Numerical Derivation

Aksel Hiorth, the National IOR Centre & Institute for Energy Resources,

University of Stavanger

Oct 27, 2020

Contents

The mathematics introduced in this chapter is absolutely essential in order to understand the development of numerical algorithms. We strongly advice you to study it carefully, implement python scripts and investigate the results, reproduce the analytical derivations and compare with the numerical solutions.

1 Truncation Errors

The solution to a physical model is usually a function. The function could describe the temperature evolution of the earth, it could be growth of cancer cells, the water pressure in an oil reservoir, the list is endless. If we can solve the model analytically, the answer is given in terms of a continuous function. Most of the models cannot be solved analytically, then we have to rely on computers to help us. The computer does not have any concept of continuous functions, a function is always evaluated at some point in space and/or time. Assume for simplicity that the solution to our problem is $f(x) = \sin x$, and we would like to

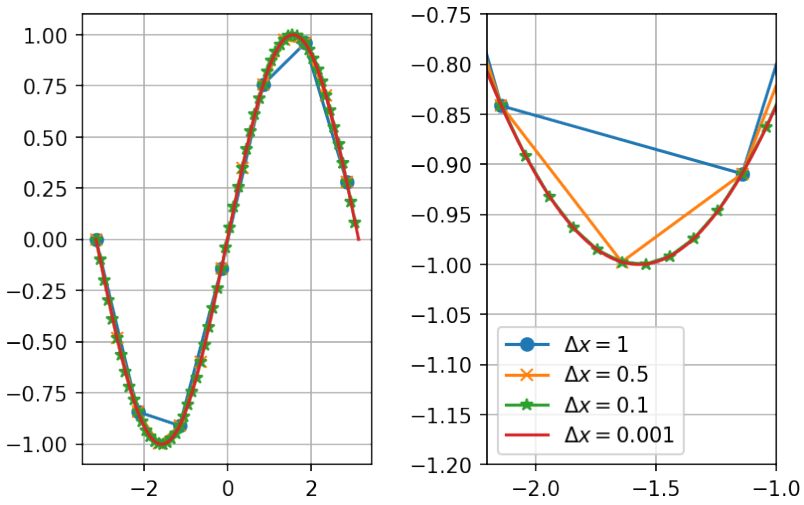

Figure 1: A plot of $\sin x$ for different spacing of the x-values.

visualize the solution. How many points do we need in our plot to approximate the true function? In figure 1, there is a plot of $\sin x$ on the interval $[-\pi, \pi]$.

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 derivative of the function changes e.g. in $[-2, -1]$, we need the points to be more closely spaced to capture the behavior of the true function.

Discretization and Truncation Error.

To represent a function of space and/or time in a computer, the function needs to be discretized. When a function is discretized it leads to discretization errors. The difference between the "true" answer and the answer obtained from a practical (numerical) calculation is called the *truncation error*.

Why do we care about the number of points? In many cases the function we would like to evaluate can take a very long time to evaluate. Sometimes simulation time is not an issue, then we can use a large number of function evaluations. However, in many applications simulation time *is an issue*, and it would be good to know where the points needs to be closely spaced, and where we can manage with only a few points.

What is a *good representation* representation of the true function? We cannot rely on visual inspection. In the next section we will show how Taylor polynomial representation of a function is a natural starting point to answer this question.

2 Taylor Polynomial Approximation

There are many ways of representing a function, but perhaps one of the most widely used is Taylor polynomials. Taylor series are the basis for solving ordinary and differential equations, simply because it makes it possible to evaluate any function with a set of limited operations: *addition, subtraction, and multiplication.* The Taylor polynomial, $P_n(x)$ of degree n of a function $f(x)$ at the point c is defined as:

Taylor polynomial:

$$\begin{aligned} P_n(x) &= f(c) + f'(c)(x-c) + \frac{f''(c)}{2!}(x-c)^2 + \cdots + \frac{f^{(n)}(c)}{n!}(x-c)^n \\ &= \sum_{k=0}^{n} \frac{f^{(k)}(c)}{k!}(x-c)^k. \end{aligned} \tag{1}$$

If the series is around the point $c = 0$, the Taylor polynomial $P_n(x)$ is often called a Maclaurin polynomial, more examples can be found here[1]. If the series converge (i.e. that the higher order terms approach zero), then we can represent the function $f(x)$ with its corresponding Taylor series around the point $x = c$:

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

The Maclaurin series of $\sin x$ is:

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

In figure 2, we show the first nine terms in the Maclaurin series for $\sin x$ (all even terms are zero).

Note that we get a decent representation of $\sin x$ on the domain, by *only knowing the function and its derivative in a single point.* The error term in Taylors formula, when we represent a function with a finite number of polynomial elements is given by:

[1] `https://en.wikipedia.org/wiki/Taylor_series`

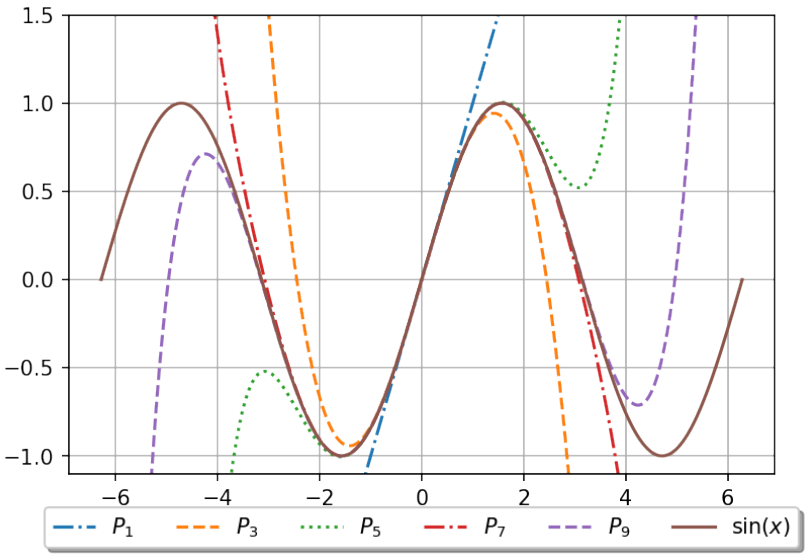

Figure 2: Up to ninth order in the Maclaurin series of $\sin x$.

Error term in Taylors formula:

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

for some η in the domain $[x, c]$.

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 a even power in x. We do not which η to use in equation (4), thus we use a trick and simply say that the error term 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. Thus, in practice we add the error term to the function evaluation, our estimate will always be better than the specified accuracy.
- We evaluate the polynomials in the Taylor series by using the previous values too avoid too many multiplications within the loop, we do this by using the following identity:

$$\begin{aligned}
\sin x &= \sum_{k=0}^{\infty}(-1)^n t_n, \text{ where: } t_n \equiv \frac{x^{2n+1}}{(2n+1)!}, \text{ hence :} \\
t_{n+1} &= \frac{x^{2(n+1)+1}}{(2(n+1)+1)!} = \frac{x^{2n+1}x^2}{(2n+1)!(2n+2)(2n+3)} \\
&= t_n \frac{x^2}{(2n+2)(2n+3)} \qquad (5)
\end{aligned}$$

2.1 Evaluation of polynomials

How to evaluate a polynomial of the type: $p_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$? We already saw a hint in the previous section that it can be done in different ways. One way is simply to do:

```
pol = a[0]
for i in range(1,n+1):
    pol = pol + a[i]*x**i
```

Note that there are n additions, whereas there are $1 + 2 + 3 + \cdots + n = n(n+1)/2$ multiplications for all the iterations. Instead of evaluating the powers all over in each loop, we can use the previous calculation to save the number of multiplications:

```
pol = a[0] + a[1]*x
```

```
power = x
for i in range(2,n+1):
    power  = power*x
    pol    = pol + a[i]*power
```

In this case there are still n additions, but now there are $2n-1$ multiplications. For $n = 15$, this amounts to 120 for the first, and 29 for the second method. Polynomials can also be evaluated using *nested multiplication*:

$$\begin{aligned} p_1 &= a_0 + a_1 x \\ p_2 &= a_0 + a_1 x + a_2 x^2 = a_0 + x(a_1 + a_2 x) \\ p_3 &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + x(a_1 + x(a_2 + a_3 x)) \\ &\vdots \end{aligned} \tag{6}$$

and so on. This can be implemented as:

```
pol = a[n]
for i in range(n-1,1,-1):
    pol  = a[i] + pol*x
```

In this case we only have n multiplications. So if you know beforehand exactly how many terms is needed to calculate the series, this method would be the preferred method, and is implemented in NumPy as `polyval`[2].

3 Calculating Numerical Derivatives of Functions

indexforward difference

The derivative of a function can be calculated using the definition from calculus:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \simeq \frac{f(x+h) - f(x)}{h}. \tag{7}$$

Not that h can be both positive and negative, if h is positive equation (7) is termed *forward difference*, because we use the function value on the right ($f(x + |h|)$). If on the other hand h is negative equation (7) is termed *backward difference*, because we use the value to the left ($f(x - |h|)$). ($|h|$ is the absolute value of h). In the computer we cannot take the limit, $h \to 0$, a natural question is then: What value to use for h? In figure 3, we have evaluated the numerical derivative of $\sin x$, using the formula in equation (7) for different step sizes h.

We clearly see that the error depends on the step size, but there is a minimum; choosing a step size too large give a poor estimate and choosing a too low step size give an even worse estimate. The explanation for this behavior is two competing effects: *mathematical approximation* and *round off errors*. Let us

[2] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyval.html#r138ee7027ddf-1`

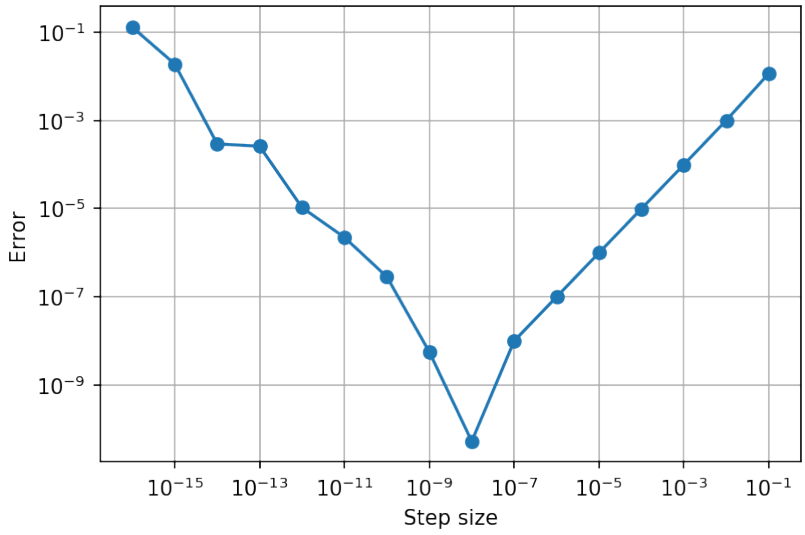

Figure 3: Error in the numerical derivative of $\sin x$ at $x = 0.2$ for different step size.

consider approximation or truncation error first. By using the Taylor expansion in equation (2) and expand about x and the error formula (4), we get:

$$f(x+h) = f(x) + f'(x)h + \frac{h^2}{2}f''(\eta) \text{ , hence:}$$
$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(\eta), \tag{8}$$

for some η in $[x, x+h]$. Thus the error to our approximation is $hf''(\eta)/2$, if we reduce the step size by a factor of 10 the error is reduced by a factor of 10. Inspecting the graph, we clearly see that this is correct as the step size decreases from 10^{-1} to 10^{-8}. When the step size decreases more, there is an increase in the error. This is due to round off errors, and can be understood by looking into how numbers are stored in a computer.

3.1 Big $\mathcal{O}$ notation

example[3]

3.2 Round off Errors

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

$$x = \pm q 2^m. \tag{9}$$

[3] `https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/`

This is very similar to our usual scientific notation where we represents 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 bit is usually termed *word*. The number of bits (or byte which almost always means a group of eight bits) in a word is handled as a unit by a processor. Most modern computers uses 64-bits (8 bytes) processors. We are not going too much into all the details, the most important message is that the units handled by the processor are *finite*. Thus we cannot store numbers in a computer with infinite accuracy, e.g. $1./3. = 0.3333\ldots$ cannot be stored accurately in a computer. In many cases this is not a problem, but in some cases, as with evaluating numerical derivatives, it is. The reason is that we subtract two numbers that are almost equal, and because of this the round off error might dominate the answer. How large is the round off error? This depends on exactly how numbers are stored in the machine. To make the following a bit more understandable, we will recap some binary number basics.

Binary numbers. Binary numbers are used in computers because processors are made of billions of transistors, the end states of a transistor is 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 this processor? Lets write down all the possible combinations, of ones and zeros and also do the translation from base 2 numerical system to base 10 numerical system:

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

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 still keep to a 4 bits word size, there are still $2^4 = 16$ numbers, but we distribute them differently. The common way to do it 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 (10), 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, negative zero 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 (9), 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

$$1.01 = 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1 + 0 + 0.25 = 1.25. \tag{11}$$

The scientific notation is commonly referred to as *floating point representation*. The term "floating point" is used because the decimal point is not in the same place, in contrast to 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 (10) we need to spend 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 uses the same standard, the IEEE Standard 754-1985[4].

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 round off error, what is the largest number that can be represented in the computer, and what is the smallest number? 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 4 for an illustration.

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 (9) is rewritten:

$$\pm q 2^{E-e}. \tag{12}$$

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 on values from 0 to $2^1 1 - 1 = 2047$. If we have an exponent of e.g.

[4] `https://standards.ieee.org/standard/754-1985.html`

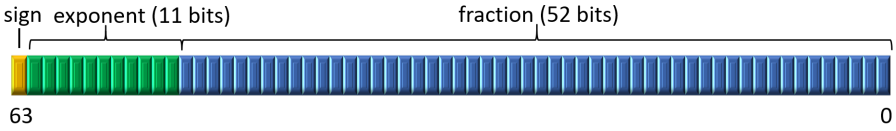

Figure 4: Representation of a 64 bits floating point number according to the IEEE 754-1985 standard. For a 32 bits floating point number, 8, is reserved for the exponent and 23 for the fraction.

-3, the computer adds 1023 to that number and store the number 1020. Two numbers are special numbers and 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 $1.111\ldots11$. Thus the lowest normalized number is

$$1.000 + \text{ (49 more zeros)} \cdot 2^{-1022} = 2^0 \cdot 2^{-1022} = 2.2250738585072014 \cdot 10^{-308}. \quad (13)$$

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

$$1.111+ \text{ (49 more ones)} \cdot 2^{1023} = 2^0 + 2^{-1} + 2^{-2} + \cdots + 2^{-52} = (2 - 2^{-52}) \cdot 2^{1023} = 1.7976931348623157 \cdot 10^{308}. \quad (14)$$

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.

There is one more important thing to mention, the round off error. The round off error is related to 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. In principle it does not sound so bad to have an answer accurate to 16 digits, and it is much better than most experimental results. So what is the problem? Consider the following example

```
h=1e-16
x = 2.1 + h
y = 2.1 - h
print((x-y)/h)
```

we expect to get the answer 2, but instead we get zero. By changing h to a higher value, the answer will get closer to 2.

Round Off Errors.

All numerical floating point operations introduces round off 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 round off 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 round off errors.

Round off error and truncation error in numerical derivatives. Armed with this knowledge of round off errors, we can continue to analyze the result in figure 3. The round off 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 (8) is modified in the following way when we take into account the round off errors:

$$f'(x) = \frac{f(x+h) - f(x)}{h} \pm \frac{2\epsilon|f(x)|}{h} - \frac{h}{2}f''(\eta), \tag{15}$$

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

$$R_2 = \frac{2\epsilon|f(x)|}{h} + \frac{h}{2}|f''(\eta)|. \tag{16}$$

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

$$\begin{aligned} \frac{dR_2}{dh} &= -\frac{2\epsilon|f(x)|}{h^2} + \frac{1}{2}f''(\eta) = 0 \\ h &= 2\sqrt{\epsilon\left|\frac{f(x)}{f''(\eta)}\right|} \simeq 2 \cdot 10^{-8}, \end{aligned} \tag{17}$$

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 the result in figure 3. we see that the minimum is located at $h \sim 10^{-8}$.

4 Higher Order Derivatives

There are more ways to calculate the derivative of a function, than the formula given in equation (8). Different formulas can be derived by using Taylors formula in (2), usually one expands about $x \pm h$:

$$f(x+h) = f(x) + f'(x)h + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(4)}(x) + \cdots$$
$$f(x-h) = f(x) - f'(x)h + \frac{h^2}{2}f''(x) - \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(3)}(x) - \cdots . \quad (18)$$

If we add these two equations, we get an expression for the second derivative, because the first derivative cancels out. But we also observe that if we subtract these two equations we get an expression for the first derivative that is accurate to a higher order than the formula in equation (7), hence:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f^{(3)}(\eta), \quad (19)$$
$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} - \frac{h^2}{12}f^{(4)}(\eta), \quad (20)$$

for some η in $[x, x+h]$. In figure 5, we have plotted equation (8), (19), and (20) for different step sizes. The derivative in equation (19), gives a higher accuracy than equation (8) for a larger step size, as can bee seen in figure 5.

We can perform a similar error analysis as we did before, and then we find for equation (19) and (20) that the total numerical error is:

$$R_3 = \frac{\epsilon|f(x)|}{h} + \frac{h^2}{6}f^{(3)}(\eta), \quad (21)$$
$$R_4 = \frac{4\epsilon|f(x)|}{h^2} + \frac{h^2}{12}f^{(4)}(\eta), \quad (22)$$

respectively. Differentiating these two equations with respect to h, and set the equations equal to zero, we find an optimal step size of $h \sim 10^{-5}$ for equation (21), 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 (22), 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 (19), that a higher step size gives a more accurate result.

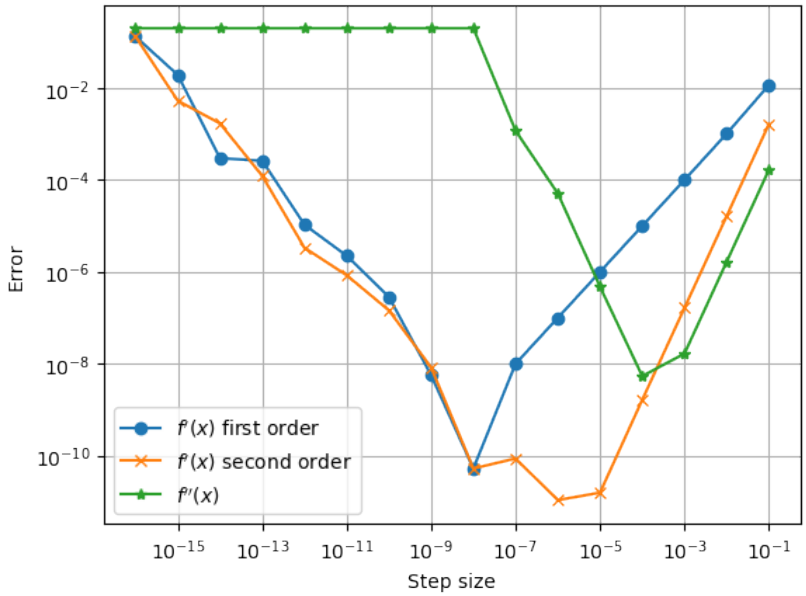

Figure 5: Error in the numerical derivative and second derivative of $\sin x$ at $x = 0.2$ for different step size.