第四章作业

目录

一、问题叙述

圆形管道中流体的速度可表示为 $v(r)=10\left(1-\frac{r}{r_0}\right)^{1/n}$ 其中 v 是速度, r 是由管道中心向外的径向距离, r0 是管道的半径。那么，管道流体的体积流量 Q 可以通过下式计算： $Q=\int_0^{r_0} 2\pi r \cdot v(r)dr$

假设 r0 = 0.75 , n = 7，请采用不同的数值积分方法计算管道流体的体积流量，并分析误差。

二、问题分析

整理函数后发现，在使用余项公式计算估计误差时，由于函数的高阶导数在 x 最大值处的取值为负无穷，积分不收敛，无法求得高阶导的平均值，所以采用事后估计法分析误差。

$$\boxed{v(r)=10\left(1-\frac{r}{r_0}\right)^{\frac{1}{n}},\quad r_0=0.75,\quad n=7}$$

$$\Downarrow \text{代入}$$

$$\boxed{Q=\int_0^{r_0} 2\pi r v(r)\,dr=\int_0^{0.75} 2\pi r \times 10\left(1-\frac{r}{0.75}\right)^{\frac{1}{7}} dr}$$

在后续代码中，d 代表区间长度，能 n 代表分段数，result 表示计算结果，Et 表示真实绝对误差，ε 表示真实相对误差，Ea 表示事后误差估计。

另外，在利用 python 库函数 sympy 计算本例中积分真值时，由于系统内部存储格式的原因，无法计算到边界点 0.75 的积分值，因此通过减去微小偏移量的方法求出近似值，设置 delta=1e-8，经验证得该近似值 14.43169110 与真实值 14.43169125 的误差小于 1e-7，对结果的影响不大。

三、复合梯形

本例中两端点的函数值都为零，无法用一阶梯形公式计算，因此采用复合梯形公式，公式推导如下：

$$\boxed{T(n)=\sum_{i=0}^{n-1}\left[\frac{h}{2}\left(f(x_i)+f(x_{i+1})\right)-\frac{h^3}{12}f''(\xi_i)\right]=h\left(\frac{1}{2}f(a)+\sum_{i=1}^{n-1}f(x_i)+\frac{1}{2}f(b)\right)-\sum_{i=0}^{n-1}\frac{h^3}{12}f''(\xi_i)}$$

$$\Downarrow$$

$$\boxed{I\approx\frac{d}{2n}\times\left(f(a)+2\sum_{i=1}^{n-1}f(x_i)+f(b)\right)}$$

事后误差估计：

$$\boxed{I - T(n) \approx \frac{1}{3}\left(T(n) - T\left(\frac{n}{2}\right)\right)}$$

$$\Downarrow$$

$$\boxed{E_a \approx \frac{1}{3}\left(T(n) - T\left(\frac{n}{2}\right)\right)}$$

代码如下：

```python
import sympy as sp

x = sp.Symbol("x")
f = 2*sp.pi*x*10*sp.root((1-x/0.75), 7)
d = 0.75                        #区间长度
n = 200                         #分段数
delta = 1e-8                    #微小偏移量
xValue1 = []                    #存放n个节点的x值
for i in range(n + 1):
    xValue1.append(i * d / n)
xValue2 = []                    #存放n/2个节点的x值
for i in range(int(n/2 + 1)):
    xValue2.append(i * 2*d / n)

def trapezoidal(xValue, n0):
    fx = []                     #存放各节点的f(x)值
    for i in range(n0+1):
        fx.append(f.evalf(subs={x: xValue[i]}))
    result = d*(fx[0]+fx[n0]+2*sum(fx[i] for i in range(1, n0)))/(2*n0)
    return result
```

```python
result = trapezoidal(xValue1, n)

Ea = (trapezoidal(xValue1, n)-trapezoidal(xValue2, int(n/2))) / 3     #计算事后误差
Real = sp.integrate(f, (x, xValue1[0], xValue1[n]-delta)).evalf()     #计算真值
Et = abs(Real - result)                         #计算绝对误差
epsilon = Et / Real * 100                       #计算相对误差

print("result = %.10f, Et = %.10f, epsilon = %.2f%%, Ea = %.10f" %
      (result, Et, epsilon, Ea))
```

改变分段 n 的值，结果如下：

分段数	计算值 result	绝对误差 Et	相对误差 ε	事后误差估计 Ea
10	13.4345714533	0.9971196506	6.91%	0.4075428038
100	14.3607324254	0.0709586785	0.49%	0.0286692524
1000	14.4265953376	0.0050957662	0.04%	0.0020534162
5000	14.4308816586	0.0008094453	0.01%	0.0003260960

复合梯形法的误差减小速度与 n 的一次放成正比。

四、Simpson 三分之一法

代码如下：

```
import numpy as np
import sympy as sp
import math

x = sp.Symbol("x")
f = 2*math.pi*x*10*sp.root((1-x/0.75), 7)
xValue = np.array([0, 0.75])
d = 0.75            #区间长度
delta = 1e-8        #微小偏移量

def simpsons(xValue, f):
    fx0 = f.evalf(subs={x: xValue[0]})
    fx1 = f.evalf(subs={x: (xValue[1]+xValue[0])/2})
    fx2 = f.evalf(subs={x: xValue[1]})
    result = d*(fx0+4*fx1+fx2)/6
    return result

result = simpsons(xValue, f)

Real = sp.integrate(f, (x, xValue[0], xValue[1]-delta)).evalf()    #计算真值
Et = abs(Real - result)                     #计算绝对误差
epsilon = Et / Real * 100                   #计算相对误差

print("result = %.10f, Et = %.10f, epsilon = %.2f%%" % (result, Et, epsilon))
```

结果如下：

分段数	计算值 result	绝对误差 Et	相对误差 ε
2	10.6703055369	3.7613853050	26.06%

三分之一法固定为 3 个节点，其精度仅比同样分段数的复合梯形法略好。

五、Simpson 八分之三法

八分之三法与三分之一法雷同，只是增加一个节点，因此需要多计算一个节点的函数值和系数，代码主体如下：

```
def simpsons(xValue, f):
    fx = []          #存放各节点的f(x)值
    for i in range(n+1):
        fx.append(f.evalf(subs={x: xValue[i]}))
    result = d*(fx[0]+3*fx[1]+3*fx[2]+fx[3])/8
    return result
```

结果如下：

分段数	计算值 result	绝对误差 Et	相对误差 ε
3	11.7215883561	2.7101027478	18.78%

其误差比三分之一法略好，但精度仍然很低，一般来说更常用三分之一法，因为计算量更少。

六、复合 Simpson 公式

复合 Simpson 即分成 n 个区间反复运用三分之一法后相加，其公式为：

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(a)+f(b)+2\sum_{k=1}^{n-1} f(x_{2k})+4\sum_{k=1}^{n} f(x_{2k-1})\right]$$

事后误差估计的公式与之前类似：

$$\boxed{I - S(n) \approx \frac{1}{15}\left(S(n) - S\left(\frac{n}{2}\right)\right)}$$

$$\Downarrow$$

$$\boxed{E_a \approx \frac{1}{15}\left(S(n) - S\left(\frac{n}{2}\right)\right)}$$

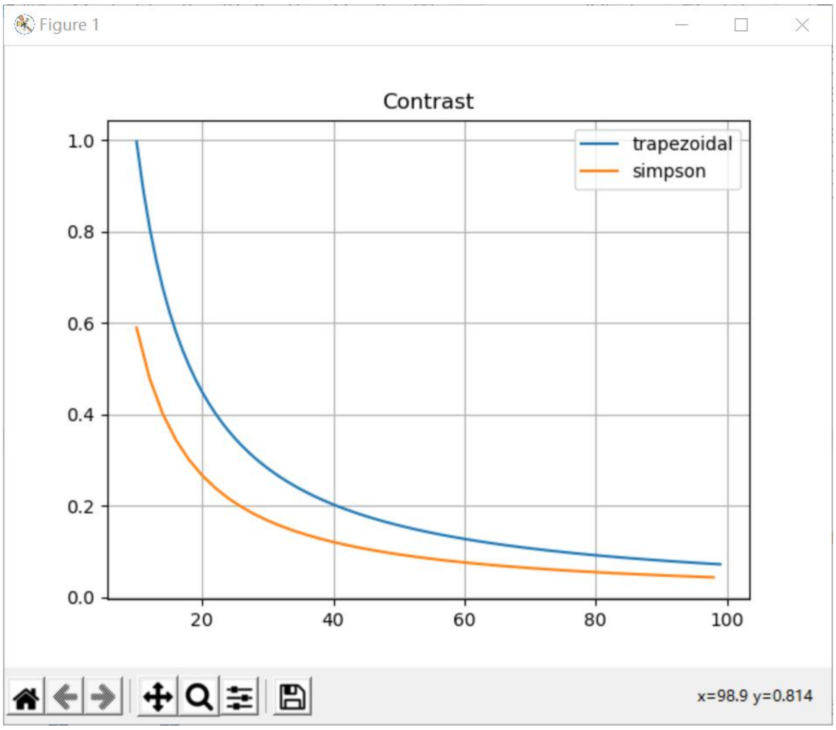
代码仍与之前类似，增加了变量 n，即分成不同的区间数量：

```
def simpsons(xValue, n0):
    fx = []          #存放各节点的f(x)值
    for i in range(n0+1):
        fx.append(f.evalf(subs={x: xValue[i]}))
    total1 = 4 * sum(fx[i] for i in range(1, n0, 2))
    total2 = 2 * sum(fx[i] for i in range(2, n0-1, 2))
    result = d/(3*n0) * (fx[0] + fx[n0] + total1 + total2)
    return result
```

改变分段 n 的值，并于梯形复合进行对比，结果如下：

分段数	计算值 result	绝对误差 Et	相对误差 ε	事后误差估计 Ea
10	13.8421142571	0.5895777919	4.09%	0.3535297836
100	14.3894016778	0.0422903712	0.29%	0.0034085463
1000	14.4286487539	0.0030432951	0.02%	0.0002450756
5000	14.4312077546	0.0004842944	0.0034%	0.0000389440

由于取点较少，无法很好比较，因此将作出二者的误差曲线图：



可以看到复合辛普森公式的误差几乎是复合梯形误差的一半，但是后续的收敛效果不明显。

七、Newton-Cotes 公式

Cotes 法在 Simpson 法的基础上继续作线性组合，使得精度进一步提高，本例中直接采用了 Cn 与 Sn 的关系进行编程：

```
def simpsons(xValue, n0):
    fx = []          #存放各节点的f(x)值
    for i in range(n0+1):
        fx.append(f.evalf(subs={x: xValue[i]}))
    total1 = 4 * sum(fx[i] for i in range(1, n0, 2))
    total2 = 2 * sum(fx[i] for i in range(2, n0-1, 2))
    result = d/(3*n) * (fx[0] + fx[n0] + total1 + total2)
    return result

def cotes(n0):
    return simpsons(xValue1, n0)*16/15 - simpsons(xValue2, int(n0/2))/15
```

结果如下：

分段数	计算值 result	绝对误差 Et	相对误差 ε
10	14.4802829574	0.0485909085	0.34%

其结果精确度更高，但方法与前两者类似，故不做过多阐述。

八、Romberg 积分

龙贝格积分同样可以由梯形公式、辛普森公式和柯特斯公式推出，其一般形式如下：

$$M(j,k) = M(j,k-1) + \frac{M(j,k-1) - M(j-1,k-1)}{4^k - 1}$$ ，其中 $M(j, 0) = T_{2^j}$

在 k > 4 时，M(j, k) - M(j, k-1)已趋近于零，再进行外推已无必要。

代码如下：

```
def trapezoidal(x0, x1, f):
    fx0 = f.evalf(subs={x: x0})
    fx1 = f.evalf(subs={x: x1})
    result = (x1-x0)*(fx0+fx1)/2
    return result

def romberg(j, k):          #根据定义推导龙贝格公式
    if k > 1:
        result = (1/(4**(k-1)-1))*(4**(k-1)*romberg(j, k-1) - romberg(j-1, k-1))
    else:
        h = (xValue[1]-xValue[0])/(2**(j-1))
        x = xValue[0]
        count = 0
        result = 0
        while count < 2**(j-1):
            result += trapezoidal(x, x+h, f)
            x += h
            count += 1
    return result
```

结果如下：

k = 4	计算值 result	绝对误差 Et	相对误差 ε
j = 4	13.7455071248	0.6861849242	4.75%
j = 6	14.2913630725	0.1403289764	0.97%
j = 8	14.4029325725	0.0287594764	0.20%
j = 10	14.4257943559	0.0058976930	0.04%
j = 12	14.4304819478	0.0012101012	0.0084%
j = 14	14.4314432463	0.0002488026	0.0017%
j = 16	14.4316403905	0.0000516584	0.0004%

可以看到阶每增加 2，绝对误差都减小一个数量级，但计算量大大增加。

九、Gauss 积分

原始一维高斯积分的被积区域被限定为[-1, 1]，公式如下：

$$\int_{-1}^{1} f(t)dt = f(t_1)w_1 + f(t_2)w_2 + \ldots + f(t_n)w_n$$

当被积区域不是[-1, 1]时，需要做如下转化：

$$\int_{a}^{b} f(x)dx = \frac{b-a}{2}\int_{-1}^{1} \varphi(t)dt$$

其中 x 与 t 存在以下的关系：

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

本例中分别使用三阶和五阶的系数进行运算，代码如下：

```
import sympy as sp

xValue = [0, 0.75]
def f(x0):
    return 2*sp.pi*x0*10*sp.root((1-x0/0.75), 7)

# 三阶和五阶的高斯系数表
GauThree = {0.7745966692: 0.555555556, 0: 0.8888888889}
GauFive = {0.9061798459: 0.2369268851, 0.5384693101: 0.4786286705, 0: 0.5688888889}
totalThree = 0.0
totalFive = 0.0

# 系数表代入公式
for key, value in GauThree.items():
    totalThree += f(((xValue[1]-xValue[0])*key + xValue[0] + xValue[1])/2) * value
    if key > 0:
        totalThree += f(((xValue[0]-xValue[1])*key + xValue[0] + xValue[1])/2) * value
totalThree = (totalThree*(xValue[1]-xValue[0])/2).evalf()

for key, value in GauFive.items():
    totalFive += f(((xValue[1]-xValue[0])*key + xValue[0] + xValue[1])/2) * value
    if key > 0:
        totalFive += f(((xValue[0]-xValue[1])*key + xValue[0] + xValue[1])/2) * value
totalFive = (totalFive*(xValue[1]-xValue[0])/2).evalf()
```

结果如下：

n	计算值 result	绝对误差 Et	相对误差 ε
三阶	14.5784579207	0.1467668168	1.02%
五阶	14.4818722842	0.0501811803	0.35%

高斯积分法的优点在于计算量少的同时保证了精度，但是改变 n 的大小时，节点和系数都需改变，编程时较为繁琐。

十、小结

梯形公式、辛普森公式、科特斯公式分别是前者的线性组合得出的精度更高的解，相对来说前两者应用更为广泛，高阶的科特斯公式使用较少，因为即使精度更高，但计算量大，编程相对不易。龙贝格公式和高斯公式都需用到节点的函数值，所以一般需要已知函数表达式，但计算量很小，精度也很高。