Finite Differences

Prepared as part of MOD510 Computational Engineering and Modeling

Sep 16, 2021

1 Numerical Integration Notebook

Learning objectives:

- being able to implement a numerical algorithm in python
- quantify numerical uncertainty
- test different methods and have basic understanding of the strength and weaknesses of each method

1.1 The Trapezoidal Rule

In the lecture notes it was shown that the algorithm for the trapezoidal rule was:

$$I(a,b) = \int_a^b f(x)dx \simeq h\left[\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{k=1}^{N-1} f(a+kh)\right]. \qquad (1)$$

Exercise 1: Implementing the trapezoidal rule in Python

Whenever implementing an algorithm it is always important to be absolutely sure that we have implemented it correctly. We always use functions that we know the true answer. As a test function you could use $\sin x$ or any other function (e.g. e^x etc.) and choose a reasonable integration domain. First:

1. Show that the analytical result is $\int_0^\pi \sin(x)dx = 2$
2. Show that if $N = 3$, equation (1) would give $I(0,\pi) = \frac{\pi}{\sqrt{3}} = 1.8137993\ldots$

Use the formula in equation (1) to develop a python function that takes as argument the integration limits (a, b), the function to be integrated, $f(x)$, and the number of integration points N:

```
import numpy as np
def f(x):
    return np.sin(x)

def int_trapez(func,lower_limit, upper_limit,N):
    """ calculates the area of func over the domain (lower_limit, upper)
        limit using N integration points """
# calculate the step size from the integration limits and N, do the sum, and return the area
    return area
```

- Test the code for $N = 3$, increase N and compare with the analytical result (2).
- By increasing N the numerical result will get closer to the true answer. How much do you need to increase N in order to reach an accuracy higher than 10^{-8}?
- Show that the error term for the trapezoidal rule is:

$$E_T \simeq \frac{h^2}{12}\left[f'(b) - f'(a)\right] = \frac{(b-a)^2}{12N^2}\left[f'(b) - f'(a)\right] = -\frac{\pi^2}{6N^2} \tag{2}$$

- How does the numerical error compares with the analytical error?

Exercise 2: Choose number of steps automatically for the trapezoidal rule

In practical applications we would like to just enter the accuracy we would like, and then expect our algorithm to figure out the number of steps. Change the code in the exercise above to calculate the value of the integral using a tolerance as input, instead of N. (The step size can be calculated from equation (2)

```
import numpy as np
def f(x):
    return np.sin(x)
#Numerical derivative of function
def df(x,func):
    dh=1e-5 # some low step size
    return (func(x+dh)-func(x))/dh

def int_trapez(func,lower_limit, upper_limit,tol):
    """ calculates the area of func over the domain (lower_limit, upper)
        limit using N integration points """
# calculate the step size h from the tolerance, do the sum, and return the ares
    return area

prec=1e-8
```

```
a=0
b=np.pi
Area = int_adaptive_trapez(a,b,f,prec)
print('Numerical value = ', Area)
print('Error           = ', (2-Area)) # Analytical result is 2
```

Exercise 3: Practical error estimate of numerical integrals

Assume that we estimate an integral using a step size of h_1 and h_2 ($h_1 = 2h_2$). The resulting estimates are I_1 and I_2 respectively. Show that the higher order error term for I_2 is:

$$E(a,b) = ch_2^2 = \frac{1}{3}(I_2 - I_1). \tag{3}$$

Make a Python implementation of the trapezoidal rule that uses this method to calculate the integral to a specific tolerance:

```
def int_adaptive_trapez(func, lower_limit, upper_limit,tol):
    N0      = 1 # first estimate of integral
    h       = (upper_limit-lower_limit)/N0
    area    = func(lower_limit)+func(upper_limit)
    area   *= 0.5
    val     = lower_limit
    # calculate the area using the trapezoidal rule
    # enter code:

    calc_tol = 2*tol # just larger than tol to enter the while loop
    while(calc_tol>tol):
        h *= .5 # half the step size
    # calculate new_area using the trapzoidal rule
    # enter code:

        calc_tol = abs(new_area-area)/3
        area     = new_area # store new values for next iteration

    print('Number of intervals = ', (upper_limit-lower_limit)/h )
    return area #while loop ended and we can return the area

prec=1e-8
a=0
b=np.pi
Area = int_adaptive_trapez(f,a,b,prec)
print('Numerical value = ', Area)
print('Error           = ', (2-Area)) # Analytical result is 2
```

Hint: To improve the efficiency of the code, you only need to calculate the odd terms in the next estimate of the area, using the algorithm in the compendium:

$$I_2(a,b) = \frac{1}{2}I_1(a,b) + h_2 \sum_{k=\text{odd values}}^{N_2-1} f(a+kh_2) \tag{4}$$

- Compare the number of function evaluation for an error of 10^{-8} using the algorithm in this exercise and the previous for the following integrals:
  - $\int_0^{\pi} \sin(x)dx$
  - $\int_0^1 e^{-x^2}dx$
  - $\int_{-1}^1 x^2 dx$

Answer.

```
#%%
import numpy as np

def g(x):
    return np.exp(3*x)*np.sin(2*x)
# Function to be integrated
def f(x):
    return np.sin(x)
# step size is chosen automatically to reach the specified tolerance
def int_adaptive_trapez(lower_limit, upper_limit,func,tol):
    """
    adaptive quadrature, integrate a function from lower_limit
    to upper_limit within tol*(upper_limit-lower_limit)
    """
    N0=1
    h       = (upper_limit-lower_limit)/N0
    area    = (func(lower_limit)+func(upper_limit))*0.5*h
    calc_tol = 2*tol + 1 # just larger than tol to enter the while loop
    TOL= tol*np.abs(h)   # note N0=1
    iterations=0
    while(calc_tol>TOL):
        iterations+=1
        N = N0*2
        h = (upper_limit-lower_limit)/N
        odd_terms=0
        for k in range (1,N,2): # 1, 3, 5, ... , N-1
            val  = lower_limit + k*h
            odd_terms += func(val)
        new_area = 0.5*area + h*odd_terms
        calc_tol = abs(new_area-area)/3
        area     = new_area # store new values for next iteration
        N0       = N        # update number of slices
    print('Number of intervals = ', N, ' iterations: ',iterations)
    return area #while loop ended and we can return the area

prec=1e-8
a=0
b=np.pi/4
Area = int_adaptive_trapez(a,b,g,prec)
```

```
print('Numerical value = ', Area)
print('Error           = ', (2-Area)) # Analytical result is 2


def int_adaptive_trapez2(lower_limit, upper_limit,func,tol):
    """
    adaptive quadrature, integrate a function from lower_limit
    to upper_limit within tol*(upper_limit-lower_limit)

    """
    S=[]
    S.append([lower_limit,upper_limit])
    I=0
    iterations=0
    while S:
        iterations +=1
        a,b=S.pop(-1) # last element
        m=(b+a)*0.5   # midpoint
        I1=0.5*(b-a)*(func(a)+func(b)) #trapezoidal for 1 interval
        I2=0.25*(b-a)*(func(a)+func(b)+2*func(m)) #trapezoidal for 2 intervals
        if(np.abs(I1-I2)<3*np.abs((b-a)*tol)):
            I+=I2     # accuarcy met
        else:
            S.append([a,m]) # half the interval
            S.append([m,b])
    print("Number of iterations: ", iterations)
    return I

Area = int_adaptive_trapez2(a,b,g,prec)
print("trapez 2", Area)
print("error", Area-2)

def simpson_step(a, b,func):
    m=0.5*(a+b)
    return (b-a)/6*(func(a)+func(b)+4*func(m))

def int_adaptive_simpson(a, b,func,tol):
    """
    adaptive quadrature, integrate a function from a
    to b within tol*(b-a) uses simpsons rule
    """
    S=[]
    S.append([a,b])
    I=0
    iterations=0
    while S:
        iterations +=1
        a,b=S.pop(-1) # last element
        m=(b+a)*0.5   # midpoint
        I1=simpson_step(a,b,func) #simpsons for 1 interval
```

```
        I2=simpson_step(a,m,func)+simpson_step(m,b,func) # ...2 intervals
        if(np.abs(I1-I2)<15*np.abs((b-a)*tol)):
            I+=I2     # accuarcy met
        else:
            S.append([a,m]) # half the interval
            S.append([m,b])
    print("Number of iterations: ", iterations)
    return I

Area = int_adaptive_simpson(a,b,g,prec)
print(Area)
print("error", Area-2)


#%timeit int_adaptive_trapez2(a,b,f,prec)

#%timeit int_adaptive_trapez2(a,b,f,prec)
#%%
```

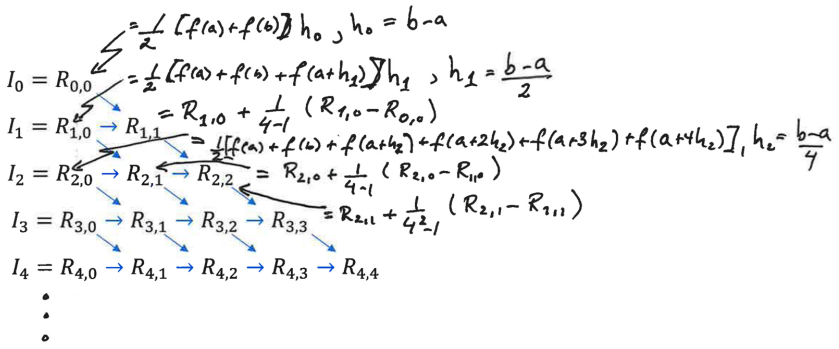
Exercise 4: Adaptive integration - Rombergs algorithm

In this exercise we will implement the Romberg algorithm, which is actually closely related to the adaptive trapezoidal rule in the previous exercise. The algorithm uses the technique from the previous exercise by halving the step size to estimate the error, but an additional trick is used: When we have the error estimate, we can add the error estimate to our numerical estimate of the integral to obtain a higher order accuracy. The algorithm is explained in the compendium and the result is:

$$I = R_{i,m+1} + \mathcal{O}(h_i^{2m+2}) \tag{5}$$

$$R_{i+1,m+1} = R_{i,m} + \frac{1}{4^{m+1}-1}(R_{i+1,m} - R_{i,m}). \tag{6}$$

Below is a graphical illustration of the algorithm:



```
def int_romberg(func,lower_limit, upper_limit,tol,show=False):
    """ calculates the area of func over the domain lower_limit
        to upper limit for the given tol, if show=True the triangular
        array of intermediate results are printed """
    Nmax = 100
    R      = np.empty([Nmax,Nmax]) # storage buffer
    h      = (upper_limit-lower_limit) # step size
    N      = 1
    R[0,0] = .5*(func(lower_limit)+func(upper_limit))*h # first estimate
    for i in range(1,Nmax):
        h /= 2
        N *= 2
    # estimate R[i,0] from the trapezoidal rule:
    # ....
    # next, estimate R[i,1], R[1,2],..,R[1,m+1]:
    #...
    # check tolerance, best guess
        calc_tol = abs(R[i,i]-R[i-1,i-1])
        if(calc_tol<tol):
            break  # estimated precision reached, exit for loop


    if(i == Nmax-1):
        print('Romberg routine did not converge after ', Nmax, 'iterations!')
    else:
        print('Number of intervals = ', N)

    if(show==True):
    # print out the triangular matrix
    # R[0,0]
    # R[1,0] R[1,1]
    # R[2,0] R[2,1] R[2,2]
    # etc.

    return R[i,i] #return the best estimate
```

You can check your implementation by comparing with the implementation in SciPy, Romberg[1]:

```
from scipy import integrate
integrate.romberg(f, a, b, show=True)
```

1. Compare the adaptive trapezoidal rule and the Romberg algorithm for $\int_{-1}^{1} x^4 dx$. Notice the extreme improvement by the Romberg algorithm.

Answer.

```
import numpy as np
```

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.romberg.html

```
# Function to be integrated
def f(x):
    return np.sin(x)
# step size is choosen automatically to reach (at least)
# the specified tolerance
def int_romberg(func,lower_limit, upper_limit,tol,show=False):
    """ calculates the area of func over the domain lower_limit
        to upper limit for the given tol, if show=True the triangular
        array of intermediate results are printed """
    Nmax = 100
    R = np.empty([Nmax,Nmax]) # storage buffer
    h = (upper_limit-lower_limit) # step size
    R[0,0]    =.5*(func(lower_limit)+func(upper_limit))*h
    N = 1
    for i in range(1,Nmax):
        h /= 2
        N *= 2
        odd_terms=0
        for k in range (1,N,2): # 1, 3, 5, ... , N-1
            val        = lower_limit + k*h
            odd_terms += func(val)
        # add the odd terms to the previous estimate
        R[i,0]   = 0.5*R[i-1,0] + h*odd_terms
        for m in range(0,i): # m = 0, 1, ..., i-1
            # add all higher order terms in h
            R[i,m+1]   = R[i,m] + (R[i,m]-R[i-1,m])/(4**(m+1)-1)
        # check tolerance, best guess
        calc_tol = abs(R[i,i]-R[i-1,i-1])
        if(calc_tol<tol):
            break  # estimated precision reached
    if(i == Nmax-1):
        print('Romberg routine did not converge after ',
              Nmax, 'iterations!')
    else:
        print('Number of intervals = ', N)

    if(show==True):
        elem = [2**idx for idx in range(i+1)]
        print("Steps StepSize Results")
        for idx in range(i+1):
            print(elem[idx],' ',
                  "{:.6f}".format((upper_limit-lower_limit)/2**idx),end = ' ')
            for l in range(idx+1):
                print("{:.6f}".format(R[idx,l]),end = ' ')
            print('')
    return R[i,i] #return the best estimate
#end
prec=1e-8
a=0
b=np.pi
```

```
Area = int_romberg(f,a,b,prec,show=True)
print('Numerical value = ', Area)
print('Error           = ', (2-Area)) # Analytical result is 2

from scipy import integrate
integrate.romberg(f, a, b, show=True)

def g(x):
    u=(1-x)
    u*=u
    return np.exp(-x*x/u)/u

w=100
def h(x):
    u=x/(1-x)
    return u*np.exp(-u)*np.cos(w*u)/(1-x)/(1-x)

Area = int_romberg(g,0.,0.99999999,prec,show=True)
print('Numerical value = ', Area)
print('Error           = ', (np.sqrt(np.pi)/2-Area)) # Analytical result is 2

Area = int_romberg(h,0.,0.99999999,prec,show=True)
print('Numerical value = ', Area)
print('Error           = ', ((1-w*w)/(w**2+1)**2-Area)) # Analytical result is 2

integrate.romberg(h, 0, 0.99999999999, show=True)

def i(x):
    return x**4-2*x+1

int_romberg(i,0.,2,prec,show=True)
```

Exercise 5: Evaluate $\int_a^b x^n f(x) x dx$

We will now look closer at an integral where the derivative has a singularity in the integration domain. We will consider the integral:

$$\int_0^1 x^{1/2} \cos x dx \simeq 0.53120268 \tag{7}$$

- Compare the adaptive trapezoidal rule and the Romberg algorithm. Note that in this case the trapezoidal rule does a better job (!). If you compare with the SciPy implementation you will also observe that an error is given because the accuracy is not reached.

Do the following substitution: $\tau = x^2$, and show that the integral can be rewritten as:

$$2 \int_0^1 \tau^2 \cos \tau^2 d\tau. \tag{8}$$

- Estimate the integral once more with the adaptive trapezoidal and Romberg algorithm. Note the greatly improvement in performance for the Romberg method

Notice.

It always wise to test out different methods, even if we expect that a specific method is supposed to be better it is not always so. Change of integration variable can greatly improve the performance.

Exercise 6: Gaussian evaluation of $\int_a^b x^n f(x)xdx$

Gaussian integration is extremely powerful, and should always be considered if speed is an issue. As explained in the compendium the idea behind the Gaussian integration is to approximate the function to be integrated on the domain as a polynomial of as *large a degree as possible*, then the numerical integral of this polynomial will be very close to the integral of the function we are seeking. In this case, considering equation (7), we can develop similar integration rules as in the compendium, and we choose $f(x) = 1$, x, x^2 x^3 to be integrated exact:

$$\int_0^1 x^{1/2}\,dx = \frac{2}{3} = \omega_0 + \omega_1\,, \tag{9}$$

$$\int_0^1 x^{1/2+1}\,dx = \frac{2}{5} = \omega_0 x_0 + \omega_1 x_1\,, \tag{10}$$

$$\int_0^1 x^{1/2+2}\,dx = \frac{2}{7} = \omega_0 x_0^2 + \omega_1 x_1^2\,, \tag{11}$$

$$\int_0^1 x^{1/2+3}\,dx = \frac{2}{9} = \omega_0 x_0^3 + \omega_1 x_1^3\,, \tag{12}$$

It is a bit cumbersome to solve the above equations, but in Python it can be done by e.g. using SymPy:

```
import sympy as sym
import numpy as np
x1,x2,w1,w2=sym.symbols('x1, x2, w1, w2')
#n=1/2 gives numerical value
n  = sym.Rational(1,2) # gives analytical result
f1 = sym.Eq(w1+w2,1/(n+1))
f2 = sym.Eq(w1*x1+w2*x2,1/(n+2))
f3 = sym.Eq( ... enter missing equation)
f4 = sym.Eq( ... enter missing equation)
sol=sym.solve(... correct syntax ...)
```

1. Find x_1 and x_2 and the corresponding weights ω_0 and ω_1

2. Implement the Gaussian integration rule in this case and estimate $\int_a^b x^{1/2}\cos x dx$
3. Compare your implementation with the standard implementation in the compendium

A correct implementation should give you $I \simeq 0.53109917759$, or an accuracy of 10^{-4}.

```
import sympy as sym
import numpy as np
x1,x2,w1,w2=sym.symbols('x1, x2, w1, w2')
#n=1/2 gives numerical value
n = sym.Rational(1,2) # gives analytical result
f1=sym.Eq(w1+w2,1/(n+1))
f2=sym.Eq(w1*x1+w2*x2,1/(n+2))
f3=sym.Eq(w1*x1**2+w2*x2**2,1/(n+3))
f4=sym.Eq(w1*x1**3+w2*x2**3,1/(n+4))
sol=sym.solve([f1,f2,f3,f4],(x1,x2,w1,w2))
print(sol)
```

Exercise 7: Oscillating integral and infinite integration limit

As you might have seen from the last exercise, it can be a challenge to estimate the integration points and the weights in the Gaussian quadrature. There are methods how to do this, using *interpolating polynomials*. The mathematics is very elegant, but we do not need to go into detail, we only need the result. The routine below calculates the weights and points for any order N, on the domain $[-1, 1]$.

```
# gausswx below adopted from
# M. Newman "Compuatiional Physics" - Appendix E
def gausswx(N):
    # initial approximation to roots of the Legendre polynomials
    a = np.linspace(3,4*N-1,N)/(4*N+2)
    x = np.cos(np.pi*a+1/(8*N*N*np.tan(a)))
    #Find Root using Newtons method
    epsilon = 1e-15
    delta = 1.0
    while delta > epsilon:
        p0 = np.ones(N, float)
        p1 = np.copy(x)
        for k in range(1,N):
            p0,p1=p1,((2*k+1)*x*p1-k*p0)/(k+1)
        dp = (N+1)*(p0-x*p1)/(1-x*x)
        dx=p1/dp
        x-=dx
        delta = np.max(abs(dx))
```

```
    # caluclate the weights
    w = 2*(N+1)*(N+1)/(N*N*(1-x*x)*dp*dp)

    return x,w

# gaussxwab below adopted from
# M. Newman "Compuatiional Physics" - Appendix E
def gaussxwab(N,a,b):
    x,w=gaussxw(N)
    return 0.5*(b-a)*x+0.5*(b+a),0.5*(b-a)*w

def gauss(f,a,b,N):
    xp,wp = gaussxwab(N,a,b)
    return np.sum(wp*f(xp))

def f(x):
    return x**4-2*x+1

def g(x):
    u=(1-x)
    u*=u
    return np.exp(-x*x/u)/u

w=5
def h(x):
    u=x/(1-x)
    return u*np.exp(-u)*np.cos(w*u)/(1-x)/(1-x)

a=0.
b=1.
N=300
x=np.arange(a,b,0.0001)
plt.plot(x,h(x),label=r'$\omega=$'+str(w))
plt.grid()
plt.xlim(a,b)
plt.legend()
plt.show()

analytical = (1-w*w)/(w**2+1)**2
Area = gauss(h,a,b,N)
eps=max(1e-8,np.abs(Area-analytical))
Area2 = int_romberg(h,a,b*.99999999,eps,show=True)
Area3 = int_trapez(h,a,b*.99999999,N)
print('Numerical value = ', Area)
print('Error  Gauss    = ', (Area-analytical))
print('Error  Romberg  = ', (Area2-analytical))
print('Error  Trapez   = ', (Area3-analytical))
```

The weights needs to be transformed to the domain $[a, b]$, and then the routine doing the Gaussian integration is quite simple, see the routine `gauss` below.

```
# gausswab below adopted from
# M. Newman "Compuatiional Physics" - Appendix E
def gaussxwab(N,a,b):
    x,w=gaussxw(N)
    return 0.5*(b-a)*x+0.5*(b+a),0.5*(b-a)*w

def gauss(f,a,b,N):
    xp,wp = gaussxwab(N,a,b)
    return np.sum(wp*f(xp))

def f(x):
    return x**4-2*x+1

def g(x):
    u=(1-x)
    u*=u
    return np.exp(-x*x/u)/u

w=5
def h(x):
    u=x/(1-x)
    return u*np.exp(-u)*np.cos(w*u)/(1-x)/(1-x)

a=0.
b=1.
N=300
x=np.arange(a,b,0.0001)
plt.plot(x,h(x),label=r'$\omega=$'+str(w))
plt.grid()
plt.xlim(a,b)
plt.legend()
plt.show()

analytical = (1-w*w)/(w**2+1)**2
Area = gauss(h,a,b,N)
eps=max(1e-8,np.abs(Area-analytical))
Area2 = int_romberg(h,a,b*.99999999,eps,show=True)
Area3 = int_trapez(h,a,b*.99999999,N)
print('Numerical value = ', Area)
print('Error  Gauss    = ', (Area-analytical))
print('Error  Romberg  = ', (Area2-analytical))
print('Error  Trapez   = ', (Area3-analytical))
```

The following integral:

$$\int_0^\infty xe^{-x}\cos(\omega x)dx \tag{13}$$

```
def f(x,w):
    return x*np.exp(-x)*np.cos(w*x)

x=np.arange(0,20,0.01)
w=40
plt.plot(x,f(x,w),label=r'$\omega$=' +str(w))
plt.grid
plt.show()
```

References