Chap 4: Numerical Differentiation and Integration

Due on May, 2022

Yuchen Ge

1. Introduction

The chapter mianly deals with the problem of numerical differentiation and integration. **When dealing with numerical differentiation**, we have the three(five)-point midpoint(endpoint) formulas. **When dealing with numerical integration**, we have closed (open) newton-cotes formulas, composite numerical integration, romberg integration, gaussian quadrature, multiple integration via former techniques.

First we claim that all function and variable names are self-clear. And a/b is interpreted to be section a question b, e.g. 4.4/7.

2. Application of Five-point Formulas (4.1/5a)

2.1 Program Running and Data Analysis

The question requires the best approximation of the differentiation. Thus we apply five-point midpoint(endpoint) fomulas for the best approximation. Considering the data types we apply the five-point endpoint formulas (4.7) for solving $x = 2.1, 2.2, 2.5, 2.6$ and the five-point midpoint formulas (4.6) for solving $x = 2.3, 2.4$. We run the program as follows. (Meanwhile we display the running time.)

```
def Fivepointmid(a,b,c,d,h):
    Y=(a-8*b+8*c-d)/(12*h)
    return Y

def Fivepointendleft(a,b,c,d,e,h):
    Y=(-25*a+48*b-36*c+16*d-3*e)/(12*h)
    return Y

def Fivepointendright(a,b,c,d,e,h):
    Y=(-3*a+16*b-36*c+48*d-25*e)/(12*h)
    return Y


print('The approximation for derivative at x=2.1 is', Fivepointendleft(-1.709847,-1.373823,-1.119214,-0.9160143,-0.7470223,0.1),'.')
print('The approximation for derivative at x=2.2 is', Fivepointendleft(-1.373823,-1.119214,-0.9160143,-0.7470223,-0.6015966,0.1),'.')
print('The approximation for derivative at x=2.3 is', Fivepointmid(-1.709847,-1.373823,-0.9160143,-0.7470223,0.1),'.')
print('The approximation for derivative at x=2.4 is', Fivepointmid(-1.373823,-1.119214,-0.7470223,-0.6015966,0.1),'.')
print('The approximation for derivative at x=2.5 is', Fivepointendright(-1.709847,-1.373823,-1.119214,-0.9160143,-0.7470223,0.1),'.')
print('The approximation for derivative at x=2.6 is', Fivepointendright(-1.373823,-1.119214,-0.9160143,-0.7470223,-0.6015966,0.1),'.')
```

```
The approximation for derivative at x=2.1 is 3.8993442499999875 .
The approximation for derivative at x=2.2 is 2.876875666666669 .
The approximation for derivative at x=2.3 is 2.249704083333334 .
The approximation for derivative at x=2.4 is 1.8377559999999993 .
The approximation for derivative at x=2.5 is -1.544209916666667 .
The approximation for derivative at x=2.6 is -1.3554963333333316 .
[Finished in 54ms]
```

We write down the results in the table below, where we **reserve seven significant digits** for the result.

Table 1: Results of 4.1/5a

x	$f(x)$	$f'(x)$	**Applied Formulas**
2.1	-1.709847	3.899344	4.1/(4.7)
2.2	-1.373823	2.876876	4.1/(4.7)
2.3	-1.119214	2.249704	4.1/(4.6)
2.4	-0.9160143	1.837756	4.1/(4.6)
2.5	-0.7470223	1.544210	4.1/(4.7)
2.6	-0.6015966	1.355496	4.1/(4.7)

3. Application of Richardson's Extraposition (4.2/8,9)

3.1 Solution of 4.2/8

Solution: We have

$$f'(x_0) = N_1(h) - \frac{f''(x_0)}{2} \cdot h - \frac{f'''(x_0)}{6} \cdot h^2 + O(h^3), \tag{1}$$

where $N_1(h) = (f(x_0 + h) - f(x_0))/h$. Replacing h with $h/2$ gives

$$f'(x_0) = N_1(\frac{h}{2}) - \frac{f''(x_0)}{2} \cdot \frac{h}{2} - \frac{f'''(x_0)}{6} \cdot \frac{h^2}{4} + O(h^3), \tag{2}$$

Therefore subtract (1) from twice (2) gives

$$f'(x_0) = 2 \cdot N_1(\frac{h}{2}) - N_1(h) + \frac{f'''(x_0)}{6} \cdot \frac{h^2}{2} + O(h^3), \tag{3}$$

Similarly we can replace h in (3) with $h/2$ gives

$$f'(x_0) = 2 \cdot N_1(\frac{h}{4}) - N_1(\frac{h}{2}) + \frac{f'''(x_0)}{6} \cdot \frac{h^2}{8} + O(h^3). \tag{4}$$

Therefore subtract (3) from four times (4) finally gives

$$3 \cdot f'(x_0) = 6 \cdot N_1(\frac{h}{2}) - 3 \cdot N_1(h) + O(h^3).$$

And thus if we write $N_1(h) = (f(x_0 + h) - f(x_0))/h$,

$$N_3(h) = 2 \cdot N_1(\frac{h}{2}) - N_1(h)$$

is the formula that we want.

3.2 Solution of 4.2/9

Solution: We have by replacing h by $h/3$

$$M = N(\frac{h}{3}) + K_1\frac{h}{3} + K_2\frac{h^2}{9} + O(h^3) \tag{5}$$

Subtracing (5) from three times the original equation in question gives

$$2M = 3N(\frac{h}{3}) - N(h) - K_2\frac{2h^2}{3} + O(h^3) = 2N_2(h) - K_2\frac{2h^2}{3} + O(h^3) \tag{6}$$

Similarly we have

$$2M = 3N(\frac{h}{9}) - N(\frac{h}{3}) - K_2\frac{2h^2}{27} + O(h^3) = 2N_2(\frac{h}{3}) - K_2\frac{2h^2}{27} + O(h^3), \tag{7}$$

and accordingly we have

$$16M = 18N_2(\frac{h}{3}) - 2N_2(h) + O(h^3). \tag{8}$$

Finally both equations together gives the results.

$$N_2(h) = \frac{3}{2} \cdot N(\frac{h}{3}) - \frac{N(h)}{2} = N(\frac{h}{3}) + \frac{N(\frac{h}{3}) - N(h)}{2}.$$
$$N_3(h) = \frac{9}{8} \cdot N_2(\frac{h}{3}) - \frac{N_2(h)}{8} = N_2(\frac{h}{3}) + \frac{N_2(\frac{h}{3}) - N_2(h)}{8}.$$

4. Application of Composite Numerical Integration (4.4/7)

To interpret the value of

$$\int_0^2 e^{2x} sin(3x) dx \quad \text{(question 4.4/7)}$$

by means of numerical integration. First we calculate an actual value of it as follows

$$\int_0^2 e^{2x} sin(3x) dx = Im(\int_0^2 e^{(2+3i)x} dx) = \frac{3 + e^4(2sin6 - 3cos6)}{13} \approx -14.2139771298625\ldots$$

Let $f(x) := e^{2x} sin(3x)$, we have $f''(x) = 12cos(3x)e^{2x} - 5sin(3x)e^{2x}$ and

$$max\{|f''(x)| : x \in [0, 2]\} = |f''(2)| = e^4(12cos6 - 5sin6) \approx 705.360102876099\ldots$$

Therefore for **composite trapezoidal rule**, **the error form** is

$$|\frac{b-a}{12} h^2 f''(\xi)| \leq |\frac{f''(2)}{6} h^2| < 10^{-4} \implies h < (\frac{6 \cdot 10^{-4}}{f''(2)})^{1/2} = 0.000922295691056397\ldots$$

and thus

$$n \geq \frac{2}{0.000922295691056397} = 2168.5 \implies n \geq 2169.$$

Similarly for **composite simpsons rule** we have

$$|\frac{b-a}{180} h^4 f^{(4)}(\xi)| < 10^{-4} \implies h < 0.037658\ldots \text{ and } n \geq 54.$$

And **composite midpoint rule** we have

$$|\frac{b-a}{6} h^2 f''(\xi)| \leq |\frac{f''(2)}{6} h^2| < 10^{-4} \implies h < 0.00065216\ldots \text{ and } n \geq 3067.$$

Finally we have

Table 2: Results of 4.4/7

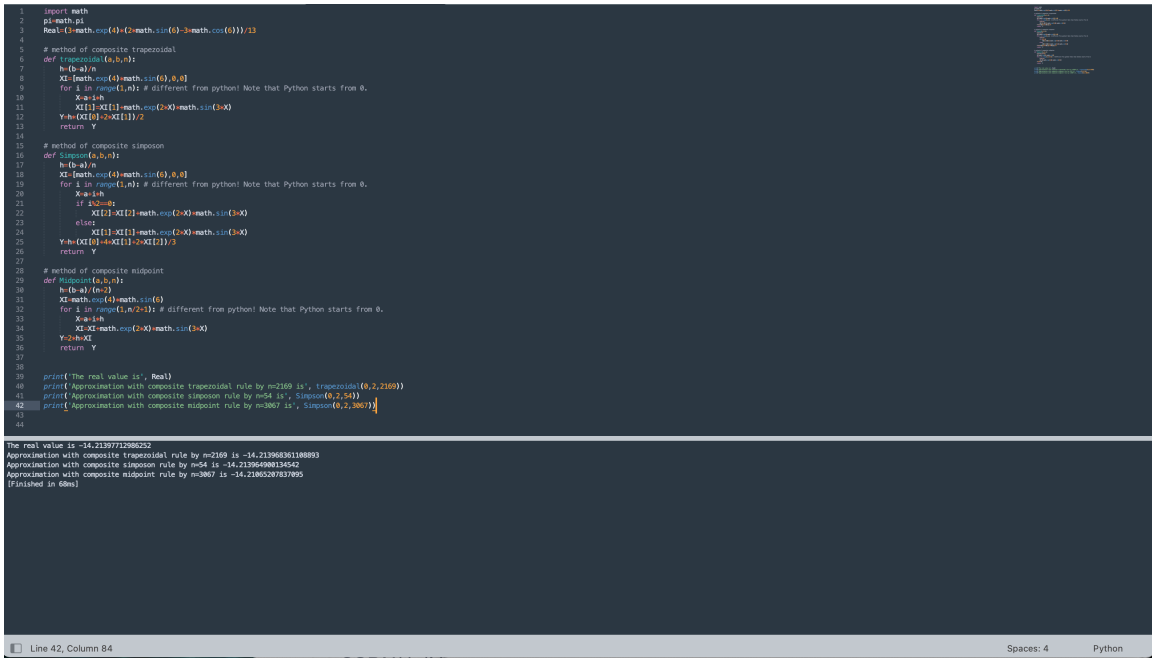
Applied Methods	**Maximum Value of h**	**Minimum Value of n**
composite trapezoidal rule	0.000922295691	2169
composite simpsons rule	0.037658	54
composite midpoint rule	0.00065216	3067

and for completeness we give the approximation

Table 3: Results of 4.4/7

Applied Methods	**n**	**Approximation Value**	**Error**
composite trapezoidal rule	2169	-14.213968361108893	-8.768753627208525e-06
composite simpsons rule	54	-14.213964900134542	-1.2229727978763094e-05
composite midpoint rule	3067	-14.21065207837095	-0.0033250514915703633

For completeness, we write down the codes and run the program for each case, e.g. n=2169 with composite trapezoidal rule.

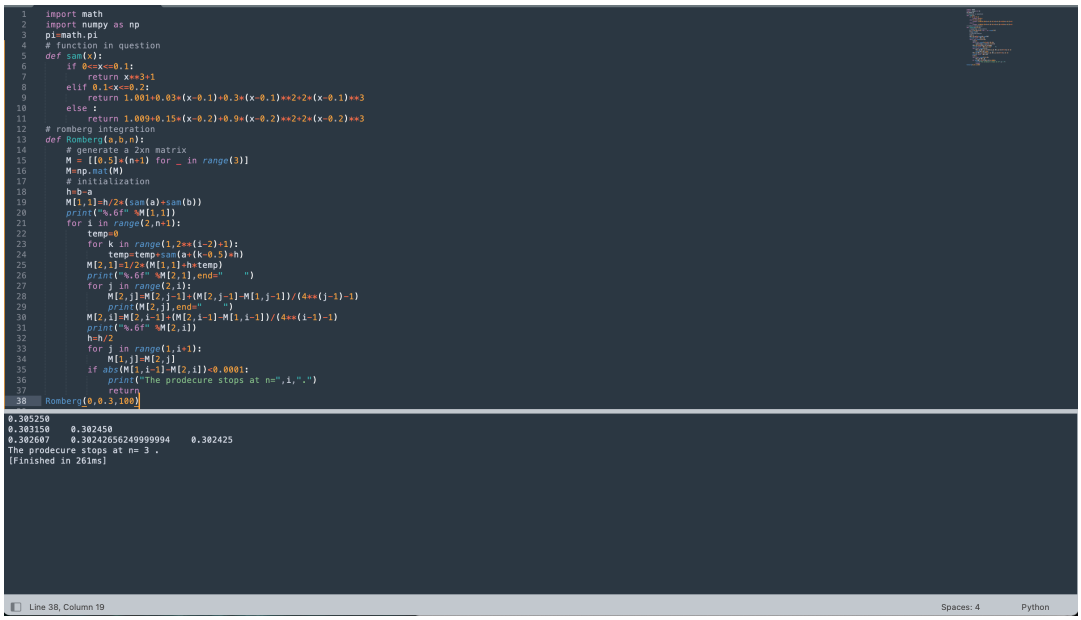


5. Application of Romberg Integration (4.5/4b)

4.5/4b requires approximation of $\int_0^{0.3} f(x)dx$ where

$$f(X) = \begin{cases} X^3 + 1 & 0 \le X \le 0.1 \\ 1.001 + 0.03(X-0.1) + 0.3(X-0.1)^2 + 2(X-0.1)^3 & 0.1 < X \le 0.2 \\ 1.009 + 0.15(X-0.2) + 0.15(X-0.2)^2 + 0.9(X-0.2)^3 & 0.2 < X \le 0.3 \end{cases}$$

And we run the program as follows. (Meanwhile we display the running time.)



The real value is

$$\int_0^{0.3} f(x)dx = (\int_0^{0.1} + \int_{0.1}^{0.2} + \int_{0.2}^{0.3}) f(x)dx = 0.100025 + 0.10031 + 0.102 = 0.30256$$

We show the generated results in a table just as table 4.9. Note: we end at n=3 since $R_{n-1,n-1}$ and $R_{n,n}$ agree to within 10^{-4}

Table 4: Results of 4.5/4b

0.305250	0	0
0.303150	0.302450	0
0.302607	0.30242656249999994	0.302425

We see that the best approximation is given by (either one)

$$\int_0^{0.3} f(x)dx - R_{31} = 0.30256 - 0.302607 = -4.7 \times 10^{-5}$$

$$\int_0^{0.3} f(x)dx - R_{32} = 0.30256 - 0.30242656249999994 = 1.3 \times 10^{-4}$$

$$\int_0^{0.3} f(x)dx - R_{33} = 0.30256 - 0.302425 = 1.3 \times 10^{-4}$$

We see that the romberg technique has two desirable features:

1.It allows an entire new row easily calculated from above.

2.It uses an averaging technique to produce formulas with higher-order tuncation error.

6. Application of Gaussian Quadrature (4.7/1b)

The question is to determine values of $\int_0^1 x^2e^{-x}dx$ with 2/3/4/5-point Gaussian Quadrature.
First the real value is

$$\int_0^1 x^2e^{-x}dx = -e^{-1} - \int_0^1 2xd(e^{-x}) = 2 - \frac{5}{e} = 0.160602794142788\ldots$$

Using 4.7/(4.42) we have

$$\int_0^1 f(x)dx = \int_{-1}^1 f(\frac{t+1}{2}) \cdot \frac{1}{2}dt$$

where $f(x) := x^2e^{-x}$ and we let $g(x) := f(\frac{t+1}{2}) \cdot \frac{1}{2}$. Using values in table 4.11 we write down and codes and run the program as follows. (Meanwhile we display the running time.)

```python
import math
import numpy as np
e=math.exp(1)
Real=2-5/e
def g(x):
    return 1/2*(((x+1)/2)**2)*math.exp(-(x+1)/2)
# input the number of points
def Gauss(n):
    if n==2:
        return g(0.5773502692)+g(-0.5773502692)
    if n==3:
        return 0.5555555556*g(0.7745966692)+0.8888888889*g(0)+0.5555555556*g(-0.7745966692)
    if n==4:
        return 0.3478548451*(g(0.8611363116)+g(-0.8611363116))+0.6521451549*(
            g(0.3399810436)+g(-0.3399810436))
    if n==5:
        return 0.2369268850*(g(0.9061798459)+g(-0.9061798459))+0.5688888889*g(0)+0.4786286705*(
            g(0.5384693101)+g(-0.5384693101))
print("The approximation applying 2-point gauss quadrature is", Gauss(2), "with error term",abs(Real-Gauss(2)))
print("The approximation applying 3-point gauss quadrature is", Gauss(3), "with error term",abs(Real-Gauss(3)))
print("The approximation applying 4-point gauss quadrature is", Gauss(4), "with error term",abs(Real-Gauss(4)))
print("The approximation applying 5-point gauss quadrature is", Gauss(5), "with error term",abs(Real-Gauss(5)))
```

```
The approximation applying 2-point gauss quadrature is 0.15941043096671137 with error term 0.0011923631760769604
The approximation applying 3-point gauss quadrature is 0.1605953868159705 with error term 7.407326817837623e-06
The approximation applying 4-point gauss quadrature is 0.16060277751425955 with error term 1.6628528776152507e-08
The approximation applying 5-point gauss quadrature is 0.16060279411372314 with error term 2.9065194695476748e-11
[Finished in 1.1s]
```

Line 22, Column 112 Spaces: 4 Python

The program give the following gaussian quadrature approximations for the problem:

$$\begin{aligned}
\text{n=2: } \int_0^1 f(x)dx =& g(0.5773502692) + g(-0.5773502692) = 0.15941043096671137\\
\text{n=3: } \int_0^1 f(x)dx =& 0.5555555556 \cdot \Big[g(0.7745966692) + g(-0.7745966692)\Big] + 0.8888888889 \cdot g(0)\\
=& 0.1605953868159705\\
\text{n=4: } \int_0^1 f(x)dx =& 0.3478548451 \cdot \Big[g(0.8611363116) + g(-0.8611363116)\Big]\\
&+ 0.6521451549 \cdot \Big[g(0.3399810436) + g(-0.3399810436)\Big]\\
=& 0.16060277751425955
\end{aligned}$$

and

$$\begin{aligned}\text{n=5: } \int_0^1 f(x)dx =& 0.2369268850 \cdot \Big[g(0.9061798459) + g(-0.9061798459)\Big] + 0.5688888889 \cdot g(0) \\ &+ 0.4786286705 \cdot \Big[g(0.5384693101) + g(-0.5384693101)\Big] \\ =& 0.16060279411372314\end{aligned}$$

where $g(x) := f(\frac{t+1}{2}) \cdot \frac{1}{2}$.
And therefore a table for results

Table 5: Results of 4.7/1b

Applied Methods	Approximation Value	Error
2-point Gaussian Quadrature	0.15941043096671137	$1.1923631760769604 \times 10^{-3}$
3-point Gaussian Quadrature	0.1605953868159705	$7.407326817837623 \times 10^{-6}$
4-point Gaussian Quadrature	0.16060277751425955	$1.6628528776152507 \times 10^{-8}$
5-point Gaussian Quadrature	0.16060279411372314	$2.9065194695476748 \times 10^{-11}$

which shows that the more nodes involves, the more accurate the results approximate the real value.

7. Double Integration (4.8/1b)

To approximate

$$\int_0^{0.5} \int_0^{0.5} e^{y-x} dxdy (4.8/1b),$$

we have simpson's double integration (n=m=4) and gaussian double integration (n=m=3). Using values in table 4.11 we write down and codes and run the program as follows. (Meanwhile we display the running time.)

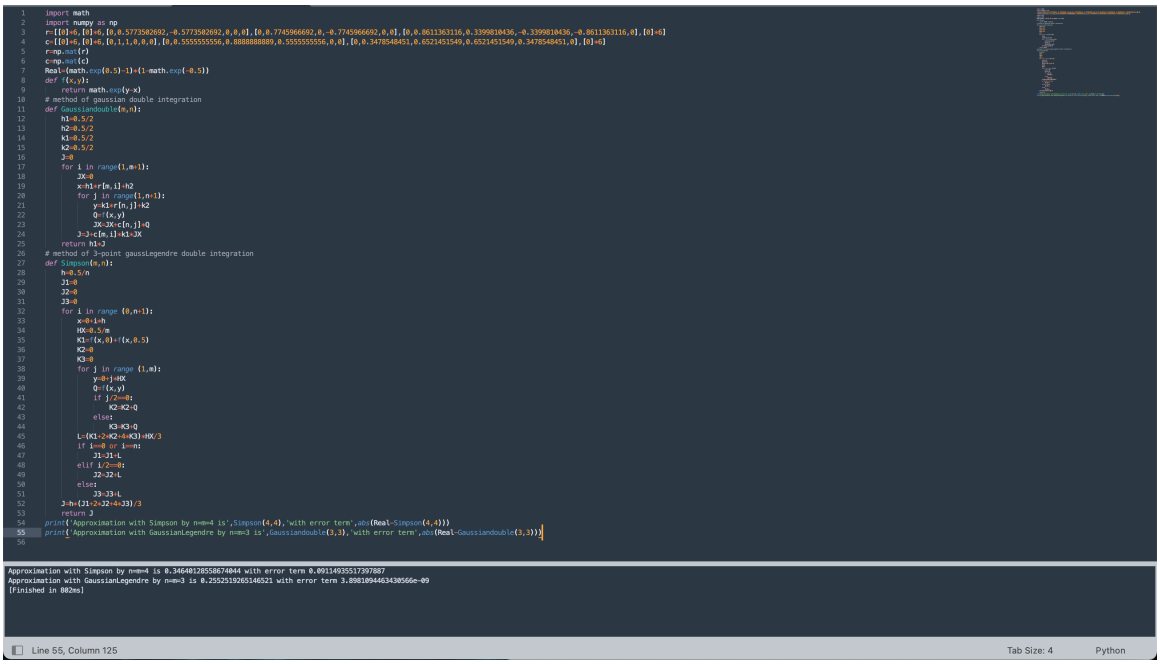


Table 6: Results of 4.8/1b

Applied Methods	Approximation Value	Error
Simpson's rule (n=m=4)	0.34640128558674044	$9.114935517397887 \times 10^{-2}$
GaussianLegendre (n=m=3)	0.2552519265146521	$3.8981094463430566 \times 10^{-9}$

And therefore a table for results above.

We see that the gaussian double inetegration involves fewer functional evaluations (9 functional evaluations) compared with 16 functional evaluations of Simpson's rule. Moreover, the gaussian double inetegration acquires itself higher-order tuncation error. Thus the result is accurate to within 3.9×10^{-9} compared to the accuracy 9×10^{-2} of Simpson's rule.

4. Code Appendix

4.1 Five-point Fomulas

```
    # method of five-point midpoint(endpoint) formula
    # midpoint version
    def Fivepointmid(a,b,c,d,h):
        Y=(a-8*b+8*c-d)/(12*h)
        return Y
    # left endpoint version
    def Fivepointendleft(a,b,c,d,e,h):
        Y=(-25*a+48*b-36*c+16*d-3*e)/(12*h)
        return Y
    # right endpoint version
    def Fivepointendright(a,b,c,d,e,h):
        Y=(-3*a+16*b-36*c+48*d-25*e)/(12*h)
        return Y


    print('The approximation for derivative at x=2.1 is', Fivepointendleft(-1.709847,
    -1.373823,-1.119214,-0.9160143,-0.7470223,0.1),'.')
    print('The approximation for derivative at x=2.2 is', Fivepointendleft(-1.373823,
    -1.119214,-0.9160143,-0.7470223,-0.6015966,0.1),'.')
    print('The approximation for derivative at x=2.3 is', Fivepointmid(-1.709847,
    -1.373823,-0.9160143,-0.7470223,0.1),'.')
    print('The approximation for derivative at x=2.4 is', Fivepointmid(-1.373823,
    -1.119214,-0.7470223,-0.6015966,0.1),'.')
    print('The approximation for derivative at x=2.5 is', Fivepointendright(-1.709847,
    -1.373823,-1.119214,-0.9160143,-0.7470223,0.1),'.')
    print('The approximation for derivative at x=2.6 is', Fivepointendright(-1.373823,
    -1.119214,-0.9160143,-0.7470223,-0.6015966,0.1),'.')
```

4.2 Composite Trapezoidal/Simposon/Midpoint Rule

```
    import math
    pi=math.pi
    Real=(3+math.exp(4)*(2*math.sin(6)-3*math.cos(6)))/13

    # method of composite trapezoidal
```

```
    def trapezoidal(a,b,n):
        h=(b-a)/n
        XI=[math.exp(4)*math.sin(6),0,0]
        for i in range(1,n): # different from python! Note that Python starts from 0.
            X=a+i*h
            XI[1]=XI[1]+math.exp(2*X)*math.sin(3*X)
        Y=h*(XI[0]+2*XI[1])/2
        return  Y

    # method of composite simposon
    def Simpson(a,b,n):
        h=(b-a)/n
        XI=[math.exp(4)*math.sin(6),0,0]
        for i in range(1,n): # different from python! Note that Python starts from 0.
            X=a+i*h
            if i%2==0:
                XI[2]=XI[2]+math.exp(2*X)*math.sin(3*X)
            else:
                XI[1]=XI[1]+math.exp(2*X)*math.sin(3*X)
        Y=h*(XI[0]+4*XI[1]+2*XI[2])/3
        return  Y

    # method of composite midpoint
    def Midpoint(a,b,n):
        h=(b-a)/(n+2)
        XI=math.exp(4)*math.sin(6)
        for i in range(1,n/2+1): # different from python! Note that Python starts from 0.
            X=a+i*h
            XI=XI+math.exp(2*X)*math.sin(3*X)
        Y=2*h*XI
        return  Y


    print('The real value is', Real)
    print('Approximation with composite trapezoidal rule by n=2168 is', trapezoidal(0,2,2168))
    print('Approximation with composite simposon rule by n=54 is', Simpson(0,2,54))
    print('Approximation with composite midpoint rule by n=3066 is', Simpson(0,2,3066))
```

4.3 Romberg Integration

```
    import math
    import numpy as np
    import sympy
    pi=math.pi

    def sam(x):
        if 0<=x<=0.1:
            return x**3+1
        elif 0.1<x<=0.2:
            return 1.001+0.03*(x-0.1)+0.3*(x-0.1)**2+2*(x-0.1)**3
        else :
            return 1.009+0.15*(x-0.2)+0.9*(x-0.2)**2+2*(x-0.2)**3

    def Romberg(a,b,n):
        # generate a 2xn matrix
        M = [[0.5]*(n+1) for _ in range(3)]
        M=np.mat(M)
        # initialization
        h=b-a
```

```
        M[1,1]=h/2*(sam(a)+sam(b))
        print("%.6f" %M[1,1])
        for i in range(2,n+1):
            temp=0
            for k in range(1,2**(i-2)+1):
                temp=temp+sam(a+(k-0.5)*h)
            M[2,1]=1/2*(M[1,1]+h*temp)
            print("%.6f" %M[2,1],end="    ")
            for j in range(2,i):
                M[2,j]=M[2,j-1]+(M[2,j-1]-M[1,j-1])/(4**(j-1)-1)
                print(M[2,j],end="    ")
            M[2,i]=M[2,i-1]+(M[2,i-1]-M[1,i-1])/(4**(i-1)-1)
            print("%.6f" %M[2,i])
            h=h/2
            for j in range(1,i+1):
                M[1,j]=M[2,j]
            if abs(M[1,i-1]-M[2,i])<0.0001:
                print("The prodecure stops at n=",i,".")
                return

    Romberg(0,0.3,100)
```

4.4 Gauss Quatrature

```
import math
import numpy as np
e=math.exp(1)
Real=2-5/e
def g(x):
    return 1/2*(((x+1)/2)**2)*math.exp(-(x+1)/2)
# input the number of points
def Gauss(n):
    if n==2:
        return g(0.5773502692)+g(-0.5773502692)
    if n==3:
        return 0.5555555556*g(0.7745966692)+0.8888888889*g(0)+0.5555555556
        *g(-0.7745966692)
    if n==4:
        return 0.3478548451*(g(0.8611363116)+g(-0.8611363116))+0.6521451549*(
        g(0.3399810436)+g(-0.3399810436))
    if n==5:
        return 0.2369268850*(g(0.9061798459)+g(-0.9061798459))+0.5688888889*g(0)
        +0.4786286705*(g(0.5384693101)+g(-0.5384693101))
print("The approximation applying 2-point gauss quadrature is", Gauss(2), "with error ...
                                                    term",abs(Real-Gauss(2)))
print("The approximation applying 3-point gauss quadrature is", Gauss(3), "with error ...
                                                    term",abs(Real-Gauss(3)))
print("The approximation applying 4-point gauss quadrature is", Gauss(4), "with error ...
                                                    term",abs(Real-Gauss(4)))
print("The approximation applying 5-point gauss quadrature is", Gauss(5), "with error ...
                                                    term",abs(Real-Gauss(5)))
```

4.5 Gauss/Simpson's double Integration

```
import math
```

```
    import numpy as np
    r=[[0]*6,[0]*6,[0,0.5773502692,-0.5773502692,0,0,0],[0,0.7745966692,0,-0.7745966692,0,0],
    [0,0.8611363116,0.3399810436,-0.3399810436,-0.8611363116,0],[0]*6]
    c=[[0]*6,[0]*6,[0,1,1,0,0,0],[0,0.5555555556,0.8888888889,0.5555555556,0,0],
    [0,0.3478548451,0.6521451549,0.6521451549,0.3478548451,0],[0]*6]
    r=np.mat(r)
    c=np.mat(c)
    Real=(math.exp(0.5)-1)*(1-math.exp(-0.5))
    def f(x,y):
        return math.exp(y-x)
    # method of gaussian double integration
    def Gaussiandouble(m,n):
        h1=0.5/2
        h2=0.5/2
        k1=0.5/2
        k2=0.5/2
        J=0
        for i in range(1,m+1):
            JX=0
            x=h1*r[m,i]+h2
            for j in range(1,n+1):
                y=k1*r[n,j]+k2
                Q=f(x,y)
                JX=JX+c[n,j]*Q
            J=J+c[m,i]*k1*JX
        return h1*J
    # method of 3-point gaussLegendre double integration
    def Simpson(m,n):
        h=0.5/n
        J1=0
        J2=0
        J3=0
        for i in range (0,n+1):
            x=0+i*h
            HX=0.5/m
            K1=f(x,0)+f(x,0.5)
            K2=0
            K3=0
            for j in range (1,m):
                y=0+j*HX
                Q=f(x,y)
                if j/2==0:
                    K2=K2+Q
                else:
                    K3=K3+Q
            L=(K1+2*K2+4*K3)*HX/3
            if i==0 or i==n:
                J1=J1+L
            elif i/2==0:
                J2=J2+L
            else:
                J3=J3+L
        J=h*(J1+2*J2+4*J3)/3
        return J
    print('Approximation with Simpson by n=m=4 is',Simpson(4,4),'with error ...
                                                    term',abs(Real-Simpson(4,4)))
    print('Approximation with GaussianLegendre by n=m=3 is',Gaussiandouble(3,3),'with error ...
                                                    term',abs(Real-Gaussiandouble(3,3)))
```