Yousif Aldossary

300231539

23/09/2019

Project 4

Numerical_differentiation_and_integration

Q1 Numerical differentiation

The Assingmnet has two parts.

introduction

This question is working on numerical differentation for a function, while taking forward, backward and centeral derivaties. The reason we work on different location of the derivative is the size of h for small derivative to obtain an approximation value of the function for part A.
The functions are listed below:

$$Centerdf(x) = f(x+h) - f(x-h)/2h$$
$$forwarddf(x) = f(x+h) - f(x)/h$$
$$backwarddf(x) = f(x) - f(x-h)/h$$

In part B is using truncation error of $O(h)$, by considering $h = h1, h2, \cdots h_n$ and that each $h_i + 1 < h_i$. Then we compute the error $E_i = |f'(x) - Df(x)|$, and take the value of E_i and $E_i + 1$ to approximate the order of p as shown below:

$$Ei/Ei + 1 = (hi/hi + 1)^p$$

rearranged

$$p = log(Ei/Ei + 1)/log(hi/hi + 1)$$

In this question we will use the function $f(x) = sin(x)$ and $x = pi$ using the values of $h = 0.2, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001$ for central forward and backward differennces, and then compute p.

In [1]:

```
"Part A"
from numpy import *
from matplotlib.pyplot import *

def f(x):
    y = sin(x)
    return y

x = pi

h = array([0.2,0.1,0.05,0.01,0.005,0.001,0.0005,0.0001])
# second order at the center
deriv_center = (f(x+h)-(f(x-h)))/h
# first order at both ends
deriv_forward = (f(x+h)-(f(x)))/h
deriv_backward = (f(x)-(f(x-h)))/h

"Part B"
# finding the error rate using Ei and Ei+1 and I already have h values

for i in range(0,7):
    Ei = abs(cos(x) - deriv_backward)
    Ei = Ei + 1
    #E1 = abs(cos(x) - deriv_forward)
    #E1 = E1+1

#print('df(x)=',deriv_center)
#print('df_f(x)=',deriv_forward)
#print('df(x)_b=',deriv_backward)

print(Ei)
#print(E1)
plot(Ei)
show()

for j in range(0,7):
    p=log(Ei[j]/Ei[j+1])/log(h[j]/h[j+1])
    print(p)
```

```
[1.00665335 1.00166583 1.00041661 1.00001667 1.00000417 1.00000017
 1.00000004 1.        ]

<Figure size 640x480 with 1 Axes>

0.007165668092728956
0.001800367625141003
0.0002484478457468599
1.80333874542364e-05
2.4853310671862074e-06
1.8033749081045149e-07
2.485450017085903e-08
```

Observation:

the results for the program above has given values that are decreasing until it reaches a constant of 1.0 for the forward and backward results, while the center is increasing from 1.98669 to a constant of 2. my prediction of this cause is the forward and backward are using the 1st order of differentiation, while the center is using the

2nd order of dervative. However, all my results share the same error values of <+- 0.

Conclusion:

I am uncertain how to obtain the results of the function other than the given formula in the assignment sheet. It would be best if we can test the method using an alternative method to check the results. However, I predicted the results would be something similar to this.

Q1 Numberical Quadrature, Midpoint rule , degree of exactness

This second part of the assingment is using the integral of a function by estimating the area. In this part we will use 3 factors to help understand the idea of the second part of the assignment:

– "Degree of exactness:" the largest value of n so that all polynomials of degree n and below are integrated exactly. (Degree of a polynomial is the highest power of x appearing in it.)

– "Order of accuracy:" the value of n so that the error is O(hn), where h measures the subinterval size.

– "Index:" a number distinguishing one of a collection of rules from another.

The concepts will help later when we program to find the midpoints, and its error using midpoint rule, trapezoidal rule, and lastly Gauss-Legendre family of rules to obtain the neccassary values to fill up the table given in the Assingments.

more endpoint than intervals, of course). Then the midpoint rule can be written as

$$\text{Midpoint rule } = \sum_{k=1}^{N-1} (x_{k+1} - x_k) f(\frac{x_k + x_{k+1}}{2}).$$

- **Exactness:** If a quadrature rule can compute exactly the integral of any polynomial up to some specific degree, we will call this its degree of exactness. Thus a rule that can correctly integrate any cubic, but not quartics, has exactness 3.

  To determine the degree of exactness of a rule, we might look at the approximations of the integrals

$$\int_0^1 1dx = [x]_0^1 = 1$$

$$\int_0^1 2xdx = [x^2]_0^1 = 1$$

$$\int_0^1 3x^2dx = [x^3]_0^1 = 1$$

$$\vdots$$

$$\int_0^1 (k+1)x^k dx = [x^{k+1}]_0^1 = 1$$

2

in part 1. of this part we will use one function $f(x) = 1/(1 + x * *2)$ with the exact answer of $2arctan(5)$ and using the intervals $[-5, 5]$. However, we will change the values of N and fill up the table using the results obtained.

In [10]:

```
import numpy
f = lambda x: 1/(1+x**2)
# number of interval intake between the given intervals

N=10001
x = numpy.linspace(-5,5,N)
y = f(x)

sum = 0.0
# taking the sum between all values intake between intervals and sum them up
for i in range(N-1):
    sum = sum + (x[i+1]-x[i])*f((x[i]+x[i+1])/2.0)

#as it exactly show those are midpoint results and error rate
print('midpoint results = ', sum, 'error', np.abs(sum - 2*np.arctan(5)))
```

```
midpoint results =  2.74680153512277 error 1.2327383558385918e-09
```

N	h	Midpoint	error
11	1.0	2.7363	0.01049
101	0.1	2.7468	1.23e-5
1001	0.01	2.7468	1.23e-7
10001	0.001	2.7468	1.23e-9

observation:

The results on the table above shows that the mid point results is almost consistant, however the more we increase N values to run the function the smaller the error becomes.

Conclusion:

We can determine if the results are correct by comparing the current results with the last results of Gauss-Legendre integration method, and trapozoidal. Thus, we can predict the results of the next two methods will be almost the same or similar

Q2

In this part of the assingmnet we will use a different approach. Instead of increasing N we will run multiple functions at each step $f(x) = (n + 1) * x^n$ but we only need four point starting from n = 0 as required from the assingment.

In [151]:

```
# this is increase in function instead while the number of order is constant
# of 2 or 3 dpending on the requirements

import numpy
f = lambda x: (n+1)*x**n
n = 3
N=10
x = numpy.linspace(0.0,1.0,N)
y = f(x)

sum = 0.0
# midpoint method/function given in the assignment in a loop
for i in range(N-1):
    sum = sum + (x[i+1]-x[i])*f((x[i]+x[i+1])/2.0)

print('midpoint results = ', sum, 'error', numpy.abs(1-sum))
```

```
midpoint results =  0.9938271604938271 error 0.006172839506172867
```

func	Midpoint	error
1	1.0	0.0
2x	1.0	0.0
$3x^2$	0.996913	0.003086
$4x^3$	0.993827	0.006172

Observation:

THe results are always 1 for this function. However, this is results is different to Q1 in part 2 since the error was exactly then it started becoming less accurate by the increase of function.

Conclusion:

Similar to Q1, we will also test the accuracy of other function such as trepzoidal and gauss-legendre methods. However, the results should be similar, so we can expect similar results such as increase in function = decrease in accuracy.

Q3

In this question we will be using trapzoidal rule for linear approximations similar to the previous questions 1,2. Therefore, we will test this method on the same function and requirements for Q1, Q2 for increase in N and functions.
Moreover, the trapzoidal rule work by estimating a b as a = x0 and b= xn where n is the number of intervals. h = (b-a) is the distance of the function between a and b and the function will look as the image below including the graph to clarify the method.

Trapezoidal rule

Trapezoidal rule uses $N = 1$, i.e. linear polynomials to approximate the function $f(x)$.

For $N = 1$ we have

$$\ell_0(x) = \frac{x - x_1}{x_0 - x_1} = -\frac{x - b}{h}$$

and

$$\ell_1(x) = \frac{x - x_0}{x_0 - x_1} = \frac{x - a}{h}$$

where $h = b - a$.

So

$$A_0 = \frac{1}{h}\int_a^b (x - b) = \frac{1}{2h}(b - a)^2 = \frac{h}{2},$$

and

$$A_1 = \frac{1}{h}\int_a^b (x - a)dx = \frac{1}{2h}(b - a)^2 = \frac{h}{2}.$$

Substitution in the formula for $I_N[f]$ gives:

$$I_1[f] = [f(a) + f(b)]\frac{h}{2}$$

Composite trapezoidal rule

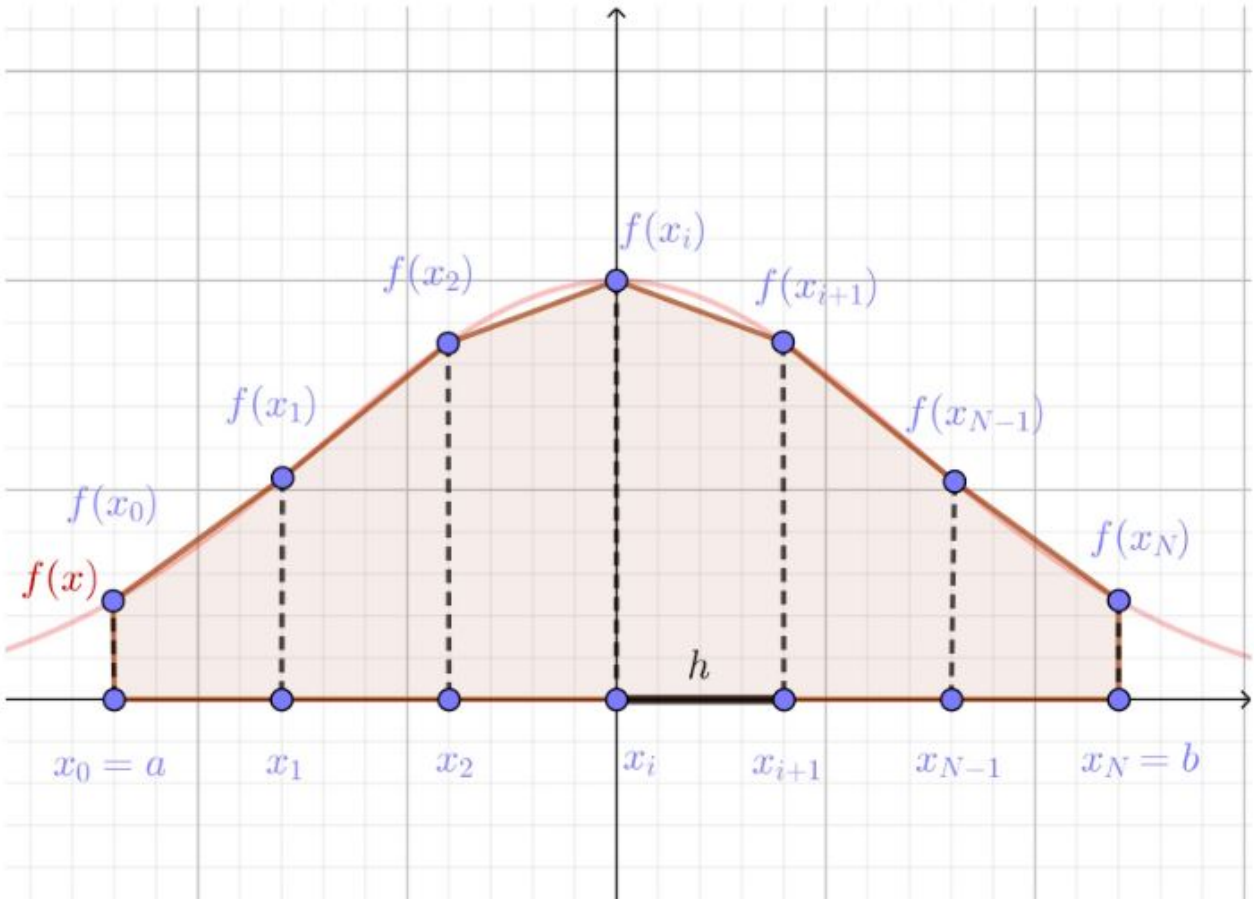
In practice the trapezoidal rule is applied separetely in divisions of the interval (a, b). For example we consider N intervals $D_i = [x_i, x_{i+1}]$, for $i = 0, 1, \cdots, N-1$ and we apply the trapezoidal rule in each interval D_i:

$$I_i = [f(x_i) + f(x_{i+1})]\frac{h}{2}$$

where $h = x_{i+1} - x_i$, for $i = 0, 1, 2, \cdots, N$.

Hence the total area is

$$I_N[f] = \sum_{i=0}^{N-1} I_i = [f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{N-1}) + f(x_N)]\frac{h}{2}$$



In [180]:

```
from numpy import *
from matplotlib.pyplot import *
#trapezoidal function for 2 points only
def MidpointQuad(f,a,b,N):

    x = linspace(0,1,N+1)
    y = f(x)
    h = (b-a)

    sum = (h)/2 * (f(a) + f(b))
    return sum

for i in range(0,4):
    f = lambda x: (i+1) * x**i
    print('midpoint' , MidpointQuad(f,0.0,1.0,2), 'error', np.abs((MidpointQuad(f,0.0,1.0,2
```

```
midpoint 1.0 error 0.0
midpoint 1.0 error 0.0
midpoint 1.5 error 0.5
midpoint 2.0 error 1.0
```

func	Midpoint	error
1	1.0	0.0
2x	1.0	0.0
$3x^2$	1.5	0.5
$4x^3$	2.5	1.0

Observation:

The results of trapezoidal method is very simial to midpoint method. However, this is more consistant to the results of 1.5 with erro of 0.5 and increasing. Therefore, for increase in function I would assume this is the least reliable method to use comparing this method with the other three methods.

Conclusion:

We can assume that midpoint method is better in accuracy and midpoint results than trapezoidal. However, we still have one more method to compare with these 2 methods. However, this shows that trapezoidal method works and the prediction are correct.

In [9]:

```
# trapezoidal rule for multiple points intakes
import numpy as np
def trapezoid(f, a, b, N):

    x = np.linspace(a, b, N+1)
    y=f(x)
    # h is sort of averaging the values to reach an approximation
    h = (b - a)/N
    sum = 0.0
    # the loop of calculating the approximation using trapezoidal method
    for i in range(1,N):
        sum+= 2.0*y[i]
    sum = (h)/2 * (f(a) + sum + f(b))
    return sum

f = lambda x: 1.0/(1.0+x**2)

print(trapezoid(f, -5, 5, 10001))
print('error', np.abs(trapezoid(f, -5, 5, 11) - 2.0*np.arctan(5)))
```

```
2.7468015314250143
error 0.008279306820373034
```

N	h	Midpoint	error
1	1.0	2.73852	0.008279
101	0.1	2.74777	2.465e-5
1001	0.01	2.7468	2.465e-7
10001	0.001	2.7468	2.465e-9

observation:

The results shows the increase of interval intake N between -5 to 5 will results to better accuracy. However, the accuracy is very satisfying to be 2.465-9, and this is the prediction I had from the previous data for midpoint method.

Conclusion:

The results are accurate, but according to the assingment hint that gaussian legendre should have better results amoung the methods. Therefore, I can assume that the results of error will be much smaller than trapezoidal rule.

Q4

This part will work on gauss-legendre, but this function is using the pythond function numpy.polynomial.legendre.leggauss this will trake (x,w) into count as shown in the formula below:

Gaussian Quadrature on Arbitrary Intervals

An integral $\int_a^b f(x)\,dx$ over an arbitrary interval $[a,b]$ can be transformed into an integral over $[-1,1]$ by using the change of variables:

$$t = \frac{2x-a-b}{b-a} \Leftrightarrow x = \frac{1}{2}[(b-a)t+a+b]$$

This permits Gaussian quadrature to be applied to any interval $[a,b]$ because

$$\int_a^b f(x)\,dx = \int_{-1}^{1} f\left(\frac{(b-a)t+(b+a)}{2}\right)\frac{(b-a)}{2}\,dt$$

So

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\sum_{i=1}^{N} w_i f\left(\frac{b-a}{2}t_i + \frac{a+b}{2}\right)$$

the function will approximate the results, and according to the assignment sheet hint that this method should work best amoung the 3 methods we are testing. Therefore, we can predict high accuracy for both Q1 and Q2 requirements to find midpoints and errors. However, for this part we will only focus on the error for the increase of function for N = 2 and N =3. But the other part where N is increasing, we will focus on the outputs and the error accuracy.

In [210]:

```
# this is gaussian legendre method found online and its mentioned in
# the assignment

import numpy as np
from numpy.polynomial.legendre import leggauss

def f(x):
    return 4*x**3

a = 0.0
b = 1.0
n = 3

# takes both values of x and w looking at it mathematically
t,w = leggauss(n)

A = (b-a)/2.0
B = (b+a)/2.0

sumi = 0.0
# the function of gaussian method
for i in range(n):
    sumi = sumi + w[i]*f(A*t[i]+B)

sumi = A*sumi

error = np.abs(sumi-1.0)

print(error)
print(sumi)
```

```
2.220446049250313e-16
1.0000000000000002
```

func	error N=2	error N=3
1	0.0	2.22e-16
2x	0.0	2.22e-16
$3x^2$	0.0	2.22e-16
$4x^3$	1e-16	2.22e-16

observation

Accoridng to the tables outputs from midpoint, trapezoidal and gaussian legendre. Gaussian legendre has an error for increase of function almost always zero. Unlike, midpoint and trapezoidal the error increased frequently. Therefore, thet prediciton based on the assignment hint is correct.

Conclusion:

it is best to use guassian-legendre function. However, in my perspective the function can be writting better since it is only used on function at time, where I could use a loop for each increase of the function based a given range.

In [207]:

```
import numpy as np
from numpy.polynomial.legendre import leggauss

def f(x):
    return 1.0/(1.0+x**2)

a = -5.0
b = 5.0
n = 10001

t,w = leggauss(n)

A = (b-a)/2.0
B = (b+a)/2.0

sumi = 0.0
for i in range(n):
    sumi = sumi + w[i]*f(A*t[i]+B)

sumi = A*sumi

error = np.abs(sumi-2.0*np.arctan(5.0))

print(error)
print(sumi)
```

```
5.773159728050814e-15
2.7468015338900376
```

N	Midpoint	error
11	2.812	0.065
101	2.7468	1.0658e-14
1001	2.7468	1.914e-13

N	Midpoint	error
10001	2.7468	5.77e-15

Observation:

The function works perfectly, and it more accurate than midpoint, and trapezoidal rules. The function works similar to the previous methods with increase of gap number between intervals the more accurate the results. Although, the midpoint function are almost the same results in all methods.

Conclusion:

We can conclude that all function with increase share almost the same accuracy of almost 0, but gaussian function has the error at very small of e-16. Therefore, we can conclude that the best method to be used among all three is guassian.

References:

Dr Dimitrios
Lesson 11 to 13
https://docs.scipy.org/doc/numpy/reference/generated/numpy.polynomial.legendre.leggauss.html