Finite Differences

Prepared as part of MOD510 Computational Engineering and Modeling

Oct 22, 2021

1 Getting familiar with Taylors formula

Learning objectives:

1. Understand how Taylors formula can be used to approximate a function, for more information on Taylors formula, see the book[1] or Wikipedia[2]
2. Understand approximation and numerical error

Below are *suggestions* for Python code, it is not the most optimal or elegant code, but meant as an inspiration. As a general rule, we *always* make the code as simple as possible (and as a consequence, usually, inefficient) the first time. When the code is working we can make it more elegant and/or efficient. The most important point is always to get a code that does what it is supposed to do.

Efficiency is usually lost whenever you write a for loop. For loops usually looks the same in any language and are easy to understand. As an example, let us consider the classical example of adding all integers from 1 to 100 (the answer is $n(n+1)/2 = 5050$ for $n = 100$), we can do this in Python several ways:

```
import numpy as np

def sum1(N):
    """ sum integers up to N
        C-type implementation """
    sum = 0.
    for i in range(N+1): # 0, 1, ..., N
        sum += i
    return sum

def sum2(N):
```

[1] http://www.ux.uis.no/ ah/CompEng/._book002.html#___sec3
[2] https://en.wikipedia.org/wiki/Taylor%27s_theorem

```
    """ sum integers up to N
        build in Python functions """
    x = [i for i in range(N+1)]
    return sum(x)

def sum3(N):
    """ sum integers up to N
        NumPy functions """
    x = np.arange(N+1)
    return np.sum(x)
```

If you are unfamiliar with `numpy.arange`, we refer to the official documentation here[3]. The first implementation is how you would to the summation in most low level languages as C, C++ or Fortran. It is straight forward to read, and it will give you the correct answer. How fast are the methods? This can be checked with the following code:

```
N=100
%timeit sum1(N)
%timeit sum2(N)
%timeit sum3(N)
```

The command `%timeit` is a built-in magic command, and will work in Jupyter notebooks and with iPython. You can read more about `%timeit` here[4]. Notice that there is hardly any difference in speed between the methods. However if you increase N to e.g. 1000, you will see that the NumPy implementation is superior.

Observe:

Testing the efficiency of code should always be done on *large* samples.

Exercise 1: Plot $\sin x$

Run the script below, and you will see a plot of $\sin x$ on the interval $[-\pi, \pi]$.

```
import matplotlib.pyplot as plt
import numpy as np

x=np.arange(-np.pi,np.pi,1)

f = np.sin(x)
plt.plot(x,f)
plt.grid()

plt.show()
```

[3] https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html
[4] https://ipython.org/ipython-doc/dev/interactive/magics.html#magic-timeit

Is this a good plot of $\sin x$? Improve the code above to produce a better plot of $\sin x$.

```
# enter code here
```

1. How many points are needed?
2. Do we need the same number of points in the whole domain? Which properties of the function determines where we need more points?

Exercise 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:

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

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[5]. 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 $x = c$:

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

Exercise 3: Maclaurin series of $\sin x$

Use equation (2) to show that the Maclaurin series ($c = 0$) of $\sin x$ is:

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

Make a plot of $\sin x$, and the first 7 terms of the Maclaurin series for $\sin x$:

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

```
import matplotlib.pyplot as plt
import numpy as np

#Maclaurin Serie of sin(x) at the power n
def mac_sin(x,n):
    """Returns the Maclaurin series up to order
       n for any x"""
    for element 0, 1, 2, ..., n-1:
    return x - x**3/3! + x**5/5! ...

# make the plot:
x  = np.arange(-np.pi,np.pi,0.01)
f1 = mac_sin(x,1)
f2 = mac_sin(x,1)
...
plt.plot(x,f1)
plt.plot(x,f1)
...
plt.show()
```

Exercise 4: Calculating $\sin x$ up to a certain error

The error term in Taylors formula, when we represent a function with a finite number of polynomial elements is given by:

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

for some η in the domain $[x, c]$.

Write an implementation of $\sin x$, that calculates $\sin x$ up to a certain accuracy, using the error formula in equation (4).

Exercise 5: Calculating derivative of functions

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

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

In the computer we cannot take the limit, $h \to 0$, a natural question is then: What value to use for h?

- Make a Python script that calculates the derivative of $\sin x$ for different step sizes h. Vary h between 10^{-15} and 0.1, and make a plot of the error versus the step size.

```
import matplotlib.pyplot as plt
import numpy as np

def f(x):
    return np.sin(x)

def df(x,h):
    return numerical derivative of f(x)

def err(x,h):
    return np.abs(df(x,h)-np.cos(x))

# to generate plot
step_size = [1/(1e1*10**h) for h in range(16)]
error =  estimate error for various step sizes
plt.loglog( ...)

plt.grid()

plt.show()
```

- Explain the shape of the plot you have made. Why does the numerical *not go to zero* as the step size is decreasing?
- Use equation (2), expand about $x \pm h$ and show that you can derive a higher order formula for the numerical derivative:

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

- Make a similar plot as before but this time also with equation (6).

Exercise 6: Round off errors vs numerical errors

The numerical error is given in equation (4), show that for the numerical derivatives in the previous exercise:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(\eta), \tag{7}$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f^{(3)}(\eta), \tag{8}$$

1. What is the value of η? Is it the same for equation (7) and (8)?
2. What is round off errors?
3. Show that the total error, including round off errors for (7) and (8) is:

$$R_3 = \frac{\epsilon|f(x)|}{h} + \frac{h^2}{6} f^{(3)}(\eta), \tag{9}$$

$$R_4 = \frac{4\epsilon|f(x)|}{h^2} + \frac{h^2}{12} f^{(4)}(\eta), \tag{10}$$

Which value of h would minimize R_3 and R_4? Compare with the plot in the previous exercise.