A basic ingredient in modeling: gradients

Aksel Hiorth

University of Stavanger

Aug 26, 2024

Contents

1 Why are gradients important? 1

2 Continuous functions and finite representation: numerical errors 3

3 Taylor polynomial approximation 4

4 Calculating Numerical Derivatives of Functions 8
4.1 Roundoff Errors 11

5 Binary numbers 13
5.1 Floating point numbers and the IEEE 754-1985 standard 15

References 17

Index 18

1 Why are gradients important?

If you are going to walk up a mountain, it is not enough to know the height of the mountain, you also want to know how steep the mountain is. Even if the mountain is low, it can still be difficult to reach the top if it is very steep. The steepness is how much the height changes as function of time (if we walk at the same pace) or how much the height changes with horizontal distance. To be more precise, let's say we move from x_a to x_b, and the height increases from h_a to h_b, the steepness is

$$\frac{h_b - h_a}{x_b - x_a}. \tag{1}$$

If we climb a ladder, the horizontal movement is small ($x_b - x_a$ is small) and the increase in height is large, hence the steepness is large. If we walk a long a flat path we have no vertical movement and the steepness is zero ($h_a = h_b$). Mathematically, if we let x_b and x_a be infinitely close, the steepness is called a *gradient*, and we denote it by $\nabla h(x)$. Note also that the sign of the gradient tells something about the direction. If we climb up a ladder the height is increasing ($h_a < h_b$) and the gradient is positive, on the other hand if we are climbing down the height is decreasing ($h_a > h_b$) and the gradient is negative.

If we consider the height of a mountain in two dimensions, $h = h(x, y)$, the height is represented by the contour lines on a map. The spacing between the contour lines is the gradient, if the spacing between the contour lines is small the mountain side is steeper than if the spacing is larger.

Gradients vs derivatives.

If we are only considering a single variable, height as a function of time or position, x, we often denote the gradient (∇h), $h'(x)$ and call it the derivative of $h(x)$. In higher dimensions, e.g. $h(x, y)$, we use the term partial derivatives, because there are now two different variables we can vary e.g. latitude and longitude. The gradient is now a *vector*, $\nabla = [\partial h/\partial x, \partial h/\partial y]$. $\partial h/\partial x$ is the partial derivative of $h(x, y)$ with respect to x, i.e. we keep y constant and only differentiate with respect to x.

Another example where gradients are important is the flow of heat. Heat flows from hot to cold places, and the amount of heat is proportional to the temperature difference, i.e. a gradient in temperature. The flow of air is also from points of high pressure to low pressure, i.e. a gradient in pressure.

A primary task of a modeler is to predict something. If there are no gradients in a system, nothing will happen and there is no reason to model anything. Hence, an extremely important task when we model something is to analyze gradients carefully. If gradients are not represented correctly, the output of the simulation will introduce errors that can be so large that one cannot trust the results.

2 Continuous functions and finite representation: numerical errors

A computer can only deal with numbers. To simulate a physical system in a computer we have to divide space and time into finite pieces, and assign numbers to different parts of time and/or space.

Numerical errors.

Whenever we divide space and/or time into finite pieces, we introduce numerical errors. These errors tend to become smaller, but not always, when we use more pieces. The difference between the "true" answer and the answer obtained from a practical (numerical) calculation is called the *numerical error*.

When we divide space and time into finite pieces to represent them in a computer, a natural question is how many pieces do we need? Consider an almost trivial example, let say you want to visualize the function $f(x) = \sin x$. To do this we need to choose where, which values of x, we want to evaluate our function. To make an efficient program, we want to use as few points as possible but still capture the shape of the function. In figure 1, we have plotted $\sin x$ for various discretization steps (Δx, spacing between the points) in the interval $[-\pi, \pi]$.

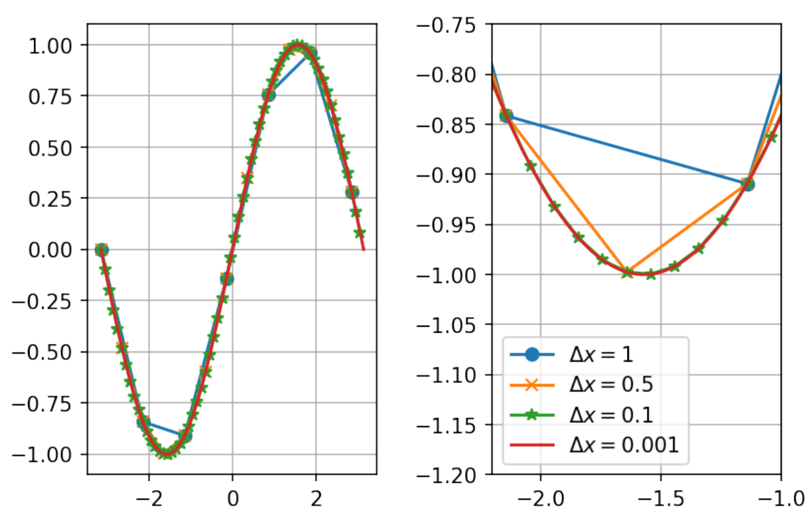

Figure 1: A plot of $\sin x$ for different spacing (Δx) of the x-values.

From the figure we see that in some areas only a couple of points are needed in order to represent the function well, and in some areas more points are needed. To state it more clearly; between $[-1, 1]$ a linear function (few points) approximate $\sin x$ well, whereas in the area where the gradient of the function changes more rapidly e.g. between $[-2, -1]$, we need the points to be more closely spaced to capture the behavior of the function.

What is a *good representation* of the function? We cannot rely on visual inspection every time, and most of the time we do not know the answer, so we would not know what to compare with. In the next section, we will show how the Taylor polynomial representation of a function is a natural starting point to answer this question.

3 Taylor polynomial approximation

How can we evaluate numerical errors if we do not know the true answer? There are at least two answers to this:

1. The pragmatic engineering approach is to do a simulation with a coarse grid, then refine the grid until the solution does not change much. This is perfectly fine *if you know that the numerical code is bug free*, because even if the simulation converges to a solution, we do not know if it is the *true solution*. In many cases this is not so. Therefore, even in well tested industrial codes, it is always good to test them on a simple test case where you know the exact solution.

2. Taylor's formula can be used to represent any continuous function with continuous gradients or most solutions to a mathematical model. Taylor's formula gives us an estimate of the numerical error introduced when we divide space and time into finite pieces.

There are many ways of representing a function, $f(x)$, like Fourier series and Legendre polynomials, but perhaps one of the most widely used is Taylor polynomials. Taylor series are perfect for computers, simply because they make possible to evaluate any function with a set of simple operations: *addition, subtraction, and multiplication.* Let us start off with the formal definition:

Taylor polynomial:

The Taylor polynomial, $P_n(x)$ of degree n of a function $f(x)$ at the point c is defined as:

$$P_n(x) = f(c) + f'(c)(x-c) + \frac{f''(c)}{2!}(x-c)^2 + \cdots + \frac{f^{(n)}(c)}{n!}(x-c)^n$$
$$= \sum_{k=0}^{n} \frac{f^{(k)}(c)}{k!}(x-c)^k. \tag{2}$$

Note that x can be anything, space, time, temperature, etc. If the series is around the point $c = 0$, the Taylor polynomial $P_n(x)$ is often called a Maclaurin polynomial. If the series converge (i.e. the higher order terms approach zero), then we can represent the function $f(x)$ with its corresponding Taylor series around the point $x = c$:

$$f(x) = f(c) + f'(c)(x-c) + \frac{f''(c)}{2!}(x-c)^2 + \cdots = \sum_{k=0}^{\infty} \frac{f^{(k)}}{k!}(x-c)^k. \tag{3}$$

The magic of Taylors formula.

Taylors formula, equation (3), states that if we know the function value and its gradients *at a single point* c, we can estimate the function everywhere *using only information from the single point* c. How can information from a single point be used to predict the behavior of the function everywhere? One way of thinking about this is to imagine an object moving in a constant gravitational field without air resistance. Newtons laws tell us that if we know the starting point e.g. ($x(0)$), the velocity ($v = dx/dt$), and the acceleration ($a = dv/dt = d^2x/dt^2$) in that point we can predict the trajectory of the object. This trajectory is exactly the first terms in Taylor's formula, $x(t) = x(0) + vt + at^2/2$.

An example of how Taylors formula works for a known function, can be seen in figure 2, where we show the first nine terms in the Maclaurin series for $\sin x$ (all even terms are zero).

Notice that close to $x = 0$ we only need one term, as we move further away from this point more and more terms need to be added. Thus, Taylors formula is only exact if we include an infinite number of terms. In practice, we only include a limited number of terms and truncate the series up to a given order. Luckily, Taylors formula include an estimate of the error we have when we truncate the series.

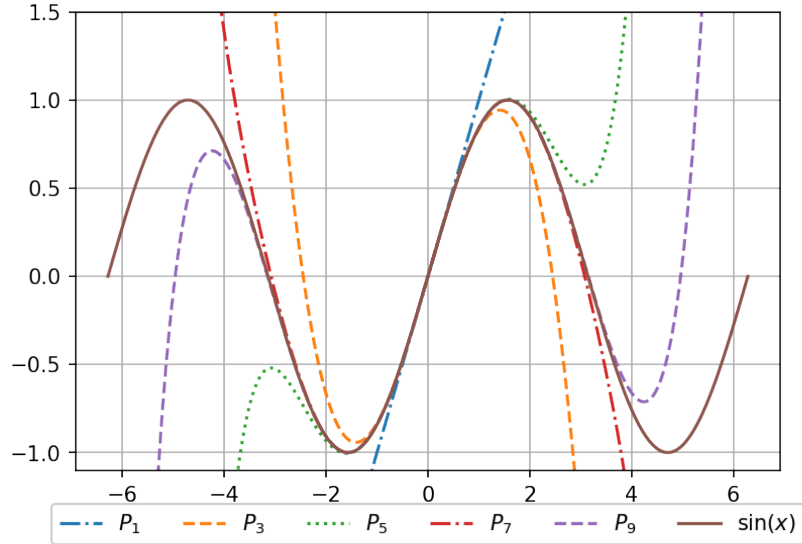

Figure 2: Nine first terms of the Maclaurin series of $\sin x$.

Truncation error in Taylors formula:

$$R_n(x) = f(x) - P_n(x) = \frac{f^{(n+1)}(\eta)}{(n+1)!}(x-c)^{n+1}$$
$$= \frac{1}{n!}\int_c^x (x-\tau)^n f^{(n+1)}(\tau)d\tau, \tag{4}$$

Notice that the mathematical formula is basically the next order term $(n+1)$ in the Taylor series, but with $f^{(n+1)}(c) \to f^{(n+1)}(\eta)$. η is an (unknown) value in the domain $[x, c]$.

Notice that if c is very far from x the truncation error increases. The fact that we do not know the value of η is usually not a problem, in many cases we just replace $f(\eta)$ with the maximum value it can take on the domain. Equation (4) gives us an direct estimate of discretization error.

Example: evaluate $\sin x$.

Whenever you do e.g. `np.sin(1)` in Python or an equivalent statement in another language, Python has to tell the computer how to evaluate $\sin x$

at $x = 1$. Write a Python code that calculates $\sin x$ up to a user specified accuracy.

Solution The Maclaurin series of $\sin x$ is:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{k=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}. \tag{5}$$

If we want to calculate $\sin x$ to a precision lower than a specified value we can do it as follows:

```
import numpy as np

# Sinus implementation using the Maclaurin Serie
# By setting a value for eps this value will be used
# if not provided
def my_sin(x,eps=1e-16):
    f = power = x
    x2 = x*x
    sign = 1
    i=0
    while(power>=eps):
        sign = - sign
        power *= x2/(2*i+2)/(2*i+3)
        f += sign*power
        i += 1
    print('No function evaluations: ', i)
    return f

x=0.8
eps = 1e-9
print(my_sin(x,eps), 'error = ', np.sin(x)-my_sin(x,eps))
```

This implementation needs some explanation:

- The error term is given in equation (4), and it is an even power in x. We do not know which η to use in equation (4), instead we simply say that the error in our estimate is smaller than the highest order term. Thus, we stop the evaluation if the highest order term in the series is lower than the uncertainty. Note that the final error has to be smaller as the higher order terms in any convergent series are smaller than the previous. Our estimate should then always be better than the specified accuracy.
- We evaluate the polynomials in the Taylor series by using the previous values to avoid too many multiplications within the loop, we do this by using the following identity:

$$\sin x = \sum_{k=0}^{\infty} (-1)^n t_n, \text{ where: } t_n \equiv \frac{x^{2n+1}}{(2n+1)!}, \text{ hence :}$$
$$t_{n+1} = \frac{x^{2(n+1)+1}}{(2(n+1)+1)!} = \frac{x^{2n+1}x^2}{(2n+1)!(2n+2)(2n+3)}$$
$$= t_n \frac{x^2}{(2n+2)(2n+3)} \tag{6}$$

4 Calculating Numerical Derivatives of Functions

As stated earlier many models are described by differential equations. Differential equations contain derivatives, and we need to tell the computer how to calculate these. By using a simple transformation, $x \to x+h$ and $c \to x$ (hence $x-c \to h$), Taylors formula in equation (3) can be written as:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots . \tag{7}$$

This is useful because this equation contains the derivative of $f(x)$ on the right hand side. To be even more explicit let's truncate the series to a certain power. Remember that you can always do this but we need to replace x with η in the last term we choose to keep

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(\eta)h^2 \tag{8}$$

where $\eta \in [x, x+h]$. Solving this equation with respect to $f'(x)$ gives:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}f''(\eta)h. \tag{9}$$

Note that if $h \to 0$, this expression is equal to the definition of the derivative. The beauty of equation (9) is that it contains an expression for the error *when h is not zero*. Equation (9) is usually called the *forward difference* . As you might guess, we can also choose to use the *backward difference* by simply replacing $h \to -h$. Is equation (9) the only formula for the derivative? The answer is no, and we are going to derive the formula for the *central difference* , by writing Taylors formula for $x+h$ and $x-h$ up to the third order:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{3!}f^{(3)}(\eta_1)h^3, \tag{10}$$
$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{3!}f^{(3)}(\eta_2)h^3. \tag{11}$$

where $\eta_1 \in [x, x+h]$, and $\eta_2 \in [x-h, x]$. Subtracting equation (10) and (11), we get the following expression for the central difference:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f^{(3)}(\eta), \tag{12}$$

where $\eta \in [x-h, x+h]$. Note that the error term in this equation is *one order higher* than in equation (9), meaning that it is expected to be more accurate. Figure 3 is a graphical representation of the finite difference approximations to the derivative.

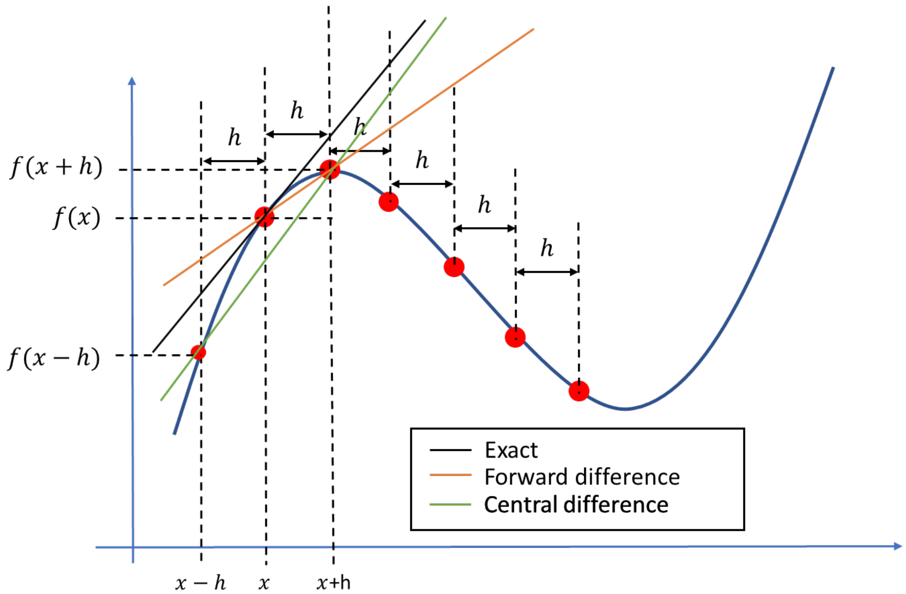

Figure 3: A graphical interpretation of the forward and central difference formula.

Higher order derivative. We are now in the position to derive a formula for the second order derivative. Instead of subtracting equation (10) and (11), we can add them. Then the first order derivative disappear and we are left with an expression for the second derivative:

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} - \frac{h^2}{12} f^{(4)}(\eta), \tag{13}$$

We can also calculate higher order derivatives by expanding about $x \pm h$ and $x \pm 2h$, adding one more term it follows from equation (12):

$$\begin{aligned} f(x+h) - f(x-h) &= 2hf'(x) + \frac{2}{3!}h^3 f^{(3)}(x) + \frac{2}{5!}h^5 f^{(5)}(\eta), \\ f(x+2h) - f(x-2h) &= 2(2h)f'(x) + \frac{2}{3!}(2h)^3 f^{(3)}(x) + \frac{2}{5!}h^5 f^{(5)}(\eta). \end{aligned} \tag{14}$$

It is now possible to find an expression for the third derivative:

$$f^{(3)}(x) = \frac{f(x-h) - f(x+h) - \frac{1}{2}f(x-2h) + \frac{1}{2}f(x+2h)}{h^3} + \frac{h^2}{4} f^{(5)}(\eta), \tag{15}$$

or a higher order first derivative:

$$f'(x) = \frac{2f(x+h) - 2f(x-h) - \frac{1}{4}f(x+2h) + \frac{1}{4}f(x-2h)}{3h} + \frac{h^4}{30}f^{(5)}(\eta). \quad (16)$$

Example: calculate the numerical derivative and second derivative of $\sin x$.

Choose a specific point, e.g. $x = 1$, and calculate the numerical error for various values of the step size h.

Solution: The derivative of $\sin x$ is $\cos x$, we can calculate the numerical derivatives using Python

```
def f(x):
    return np.sin(x)
def fd(f,x,h):
    """ f'(x) forward difference """
    return (f(x+h)-f(x))/h

def fc(f,x,h):
    """ f'(x) central difference """
    return 0.5*(f(x+h)-f(x-h))/h

def fdd(f,x,h):
    """ f''(x) second order derivative """
    return (f(x+h)+f(x-h)-2*f(x))/(h*h)

def fd3(f,x,h):
    """ f'''(x) third order derivative """
    return (2*f(x-h)-2*f(x+h)-f(x-2*h)+f(x+2*h))/(2*h*h*h)

def fd_4(f,x,h):
    """ f'(x) fourth order """
    return (8*f(x+h)-8*f(x-h)-f(x+2*h)+f(x-2*h))/(12*h)
x=1
h=np.logspace(-15,0.1,10)
plt.plot(h,np.abs(np.cos(x)-fd(f,x,h)), '-o',label='forward difference')
plt.plot(h,np.abs(np.cos(x)-fc(f,x,h)),'-x', label='central difference')
plt.plot(h,np.abs(np.cos(x)-fd_4(f,x,h)),'-*',label='derivative - fourth order')
plt.plot(h,np.abs(-np.sin(x)-fdd(f,x,h)),'-*',label='second derivative')
h=np.logspace(-7,0.1,10)
plt.plot(h,np.abs(-np.cos(x)-fd3(f,x,h)),'-*',label='third derivative')

plt.grid()
plt.legend()
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Step size $h$')
plt.ylabel('Numerical error')
```

Figure 4 shows the result of the code above.

There are several important lessons from figure 4:

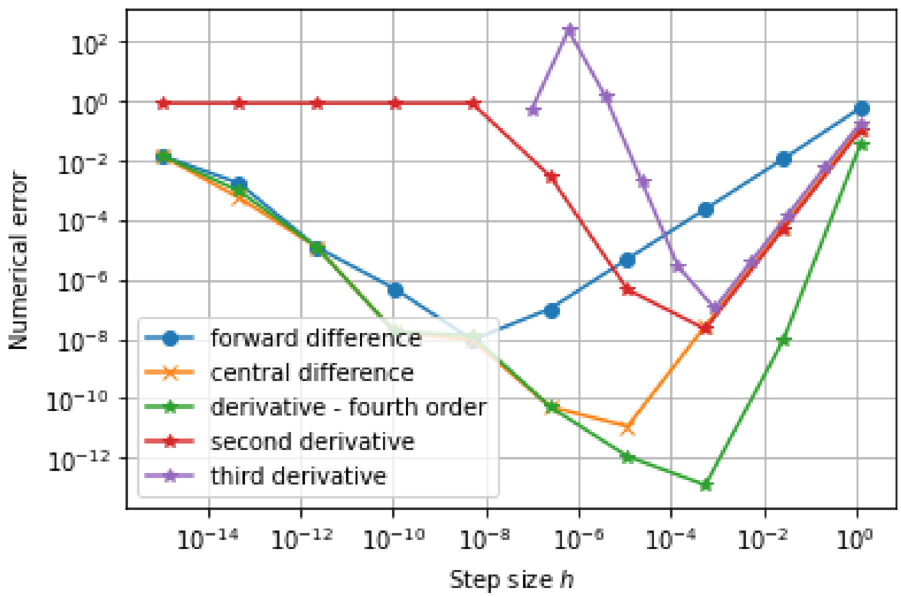

Figure 4: Numerical error of derivatives of $\sin x$ for various step sizes.

1. When the step size is high and decreasing (from right to left in the figure), we clearly see that the numerical error *decreases*.
2. The numerical error scales as expected from right to left. The forward difference formula scales as h, i.e. decreasing the step size by 10 reduces the numerical error by 10. The central difference and second order derivative formula scales as h^2, reducing the step size by 10 reduces the numerical error by 100
3. At a certain step size the numerical error starts to *increase*. For the forward difference formula this happens at 10^{-8}.

The numerical error has a minimum, *it does not continue to decrease when h decreases*. The explanation for this behavior are two competing errors: *truncation errors* and *roundoff errors*. The truncation errors have already been discussed in great detail, in the next section we will look at roundoff errors.

4.1 Roundoff Errors

In a computer a floating point number, x, is represented as:

$$x = \pm q 2^m. \tag{17}$$

This is very similar to our usual scientific notation where we represent large (or small numbers) as $\pm qEm = \pm q10^m$. The processor in a computer handles a chunk of bits at one time, this chunk of bits is usually called a *word*. The number of bits (or bytes which almost always means a group of eight bits) in a word is handled as a unit by a processor. Most modern computers use 64-bits (8 bytes) processors. We are not going into all the details, the most important message is that the units handled by the processor are *finite*. Thus we cannot, in general, store numbers in a computer with infinite accuracy.

Machine Precision.

In the next section we explain why the machine precision has this value, but if you just accept this for a moment, we can demonstrate why the machine precision is important and you need to care about it. First, just to convince you that the machine precision has the value of 2^{-52}, you can do the following in Python:

```
print(1+2**-52) # prints a value larger than 1
print(1+2**-53) # prints 1.0
```

Next, consider the simple calculation

```
a=0.1+0.2
b=0.3
print(a==b) # gives False
```

Why is `a==b` false if this calculation involves only numbers with one decimal? The reason is that the computer uses the binary system, and in the binary system there is no way of representing 0.2 and 0.3 with a finite number of bits. As an example 0.2 in the binary system is:

$$0.2_{10} = 0.0011001100\ldots_2 \, (= 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + \cdots) \tag{18}$$

Note that we use the subscript $_{10}$ and $_2$ to represent the decimal and binary system respectively. Thus in the computer 0.2 will be represented as $0.1999\ldots$ and when we add 0.1 we will get a number really close to 0.3 but not equal to 0.3. Some floats have an exact binary representation e.g. $0.125_{10} = 2^{-3}_{10} = 0.001_2$. Thus the following code will produce the expected result

```
a=0.125+0.25
b=0.375
print(a==b) # gives True
```

Comparing two floats.

Whenever you want to compare if two floats, a and b, are equal in a computer program, you should never do $a == b$ because of roundoff errors. Rather you should choose a variant of $|a - b| < \epsilon$, where you check if the numbers are *close enough*. In practice, you may also want to normalize the values and do $|1 - b/a| < \epsilon$.

Roundoff errors can play a big role in calculations. This is particularly apparent when subtracting two numbers of similar magnitude as illustrated in the following code:

```
h=2**-53
a=1+h
b=1-h
print((a-b)/h) # analytical result is 2
```

The calculation above is very similar to the calculation performed when evaluating derivatives, and if you run the code you will see that Python does not give the expected value of 2.

Choosing the right step size.

A step size that is too low will give higher numerical error because roundoff errors dominate the numerical error.

There is a simple trick you can use sometimes to avoid roundoff errors [1]. In practice, we can never get rid of roundoff errors in the calculation $f(x+h)$, but since we can choose the step size h, we can choose values such that x and $x+h$ differ by an exact binary number

```
x=1
h=0.0002
temp = x+h
h=temp-x
print(h) # improved value of h with exact binary representation
```

In the next sections we will show why $\epsilon_M = 2^{-52}$, and why a finite word size implies a maximum and minimum number.

5 Binary numbers

Binary numbers are used in computers because processors are made of billions of transistors, the end states of a transistor are off or on, representing a 0 or 1 in the binary system. Assume, for simplicity, that we have a processor that uses a word size of 4 bits (instead of 64 bits). How many *unsigned* (positive) integers can we represent in such processor? Lets write down all the possible combinations of ones and zeros and also do the translation from the base 2 numerical system to the base 10 numerical system:

$$\begin{array}{ccccc}
0 & 0 & 0 & 0 = 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 = 0 \\
0 & 0 & 0 & 1 = 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 1 \\
0 & 0 & 1 & 0 = 0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 = 2 \\
0 & 0 & 1 & 1 = 0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 3 \\
0 & 1 & 0 & 0 = 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 = 4 \\
0 & 1 & 0 & 1 = 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 5 \\
0 & 1 & 1 & 0 = 0\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 = 6 \\
0 & 1 & 1 & 1 = 0\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 7 \\
1 & 0 & 0 & 0 = 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 = 8 \\
1 & 0 & 0 & 1 = 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 9 \\
1 & 0 & 1 & 0 = 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 = 10 \\
1 & 0 & 1 & 1 = 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 11 \\
1 & 1 & 0 & 0 = 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 = 12 \\
1 & 1 & 0 & 1 = 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 13 \\
1 & 1 & 1 & 0 = 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 = 14 \\
1 & 1 & 1 & 1 = 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 15
\end{array}. \tag{19}$$

Hence, with a 4 bits word size, we can represent $2^4 = 16$ integers. The largest number is $2^4 - 1 = 15$, and the smallest is zero. What about negative numbers? If we use a 4 bits word size, there are still $2^4 = 16$ numbers, but they are distributed differently. The common way to do this is to reserve the first bit to be a *sign* bit, a "0" is positive and "1" is negative, i.e. $(-1)^0 = 1$, and $(-1)^1 = -1$. Replacing the first bit with a sign bit in equation (19), we get the following sequence of numbers 0,1,2,3,4,5,6,7,-0,-1,-2,-3,-4,-5,-6,-7. The "-0" might seem strange but is used in the computer to extend the real number line $1/0 = \infty$, whereas $1/-0 = -\infty$. In general when there are m bits, we have a total of 2^m numbers. If we include negative numbers, we can choose to have $2^{m-1} - 1$ negative and $2^{m-1} - 1$ positive numbers, and negative and positive zero, i.e. $2^{m-1} - 1 + 2^{m-1} - 1 + 1 + 1 = 2^m$.

What about real numbers? As stated earlier we use the scientific notation as in equation (17), but still the scientific notation might have a real number in front, e.g. $1.25 \cdot 10^{-3}$. To represent the number 1.25 in binary format we use a decimal separator, just as with base 10. In this case 1.25 is 1.01 in binary format

$$1.01 = 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1 + 0 + 0.25 = 1.25. \tag{20}$$

The scientific notation is commonly referred to as a *floating point representation*. The term "floating point" is used because the decimal point is not in the same place, in contrast to the fixed point where the decimal point is always in the same place. To store the number 1e-8=0.00000001 in floating point format, we only need to store 1 and -8 (and possibly the sign), whereas in fixed point format we need to store all 9 numbers. In equation (19), we need to use one bit to store the sign, leaving (in the case of 4 bits word size) three bits to be distributed among the *mantissa*, q, and the exponent, m. It is not given how many bits should be used for the mantissa and the exponent. Thus there are choices to be made, and all modern processors use the same standard, the IEEE Standard 754-1985.

5.1 Floating point numbers and the IEEE 754-1985 standard

A 64 bits word size is commonly referred to as *double precision*, whereas a 32 bits word size is termed *single precision*. In the following, we will consider a 64 bits word size. We would like to know: what is the roundoff error, what is the largest number, and what is the smallest number that can be represented in the computer? Almost all floating point numbers are represented in *normalized* form. In normalized form, the mantissa is written as $M = 1.F$, and it is only F that is stored, F is termed the *fraction*. We will return to the special case of some of the unnormalized numbers later. In the IEEE standard one bit is reserved for the sign, 52 for the fraction (F), and 11 for the exponent (m), see figure 5 for an illustration.

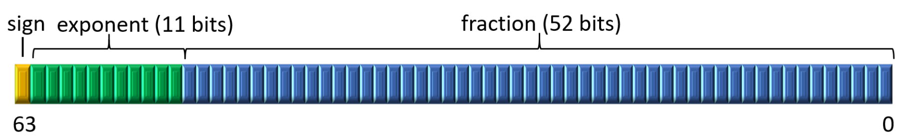

Figure 5: Representation of a 64 bits floating point number according to the IEEE 754-1985 standard. For a 32 bits floating point number, 8 bits are reserved for the exponent and 23 for the fraction.

The exponent must be positive to represent numbers with absolute value larger than one, and negative to represent numbers with absolute value less than one. To make this more explicit the simple formula in equation (17) is rewritten as:

$$\pm q 2^{E-e}. \tag{21}$$

The number e is called the *bias* and has a fixed value. For 64 bits it is $2^{11-1}-1 = 1023$ (32-bits: $e = 2^{8-1} - 1 = 127$). The number E is represented by 11 bits and can thus take values from 0 to $2^{11} - 1 = 2047$. If we have an exponent of e.g. -3, the computer adds 1023 to that number and store the number 1020. Two numbers are special numbers and they are reserved to represent infinity and zero, $E = 0$ and $E = 2047$. Thus *the largest and smallest possible numerical value of the exponent is: 2046-1023=1023, and 1-1023=-1022, respectively.* The fraction of a normalized floating point number takes on values from $1.000\ldots00$

to 1.111...11. Thus the lowest normalized number is:

$$
\begin{aligned}
1.000 + \text{ (49 more zeros)} \cdot 2^{-1022} &= 2^0 \cdot 2^{-1022} \\
&= 2.2250738585072014 \cdot 10^{-308}. \qquad (22)
\end{aligned}
$$

It is possible to represent smaller numbers than $2.22 \cdot 10^{-308}$, by allowing *unnormalized* values. If the exponent is -1022, then the mantissa can take on values from $1.000\ldots00$ to $0.000\ldots01$, but then accuracy is lost. So the smallest possible number is $2^{-52} \cdot 2^-1022 \simeq 4.94 \cdot 10^{-324}$. The highest normalized number is

$$
1.111+ \text{ (49 more ones)} \cdot 2^{1023} = (2^0 + 2^{-1} + 2^{-2} + \cdots + 2^{-52}) \cdot 2^{1023}
$$
$$
= (2 - 2^{-52}) \cdot 2^{1023} = 1.7976931348623157 \cdot 10^{308}. \qquad (23)
$$

If you enter `print(1.8*10**(308))` in Python, the answer will be `Inf`. If you enter `print(2*10**(308))`, Python will (normally) give an answer. This is because the number $1.8 \cdot 10^{308}$ is floating point number, whereas $2 \cdot 10^{308}$ is an *integer*, and Python does something clever when it comes to representing integers. Python has a third numeric type called long int, which can use the available memory to represent an integer.

What about the machine precision? The machine precision, ϵ_M, is the *smallest possible number that can be added to one, and get a number larger than one*, i.e. $1 + \epsilon_M > 1$. The smallest possible value of the mantissa is $0.000\ldots01 = 2^{-52}$, thus the lowest number must be of the form $2^{-52} \cdot 2^m$. If the exponent , m, is lower than 0 then when we add this number to 1, we will only get 1. Thus the machine precision is $\epsilon_M = 2^{-52} = 2.22 \cdot 10^{-16}$ (for 32 bits $2^{-23} = 1.19 \cdot 10^{-7}$). In practical terms this means that e.g. the value of π is $3.14159265358979323846264338\ldots$, but in Python it can only be represented by 16 digits: 3.141592653589793.

Roundoff error and truncation error in numerical derivatives.

Roundoff Errors.

All numerical floating point operations introduce roundoff errors at each step in the calculation due to finite word size. These errors accumulate in long simulations and introduce random errors in the final results. After N operations, the error is at least $\sqrt{N}\epsilon_M$ (the square root is a random walk estimate, and we assume that the errors are randomly distributed). The roundoff errors can be much, much higher when numbers of equal magnitude are subtracted. You might be so unlucky that after one operation the answer is completely dominated by roundoff errors.

The roundoff error when we represent a floating point number x in the machine will be of the order $x/10^{16}$ (*not* 10^{-16}). In general, when we evaluate a function, the error will be of the order $\epsilon|f(x)|$, where $\epsilon \sim 10^{-16}$. Thus equation (9) is modified in the following way when we take into account the roundoff errors:

$$f'(x) = \frac{f(x+h) - f(x)}{h} \pm \frac{2\epsilon|f(x)|}{h} - \frac{h}{2}f''(\eta), \tag{24}$$

we do not know the sign of the roundoff error, so the total error R_2 is:

$$R_2 = \frac{2\epsilon|f(x)|}{h} + \frac{h}{2}|f''(\eta)|. \tag{25}$$

We have put absolute values around the function and its derivative to get the maximal error, it might be the case that the roundoff error cancel part of the truncation error. However, the roundoff error is random in nature and will change from machine to machine, and each time we run the program. Note that the roundoff error increases when h decreases, and the approximation error decreases when h decreases. This is exactly what we saw in figure 4. We can find the best step size, by differentiating R_2 and put it equal to zero:

$$\begin{aligned} \frac{dR_2}{dh} &= -\frac{2\epsilon|f(x)|}{h^2} + \frac{1}{2}f''(\eta) = 0 \\ h &= 2\sqrt{\epsilon\left|\frac{f(x)}{f''(\eta)}\right|} \simeq 2 \cdot 10^{-8}, \end{aligned} \tag{26}$$

In the last equation we have assumed that $f(x)$ and its derivative is 1. This step size corresponds to an error of order $R_2 \sim 10^{-8}$. Inspecting figure 4 we see that the minimum is located at $h \sim 10^{-8}$.

We can perform a similar error analysis as we did before, and then we find for equation (12) and (13) that the total numerical error is:

$$R_3 = \frac{\epsilon|f(x)|}{h} + \frac{h^2}{6}f^{(3)}(\eta), \tag{27}$$

$$R_4 = \frac{4\epsilon|f(x)|}{h^2} + \frac{h^2}{12}f^{(4)}(\eta), \tag{28}$$

respectively. Differentiating these two equations with respect to h, and setting the equations equal to zero, we find an optimal step size of $h \sim 10^{-5}$ for equation (27), which gives an error of $R_2 \sim 10^{-16}/10^{-5} + (10^{-5})^2/6 \simeq 10^{-10}$, and $h \sim 10^{-4}$ for equation (28), which gives an error of $R_4 \sim 4 \cdot 10^{-16}/(10^{-4})^2 + (10^{-4})^2/12 \simeq 10^{-8}$. Note that we get the surprising result for the first order derivative in equation (12), that a higher step size gives a more accurate result.

References

[1] Brian P. Flannery, William H. Press, Saul A. Teukolsky, and William Vetterling. Numerical recipes in c. *Press Syndicate of the University of Cambridge, New York*, 24(78):36, 1992.

Index

backward difference, 8

central difference, 8

forward difference, 8

IEEE 754-1985 standard, 15

machine precision, 12
Maclaurin series, 7

numerical error, 3

roundoff errors, 16
roundoff erros, 11

Taylor polynomial, 4
Taylor polynomial, error term, 6
truncation error, 5