L5: Errors and Computer Arithmetic

1 Brief Introduction

When numbers are processed in a computer, they must somehow be transferred to a finite memory space. This means that the numbers are usually rounded, which means that an error is introduced. This error can grow when the numbers are used repeatedly in a program. Therefore, it is important to know how errors resulting from rounding numbers affect the accuracy of numerical methods when they are implemented. Here we will look at the types of errors involved and how to measure the size of the error. In principle, there are two types of errors: discretization errors and rounding errors.

- *The discretization error* occurs when you represent something continuous, for example a function or a differential equation, as something finite in a computer.
- In addition to this error, the computer rounds all real numbers that are stored because the storage space is limited, this leads to a *rounding error.*
- Together, these two types of error make up the total error that we see in numerical calculations. In this lab, we discuss the two types of errors, how they occur and how they interact during a numerical simulation.

Start by downloading the Python file for task 5 of this lab: `eriv`emo.py . The rest of the tasks can be done directly in the Python prompt, so you don't need to write your own programs.

2 Measuring Error

Before we get into how errors occur in a computer, we need a way to measure the errors. One can either look at the *absolute* or the *relative error.* They are defined as

$$\epsilon_{abs} = |x_{approx} - x_{exact}| \quad \text{and} \quad \epsilon_{rel} = \frac{|x_{approx} - x_{exact}|}{|x_{exact}|}. \tag{1}$$

The absolute error depends on the size of the numbers you work with, i.e. on which units are used. The relative error is dimensionless, i.e. independent of which units you work in, and is sometimes expressed as a percentage.

1. Suppose you buy a hot dog in town. It costs SEK 15, but you leave SEK 20 and forget to receive the change. How much have you lost in absolute and relative terms, ie. what is the absolute and relative error?
2. Suppose you buy a new house that costs SEK 1,975,000. The seller persuades you to pay SEK 1,980,000 in exchange for some old furniture that is in the house. You later regret paying extra, because you don't really need the furniture, but then it's too late. How much have you lost in absolute and relative terms, i.e. what is the absolute and relative error?

In one case, you have lost much more money, but it may not feel like that. Both ways of measuring, absolute and relative, are of course correct, but possibly the relative error corresponds more to how one experiences it.

3 Round-off and discretization errors

When calculations with real numbers are performed on a computer, the result of each calculation is rounded. This usually leads to rounding error.

3. You can see the rounding error in simple calculations. Type the following in the Python prompt:

```python
a = 4/3
b = a - 1
c = b + b + b
print(c)
```

   The exact value of `is 1, what value does the` omputer give? Approximately how big is the difference?

4. When Python performs larger calculations, successive partial results are rounded in the calculations. This means that you get an accumulated rounding error. Investigate the order of magnitude of the error by performing the calculation $\|I_N - A^{-1}A\|_{\max}$, where I_N is the identity matrix and A is a random matrix of size N. The max-norm of a matrix X is defined as

$$\|X\|_{\max} = \max_{ij} |x_{ij}| \tag{2}$$

   where x_{ij} are the matrix elements. Using NumPy, a possible implementation is

```python
A = np.random.rand(N,N)
err = np.max(np.abs(np.eye(N)-np.matmul(np.linalg.inv(A),A)))
```

   The exact answer is zero. How does the round-off error change as the matrix size increases? Note that computing the inverse of a large matrix is very expensive, so don't use too large values for N. Start, for example, with $N = 10$.

To see how discretization error occurs and how it interacts with rounding error, we shall study two different numerical methods for approximating the derivative of a function $f(x)$:

$$\text{Forward difference:} \quad f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{3}$$

$$\text{Central difference:} \quad f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{4}$$

Compare with the definition of the derivative:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{5}$$

You would expect the approximation to become the better the smaller h is chosen.

Suppose we want to approximate the derivative of a function on a finite interval $[a,\, b]$. We introduce a discretization of the interval with $n + 1$ points,

$$x_i = a + ih, \; i = 0, 1, \ldots, n, \quad h = (b - a)/n, \tag{6}$$

and apply one of the above difference formulae at each point. Obviously, we cannot let h go to zero, that would be equivalent to n going to infinity. Hence an error occurs in the calculation of the derivative, this is the so-called *discretization error*. This is the type of error you have encountered earlier in the course.

5. Run the program erivemo.py . The program plots the relative error when calculating derivatives according to the formulae above as a function of the step length h. Which formula is most accurate? Reason about which type of error (rounding or discretization) is dominant for large h, and which error is dominant for very small h. Also note that the error first decreases for decreasing h, but then increases again. Think about why that is. The results will be discussed during the lecture.

4 Computer Arithmetic

There are a number of concepts associated with the way computers store numbers. It is important to understand these concepts in order to be able to interpret the solutions you get. This part of the lab introduces these concepts.

4.1 Machine epsilon

The relative error you get when numbers are stored in a computer (or when calculations are performed) depends on how the numbers are stored internally in the computer. There is a machine constant that specifies the relative size of this error. This is called machine epsilon and denoted as ε_M.

6. In Python, the machine epsilon is obtained through the command

```
import sys
print(sys.float_info.epsilon)
```

   What is the order of magnitude of machine epsilon? Relate machine epsilon to your investigation of the error in task 3 and to the inverse calculation in task 4.

In principle, rounding errors occur because a computer cannot store infinitely many decimal places. If it tried to do so, a number with infinite decimal expansion would fill the entire memory, such as π or $\frac{10}{3}$. This means that the computer can only represent certain numbers. These are detailed on the number line so that the relative distance between two numbers is approximately constant. This distance is precisely machine epsilon. For example, Python can represent the number 1. The next number that Python can represent is $1 + \varepsilon_M$. The numbers inbetween cannot be represented, so if we input such a number it will be rounded to either 1 or $1 + \varepsilon_M$.

Does the above mean that you cannot store numbers smaller than ε_M in Python? Can't one perform operations, for example addition, between numbers smaller than ε_M?

7. Check if the statement above is true by entering numbers less than ε_M, for example 10^{-25}, 10^{-50}, 10^{-200} and 10^{-400}. If it is too small to represent, the result should be zero.

8. Does it work to perform the additions $10^{-20} + 10^{-35}$, $10^{-20} + 10^{-36}$ and $10^{-20} + 10^{-37}$? What happens?

4.2 Overflow, underflow, inf and nan

Here you will investigate how small and large numbers can actually be stored (represented) in a computer. When you exceed the largest number, it is called overflow, and when you fall below the smallest normalized number, it is called underflow. In Python, the numbers corresponding to overflow and underflow are obtained through the commands `sys.float_info.max` and `sys.float_info.min`. Use these to solve the tasks below.

9. Investigate what happens if you try to store a larger number by adding a number to `sys.float_info.max`. Note that one must add a number so large that it affects the last decimal place (16th).

10. Is the number obtained with `sys.float_info.min` really the smallest? Investigate whether it is possible to represent smaller numbers. Note that `sys.float_info.min` returns the smallest *normalized* number that can be represented.

Sometimes longer calculations are interrupted and when you look at the result you see that `inf` or `nan` are included. Sometimes error messages related to computer arithmetic are also given. To understand what has happened, it is important to be able to interpret such results. Begin by importing the following:

```python
import math
import numpy as np
```

11. Calculate $\sqrt{-1}$ using the functions `math.sqrt` and `np.sqrt`. Is there a difference between how the math module and NumPy handle the operation?

12. Perform division by zero, for example 1/0. Also test the forbidden operation 0/0.

13. Infinity in Python is denoted `inf` and can be used as a variable through the calls `math.inf`. Investigate what `inf+inf`, `exp(inf)`, `cos(inf)` and other operations with `inf` give as their results. Try to understand, for example, why `inf+inf` and `inf-inf` give different answers.