prefer: `if fabs(x - 8.932) < tol` where `tol` is some small tolerance

  Exception: Tests against 0.0 are alright, if you know you will explicitly set a value to 0.0. However, they should still be avoided if you are approaching the value for 0.0 by computation.
- Try to avoid formulas and expressions with subtraction of nearly equal quantities.

Today's summary

Today we covered:

- Details of the course.
- Numerical problem solving, combining modern and traditional approaches.
- Types of errors to be aware of when writing computer algorithms.
- A bit of background as to how computers represent numbers and perform operations.

From here:

- Familiarise yourself with the Course Profile (assessment due dates, policies, etc.).
- Familiarise yourself with the Blackboard (and complete poll for online office hour).
- Familiarise yourself with the Ed Discussion Board.
- Come to tomorrow's Contact.
- Be sure to sign onto an ICT session.

Some further reading

Bradie, *A Friendly Introduction to Numerical Analysis*
Ch. 1 Getting Started

Gerald and Wheatley, *Applied Numerical Analysis*
Ch. 0 Preliminaries: Computer Arithmetic and Errors

Hamming, *Numerical Methods for Scientists and Engineers*
Ch. 2 Numbers

Goldberg, What every computer scientist should know about floating-point arithmetic,
Computer Surveys, March 1991