**北京科技大学实验报告**

学院：计算机与通信工程学院 专业：计算机科学与技术 班级： 计184

姓名： 雷方雨 学号： 41823205 实验日期： 2020 年 4 月 15 日

**《数值计算方法》课程实验**

**实验一：非线性方程求解**

1. **实验目的**

探究非线性方程的解法，比较不同解法的优劣性，针对具体问题对解法进行实践，并迭代达到指定的精度

1. **实验仪器**

设备：ASUS VivoBook S14

实验1，2，3都采用**Python**语言，使用**Anaconda**自带IDE **Spider**。

用到Math, Numpy 和 Matplotlib 库。

Numpy 用于矩阵计算，Matplotlib用于绘图。

1. **实验原理：**

二分法

若f(x)在[a, b]上连续，且f(a)f(b)<0 取，计算f(x0)

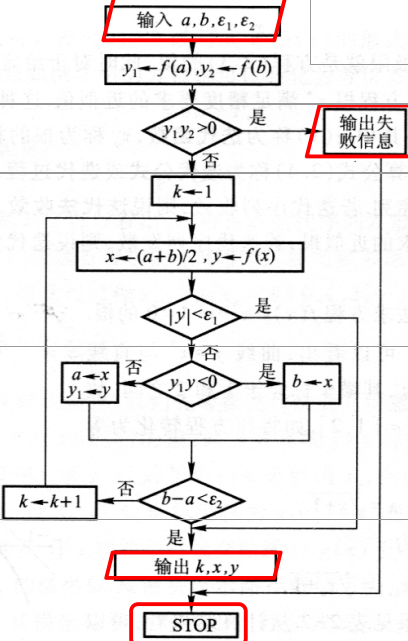
若f(a)f(x0)<0，则根位于[a, x0]取a1 = a, b1 = x0

若f(a)f(x0)>0，则根位于[x0, b]取 a1 = x0, b1 = b

取，计算f(x1)

估计所需迭代次数 

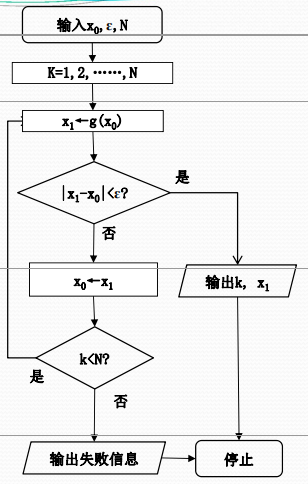




简单迭代法

将方程f(x) = 0化位另一个与它同解的方程 x = g(x)

取初值x0代入右边得到x1 = g(x0)，如果迭代收敛，则结果为所求根



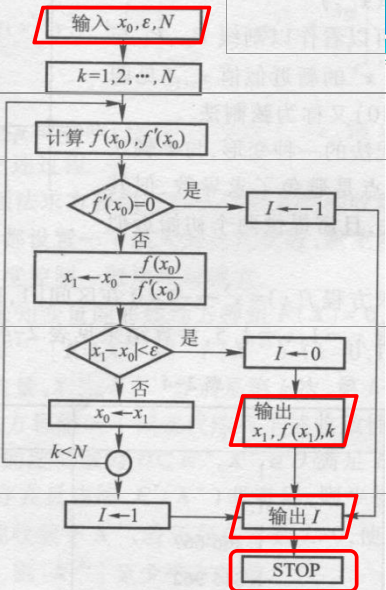
牛顿迭代法

(1)输入精度,，最大迭代次数N、初值x0，计算f(x0)、f’(x0),记为k=0

(2)若k>=N或f’(xk)=0,终止并输出失败标志

(3)计算

(4)若或,输出否则k=k+1, 转(2)



## 四、实验内容与步骤：

（1）实验内容

至少采用两种不同的算法求解在[0,1]范围内的一个根，要求两次迭代误差小于10-4。

1. 主要步骤

本实验由**Python**语言实现。

选用三种不同的算法，二分法，简单迭代法和牛顿迭代法。简单迭代法是将方程化成一个与原方程同解的方程，方程一端化成自变量x，然后判断迭代函数是否收敛，如果收敛的话，不停地迭代将使x区域一个定值，这个定值就是原方程的近似解。而牛顿迭代法是直接给出形如的迭代函数进行迭代，如果满足两个端点异号，f’(x)在[a, b]上不等于0，f’’(x) 在[a, b]上不变号且初值X0 满足f(x0)f’’(x0)>=0, 则由牛顿迭代法产生的序列单调收敛于[a, b]内的唯一根。

牛顿迭代法的收敛速度要大于简单迭代法，以下进行迭代解非线性方程组并验证收敛速度。

首先需要把问题的原函数和导数定义两个方法

def solve\_function(x):

ex = math.exp(x)

return ex + 3 \* x \*\* 3 - 2 \* x \*\* 2 + x - 2

def solve\_derivatives(x):

ex = math.exp(x)

return ex + 9 \* x \*\* 2 - 4 \* x + 1

1. 二分法

def dichotomy(left, right, eps):

print("二分法：")

print('n--------a--------b-------x-------y')

middle = (left + right) /2

count = 0

while abs(solve\_function ( middle ) ) > eps:

middle = ( left + right ) / 2

if solve\_function(left) \* solve\_function( middle ) <= 0:

right = middle

else:

left = middle

count = count + 1

print("%d---%8.6f----%8.6f---%8.6f---%8.6f"

%( count, left, right, middle, solve\_function( middle ) ) )

return count, middle

1. 简单迭代法

将x3  移项，整理，得到迭代函数如下：

，选取收敛点x = 0. 在实现上利用for循环进行迭代，知道相邻两次的误差小于10-4 。Python代码如下：

简单迭代函数：

def function\_diedai(x):

ex = math.exp(x)

return ( ( 2 \* x \*\* 2 - x + 2 -ex ) / 3 ) \*\* ( 1 / 3 );

主函数中的简单迭代法部分：

def SimpleIteration(x0, eps):

print("迭代法：")

print('n--------x--------y')

count = 0;

x1 =function\_diedai(x0)

print("%d----%8.6f----%8.6f" % (count, x0, x1))

while abs(x1 - x0)>eps:

x0 =x1

x1 = function\_diedai(x0)

count = count + 1

print("%d----%8.6f----%8.6f"%(count,x0,x1))

return count,x1

1. 牛顿迭代法

根据牛顿迭代法迭代函数的一般形式可以得到具体的迭代函数如下：

，代码如下：

def Newton(x,eps):

print("牛顿法：")

print('n--------x--------y')

count = 0;

while abs(solve\_function(x))>eps:

x = x-solve\_function(x)/solve\_derivatives(x)

count = count+1

print("%d----%8.6f----%8.6f"%(count,x,solve\_function(x)))

return count,x

完整的代码如下：

import numpy as np

import matplotlib.pyplot as plt

import math

def solve\_function(x):

ex = math.exp(x)

return ex + 3 \* x \*\* 3 - 2 \* x \*\* 2 + x - 2

def solve\_derivatives(x):

ex = math.exp(x)

return ex + 9 \* x \*\* 2 - 4 \* x + 1

def dichotomy(left, right, eps):

print("二分法：")

print('n--------a--------b-------x-------y')

middle = (left + right) /2

count = 0

while abs(solve\_function ( middle ) ) > eps:

middle = ( left + right ) / 2

if solve\_function(left) \* solve\_function( middle ) <= 0:

right = middle

else:

left = middle

count = count + 1

print("%d---%8.6f----%8.6f---%8.6f---%8.6f"

%( count, left, right, middle, solve\_function( middle ) ) )

return count, middle

def function\_diedai(x):

ex = math.exp(x)

return ( ( 2 \* x \*\* 2 - x + 2 -ex ) / 3 ) \*\* ( 1 / 3 );

def SimpleIteration(x0, eps):

print("迭代法：")

print('n--------x--------y')

count = 0;

x1 =function\_diedai(x0)

print("%d----%8.6f----%8.6f" % (count, x0, x1))

while abs(x1 - x0)>eps:

x0 =x1

x1 = function\_diedai(x0)

count = count + 1

print("%d----%8.6f----%8.6f"%(count,x0,x1))

return count,x1

def Newton(x,eps):

print("牛顿法：")

print('n--------x--------y')

count = 0;

while abs(solve\_function(x)) > eps:

x = x-solve\_function(x) / solve\_derivatives(x)

count = count + 1

print("%d----%8.6f----%8.6f"%(count, x, solve\_function(x)))

return count,x

left = 0

right = 1

eps = 0.0001

Binary\_count, answer = dichotomy(left, right, eps)

print("迭代%d次得到的根是%f"%(Binary\_count, answer))

x = 0.5

eps = 0.0001

SimpleIteration\_count, answer = SimpleIteration(x, eps)

print("迭代%d次得到的根是%f"%(SimpleIteration\_count, answer))

x = 0.5

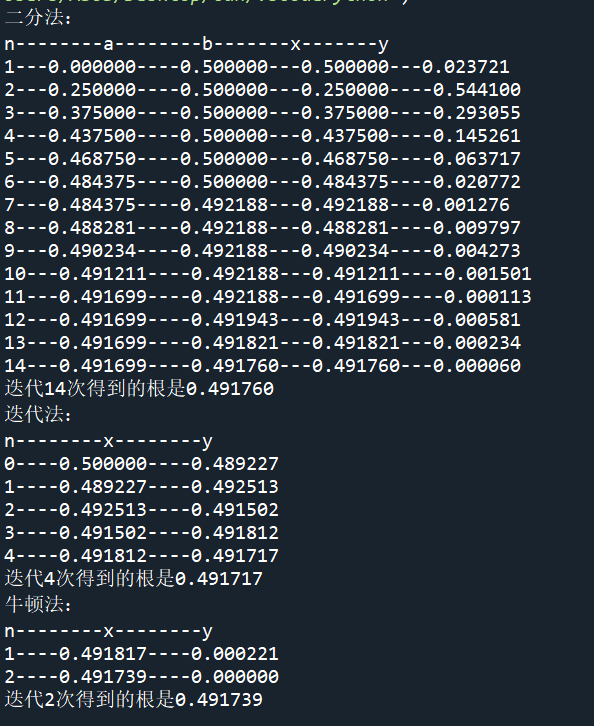
eps = 0.0001

Newton\_count, answer = Newton(x,eps)

print("迭代%d次得到的根是%f"%(Newton\_count, answer))

**五、实验数据**

执行程序，结果如下：



可见，在精度要求内，二分法14次迭代，得到结果0.491760，简单迭代法迭代4次得到的根是0.491717，牛顿迭代法迭代2次得到的根为0.491739

优劣性分析见实验结果与分析。

**六、实验数据处理：**

通过实验数据，我们可以清楚的得出看到三种方法的迭代次数，可以用来比较不同迭代方法的效率。此时，我又进行了一个额外的实验，用来比较随着精度要求的增加，不同方法的迭代次数的变化情况，以下是Python代码：

//精度要求从10-1 到10-11精度进行计算

N = 11

epss = [0.1 \*\* x for x in range(1, N)]

left = 0

right = 1

Binary\_count = []

SimpleIteration\_count = []

Newton\_count = []

for eps in epss:

count, ans = dichotomy(left, right, eps)

Binary\_count.append(count)

count, ans = Newton(0.5, eps)

Newton\_count.append(count)

count, ans = SimpleIteration(0.5, eps)

SimpleIteration\_count.append(count)

# 画图比较三种方法的性能

number=[x for x in range(1, N)]

plt.figure(figsize=(8, 6))

line1,=plt.plot(number,Binary\_count, label="Line 1",color="red",linewidth=2)

line2,=plt.plot(number,Newton\_count, label="Line 2",color="blue",linewidth=2)

line3,=plt.plot(number,SimpleIteration\_count, label="Line 3",color="green",linewidth=2)

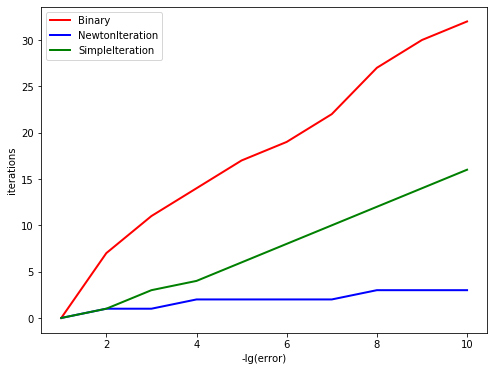
l1 = plt.legend([line1, line2,line3], ["Binary", "NewtonIteration","SimpleIteration"], loc='upper left')

plt.xlabel("-lg(error)")

plt.ylabel("iterations")

**七、实验结果与分析**

通过二分法，简单迭代法和牛顿迭代法处理同一问题，观察达到指定精度所需不同的迭代次数，二分法经过14次误差小于10-4，简单迭代法经过4次得到误差小于10-4，牛顿迭代法经过2次迭代误差小于10-4



通过这一个图表可以更加直观的看出，随着精度要求的增加，不同迭代方法所需迭代次数的变化情况，可以得到结论。从收敛速度上讲，牛顿迭代法优于简单迭代法优于二分法。

**实验二：最小二乘法拟合**

**一、实验目的**

探究非线性方程的解法，比较不同解法的优劣性，针对具体问题对解法进行实践，并迭代达到指定的精度

**二、实验仪器**

设备：ASUS VivoBook S14

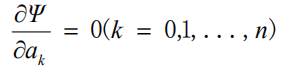
实验1，2，3都采用Python语言，使用Anaconda自带IDE Spider。

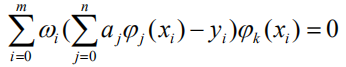
用到Math, Numpy 和 Matplotlib 库。

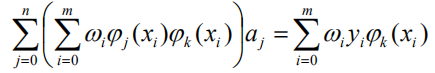
Numpy 用于矩阵计算，Matplotlib用于绘图。

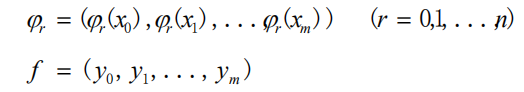
**三、实验原理**

记：

由：

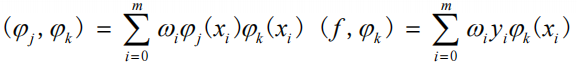
得：

即：

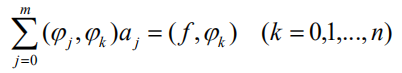


令：

定义内积：



有：



有：

最小二乘法是在确定函数形式的情况下，找出一条最靠近所有数据点的直线，其判定规则是达到最小。在求解的过程中，用最小二乘法拟合复合函数的过程实际上就是求法方程组的过程。分别写出各项的内积和，然后求解方程组

## 四、实验内容与步骤：

（1）实验内容

x:   1.0000      1.4000    1.8000      2.2000      2.6000      3.0000      3.4000      3.8000    4.2000      4.6000      5.0000

y:    4.7187     9.4496    13.3248    16.0722    17.4894    17.5794    16.6755    15.6332    16.0858    20.8430    34.4605

数据可能来自于不超过3阶多项式和指数、对数函数线性组合形成的复合函数([1, x, x2, x3, ex, ln(x)])，请采用最小二乘算法确定复合函数中各个函数项的系数(保留到小数点两位)，有能力可图示对比拟合值与真实值。

（2）主要步骤

最小二乘法是在确定函数形式的情况下，找出一条最靠近所有数据点的直线，其判定规则是达到最小。在求解的过程中，用最小二乘法拟合复合函数的过程实际上就是求发方程组的过程。

具体的Python代码如下：

def f(x, n):

if n == 0:

return 1

elif n == 1:

return x

elif n == 2:

return x \* x

elif n == 3:

return x \* x \* x

elif n == 4:

return np.exp(x)

elif n == 5:

return np.log(x)

整个法方程组的系数矩阵可以用一个三重for循环实现，等号右侧的矩阵用另一个二重循环实现

for k in range(0, 6):

for i in range(0, 6):

for j in range(0, 11):

D[k, i] = f(X[0, j], k) \* f(X[0, j], i) + D[k, i]

等式右边的矩阵的Python代码如下：

for l in range(0, 6):

for m in range(0, 11):

F[l, 0] = f(X[0, m], l) \* Y[0, m] + F[l, 0]

完整的Python代码如下：

import numpy as np

import matplotlib.pyplot as plt

def f(x, n):

if n == 0:

return 1

elif n == 1:

return x

elif n == 2:

return x \* x

elif n == 3:

return x \* x \* x

elif n == 4:

return np.exp(x)

elif n == 5:

return np.log(x)

X = np.mat([[1,1.4,1.8,2.2,2.6,3,3.4,3.8,4.2,4.6,5.0]])

Y=np.mat([[4.7187,9.4496,13.3248,16.0722,17.4894,17.5794,16.6755,15.6332,16.0858,20.8430,34.4605]])

D = np.mat(np.zeros((6,6)))

F = np.mat(np.zeros((1,6)).T)

for k in range(0, 6):

for i in range(0, 6):

for j in range(0, 11):

D[k, i] = f(X[0, j], k) \* f(X[0, j], i) + D[k, i]

for l in range(0, 6):

for m in range(0, 11):

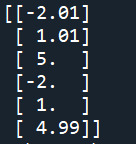
F[l, 0] = f(X[0, m], l) \* Y[0, m] + F[l, 0]

ans = np.around(D.I \* F, decimals = 2)

print(ans)

**五、实验数据**

得出的结果如下：



所以，拟合出的复合函数应为：



**六、实验数据处理**

我们已经得到了所要求的原函数，现在要对他进行检验，将拟合值与真实值进行比较，Python代码如下：

def function(x):

return -2.01+1.01\*x+5\*x\*x-2\*x\*x\*x+1\*math.exp(x)+4.99\*math.log(x)

print("x-------------拟合值--------------真实值-------------差")

for i in range(0, 11):

print("%f------%f----------%f--------%f"%(X[i], Y[i], function(X[i]), Y[i]-function(X[i])))

plt.figure(figsize=(9,6),dpi=100)

X = [1,1.4,1.8,2.2,2.6,3,3.4,3.8,4.2,4.6,5.0]

Y=[4.7187,9.4496,13.3248,16.0722,17.4894,17.5794,16.6755,15.6332,16.0858,20.8430,34.4605]

plt.plot(X,Y,'o',color = 'b',label='scatter')

x = np.linspace(0,5,200)

y=ans[0][0]+ans[1][0]\*f(x,1)+ans[2][0]\*f(x,2)+ans[3][0]\*f(x,3)+ans[4][0]\*f(x,4)+ans[5][0]\*f(x,5)

plt.plot(x,y,color = 'g',label='fitting curve')

plt.legend()

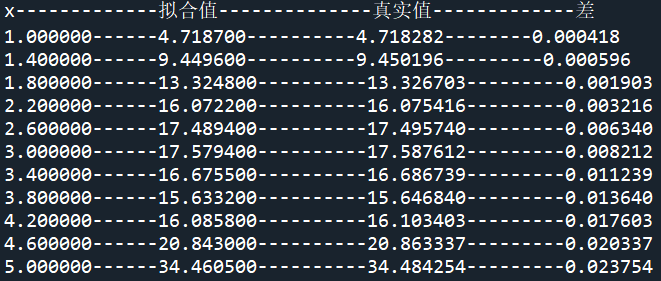
plt.show()

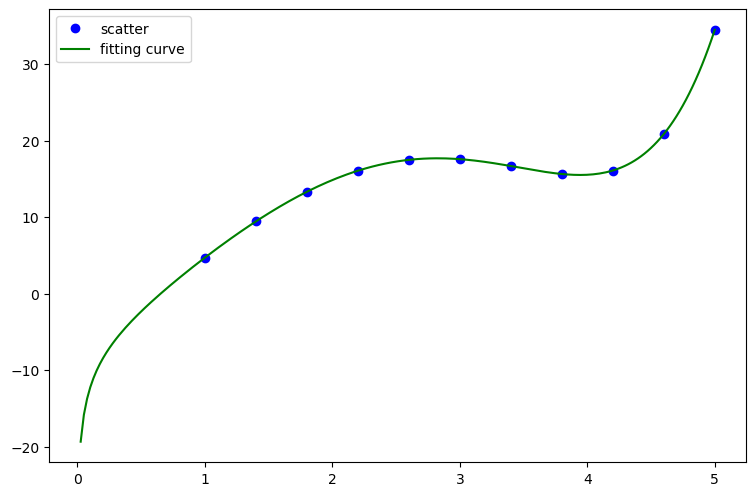
**七、实验结果与分析**

根据最小二乘法计算得到的函数式为：



根据此函数算出的真实值与拟合值相比较，发现随着x的增大，真实值与拟合值的差逐渐增大，下图为计算结果和拟合曲线图。





**实验三：数值积分**

**一、实验目的**

探究多种积分的数值解法，根据给定的精度，选择恰当的数值积分方法，确定迭代步数与步长，分析积分数值解法的优劣性

1. **实验仪器：**

设备：ASUS VivoBook S14

实验1，2，3都采用Python语言，使用Anaconda自带IDE Spider。

用到Math, Numpy 和 Matplotlib 库。

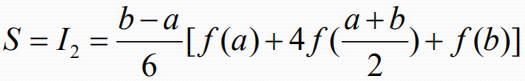
Numpy 用于矩阵计算，Matplotlib用于绘图。

1. **实验原理**

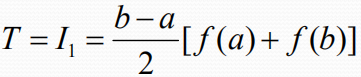
牛顿-科茨公式

将[a, b]等分成n等分，步长为h =(b-a)/n。

最重要的是n=1、2、4时的3个公式。

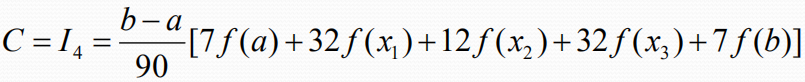


梯形公式



辛普森公式

科茨公式



在积分难求出解析解时通常用数值解的方法求积分结果的近似值，在本题的实现方法上选用复合梯形公式，复合辛普森公式，复合科茨公式进行数值积分的计算，并从步数n=1,2,4, 步长h = 1, 1/2, 1/4 开始逐步细化步长。

## 四、实验内容与步骤：

（1）实验内容

选择一种数值积分方法求解下列函数在区间[0,2]内的积分值，要求精度小于10-5：

，给出迭代步数和最终的步长，并分析你所采用方法的优缺点，有能力可对比不同算法。

（2）主要步骤

在积分难求出解析解时通常用数值解的方法求积分结果的近似值，在本题的实现方法上选用复合梯形公式，复合辛普森公式，复合科茨公式进行数值积分的计算，并从步数n=2, 步长h = 1 / 2 开始逐步细化步长。

首先定义一个题目中给出的待积分的函数，Python代码如下：

def f(x):

return math.sqrt(x - 3 \* x \* x + 2 \* x \* x \* x + math.exp(x))

编写复合辛普森公式，将其定义在一个独立的函数中

def ReiterationOfSimpson(a, b, n):

h = (b - a) / n

fa = f(a)

fb = f(b)

s1 = 0.0

s2 = 0.0

for k in range(1, n):

xk = a + k \* h

s1 = s1 + f(xk)

for j in range(0, n):

xj = a + (j + 0.5) \* h

s2 = s2 + f(xj)

sn = h / 6 \* (fa + fb + 2 \* s1 + 4 \* s2)

return sn

编写复合科茨公式，将其定义在一个独立的函数中

def Cotes(a, b, n):

h = (b - a) / n

fa = f(a)

fb = f(b)

s1 = 0.0

s2 = 0.0

s3 = 0.0

s4 = 0.0

for k in range(1, n):

xk = a + k \* h

s1 = s1 + f(xk)

for j in range(0, n):

xj = a + (j + 0.5) \* h

s2 = s2 + f(xj)

for t in range(0, n):

xt = a + (t + 0.25) \* h

s3 = s3 + f(xt)

for w in range(0, n):

xw = a + (w + 0.75) \* h

s4 = s4 + f(xw)

sn = h / 90 \* (7 \* fa + 32 \* s3 + 12 \* s2 + 32 \* s4 + 14 \* s1 + 7 \* fb)

return sn

编写复合梯形公式，将其定义在一个独立的函数中

def Trapezoid(a, b, n):

h = (b - a) / n

fa = f(a)

fb = f(b)

s1 = 0.0

for k in range(1, int(n)):

xk = a + k \* h

s1 = s1 + f(xk)

sn = h / 2 \*(fa + 2 \* s1 + fb)

return sn

最后在主函数中定义初始步数为2，初始步长h为0.5，并通过while循环逐步增大n，直到相邻两次误差小于10-5 ，具体Python代码如下：

n = 2

h = 1 / n

S = []

i = 0

print("===============复合Simpson公式法=============\n\n")

print("迭代步数\t步长\t积分数值\t\t误差\n")

S.append(ReiterationOfSimpson(0, 2, n))

print("%d\t%f\t%f\n"%(n, h, S[0]))

value\_Simpson.append(S[0])

n = n + 1

h = 1 / n

S.append(ReiterationOfSimpson(0, 2, n))

print("%d\t%f\t%f\t%f\n"%(n, h, S[1], S[1] - S[0]))

value\_Simpson.append(S[1])

i = 2

while(abs(S[i-1] - S[i-2]) >=1e-5):

n = i + 2

h = 1 / n

S.append(ReiterationOfSimpson(0, 2, n))

print("%d\t%f\t%f\t%f\n"%(n, h, S[i], S[i] - S[i-1]))

value\_Simpson.append(S[i])

i = i + 1

print()

n = 2

h = 1 / n

s = []

i = 0

print("===============复合科茨公式法==============\n\n")

print("迭代步数\t步长\t积分数值\t\t误差\n")

s.append(Cotes(0, 2, n))

print("%d\t%f\t%f\n"%(n, h, s[0]))

value\_Cotes.append(s[0])

n = n + 1

h = 1 / n

s.append(Cotes(0, 2, n))

print("%d\t%f\t%f\t%f\n"%(n, h, s[1], s[1] - s[0]))

value\_Cotes.append(s[1])

i = 2

while(s[i-1] - s[i-2] >=1e-5):

n = i + 2

h = 1 / n

s.append(Cotes(0, 2, n))

print("%d\t%f\t%f\t%f\n"%(n, h, s[i], s[i] - s[i-1]))

value\_Cotes.append(s[i])

i = i + 1

print()

n = 2

h = 1 / n

s = []

i = 0

print("===============复合梯形公式法==============\n\n")

print("迭代步数\t步长\t积分数值\t\t误差\n")

s.append(Trapezoid(0, 2, n))

print("%d\t%f\t%f\n"%(n, h, s[0]))

value\_Trapezoid.append(s[0])

n = 4

h = 1 / n

s.append(Trapezoid(0, 2, n))

print("%d\t%f\t%f\t%f\n"%(n, h, s[1], s[1] - s[0]))

value\_Trapezoid.append(s[1])

i = 2

while(abs(s[i-1] - s[i-2]) >=1e-5):

n = math.pow(2.0, i + 1)

h = 1 / n

s.append(Trapezoid(0, 2, n))

print("%d\t%f\t%f\t%f\n"%(n, h, s[i], s[i] - s[i-1]))

value\_Trapezoid.append(s[i])

i = i + 1

**五、实验数据**

程序执行的结果如下：



**六、实验数据处理：**

为了比较三种方法的收敛速度，使用以下Python代码作图：

iteration = []

for i in range(1, 10):

iteration.append(i)

for i in range(4, 9):

value\_Cotes.append(value\_Cotes[3])

for i in range(8, 9):

value\_Simpson.append(value\_Simpson[3])

plt.figure(figsize=(9,6),dpi=100)

plt.plot(iteration, value\_Trapezoid,color='red',label='Trapezoid')

plt.plot(iteration, value\_Simpson,color='blue',label='Simpson')

plt.plot(iteration, value\_Cotes,color='black',label='Cotes')

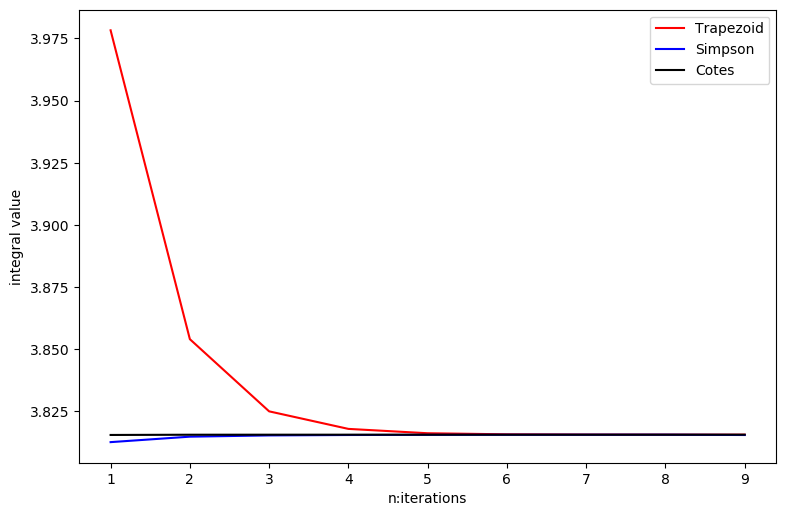
plt.legend()

plt.xlabel('n:iterations')

plt.ylabel('integral value')

**七、实验结果与分析：**

经过三种方法的计算，三种方法分别在迭代步长为9，5，512时达到了精度要求，此时的积分值约为3.815587， 3.815599， 3.815603



通过用数值积分方法求解一个函数在一个积分区间的积分值,并根据具体的精度。要求给出迭代步数和最终步长。在方法上我选用了复合Simpson 公式法，复合梯形公式法和复合Cotes公式法。复合Simpson公式在Simpson 公式的基础上提高了求积的精度，将[a, b]等分成 n个子区间，在每个子区间上使用低阶求积公式计算，然后把所有子区间上的计算结果求和。复合Simpson公式的优点在于通过增加子区间的个数可以缩小误差提高精度，这一点比单纯的Simpson公式，梯形公式和Cotes公式要好。但是缺点在于复合Simpson公式是4阶收敛的，在收敛速度上没有复合Cotes公式收敛的快，所以需要n=9时才能达到精度要求。

**结论（讨论）**

1. **实验结论**

在本次数值计算实验课中一共完成了三个实验，分别对应课程所学中三章的

内容，分别复习并实践了非线性方程的迭代解法，插值与拟合，积分的数值解法等内容。这三个实验涉及到了数值计算方法的主要内容，熟悉了数值计算方法的理论知识，并加以应用，在有一定创新度并结合各种具体编程环境的基础上，在实践中体会到了数值计算方法在实际问题中的作用。

在具体实现上，分别用Python语言实现，锻炼了把数值计算方法结合到不同应用场景的能力，为今后在各领域的使用打下基础。

在具体的实验上，在第一个实验中，应用了简单迭代法和牛顿迭代法解常见

的非线性方程，熟悉了各种非线性方程的解法，包括二分法，简单迭代法，牛顿迭代法弦截法和牛顿下山法等。其中应用二分法，简单迭代法和牛顿迭代法求解了题目中的问题，理解了两者的区别，牛顿迭代法直接通过迭代函数的一般形式给出具体的迭代函数，且收敛速度比简单迭代法要快。

在第二个实验中，应用了最小二乘法拟合一个由不超过3阶多项式和指数、对数函数线性组合形成的复合函数，熟悉了最小二乘法的概念及求解方法，通过构造法方程组来求解最小二乘法拟合的问题，并在一定程度上了解了最小二乘法拟合后平方误差的计算方法。

在第三个实验中，应用了数值积分方法中的复合梯形公式，复合Simpson公式法，复合Cotes方法解一个函数在固定区间中的积分值。由于复杂的函数在一定情况下难以找到解析解，所以要通过数值积分的方法求数值解。在求数值解的方法上主要有梯形公式，Simpson公式，Cotes 公式，复合梯形公式，复合Simpson公式，复合Cotes公式等，其中复合的公式通过在区间内等分子区间提高数值积分的精度，子区间个数越多，精度越高，直到达到目标的精度为止。熟悉并理解了相关的理论知识并加以实践，在一定程度上掌握了相关的方法。

1. **讨论**

在本次数值计算实验课程中，完成了课程中要求的实验，进一步掌握了实验中涉及的知识点，包括非线性方程的解法，最小二乘法拟合，数值积分法等等,但是对于实验题目中未涉及到的内容仍有些掌握不牢，比如说线性方程组的解法,插值，常微分方程的数值解法等等，所以我认为实验课中涉及的知识点可以覆盖到各章最好，这将在熟悉知识点上提供很大的帮助。