Numerical Derivation

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

University of Stavanger

Sep 5, 2019

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 Numerical derivatives

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

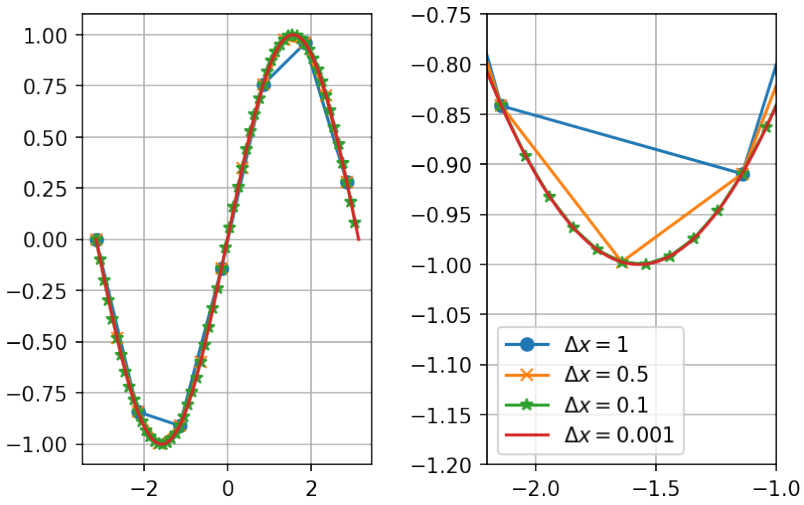

Figure 1: A plot of $\sin x$ for different spacing of the x-values.

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.

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.

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:

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

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 $x = c$:

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

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:

Error term in Taylors formula:

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

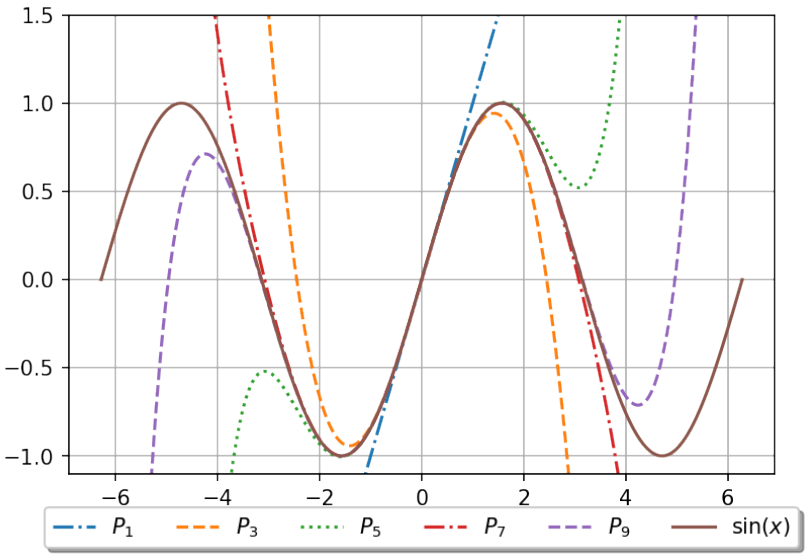

Figure 2: Up to ninth order in the Maclaurin series of $\sin x$.

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

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:

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

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 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 consider approximation or truncation error first. By using the Taylor expansion

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

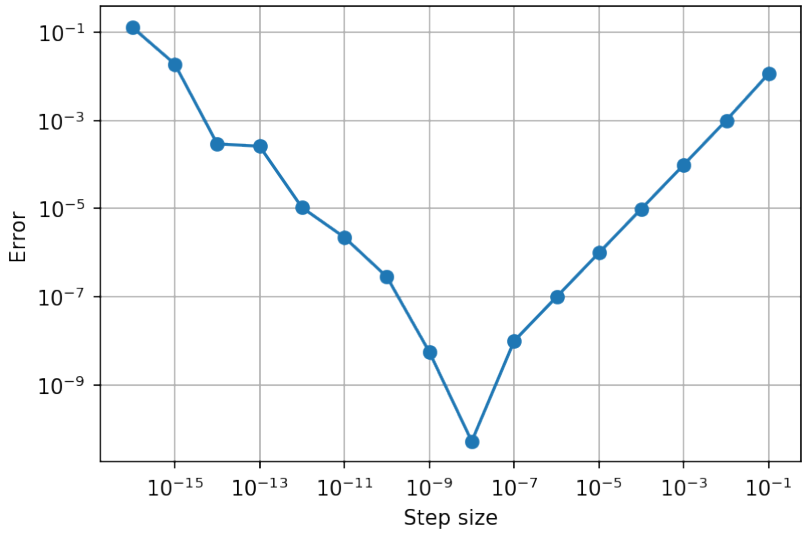

Figure 3: Error in the numerical derivative of $\sin x$ at $x = 0.2$ for different step size.

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/`

Most computers are 64-bits, then one bit is reserved for the sign, 52 for the fraction (q) and 11 for the exponent (m) (for a graphic illustration see Wikipedia[4]). what is the largest *floating point* number the computer can represent? Since m contains 11 bits, m can have the maximal value $m = 2^{11} = 1024$, and then the largest value is close to $2^{1024} \simeq 10^{308}$. If you enter `print(10.1*10**(308))` in Python the answer will be `Inf`. If you enter `print(10*10**(308))`, Python will give an answer. This is because the number $10.1 \cdot 10^{308}$ is floating point number, whereas 10^{309} 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.

10^{308} is the largest number, but what is the highest precision we can use, or how many decimal places can we use for a floating point number? Since there are 52 bits for the fraction, there are $1/2^{52} \simeq 10^{-16}$ decimal places. As an example 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? One problem that you should be aware of is that round off errors can be a serious problem when we subtract two numbers that are very close to one another. If we implement the following program in Python:

```
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.

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:

$$f'(x) = \frac{f(x+h) - f(x)}{h} \pm \frac{2\epsilon|f(x)|}{h} - \frac{h}{2}f''(\eta), \tag{10}$$

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

$$R_2 = \frac{2\epsilon|f(x)|}{h} + \frac{h}{2}|f''(\eta)|. \tag{11}$$

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

[4] `https://en.wikipedia.org/wiki/Double-precision_floating-point_format`

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:

$$\frac{dR_2}{dh} = -\frac{2\epsilon|f(x)|}{h^2} + \frac{1}{2}f''(\eta) = 0$$
$$h = 2\sqrt{\epsilon\left|\frac{f(x)}{f''(\eta)}\right|} \simeq 2 \cdot 10^{-8}, \tag{12}$$

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 $x \pm h$:

$$f(x+h) = f(x) + f'(x)h + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(4)}(x) + \cdots$$
$$f(x-h) = f(x) - f'(x)h + \frac{h^2}{2}f''(x) - \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(3)}(x) - \cdots . \tag{13}$$

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:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f^{(3)}(\eta), \tag{14}$$
$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + \frac{h^2}{12}f^{(4)}(\eta), \tag{15}$$

for some η in $[x, x+h]$. In figure 4, we have plotted equation (8), (14), and (15) for different step sizes. The derivative in equation (14), gives a higher accuracy than equation (8) for a larger step size, as can bee seen in figure 4.

We can perform a similar error analysis as we did before, and then we find for equation (14) and (15) that the total numerical error is:

$$R_3 = \frac{\epsilon|f(x)|}{h} + \frac{h^2}{6}f^{(3)}(\eta), \tag{16}$$
$$R_4 = \frac{4\epsilon|f(x)|}{h^2} + \frac{h^2}{12}f^{(4)}(\eta), \tag{17}$$

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 (16),

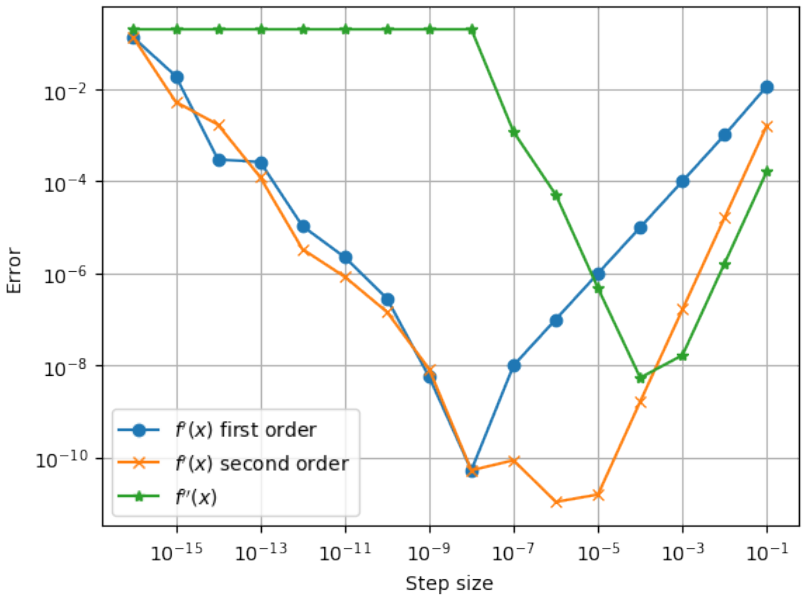

Figure 4: Error in the numerical derivative and second derivative of $\sin x$ at $x = 0.2$ for different step size.

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 (17), 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 (14), that a higher step size gives a more accurate result.