

# **计算方法实验报告**

**姓名：吴桐**

**学号：200111132**

**院系：计算机技术与科学**

**专业：计算机技术与科学**

**班级：11**

# 实验报告一

第一部分：问题分析 *(描述并总结出实验题目)*

## 实验题目 5 高斯(Gauss)列主元消去法

高斯 (Gauss) 列主元消去法：对给定的  $n$  阶线性方程组  $Ax=b$ ，首先进行列主 元消元过程，然后进行回代过程，最后得到解或确定该线性方程组是奇异的。如果系数矩阵的元素按绝对值在数量级方面相差很大，那么，在进行列主元消元过程前，先把系数矩阵的元素进行行平衡：系数矩阵的每行元素和相应的右端向量元素同除以该行元素绝对值最大的元素。这就是所谓的平衡技术。然后再进行列主元消元过程。

需要我们利用高斯列主元消除法的算法，设计程序，通过输入规定的系数矩阵以及右向量，可以返回一个解向量。

对于指导书中的 4\*2 个实例进行计算验证，并得出结果

## 第二部分：数学原理

高斯消元法的核心在于得到一个上三角矩阵，也就是要消去一个下三角矩阵。我们选择用对角线上的元素，进行等价变化，消除下方列的元素。而列主元的要求是为了增加算法的精度。得到上三角矩阵之后，我们从底向上回代便于我们求出结果

主要算法如下：

默认系数矩阵为 $n$ 阶矩阵

对于 $k = 1, 2, \dots, n - 1$

get  $P$  ( $k \leq p \leq n$ ) and  $|a_{pk}| = \max_{k \leq j \leq n} |a_{jk}|$

if  $p! = k$  交换 $pk$

如果 $\max = 0$ ， 则矩阵奇异

for  $i$  in  $k + 1$  to  $n$  for  $j$  in  $k + 1$  to  $n$

$$a_{ij} = a_{ij} - a_{kj} m_{ik}$$

$$b_i = b_i - b_k m_{ik}$$

从 $x_n = \frac{b_n}{a_{nn}}$  开始回代

$$x_k = \left( b_k - \sum_{j=k+1}^n a_{kj} x_j \right) / a_{kk}$$

### 第三部分：程序设计流程

问题1 (1) (2) (3) (4) 在/lab5/lab5\_1.ipynb

问题2 (1) (2) (3) (4) 在/lab5/lab5\_2.ipynb

主要高斯列主元消除法的函数 def

```
def GuassP(A,b)->np.ndarray:
    r=len(A)
    c=len(A[0])

    # 如果输入的是习惯的np.array, 则转为list进行计算, 因为最后结果会有一点不一样
    if isinstance(A,np.ndarray):
        A=A.tolist()
    if isinstance(b,np.ndarray):
        b=b.flatten().tolist()
    print("原系数矩阵: ",np.matrix(A))
    print("原右端项: ",b)
    # 开始消元
    for k in range(r-1):
        max_n=abs(A[k][k])
        max_i=k
        for i in range(k+1,r):
            if abs(A[i][k])>max_n:
                max_n=abs(A[i][k])
                max_i=i
        if max == 0:
            print("系数矩阵为奇异矩阵, 无解")
            return False, []
        # 交换行
        if max_i != k:
            A[k],A[max_i]=A[max_i],A[k]
            b[k],b[max_i]=b[max_i],b[k]

        for i in range(k+1,r):
            m=A[i][k]/A[k][k]
            for j in range(k,c):
                A[i][j]=A[i][j]-m*A[k][j]
            b[i]=b[i]-m*b[k]
    print("消元后系数矩阵: ",np.matrix(A))
    print("消元后右端项: ",b)
    if A[r-1][c-1]==0:
        print("系数矩阵为奇异矩阵, 无解")
        return False, []

    # 初始化返回答案矩阵
    ans=np.zeros(r)
    # 开始回代求解
    for i in range(r-1,-1,-1):
        ans[i]=b[i]
        for j in range(i+1,r):
            ans[i]=ans[i]-A[i][j]*ans[j]
        ans[i]=ans[i]/A[i][i]
    print("回代求解后的结果: ",ans)
    return True,ans
```

然后输入指导书给的实例，输入进函数得到结果：

(以下为问题 1, 2) 其中两个线性方程组的例子，其余雷同

```
A=[[197,305,-206,-804],  
[46.8,71.3,-47.4,52],  
[88.6,76.4,-10.8,802],  
[1.45,5.9,6.13,36.5]]  
b=[136,11.7,25.1,6.60]  
jud,ans1_1=GuassP(A,b)  
if jud:  
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

```
A=[[197,305,-206,-804],  
[46.8,71.3,-47.4,52],  
[88.6,76.4,-10.8,802],  
[1.45,5.9,6.13,36.5]]  
b=[136,11.7,25.1,6.60]  
jud,ans1_1=GuassP(A,b)  
if jud:  
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

打印结果实例：

```
原系数矩阵: [[0.4096 0.1234 0.3678 0.2943]  
 [0.2246 0.3872 0.4015 0.1129]  
 [0.3645 0.192 0.3781 0.0643]  
 [0.1784 0.4002 0.2786 0.3927]]  
原右端项: [1.1951, 1.1262, 0.9989, 1.2499]  
消元后系数矩阵: [[ 4.09600000e-01 1.23400000e-01 3.67800000e-01 2.94300000e-01]  
 [ 0.00000000e+00 3.46453516e-01 1.18405859e-01 2.64518555e-01]  
 [-2.77555756e-17 0.00000000e+00 9.06146128e-02 -2.92442479e-01]  
 [ 0.00000000e+00 0.00000000e+00 0.00000000e+00 -1.87057257e-01]]  
消元后右端项: [1.1951, 0.7293779296875, -0.2018278661797387,  
 -0.18705725686919192]  
回代求解后的结果: [1. 1. 1. 1.]  
解为(保留四位小数): [1.0, 1.0, 1.0, 1.0]
```

## 第四部分：实验结果、结论与讨论

结果也可以看代码文件里面

### 问题 1

(1)

(1):

```
A=[[0.4096,0.1234,0.3678,0.2943],[0.2246,0.3872,0.4015,0.1129],[0.3645,0.1920,0.  
3781,0.0643],[0.1784,0.4002,0.2786,0.3927]]  
b=[1.1951,1.1262,0.9989,1.2499]  
jud,ans1_1=GuassP(A,b)  
if jud:  
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

✓ 0.6s

Python

```
消元后系数矩阵: [[ 4.09600000e-01  1.23400000e-01  3.67800000e-01  2.94300000e-01]  
[ 0.00000000e+00  3.46453516e-01  1.18405859e-01  2.64518555e-01]  
[-2.77555756e-17  0.00000000e+00  9.06146128e-02 -2.92442479e-01]  
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00 -1.87057257e-01]]  
消元后右端项: [1.1951, 0.7293779296875, -0.2018278661797387,  
-0.18705725686919192]  
回代求解后的结果: [1. 1. 1. 1.]  
解为(保留四位小数): [1.0, 1.0, 1.0, 1.0]
```

(2)

(2):

```
A=[[136.01,90.860,0,0],  
[90.860,98.810,-67.590,0],  
[0,-67.590,132.01,46.260],  
[0,0,46.26,177.17]]  
b=[226.87,122.08,110.68,223.43]  
jud,ans1_1=GuassP(A,b)  
if jud:  
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

✓ 0.6s

Python

```
消元后系数矩阵: [[ 1.36010000e+02  9.08600000e+01  0.00000000e+00  0.00000000e+00]  
[ 0.00000000e+00 -6.75900000e+01  1.32010000e+02  4.62600000e+01]  
[ 0.00000000e+00  0.00000000e+00  4.62600000e+01  1.77170000e+02]  
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00 -1.36479429e-01]]  
消元后右端项: [226.87, 110.68, 223.43, -0.13647942935829604]  
回代求解后的结果: [1. 1. 1. 1.]  
解为(保留四位小数): [1.0, 1.0, 1.0, 1.0]
```

(3)

(3)

+ 代码 + 标记

```
A=[[1,1/2,1/3,1/4],  
[1/2,1/3,1/4,1/5],  
[1/3,1/4,1/5,1/6],  
[1/4,1/5,1/6,1/7]]  
b=[25/12,77/60,57/60,319/420]  
jud,ans1_1=GuassP(A,b)  
if jud:  
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

✓ 0.4s

Python

消元后系数矩阵: [[ 1.00000000e+00 5.00000000e-01 3.3333333e-01 2.50000000e-01]  
[ 0.00000000e+00 8.3333333e-02 8.88888889e-02 8.3333333e-02]  
[ 0.00000000e+00 0.00000000e+00 -5.5555556e-03 -8.3333333e-03]  
[ 0.00000000e+00 0.00000000e+00 0.00000000e+00 3.57142857e-04]]  
消元后右端项: [2.0833333333333335, 0.2555555555555554, -0.01388888888888729,  
0.0003571428571429204]  
回代求解后的结果: [1. 1. 1. 1.]  
解为(保留四位小数): [1.0, 1.0, 1.0, 1.0]

(4)

(4)

```
A=[[10,7,8,7],  
[7,5,6,5],  
[8,6,10,9],  
[7,5,9,10]]  
b=[32,23,33,31]  
jud,ans1_1=GuassP(A,b)  
if jud:  
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

✓ 0.4s

Python

消元后系数矩阵: [[10. 7. 8. 7. ]  
[ 0. 0.4 3.6 3.4 ]  
[ 0. 0. 2.5 4.25]  
[ 0. 0. 0. 0.1 ]]  
消元后右端项: [32, 7.399999999999999, 6.749999999999989, 0.1000000000000052]  
回代求解后的结果: [1. 1. 1. 1.]  
解为(保留四位小数): [1.0, 1.0, 1.0, 1.0]

问题 2

(1)

(1):

+ 代码 + 标记

```
A=[[197,305,-206,-804],  
[46.8,71.3,-47.4,52],  
[88.6,76.4,-10.8,802],  
[1.45,5.9,6.13,36.5]]  
b=[136,11.7,25.1,6.60]  
jud,ans1_1=GuassP(A,b)  
if jud:  
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

✓ 0.6s

Python

消元后系数矩阵: [[ 197. 305. -206. -804. ]]

```
[ 0. -60.77258883 81.84771574 1163.59593909]  
[ 0. 0. 12.56885167 112.40049782]  
[ 0. 0. 0. 221.02959897]]
```

消元后右端项: [136, -36.065482233502536, 3.4298804730960057, -19.916647727756796]

回代求解后的结果: [ 0.95367911 0.32095685 1.07870808 -0.09010851]

解为(保留四位小数): [0.9537, 0.321, 1.0787, -0.0901]

(2)

(2)

```
A=[[0.5398,0.7161,-0.5554,-0.2982],  
[0.5257,0.6924,0.3565,-0.6255],  
[0.6465,-0.8187,-0.1872,0.1291],  
[0.5814,0.94,-0.7779,-0.4042]]  
b=[0.2058,-0.0503,0.1070,0.1859]  
jud,ans1_1=GuassP(A,b)  
if jud:  
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

✓ 0.4s

Python

消元后系数矩阵: [[ 0.6465 -0.8187 -0.1872 0.1291 ]]

```
[ 0. 1.67626014 -0.6095503 -0.52030014]  
[ 0. 0. 1.00258548 -0.30892459]  
[ 0. 0. 0. 0.06231529]]
```

消元后右端项: [0.107, 0.08967447795823666, -0.20996203473079655,  
0.06459224565073099]

回代求解后的结果: [0.5161773 0.41521947 0.1099661 1.03653922]

解为(保留四位小数): [0.5162, 0.4152, 0.11, 1.0365]

(3)

- A=[[10,1,2],[1,10,2],[1,1,5]]  
b=[13,13,7]

```
jud,ans1_1=GuassP(A,b)  
if jud:  
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

✓ 0.6s

Python

```
消元后系数矩阵: [[10.          1.          2.          ]
 [ 0.          9.9         1.8         ]
 [ 0.          0.          4.63636364]]
消元后右端项: [13, 11.7, 4.6363636363637]
回代求解后的结果: [1. 1. 1.]
解为(保留四位小数): [1.0, 1.0, 1.0]
```

(4)

(4)

```
A=[[4,-2,-4],[-2,17,10],[-4,10,9]]
b=[-2,25,15]

jud,ans1_1=GuassP(A,b)
if jud:
    print("解为(保留四位小数):",[round(a,4) for a in ans1_1])
```

✓ 0.6s

Python

```
消元后系数矩阵: [[ 4. -2. -4.]
 [ 0. 16.  8.]
 [ 0.  0.  1.]]
消元后右端项: [-2, 24.0, 1.0]
回代求解后的结果: [1. 1. 1.]
解为(保留四位小数): [1.0, 1.0, 1.0]
```

总结:

解的速度还是挺快的，不过还是比不过 sympy 自带的解线性方程组，不过结果也相对经得起验证

## 实验报告二

第一部分：问题分析 *(描述并总结出实验题目)*

### 实验题目 4 牛顿(Newton)迭代法

求非线性方程  $f(x)=0$  的根 \* $x$ , 利用牛顿迭代法计算公式

在迭代过程中，需要设置相应的迭代次数和迭代的精度，用于控制迭代的停止，以及判断是否得出结果

于此同时我们要考虑失败的因素：

如果迭代次数超过预定，可能方法并不适用

如果分母数过小，可能使得数据越界，数据不再可靠，方法失败

我们同时利用  $f(X)=0$  以及  $Tol$  (两次迭代值的差值) 小于预设精度后，我们认为得到了一个足够精确的答案。

## 第二部分：数学原理

我们选择  $(x, f(x))$  点，以  $f'(x)$  为斜率做一条直线

$$0 = (x - x_0) \cdot f'(x_0) + f(x_0)$$

解得的  $x$  通常更加靠近  $f(x)=0$  的  $x$  值

所以我们有了迭代公式

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

最后的结果由于实际性，我们需要设定一个收敛界限，到一定精度的时候就可以停止收敛  
算法实际实现的过程如下：

*for n in 1 to N*

$$F(x) = f(x_0), DF = f'(x_0)$$

*if abs(F) < ε<sub>1</sub> output x<sub>0</sub>*

*if abs(DF) < ε<sub>2</sub> output wrong*

$$x_1 = x_0 - \frac{F}{DF}$$

$$Tol = \text{abs}(x_1 - x_0)$$

*if Tol < ε<sub>1</sub> output x<sub>1</sub>*

$$x_0 = x_1$$

### 第三部分：程序设计流程

问题 1 (1) (2) 代码/lab5/lab5\_1.ipynb

问题 2 (1) (2) 代码/lab5/lab5\_2.ipynb

核心牛顿迭代函数定义：（后面针对特殊题二重根有单独的修复优化）  
思考题和总结在最后

```
def Newton(N,e_1,e_2,init,exp:sp.Expr):  
  
    """  
    param N: number of iterations  
    param e_1: first er  
    param e_2: second er  
    param init: initial value  
    param f: exp  
    """  
    cur = init  
    dexp=sp.diff(exp,x)  
  
    print(f"初始值为{round(cur,9)}")  
  
    for i in range (N):  
        F=exp.evalf(subs={x:cur})  
  
        DF=dexp.evalf(subs={x:cur})  
  
        if abs(F)<e_1:  
            print("f(x)的值小于e_1, 结束迭代")  
            return i,cur,True  
  
        elif abs(DF)<e_2:  
            print("f'(x)的值小于e_2, 结束迭代")  
            return i,-1,False  
  
        next=cur-F/DF  
  
        Tol=abs(next-cur)  
  
        if Tol<e_1:  
            print(f"迭代差小于e_1, 结束迭代")  
            return i,cur,True  
  
        cur=next  
  
        print(f"第{i+1}次迭代, 当前值x{i}为{cur}, 误差为{Tol}  
, 继续迭代")  
    return i,-1,False
```

对于给定函数的输入案例：

问题 1 (2)

根据题目构建表达式后，按照构建核心迭代函数传参后得到结果，并且打印过程

### 问题1 (2)

$$e^{-x} - \sin x = 0$$

$$\varepsilon_1 = 10^{-6} \quad \varepsilon_2 = 10^{-4} \quad N = 10 \quad x_0 = 0.6$$

```
↳ x=sp.Symbol('x')
    exp2=sp.exp(-x)-sp.sin(x)
    idx,ans,jud = Newton(N=10,e_1=1e-6,e_2=1e-4,init=0.6,
    exp=exp2)
    if jud:
        print(f"第{idx+1}次迭代结束, 结果为{ans}")
        print(f"保留四位小数为{round(ans,4)}")
    else:
        print("迭代结束, 迭代失败")

    print("利用内置函数检验", sp.nsolve(exp2,x,0.6))

[23] ✓ 0.1s
```

Python

```
… 初始值为0.6
第1次迭代, 当前值x0为0.588479518996639, 误差为0.0115204810033610, 继续迭代
第2次迭代, 当前值x1为0.588532742847979, 误差为0.0000532238513399896, 继续迭代
f(x)的值小于e_1, 结束迭代
第3次迭代结束, 结果为0.588532742847979
保留四位小数为0.5885
利用内置函数检验 0.588532743981861
```

## 第四部分：实验结果、结论与讨论

结果也可以看代码文件里面

问题 1：

问题1 (1)

$$\cos x - x = 0$$
$$\varepsilon_1 = 10^{-6} \quad \varepsilon_2 = 10^{-4} \quad N = 10 \quad x_0 = \frac{\pi}{4}$$

▷ ✓  $x=\text{sp.Symbol('x')}$  ...

... 初始值为 0.785398163  
第1次迭代，当前值x0为 0.739536133515238，误差为 0.0458620298822100，继续迭代  
第2次迭代，当前值x1为 0.739085178106010，误差为 0.000450955409228126，  
继续迭代  
 $f(x)$ 的值小于e\_1，结束迭代  
第3次迭代结束，结果为 0.739085178106010  
保留四位小数为 0.7391  
利用内置函数检验 0.739085133215161

问题1 (2)

$$e^{-x} - \sin x = 0$$
$$\varepsilon_1 = 10^{-6} \quad \varepsilon_2 = 10^{-4} \quad N = 10 \quad x_0 = 0.6$$

✓  $x=\text{sp.Symbol('x')}$  ...

... 初始值为 0.6  
第1次迭代，当前值x0为 0.588479518996639，误差为 0.0115204810033610，继续迭代  
第2次迭代，当前值x1为 0.588532742847979，误差为 0.0000532238513399896，  
继续迭代  
 $f(x)$ 的值小于e\_1，结束迭代  
第3次迭代结束，结果为 0.588532742847979  
保留四位小数为 0.5885  
利用内置函数检验 0.588532743981861

## 问题 2:

### 问题2 (1)

$$x - e^{-x}$$

$$\varepsilon_1 = 10^{-6} \quad \varepsilon_2 = 10^{-4} \quad N = 10 \quad x_0 = 0.5$$

```
▷ v ✓ x=sp.Symbol('x') ...
...
· 初始值为0.5
第1次迭代, 当前值x0为0.566311003197218, 误差为0.0663110031972182, 继续
迭代
第2次迭代, 当前值x1为0.567143165034862, 误差为0.000832161837644008,
继续迭代
f(x)的值小于e_1, 结束迭代
第3次迭代结束, 结果为0.567143165034862
保留四位小数为0.5671
利用内置函数检验 0.567143290409784
```

(思考题后补充二重根修复的牛顿法)

### 问题2 (2)

$$x^2 - 2xe^{-x} + e^{-2x}$$

$$\varepsilon_1 = 10^{-6} \quad \varepsilon_2 = 10^{-4} \quad N = 20 \quad x_0 = 0.5$$

```
▷ v ✓ x=sp.Symbol('x') ...
...
· 初始值为0.5
第1次迭代, 当前值x0为0.533155501598609, 误差为0.0331555015986090, 继续
迭代
第2次迭代, 当前值x1为0.550043805622888, 误差为0.0168883040242792, 继续
迭代
第3次迭代, 当前值x2为0.558566956550440, 误差为0.00852315092755196, 继
续迭代
第4次迭代, 当前值x3为0.562848451422042, 误差为0.00428149487160145, 继
续迭代
第5次迭代, 当前值x4为0.564994199880568, 误差为0.00214574845852655, 继
续迭代
第6次迭代, 当前值x5为0.566068327008947, 误差为0.00107412712837895, 继
续迭代
第7次迭代, 当前值x6为0.566605704128148, 误差为0.000537377119200655,
继续迭代
f(x)的值小于e_1, 结束迭代
第8次迭代结束, 结果为0.566605704128148
保留四位小数为0.5666
利用内置函数检验 0.567143054564998
```

思考题 1: 牛顿法是一个局部收敛, 初值的选取得满足在收敛法中能够收敛到方程根的区间内, 并且尽可能地靠近解所在  $x$  处。

在实际过程中, 我们也可以通过程序作图, 绘制不同步伐长度的函数点折线图, 通过观察函数图像与  $x$  轴的交点。得到一个根所在大致区间, 并且缩的尽可能小, 靠近解。

思考题 2：

对于问题②尤其是第二问的收敛次数（迭代速度显然慢于其他）原因在于该方程显然是一个 x 非线性式子，有二重根，我们使用的是普通的牛顿法，所以只能有线性收敛速度，不能达到平方阶。

所以这里补充一下用修正后的牛顿法收敛的结果

## 修正一下二重根的牛顿法

```
✓ def Newton_2(N,e_1,e_2,init,exp:sp.Expr): ...
    x=sp.Symbol('x')
    exp2=x**2 - 2*x*sp.exp(-x) + sp.exp(-2*x)
    idx,ans,jud = Newton_2(N=20,e_1=1e-6,e_2=1e-4,init=0.5,
                           exp=exp2)
    if jud:
        print(f"第{idx+1}次迭代结束, 结果为{ans}")
        print(f"保留四位小数为{round(ans,4)}")
    else:
        print("迭代结束, 迭代失败")
    print("利用内置函数检验", sp.nsolve(exp2,x,0.5))
[21] ✓ 0.5s Python
...
... 初始值为0.5
第1次迭代, 当前值x0为0.566311003197218, 误差为0.0663110031972182, 继续
迭代
第2次迭代, 当前值x1为0.567143165034862, 误差为0.000832161837644008,
继续迭代
f(x)的值小于e_1, 结束迭代
第3次迭代结束, 结果为0.567143165034862
保留四位小数为0.5671
利用内置函数检验 0.567143054564998
```

这里修复之后的牛顿法显然快了很多解决了迭代速度线性的问题。

迭代系数改为了 2，详细修改可看代码 [lab5/lab5\\_2](#) 最后

# 实验报告三

第一部分：问题分析 *(描述并总结出实验题目)*

## 实验题目 1 拉格朗日(Lagrange)插值

实验题目给出一些非多项式的函数，然后在规定区域内给定我们多个点，我们要利用这些点进行插值，得到一个理想的多项式函数，进而能够拟合，预测其他点的数值

这里我们使用拉格朗日插值法进行插值得到多项式函数

$$L(x) := \sum_{j=0}^k y_j \ell_j(x)$$

## 第二部分：数学原理

插值的核心为：对于每一个插值点

$$L(x) = f(x) \text{ 插值原则}$$

我们在构建的时候令

$$\ell_j(x) := \prod_{i=0, i \neq j}^k \frac{x - x_i}{x_j - x_i} = \frac{(x - x_0)}{(x_j - x_0)} \dots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \dots \frac{(x - x_k)}{(x_j - x_k)}.$$

$$L(x) := \sum_{j=0}^k y_j \ell_j(x)$$

实际算法构建的时候，逐项相加构建表达式

```
y = 0; k = 0;  
for k in 0 to n  
    for j in 0 to n not k  
        let l = l * (x - xj) / (xk - xj)  
        y = y + l * f(xk)
```

### 第三部分：程序设计流程

问题1 代码在 lab1/lab1\_1

问题2 代码在 lab1/lab1\_2

问题4 代码在 lab1/lab1\_4

思考题和总结在最后

核心通过  $f(x)$  构建  $L(x)$  多项式表达式并且返回定义函数

```
def Linterpolation(ra:np.ndarray,f,x:sp.symbols):
    """
    Linear interpolation.
    :param ra:插值x序列
    :param r:由于是一元表达式，所以只用传入一个一元变量
    :param f:传入函数表达式
    :return:返回一个sympy表达式，后续可以带入实际值运算
    """
    y = 0

    n=ra.size
    for k in range(0,n):
        l=1
        for j in range(0,n):
            if j==k:
                continue
            else:
                l=l*(x-ra[j])/(ra[k]-ra[j])

        y=y+l*f(ra[k])
    return sp.simplify(y)
```

这里展示其中一个问题的解决代码

首先定义基础  $f(x)$  函数

#### 问题一

ˇ 拉格朗日插值多项式的次数n越大越好吗？

(1) 定义  $f_1(x)$

$$f(x) = \frac{1}{1+x^2}$$

```
def f1(x):
    """
    :param x: 运算参数
    :return: 返回计算结果
    """
    return 1/(1+x**2);
```

Python

然后根据题目设置插值点以及验证预测点

在得到  $L(x)$  函数后，进行预测，得到值与真实值进行对比

这里我使用相对误差用于判断一个准度

最后我还做出  $f(x)$  与  $L(x)$  相对应的图像来看拟合效果

设置变量和验证数据列

```
evals = np.array([0.75, 1.75, 2.75, 3.75, 4.75])
# result_L=np.zeros(evals.size)
result_R=np.array([i for i in f1(evals)])
[12]   ✓ 0.3s                                     Python
```

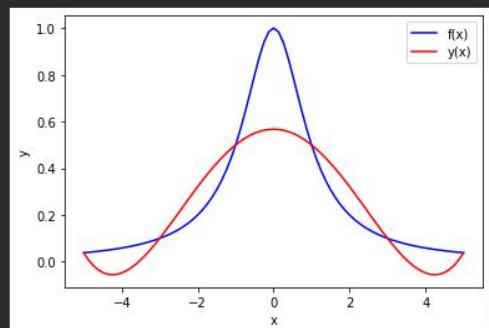
根据不同差值次数n得出结果

```
for n in [5, 10, 20]:
    print('n=', n)

    ra=np.linspace(-5, 5, n+1)
    x=sp.symbols('x')
    y=Linterpolation(ra, f1, x)
    print('多项式表达式: ', y)
    result_L=np.array([y.evalf(subs={x:i}) for i in evals])
    for i in range(0,evals.size):

        js = abs((result_L[i]-result_R[i])/result_R[i])
        print("x:", evals[i], "估计值:", round(result_L[i], 4), "真实值:", round(result_R[i], 4), "相对误差: ", round(js, 4))
    rx = np.linspace(-5, 5, 2*40 + 1)
    ry1 = np.array([f1(i) for i in rx])
    ry2 = np.array([y.evalf(subs={x:i}) for i in rx])
    plt.plot(rx, ry1, 'b-', label='f(x)')
    plt.plot(rx, ry2, 'r-', label='y(x)')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()
[13]   ✓ 3.1s                                     Python
```

```
n= 5
多项式表达式:  0.00192307692307692*x**4 - 0.0692307692307692*x**2 - 5.55111512312578e-17*x + 0.567307692307692
x: 0.75 估计值: 0.5290 真实值: 0.64 相对误差: 0.1735
x: 1.75 估计值: 0.3733 真实值: 0.2462 相对误差: 0.5166
x: 2.75 估计值: 0.1537 真实值: 0.1168 相对误差: 0.3163
x: 3.75 估计值: -0.0260 真实值: 0.0664 相对误差: 1.3909
x: 4.75 估计值: -0.0157 真实值: 0.0424 相对误差: 1.3708
```



## 第四部分：实验结果、结论与讨论

结果也可以看代码文件里面

### 问题一

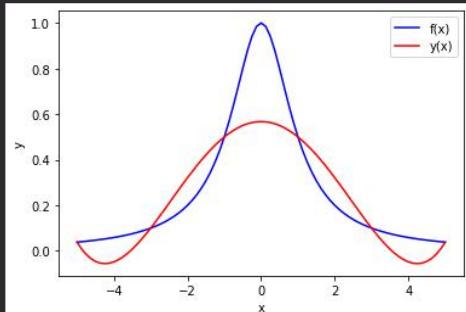
#### 1. 1

(1) 定义  $f_1(x)$

$$f(x) = \frac{1}{1+x^2}$$

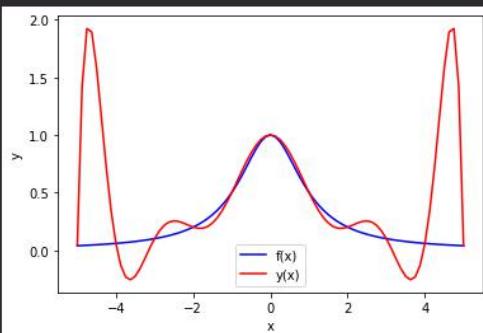
n= 5

多项式表达式: 0.00192307692307692\*x\*\*4 - 0.0692307692307692\*x\*\*2 - 5.55111512312578e-17\*x + 0.567307692307692  
x: 0.75 估计值: 0.5290 真实值: 0.64 相对误差: 0.1735  
x: 1.75 估计值: 0.3733 真实值: 0.2462 相对误差: 0.5166  
x: 2.75 估计值: 0.1537 真实值: 0.1168 相对误差: 0.3163  
x: 3.75 估计值: -0.0260 真实值: 0.0664 相对误差: 1.3909  
x: 4.75 估计值: -0.0157 真实值: 0.0424 相对误差: 1.3708



n= 10

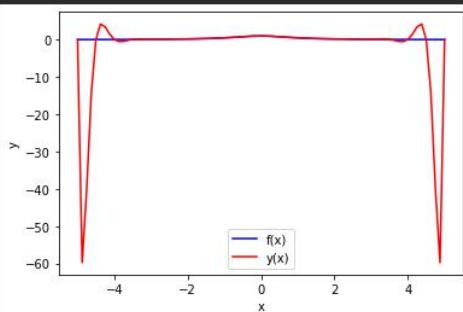
多项式表达式: -2.26244343891403e-5\*x\*\*10 - 1.53524721689842e-20\*x\*\*9 + 0.00126696832579186\*x\*\*8 + 1.6940658945086e-20\*x\*\*7 - 0.0244117647058824\*x\*\*6 + 1.73472347597681e-17\*x\*\*5 + 0.19737556561086\*x\*\*4 + 6.96057794735694e-17\*x\*\*3 - 0.67420814479638\*x\*\*2 - 1.52764086103208e-16\*x + 1.0  
x: 0.75 估计值: 0.6790 真实值: 0.64 相对误差: 0.0609  
x: 1.75 估计值: 0.1906 真实值: 0.2462 相对误差: 0.2258  
x: 2.75 估计值: 0.2156 真实值: 0.1168 相对误差: 0.8460  
x: 3.75 估计值: -0.2315 真实值: 0.0664 相对误差: 4.4864  
x: 4.75 估计值: 1.9236 真实值: 0.0424 相对误差: 44.3256



```

n= 20
多项式表达式: 2.72817068149798e-9*x**20 - 4.66581564268192e-23*x**19 -
2.65314598775676e-7*x**18 - 6.40237794116043e-20*x**17 + 1.07425130797328e-5*x**16 +
5.45155698808788e-18*x**15 - 0.000236412102809867*x**14 - 1.08535413729377e-16*x**13 +
0.00310184793200539*x**12 - 4.4581444655703e-15*x**11 - 0.0251135266738683*x**10 -
1.77814577298485e-15*x**9 + 0.126252909857137*x**8 - 2.23460572962297e-14*x**7 -
0.391630076762947*x**6 + 4.57533316788883e-17*x**5 + 0.753353962815145*x**4 -
8.47477470145019e-15*x**3 - 0.965739184991246*x**2 + 2.76471553983804e-17*x + 1.0
x: 0.75 估计值: 0.6368 真实值: 0.64 相对误差: 0.0051
x: 1.75 估计值: 0.2384 真实值: 0.2462 相对误差: 0.0313
x: 2.75 估计值: 0.0807 真实值: 0.1168 相对误差: 0.3093
x: 3.75 估计值: -0.4471 真实值: 0.0664 相对误差: 7.7337
x: 4.75 估计值: -39.9524 真实值: 0.0424 相对误差: 942.3796

```



## 1.2

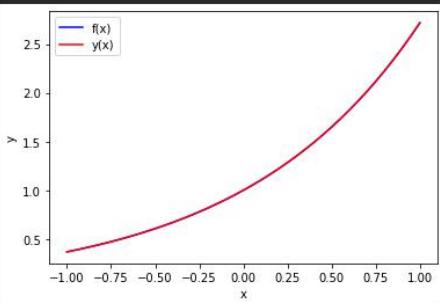
(1) 定义  $f_2(x)$

$$f(x) = e^x$$

```

n= 5
多项式表达式: 0.00861541066655919*x**5 + 0.0436498882268756*x**4 +
0.166582869717544*x**3 + 0.499410240406725*x**2 + 1.0000029132597*x + 1.00002050618164
x: -0.95 估计值: 0.3868 真实值: 0.3867 相对误差: 0.0001477137
x: -0.05 估计值: 0.9512 真实值: 0.9512 相对误差: 1.98780e-5
x: 0.05 估计值: 1.0513 真实值: 1.0513 相对误差: 1.82437e-5
x: 0.95 估计值: 2.5858 真实值: 2.5857 相对误差: 2.89628e-5

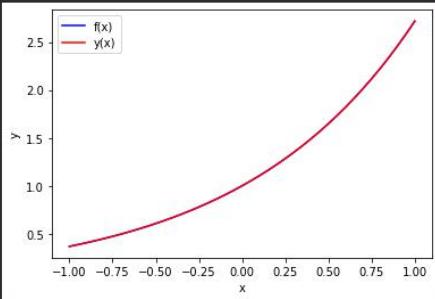
```



```

n= 10
多项式表达式: 2.80203316549432e-7*x**10 + 2.81136410507088e-6*x**9 + 2.47981352572424e-
5*x**8 + 0.000198371189739532*x**7 + 0.0013888899215857*x**6 +
0.0083334575544598*x**5 + 0.0416666665523096*x**4 + 0.166666665295964*x**3 +
0.500000000003148*x**2 + 1.00000000003747*x + 1.0
x: -0.95 估计值: 0.3867 真实值: 0.3867 相对误差: 5.e-10
x: -0.05 估计值: 0.9512 真实值: 0.9512 相对误差: 0.0
x: 0.05 估计值: 1.0513 真实值: 1.0513 相对误差: 0.0
x: 0.95 估计值: 2.5857 真实值: 2.5857 相对误差: 1.e-10

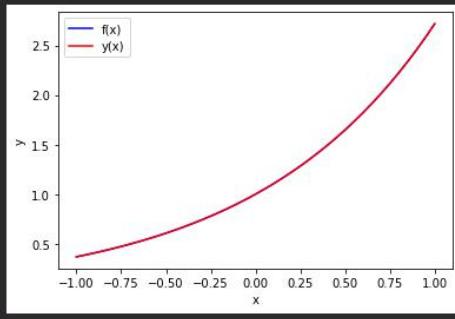
```



```

n= 20
多项式表达式: -8.13565748103429e-9*x**20 + 4.20595824834891e-9*x**19 -
4.82777977595106e-8*x**18 + 1.19896867545322e-7*x**17 - 5.44203430763446e-7*x**16 +
4.39896211901214e-7*x**15 + 7.50742401578464e-8*x**14 - 1.43386387208011e-7*x**13 +
5.52868414160912e-7*x**12 - 5.67112692806404e-7*x**11 + 6.11651444160088e-7*x**10 +
2.71032416776507e-6*x**9 + 2.48604811972086e-5*x**8 + 0.00019841482450289*x**7 +
0.00138889185378099*x**6 + 0.0083333312242956*x**5 + 0.0416666666805803*x**4 +
0.166666666667044*x**3 + 0.50000000000000113*x**2 + 1.000000000000001*x + 1.0
x: -0.95 估计值: 0.3867 真实值: 0.3867 相对误差: 1.2861e-6
x: -0.05 估计值: 0.9512 真实值: 0.9512 相对误差: 0.0
x: 0.05 估计值: 1.0513 真实值: 1.0513 相对误差: 0.0
x: 0.95 估计值: 2.5857 真实值: 2.5857 相对误差: 5.14e-8

```



## 问题二

### 2.1

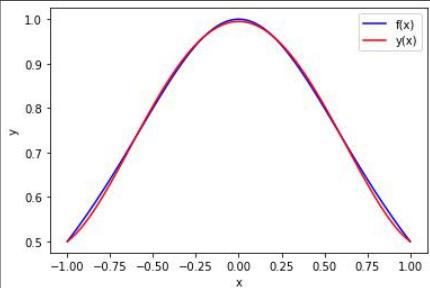
(1) 定义  $f_1(x)$

$$f(x) = \frac{1}{1+x^2}$$

```

n= 5
多项式表达式: 1.33226762955019e-15*x**5 + 0.353506787330317*x**4 + 4.44089209850063e-
15*x**3 - 0.848416289592761*x**2 + 2.91433543964104e-16*x + 0.994909502262444
x: -0.95 估计值: 0.5171 真实值: 0.5256 相对误差: 0.0161272834
x: -0.5 估计值: 0.8049 真实值: 0.8 相对误差: 0.0061245051
x: 0.5 估计值: 0.8049 真实值: 0.8 相对误差: 0.0061245051
x: 0.95 估计值: 0.5171 真实值: 0.5256 相对误差: 0.0161272834

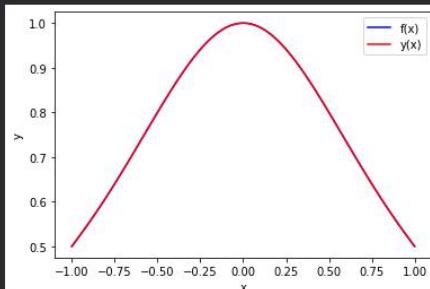
```



```

n= 10
多项式表达式: -0.185821481985784*x**10 + 5.68434188600808e-14*x**9 +
0.594628742356576*x**8 - 7.38964445190504e-13*x**7 - 0.898781344070642*x**6 -
5.96855898038484e-13*x**5 + 0.989700078776167*x**4 - 8.17124146124115e-14*x**3 -
0.99972599507549*x**2 - 8.88178419700125e-16*x + 1.0
x: -0.95 估计值: 0.5264 真实值: 0.5256 相对误差: 0.0014911879
x: -0.5 估计值: 0.8000 真实值: 0.8 相对误差: 2.82499e-5
x: 0.5 估计值: 0.8000 真实值: 0.8 相对误差: 2.82499e-5
x: 0.95 估计值: 0.5264 真实值: 0.5256 相对误差: 0.0014911879

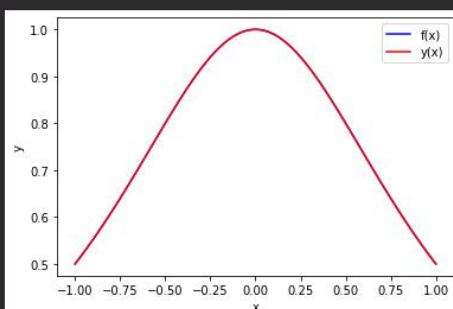
```



```

n= 20
多项式表达式: 0.0500694840447977*x**20 + 5.67524693906307e-9*x**19 -
0.242837078869343*x**18 + 1.34343281388283e-7*x**17 + 0.550493447575718*x**16 +
4.83123585581779e-7*x**15 - 0.815560380462557*x**14 - 2.05123797059059e-7*x**13 +
0.950188042013906*x**12 - 6.02216459810734e-7*x**11 - 0.991554210108006*x**10 -
5.41331246495247e-8*x**9 + 0.99915755526672*x**8 + 1.50612322613597e-9*x**7 -
0.999955461609034*x**6 - 2.4056134861894e-10*x**5 + 0.999998971619647*x**4 +
3.12638803734444e-13*x**3 - 0.99999993406618*x**2 + 4.82947015711943e-15*x + 1.0
x: -0.95 估计值: 0.5256 真实值: 0.5256 相对误差: 6.3146e-6
x: -0.5 估计值: 0.8000 真实值: 0.8 相对误差: 1.4e-9
x: 0.5 估计值: 0.8000 真实值: 0.8 相对误差: 4.e-10
x: 0.95 估计值: 0.5256 真实值: 0.5256 相对误差: 7.0716e-6

```

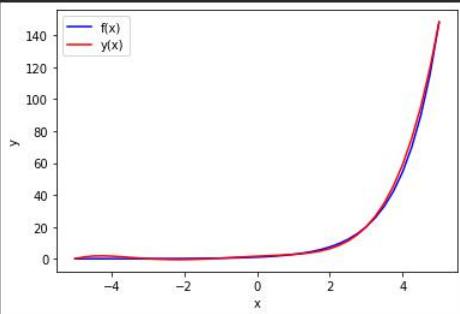


## 2.2

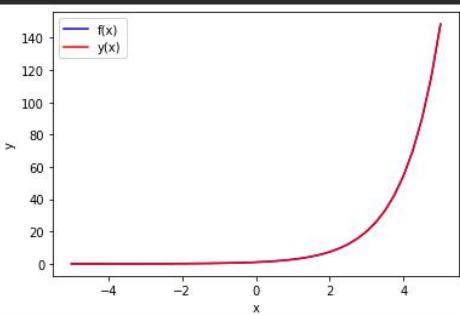
(2) 定义  $f_2(x)$

$$f(x) = e^x$$

```
n= 5
多项式表达式: 0.0186801291026965*x**5 + 0.122638343247617*x**4 + 0.0837100150763059*x**3
- 0.160810762355857*x**2 + 1.0728110494648*x + 1.58125305392348
x: -4.75 估计值: 1.1470 真实值: 0.0087 相对误差: 131.5791887676
x: -0.25 估计值: 1.3022 真实值: 0.7788 相对误差: 0.6719968597
x: 0.25 估计值: 1.8412 真实值: 1.284 相对误差: 0.4339361099
x: 4.75 估计值: 119.6210 真实值: 115.5843 相对误差: 0.0349244929
```



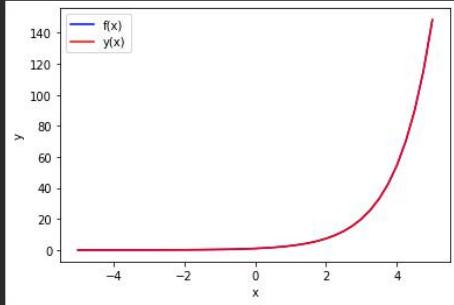
```
n= 10
多项式表达式: 4.16589702264156e-7*x**10 + 4.50740354171357e-6*x**9 + 2.20211913964803e-
5*x**8 + 0.000163566907937791*x**7 + 0.00141017955506279*x**6 +
0.00860113550358025*x**5 + 0.0416073430658901*x**4 + 0.165919291540933*x**3 +
0.500040674413193*x**2 + 1.00051269228781*x + 1.0
x: -4.75 估计值: -0.0020 真实值: 0.0087 相对误差: 1.2261464851
x: -0.25 估计值: 0.7787 真实值: 0.7788 相对误差: 0.0001469428
x: 0.25 估计值: 1.2841 真实值: 1.284 相对误差: 9.27321e-5
x: 4.75 估计值: 115.6074 真实值: 115.5843 相对误差: 0.0001996425
```



```

n= 20
多项式表达式: 5.06036221700607e-19*x**20 + 1.03305816238341e-17*x**19 +
1.52279752572525e-16*x**18 + 2.72434730523045e-15*x**17 + 4.78788316690679e-14*x**16 +
7.66631645071231e-13*x**15 + 1.14696519796008e-11*x**14 + 1.60565547074044e-10*x**13 +
2.08768259057198e-9*x**12 + 2.50522875234792e-8*x**11 + 2.7557314103604e-7*x**10 +
2.75573102075943e-6*x**9 + 2.48015873925438e-5*x**8 + 0.00019841270063755*x**7 +
0.0013888888875843*x**6 + 0.0083333333018501*x**5 + 0.0416666666668005*x**4 +
0.166666666668497*x**3 + 0.4999999999988*x**2 + 0.99999999999702*x + 1.0
x: -4.75 估计值: 0.0087 真实值: 0.0087 相对误差: 1.08005e-5
x: -0.25 估计值: 0.7788 真实值: 0.7788 相对误差: 0.0
x: 0.25 估计值: 1.2840 真实值: 1.284 相对误差: 0.0
x: 4.75 估计值: 115.5843 真实值: 115.5843 相对误差: 7.3e-9

```



## 问题 4

(1) 定义  $f(x)$

$$f(x) = \sqrt{x}$$

第四问补充一个辅助函数来帮我做

```

def work(ra:np.array):
    x=sp.symbols('x')
    print("插入值为",ra)
    y=Linterpolation(ra,f1,x)
    print("多项式表达式: ",y)
    result_L=np.array([y.evalf(subs={x:i}) for i in evals])

    for i in range(0,evals.size):
        js = abs((result_L[i]-result_R[i])/result_R[i])
        print("x:",evals[i],"估计值:",round(result_L[i],4),"真实值:",round(result_R[i],4),"相对误差: ",round(js,4))

    rx = np.linspace(0, 200, 2*40 + 1)
    ry1 = np.array([f1(i) for i in rx])
    ry2 = np.array([y.evalf(subs={x:i}) for i in rx])
    plt.plot(rx, ry1, 'b-', label='f(x)')
    plt.plot(rx, ry2, 'r-', label='y(x)')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()

```

Python

```

work(np.array([1,4,9]))
Python

插入值为 [1 4 9]
多项式表达式: -0.0166666666666667*x**2 + 0.416666666666667*x + 0.6
x: 5 估计值: 2.2667 真实值: 2.2361 相对误差: 0.0137
x: 50 估计值: -20.2333 真实值: 7.0711 相对误差: 3.8614
x: 115 估计值: -171.9000 真实值: 10.7238 相对误差: 17.0298
x: 185 估计值: -492.7333 真实值: 13.6015 相对误差: 37.2265

```

```

work(np.array([36,49,64]))
Python

插入值为 [36 49 64]
多项式表达式: -0.000366300366300372*x**2 + 0.108058608058609*x + 2.58461538461538
x: 5 估计值: 3.1158 真实值: 2.2361 相对误差: 0.3934
x: 50 估计值: 7.0718 真实值: 7.0711 相对误差: 0.0001
x: 115 估计值: 10.1670 真实值: 10.7238 相对误差: 0.0519
x: 185 估计值: 10.0388 真实值: 13.6015 相对误差: 0.2619

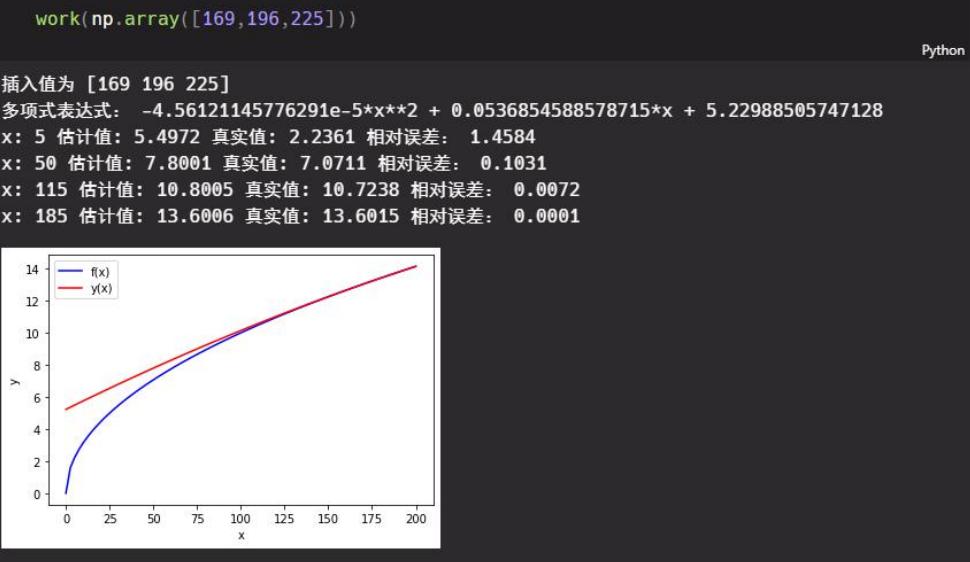
```

```

work(np.array([100,121,144]))
Python

插入值为 [100 121 144]
多项式表达式: -9.4108789760964e-5*x**2 + 0.0684170901562204*x + 4.0993788819876
x: 5 估计值: 4.4391 真实值: 2.2361 相对误差: 0.9852
x: 50 估计值: 7.2850 真实值: 7.0711 相对误差: 0.0302
x: 115 估计值: 10.7228 真实值: 10.7238 相对误差: 0.0001
x: 185 估计值: 13.5357 真实值: 13.6015 相对误差: 0.0048

```



总结（思考题）：

显然无论是插值区间还是插值的次数过多或者过少都会影响精度，并且影响最大的地方在于区间的左右端点附近容易有巨大波动。

补充：特别是  $1/(x^2+1)$  [-5, 5] 区间内 n 在 10 后出现了龙格现象

不过相对而言，如果只考虑内插，还是相对而言多的插值次数在区间中间更为稳定

尽量不选择区间左右端点附近的值

内插通过我们插值样本区间中，再预测我们插值区间内的点，外推就是我们利用我们已经有过的数据，经过一系列处理之后，来预测我们样本之外，插值区间之外的值。

外推如果是隆贝格积分类的话也可以理解成为对于估计的一个精度的提升。

经过问题四的问题，显然内插的误差明显小于外推，并且整体函数图像也可以看出，在区间内拟合效果明显强于外推。我们的拟合也具有一定局部性，在我们函数插值点附近显然稳定性更好。

## 实验报告四

第一部分：问题分析 *(描述并总结出实验题目)*

### 实验题目 2 龙贝格(Romberg)积分法

利用复化梯形求积公式，求解积分

核心在于公式外推的过程，利用已知误差精度和插值点的积分公式，得到新的误差精度和插值点的公式。

新的插值点为新的二分，即之前小区间的中点；我们在外推的时候通过线性组合来得到一个阶方更大的误差精度。（精度更高）

外推的时候通过构建一个 T-数表，来方便我们利用之前的计算结果

（指导书上应该有误，所以自己的操作参照老师上课的时候讲述的递推以及做 T-数表法）

## 第二部分：数学原理

令复化梯形公式用  $T(h)$  表示， $h$  为当前小区间的长度 ( $h = (b-a) / 2^k$ )

在我们对小区间进行继续二分的时候

$$T\left(\frac{h}{2}\right) = \frac{T(h)}{2} + \frac{U(h)}{2}$$

$$U(h) = h \sum_{i=0}^{n-1} f\left(x_{i+\frac{1}{2}}\right)$$

但是我们对其进行误差分析的时候：

$$T(h) = I(f) + a_1 h^2 + a_2 h^4 + a_3 h^6 \dots$$

$$T\left(\frac{h}{2}\right) = I(f) + \frac{a_1}{4} h^2 + \frac{a_2}{16} h^4 + \frac{a_3}{2^6} h^6 \dots$$

发现误差并没有发生实质性的变化，所以我们把这个式子线性组合一下

$$S(h) = \frac{4}{3}T\left(\frac{h}{2}\right) - \frac{1}{3}T(h) = I(f) + b_1 h^4 + b_2 h^6 \dots$$

这样误差就被我们缩小到  $O(h^4)$

如果我们令  $T_m(h)$  为外推  $m$  次，小区间为  $h$  的积分，我们可以获得一个普适性的递推公式：

$$T_m(h) = \frac{4^m T_{m-1}\left(\frac{h}{2}\right) - T_{m-1}(h)}{4^m - 1}$$

这样我们显然做表，方便我们递推，保留结果是比较好的选择：

(下表为老师上课时候所教，其实 wiki 上也搜过其他表示的表，但是本质都是以上的递推公式，由于我代码也是这样存的表，就描述的这个表)

$T_{m,k}$  代表外推  $m$  次，这里的  $k = i - m$ , 用于三角矩阵表示位置

$i$  为当前最大二分次数

$T_{0,0}, T_{0,1}, T_{0,2} \dots T_{0,i}$

$T_{1,0}, T_{1,1} \dots$

$T_{2,0} \dots$

...

$T_{i,0}$

最后通过比较  $T_{i,0}$   $T_{i-1,0}$  的差值和预设界限判断是否结束外推

### 第三部分：程序设计流程

代码在 lab2/lab2\_1.ipynb 中

核心 Romberg 积分函数定义

```
def Romberg(a,b,f,e,n):
    mat = np.zeros((n,n))
    h=(b-a)
    mat[0,0] = h/2*(f.evalf(subs={'x':a})+f.evalf(subs={'x':b}))
    for i in range(1,n):

        # 先算出第一行最右端新元素（新内二分的结果）
        sum=0

        for j in range(0,2***(i-1)):
            sum=sum+f.evalf(subs={'x':a+(j+1/2)*h})

        mat[0,i]=mat[0,i-1]/2+sum*h/2

        # 然后在利用T-数阵已有结果外推

        for j in range(1,i+1):
            mat[j,i-j]=(mat[j-1,i-j+1]*4**j-mat[j-1,i-j])/(4**j-1)

        # 判断是否结束
        if abs(mat[i,0]-mat[i-1,0])<e:
            print("结束:")
            print("T矩阵为:")
            print(mat[0:i+1,0:i+1])
            return True,mat[i,0]

        h=h/2

    print("停机")
    return False,mat[n-1,0]
```

## 实际问题求解范例

1.1

$$\int_0^1 e^x x^2 dx \quad \varepsilon = 10^{-6}$$

```
# 定义函数和精度
x=sp.symbols('x')
exp1 = sp.exp(x)*x**2
e=1e-6

# 调用函数
bo,ans=Romberg(0,1,exp1,e,20)

if(bo):
    print("结果为: ",ans)
    print("保留四位小数: ",round(ans,4))
# 验证
ss=sp.integrate(exp1,(x,0,1))

print("验证误差为: ",abs(ss.evalf()-ans))
```

✓ 0.1s

Python

## 第四部分：实验结果、结论与讨论

问题 1 计算结果：

1.1

$$\int_0^1 e^x x^2 dx \quad \varepsilon = 10^{-6}$$

结束:

T矩阵为:

```
[[1.35914091 0.88566062 0.76059633 0.72889018 0.72093578]
 [0.72783385 0.71890824 0.71832146 0.71828431 0.          ]
 [0.7183132 0.71828234 0.71828184 0.          0.          ]
 [0.71828185 0.71828183 0.          0.          0.          ]
 [0.71828183 0.          0.          0.          0.        ]]
```

结果为: 0.7182818284623739

保留四位小数: 0.7183

验证误差为: 3.32867067243114e-12

1.2

$$\int_1^3 e^x \sin x dx \quad \varepsilon = 10^{-6}$$

✓ # 定义函数和精度 ...

结束:

T矩阵为:

```
[[ 5.12182642  9.27976291 10.52055428 10.84204347 10.92309389 10.94339842]
 [10.66574174 10.93415141 10.94920653 10.9501107 10.9501666  0.      ]
 [10.95204539 10.9502102 10.95017097 10.95017033 0.      0.      ]
 [10.95018107 10.95017035 10.95017031 0.      0.      0.      ]
 [10.95017031 10.95017031 0.      0.      0.      0.      ]
 [10.95017031 0.      0.      0.      0.      0.      ]]
```

结果为: 10.950170314683838

保留四位小数: 10.9502

验证误差为: 1.68043357007264e-12

1.3

$$\int_0^1 \frac{4}{1+x^2} dx \quad \varepsilon = 10^{-6}$$

✓ # 定义函数和精度 ...

结束:

T矩阵为:

```
[[3.          3.1          3.13117647 3.13898849 3.14094161 3.14142989]
 [3.13333333 3.14156863 3.1415925 3.14159265 3.14159265 0.      ]
 [3.14211765 3.14159409 3.14159266 3.14159265 0.      0.      ]
 [3.14158578 3.14159264 3.14159265 0.      0.      0.      ]
 [3.14159267 3.14159265 0.      0.      0.      0.      ]
 [3.14159265 0.      0.      0.      0.      0.      ]]
```

结果为: 3.141592653638244

保留四位小数: 3.1416

验证误差为: 4.84510209730615e-11

1.4

$$\int_0^1 \frac{1}{1+x} dx \quad \varepsilon = 10^{-6}$$

✓ # 定义函数和精度 ...

结束:

T矩阵为:

```
[[0.75          0.70833333 0.69702381 0.69412185 0.6933912 ]
 [0.69444444 0.69325397 0.69315453 0.69314765 0.      ]
 [0.6931746 0.6931479 0.69314719 0.      0.      ]
 [0.69314748 0.69314718 0.      0.      0.      ]
 [0.69314718 0.      0.      0.      0.      ]]
```

结果为: 0.6931471819167452

保留四位小数: 0.6931

验证误差为: 1.35679989465842e-9

思考题：

在实验 1 中二分次数和精度的关系如何？

二分次数的增加有利于我们减小精度，但是不能将精度提高到另一个阶次，误差仍然是在同一个  $O(h^k)$  下。所以我们才会想到利用外推，将两个积分表达式组合后，消去较大误差的项式，达到更有效地提高精度。