****

中国地质大学（北京）

科学与工程计算

实验报告

学 院：信息工程学院

专 业：计算机科学与技术

班 级：10041811

学 号：1005183121

姓 名：周子杰

联系方式：18006163783

邮 箱：2476331552@qq.com

指导老师：王玉柱

日 期：2020年5月17日

**摘 要**

实验一：非线性方程求根

------二分法+牛顿法+牛顿下山法+弦截法

实验二：线性方程组的直接解法

------改进的Cholesky方法

实验三：线性方程组的迭代解法

------雅可比+SOR+共轭梯度法

实验四：函数插值

------三次样条插值+拉格朗日插值法

实验五：最小二乘数拟合

------最小二乘法+共轭梯度法+三次样条插值+拉格朗日插值法

实验六：数值积分

------龙格公式

关键词：二分法、牛顿法、牛顿下山法、共轭梯度法、改进的Cholesky方法、雅可比、SOR、共轭梯度法、三次样条插值、拉格朗日插值、龙格公式

**目 录**

[实验一、非线性方程求根 5](#_Toc40344994)

[一、实验内容： 5](#_Toc40344995)

[二、算法基本思想及复杂度分析： 5](#_Toc40344996)

[三、源程序及注释： 5](#_Toc40344997)

[四、运行输出结果： 13](#_Toc40344998)

[五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训： 16](#_Toc40344999)

[实验二、线性方程组的直接解法 18](#_Toc40345000)

[一、实验内容： 18](#_Toc40345001)

[二、算法基本思想及复杂度分析： 18](#_Toc40345002)

[三、源程序及注释： 18](#_Toc40345003)

[四、运行输出结果： 21](#_Toc40345004)

[五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训： 22](#_Toc40345005)

[实验三、线性方程组的迭代解法 23](#_Toc40345006)

[一、实验内容： 23](#_Toc40345007)

[二、算法基本思想及复杂度分析： 23](#_Toc40345008)

[三、源程序及注释： 24](#_Toc40345009)

[四、运行输出结果： 38](#_Toc40345010)

[五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训： 43](#_Toc40345011)

[实验四、函数插值 44](#_Toc40345012)

[一、实验内容： 44](#_Toc40345013)

[二、算法基本思想及复杂度分析： 44](#_Toc40345014)

[三、源程序及注释： 44](#_Toc40345015)

[四、运行输出结果： 47](#_Toc40345016)

[五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训： 48](#_Toc40345017)

[实验五、最小二乘拟合 49](#_Toc40345018)

[一、实验内容： 49](#_Toc40345019)

[二、算法基本思想及复杂度分析： 49](#_Toc40345020)

[三、源程序及注释： 50](#_Toc40345021)

[四、运行输出结果： 57](#_Toc40345022)

[五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训： 68](#_Toc40345023)

[实验六、数值积分 69](#_Toc40345024)

[一、实验内容： 69](#_Toc40345025)

[二、算法基本思想及复杂度分析： 69](#_Toc40345026)

[三、源程序及注释： 69](#_Toc40345027)

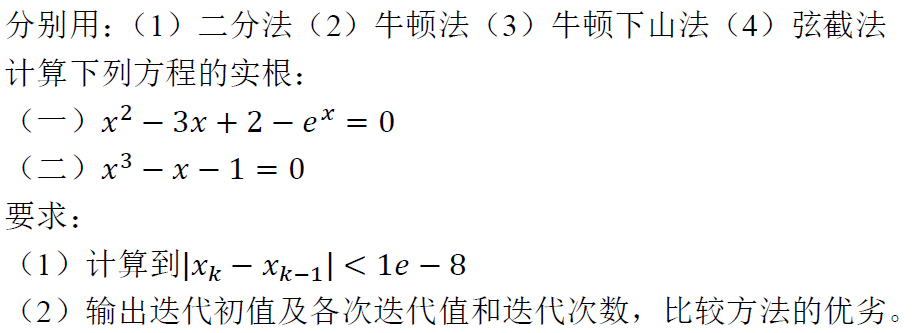
[四、运行输出结果： 72](#_Toc40345028)

[五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训： 72](#_Toc40345029)

[心得体会 73](#_Toc40345030)

# 实验一、非线性方程求根

## 一、实验内容：



## 二、算法基本思想及复杂度分析：

（一）二分法

将有根区间对分，并找出根所在的小区间，然后再对该小区间对分，依次类推，直到有根区间的长度足够小为止。

（二）牛顿法

牛顿法是一种线性化方法，其基本思想是将非线性方程f(x)=0逐步归结为某种线性方程来求解。

（三）牛顿下山法

要求每一步迭代满足下降条件（保证全局有效）。也就是由于牛顿法的复杂度依赖于x0的选取，所以在牛顿法中加入了下山因子，且逐次减半。

（四）弦截法

当f(x)比较复杂的时候，f’(x)会比较难算，该方法就是用差商来取代牛顿法中的

f’(x)。

## 三、源程序及注释：

（一）

1. 二分法

1. //实验一（一）二分法
2. //f(x)=x^2-3x+2-e^x
3. //f(0)=1>0
4. //f(1)=-e<0
5. //由此 有根区间(a,b)确定为(0,1)
7. #include <iostream>
8. #include <iomanip>
9. #include <cmath>
10. **using** **namespace** std;
12. //f(x)=x^2-3x+2-e^x
13. **double** f(**double** x){
14. **return** x\*x-3\*x+2-exp(x);
15. }
17. **int** main(){
18. **double** a=0;     //(a,b)赋初值为(0,1)
19. **double** b=1;
20. **double** x=0.5;   //x=(a+b)/2
21. **double** xt=0;    //xt用来存储上一个x
22. **int** k=0;        //k用来计数
23. **while**(fabs(f(x)-f(xt))>=1e-8){ //设置精度
24. cout<<"k:"<<k<<"\t"<<fixed<<setprecision(10)
25. <<" a:"<<a<<"\t b:"<<b<<"\t x:"<<x<<"\t f(x):"<<f(x)
26. <<endl;
27. xt=x;
28. **if**(f(x)<0){
29. b=x;
30. x=(a+x)/2;
31. }
32. **else** **if**(f(x)>0){
33. a=x;
34. x=(b+x)/2;
35. }
36. k++;
37. }
38. **return** 0;
39. }

2. 牛顿法

1. //实验一（一）牛顿法
2. //x1=x0-f(x0)/f'(x0)
4. #include <iostream>
5. #include <iomanip>
6. #include <cmath>
7. **using** **namespace** std;
9. //f(x)=x^2-3x+2-e^x
10. **double** f(**double** x){
11. **return** x\*x-3\*x+2-exp(x);
12. }
14. //f'(x)=2x-3-e^x
15. **double** f\_(**double** x){
16. **return** 2\*x-3-exp(x);
17. }
19. **int** main(){
20. **double** x[100]={0};
21. x[0]=0;
22. x[1]=x[0]-f(x[0])/f\_(x[0]);
24. //循环并判断精度
25. **int** k=1;
26. cout<<"x0: "<<x[0]<<endl;
27. **while**(fabs(x[k]-x[k-1])>=1e-8){ //设置精度
28. x[k+1]=x[k]-f(x[k])/f\_(x[k]);
29. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
30. k++;
31. }
32. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
33. **return** 0;
34. }

3. 牛顿下山法

1. //实验一（一）牛顿下山法
2. //x1=x0-f(x0)/f'(x0)
4. #include <iostream>
5. #include <iomanip>
6. #include <cmath>
7. **using** **namespace** std;
9. //f(x)=x^2-3x+2-e^x
10. **double** f(**double** x){
11. **return** x\*x-3\*x+2-exp(x);
12. }
14. //f'(x)=2x-3-e^x
15. **double** f\_(**double** x){
16. **return** 2\*x-3-exp(x);
17. }
19. **int** main(){
20. **double** lambda=1.0;  //下山因子
22. **double** x[100]={0};
23. x[0]=0;
24. x[1]=x[0]-f(x[0])/f\_(x[0]);
26. //循环并判断精度
27. **int** k=1;
28. cout<<"x0: "<<x[0]<<endl;
29. **while**(fabs(x[k]-x[k-1])>=1e-8){ //设置精度
30. x[k+1]=x[k]-f(x[k])/f\_(x[k]);
31. **while**(fabs(f(x[k+1]))>=fabs(f(x[k]))){
32. lambda/=2.0;  //下山因子逐次减半
33. x[k+1]=x[k]-lambda\*f(x[k])/f\_(x[k]);
34. }
35. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
36. k++;
37. }
38. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
39. **return** 0;
40. }

4. 弦截法

1. //实验一（一）弦截法
2. //x[k+1]=x[k]-f(x[k])/(f(x[k])-f(x[k-1]))\*(x[k]-x[k-1])
4. #include <iostream>
5. #include <iomanip>
6. #include <cmath>
7. **using** **namespace** std;
9. //f(x)=x^2-3x+2-e^x
10. **double** f(**double** x){
11. **return** x\*x-3\*x+2-exp(x);
12. }
14. **int** main()
15. {
16. **double** x[100]={0};
17. x[0]=0;     //近似根
18. x[1]=1;     //近似根
19. **int** k=1;    //计数
21. cout<<"x0: "<<setprecision(10)<<x[0]<<endl;
23. **while**(fabs(x[k]-x[k-1])>=1e-8){ //设置精度
24. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
25. x[k+1]=x[k]-f(x[k])/(f(x[k])-f(x[k-1]))\*(x[k]-x[k-1]);
26. k++;
27. }
28. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
29. **return** 0;
30. }

（二）

1. 二分法

1. //实验一（二）二分法
2. //f(x)=x^3-x-1
3. //f(1)=-1<0
4. //f(2)=5>0
5. //由此 有根区间(a,b)确定为(1,2)
7. #include <iostream>
8. #include <iomanip>
9. #include <cmath>
10. **using** **namespace** std;
12. //f(x)=x^3-x-1
13. **double** f(**double** x){
14. **return** x\*x\*x-x-1;
15. }
17. **int** main(){
18. **double** a=1;     //(a,b)赋初值为(1,2)
19. **double** b=2;
20. **double** x=1.5;   //x=(a+b)/2
21. **double** xt=0;    //xt用来存储上一个x
22. **int** k=0;        //k用来计数
23. **while**(fabs(f(x)-f(xt))>=1e-8){ //设置精度
24. cout<<"k:"<<k<<"\t"<<fixed<<setprecision(10)
25. <<" a:"<<a<<"\t b:"<<b<<"\t x:"<<x<<"\t f(x):"<<f(x)
26. <<endl;
27. xt=x;
28. **if**(f(x)<0){
29. a=x;
30. x=(b+x)/2;
31. }
32. **else** **if**(f(x)>0){
33. b=x;
34. x=(a+x)/2;
35. }
36. k++;
37. }
38. **return** 0;
39. }

2. 牛顿法

1. //实验一（二）牛顿法
2. //x1=x0-f(x0)/f'(x0)
4. #include <iostream>
5. #include <iomanip>
6. #include <cmath>
7. **using** **namespace** std;
9. //f(x)=x^3-x-1
10. **double** f(**double** x){
11. **return** x\*x\*x-x-1;
12. }
14. //f'(x)=3x^2-1
15. **double** f\_(**double** x){
16. **return** 3\*x\*x-1;
17. }
19. **int** main(){
20. **double** x[100]={0};
21. x[0]=1;
22. x[1]=x[0]-f(x[0])/f\_(x[0]);
24. //循环并判断精度
25. **int** k=1;
26. cout<<"x0: "<<x[0]<<endl;
27. **while**(fabs(x[k]-x[k-1])>=1e-8){ //设置精度
28. x[k+1]=x[k]-f(x[k])/f\_(x[k]);
29. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
30. k++;
31. }
32. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
33. **return** 0;
34. }

3. 牛顿下山法

1. //实验一（二）牛顿下山法
2. //x1=x0-f(x0)/f'(x0)
4. #include <iostream>
5. #include <iomanip>
6. #include <cmath>
7. **using** **namespace** std;
9. //f(x)=x^3-x-1
10. **double** f(**double** x){
11. **return** x\*x\*x-x-1;
12. }
14. //f'(x)=3x^2-1
15. **double** f\_(**double** x){
16. **return** 3\*x\*x-1;
17. }
19. **int** main(){
20. **double** lambda=1;  //下山因子
22. **double** x[100]={0};
23. x[0]=1;
24. x[1]=x[0]-f(x[0])/f\_(x[0]);
26. //循环并判断精度
27. **int** k=1;
28. cout<<"x0: "<<x[0]<<endl;
29. **while**(fabs(x[k]-x[k-1])>=1e-8){ //设置精度
30. x[k+1]=x[k]-f(x[k])/f\_(x[k]);
31. **while**(fabs(f(x[k+1]))>=fabs(f(x[k]))){
32. lambda/=2.0;  //下山因子逐次减半
33. x[k+1]=x[k]-lambda\*f(x[k])/f\_(x[k]);
34. }
35. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
36. k++;
37. }
38. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
39. **return** 0;
40. }

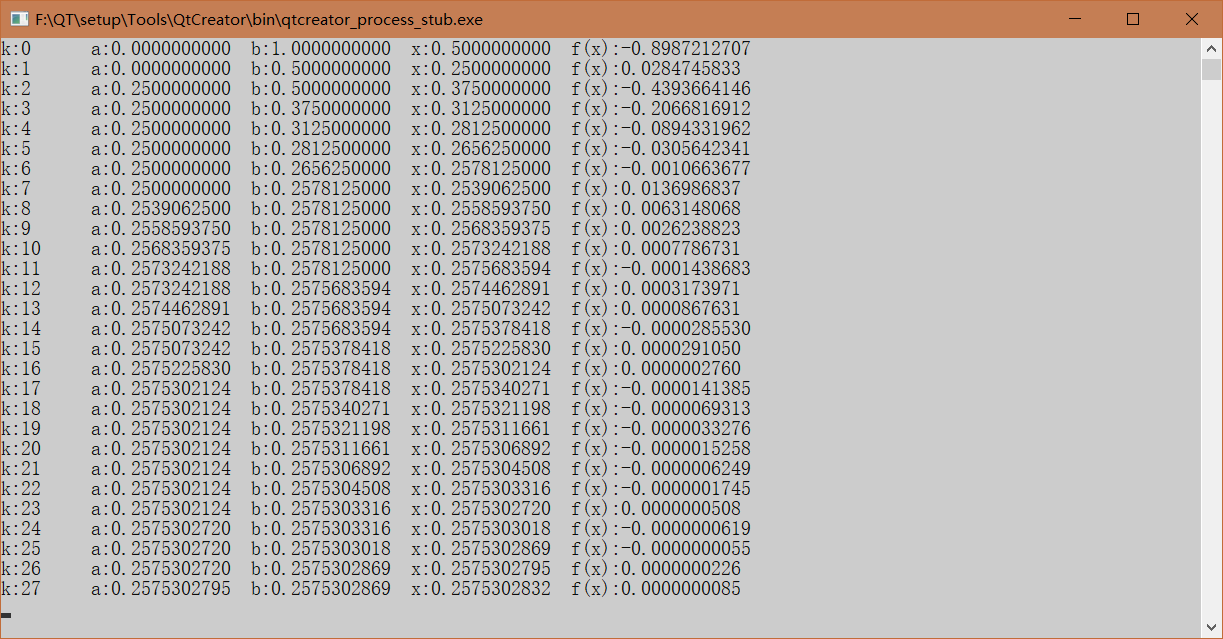
4. 弦截法

1. //实验一（二）弦截法
2. //x[k+1]=x[k]-f(x[k])/(f(x[k])-f(x[k-1]))\*(x[k]-x[k-1])
4. #include <iostream>
5. #include <iomanip>
6. #include <cmath>
7. **using** **namespace** std;
9. //f(x)=x^3-x-1
10. **double** f(**double** x){
11. **return** x\*x\*x-x-1;
12. }
14. **int** main()
15. {
16. **double** x[100]={0};
17. x[0]=1;     //近似根
18. x[1]=2;     //近似根
19. **int** k=1;    //计数
21. cout<<"x0: "<<setprecision(10)<<x[0]<<endl;
23. **while**(fabs(x[k]-x[k-1])>=1e-8){ //设置精度
24. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
25. x[k+1]=x[k]-f(x[k])/(f(x[k])-f(x[k-1]))\*(x[k]-x[k-1]);
26. k++;
27. }
28. cout<<"x"<<k<<": "<<setprecision(10)<<x[k]<<endl;
29. **return** 0;
30. }

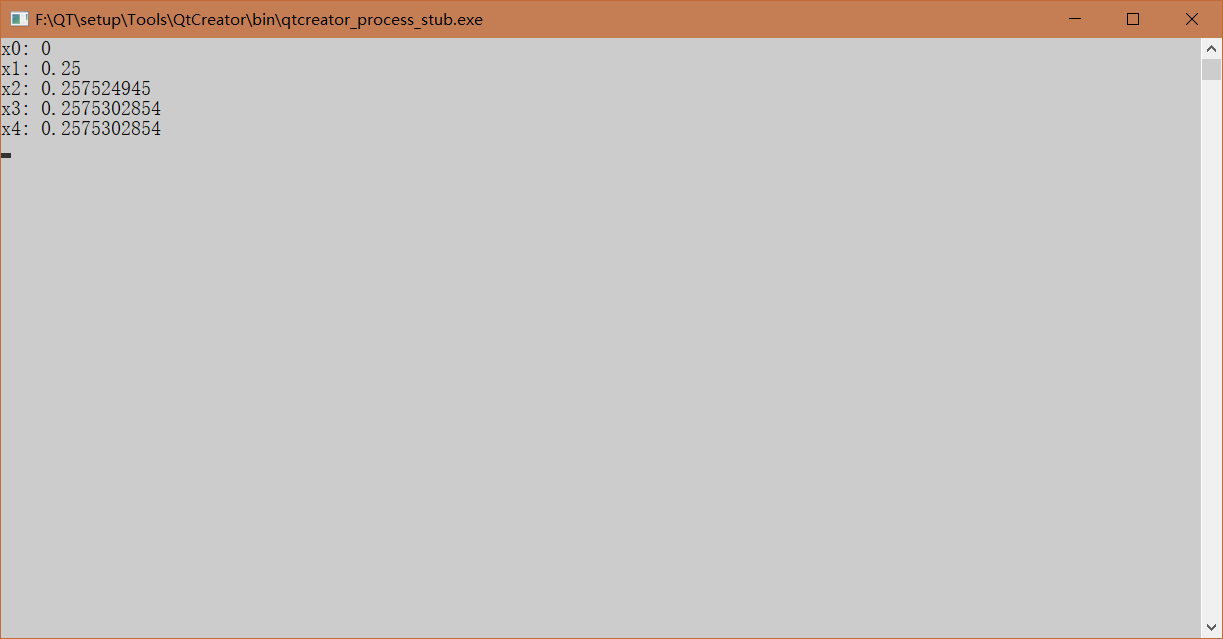
## 四、运行输出结果：

（一）

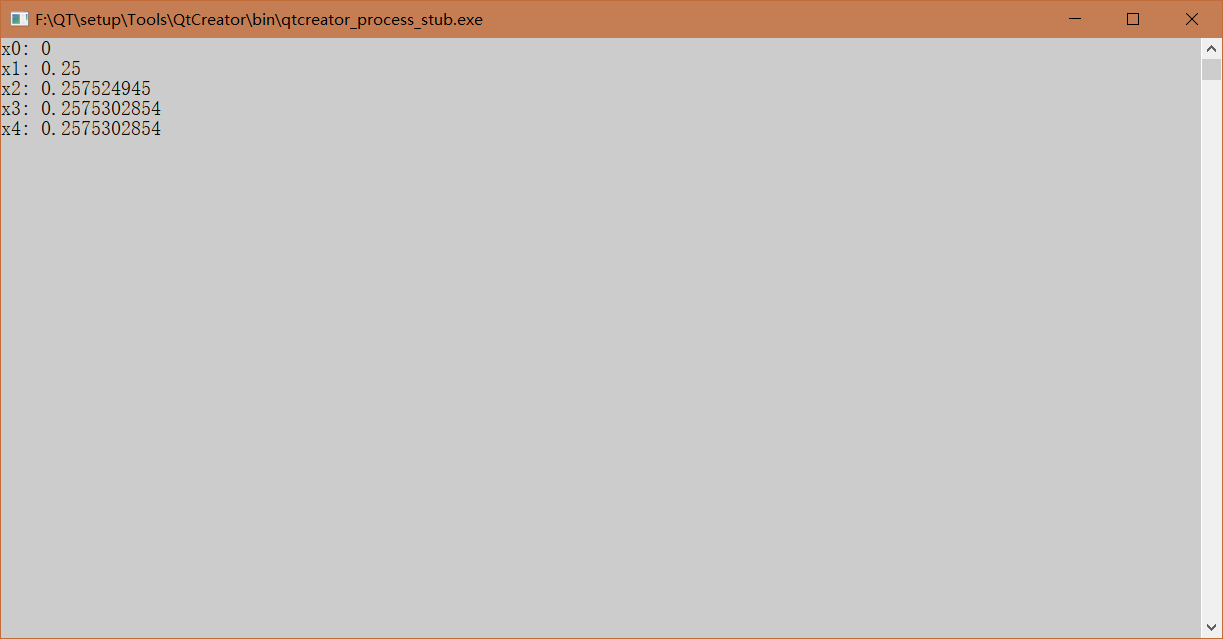
1. 二分法



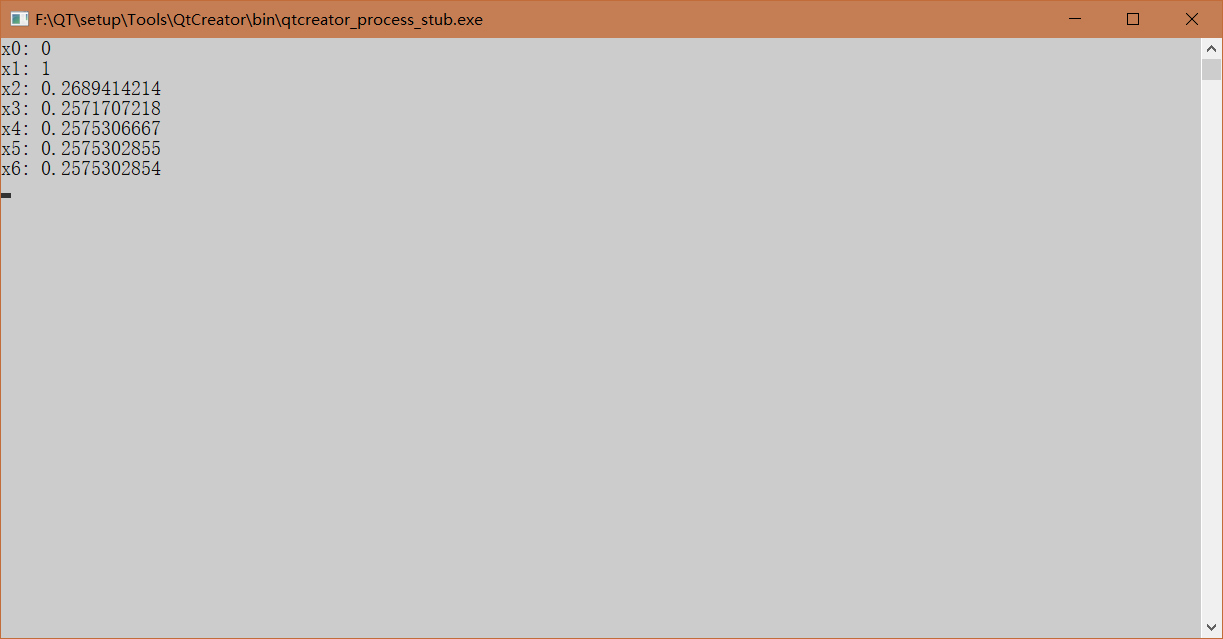
2. 牛顿法



3. 牛顿下山法

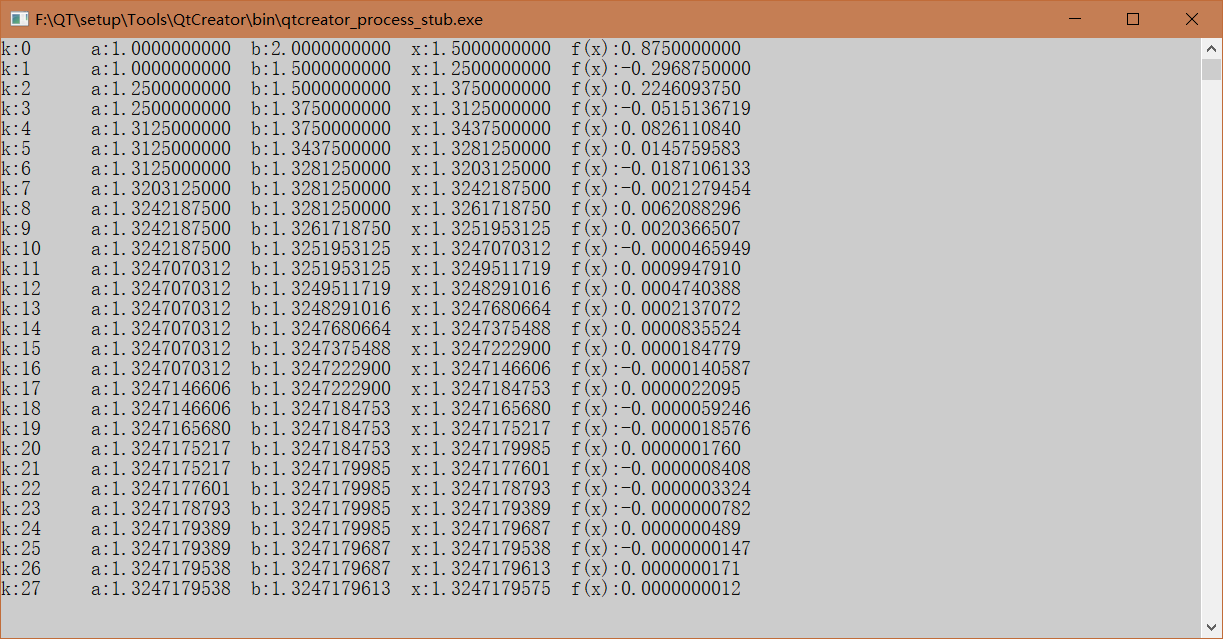


4. 弦截法

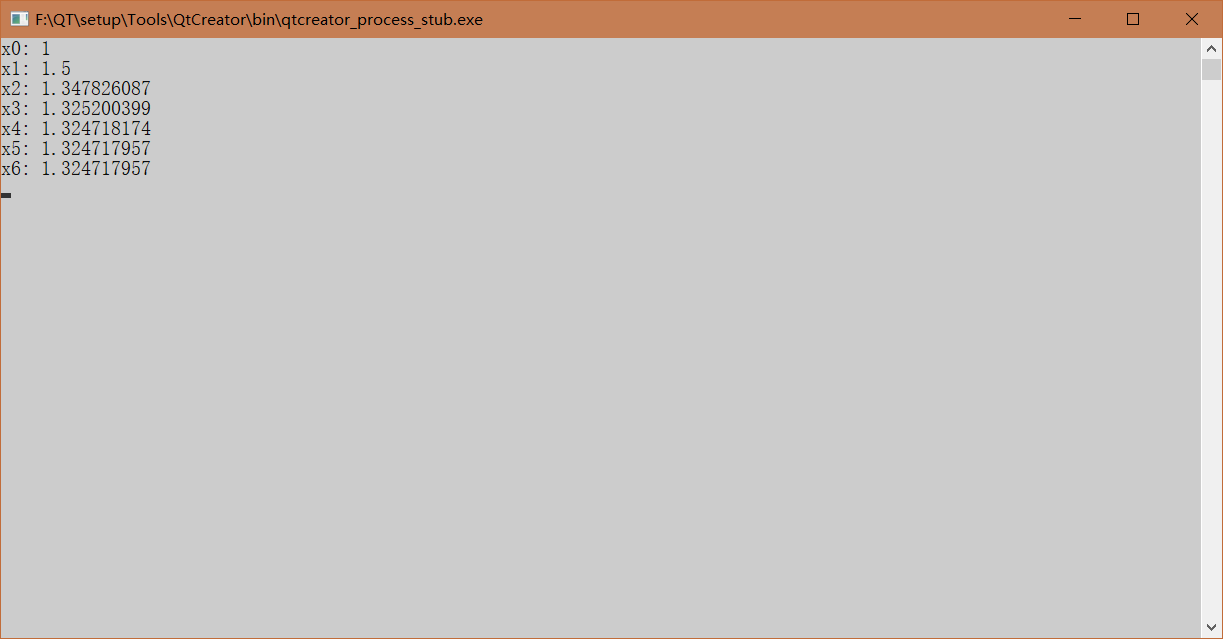


（二）

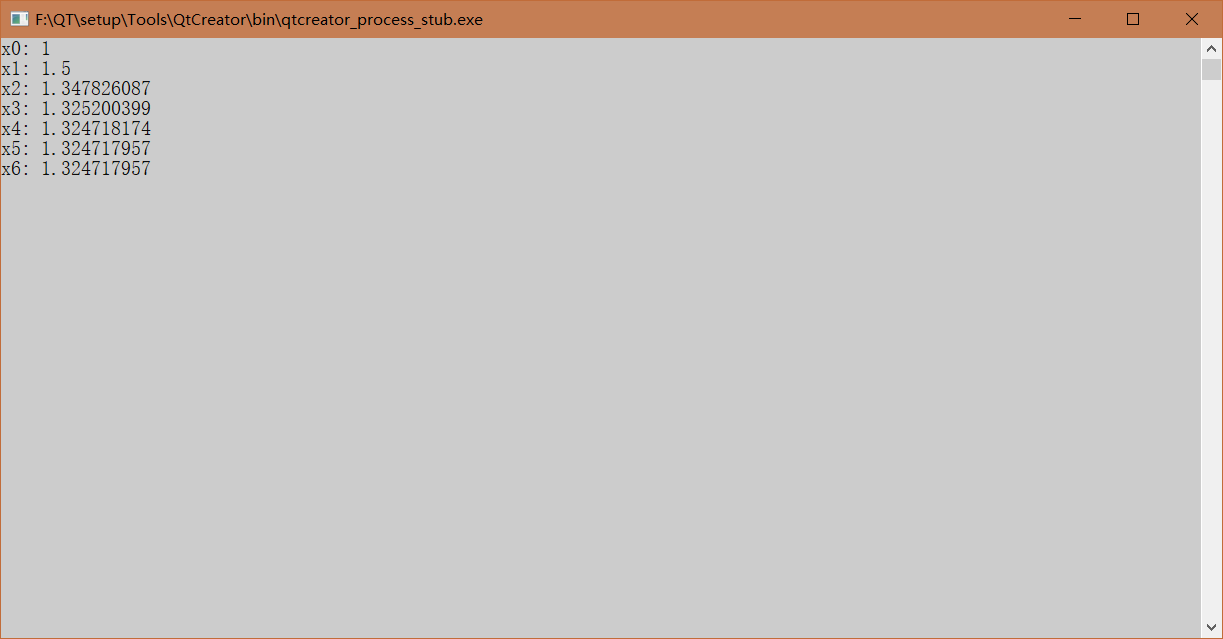
1. 二分法



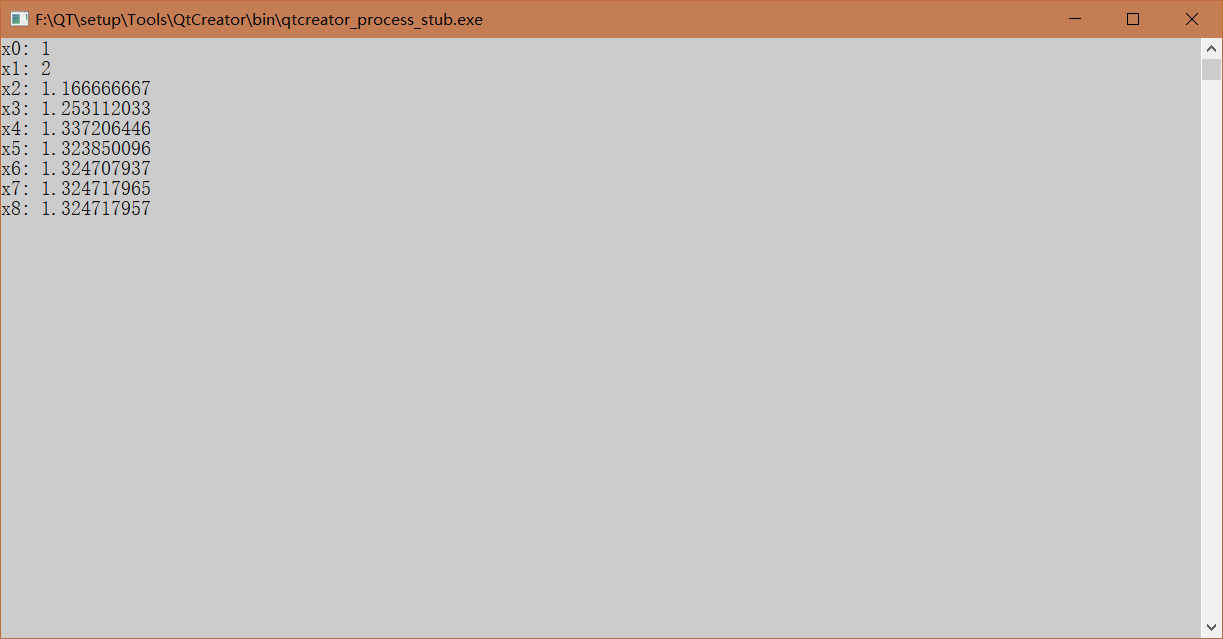
2. 牛顿法



3. 牛顿下山法



4. 弦截法



由结果可见，对于牛顿法的速度相比二分法有了很大的提升，而牛顿下山法由于加入了下山因子而对于初值的要求没有了过大的依赖。对于f（x）的导数比较难算的式子，弦截法是不错的选择。

## 五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训：

产生问题：在二分法中，当f(a)<0, f(b)>0时，与当f(a)>0, f(b)<0时，所选取的下一次迭代的x是不一样的。在一开始的时候，我直接将（二）式套用了（一）式的代码，发现结果有问题。

采取措施：在（一）式中，f(a)>0, f(b)<0，当f(x)>0时，要另a=x, x=(b+x)/2。

在（二）式中，f(a)<0, f(b)>0，当f(x)>0时，与（一）式相反。

吸取教训：不能因为用的是同一种方法而疏忽大意。要从原理出发，仔细地重新捋一遍。

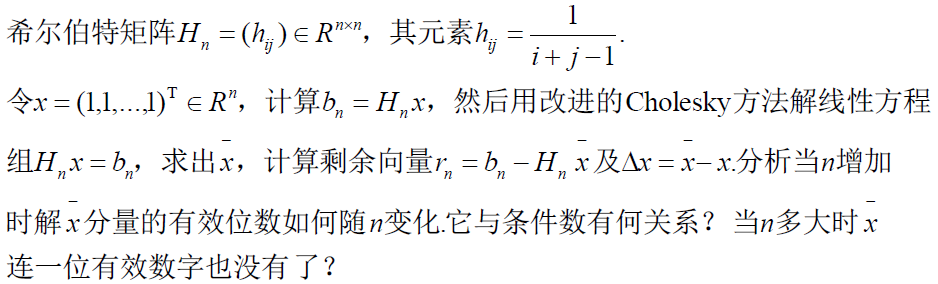
产生问题：在牛顿法中，（二）式如果选取为0的时候需要迭代21次

采取措施：将改为1

得到结论：牛顿法的复杂度和的选取有关

# 实验二、线性方程组的直接解法

## 一、实验内容：



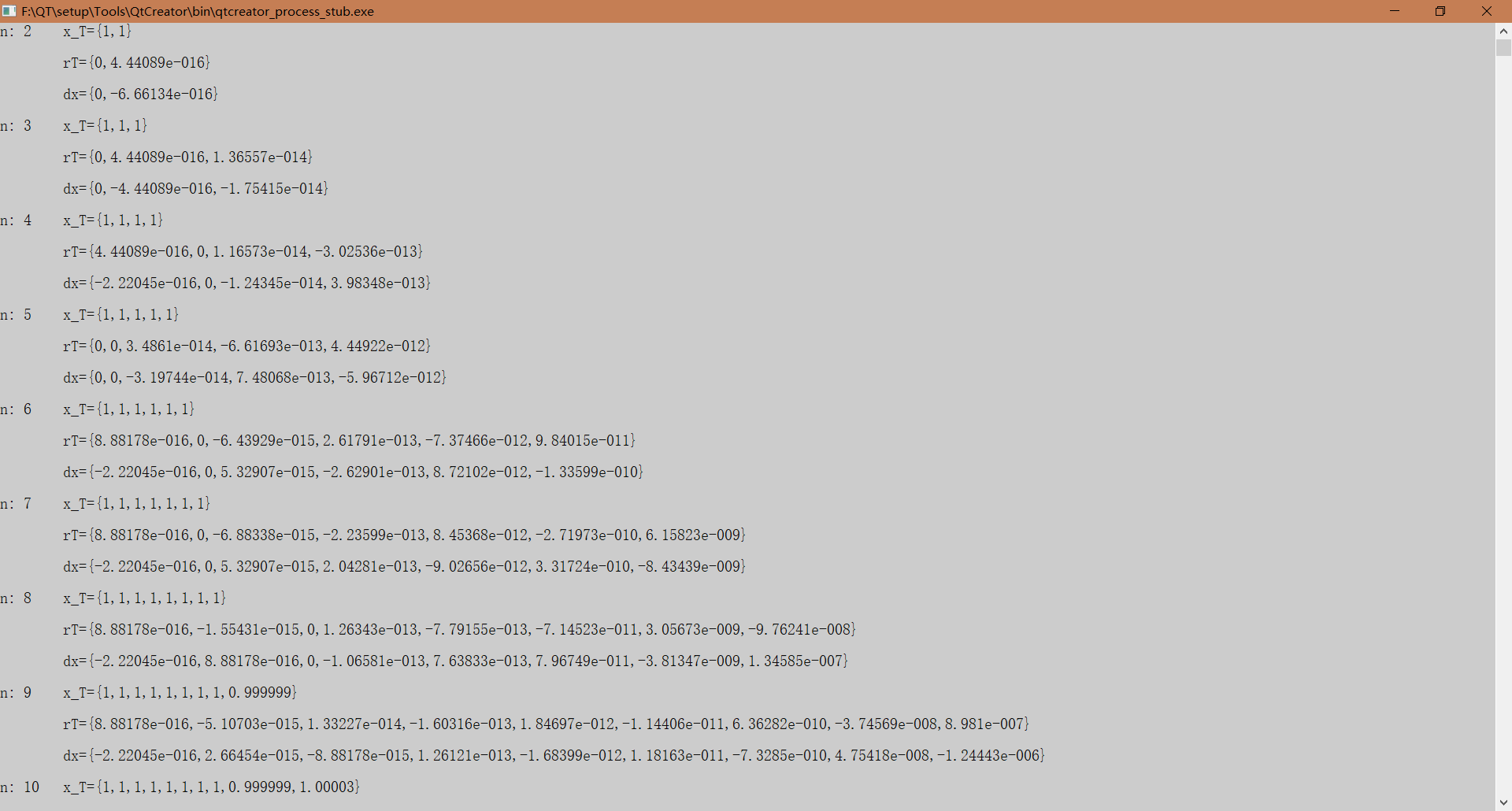
## 二、算法基本思想及复杂度分析：

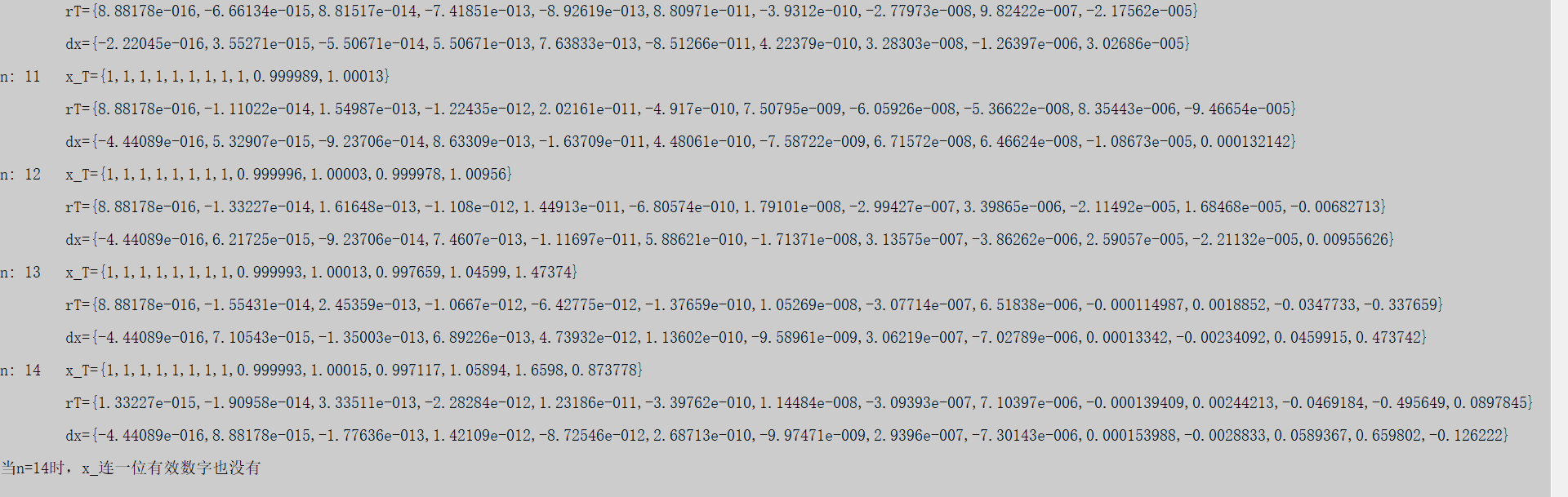
将矩阵分解为，相比将矩阵分解的Cholesky方法优化了计算方法，不用再进行开方运算。

## 三、源程序及注释：

1. //实验二
2. //改进的Cholesky方法
4. #include <iostream>
5. #include <iomanip>
6. #include <cmath>
7. **using** **namespace** std;
9. **int** main()
10. {
11. **double** h[101][101]; //h 希尔伯特矩阵
12. **double** x[101]={1};  //x hn\*x=bn
13. **double** x\_[101]={0}; //x\_ 所求值
14. **double** dx[101]={0}; //吊塔x x\_-x
15. **double** b[101]={0};  //b bn=hn\*x
16. **double** l[101][101]; //L
17. **double** d[101]={0};  //D
18. **double** c[101][101]; //c
19. **double** y[101]={0};  //y
20. **double** r[101]={0};  //r
22. //h 希尔伯特矩阵 和 x 赋值
23. //实际使用从h[1][1]开始
24. **for**(**int** i=1; i<=100; ++i){
25. **for**(**int** j=1; j<=100; ++j)
26. h[i][j]=1.0/(i+j-1);
27. x[i]=1;
28. }
30. **int** n=2;    //初始阶数
31. **while**(n<=100){
32. //为b清0
33. **for**(**int** i=1; i<=n; ++i)
34. b[i]=0;
36. //b bn=hn\*x 赋值
37. **for**(**int** i=1; i<=n; ++i)
38. **for**(**int** j=1; j<=n; ++j)
39. b[i]+=h[i][j];
41. d[1]=h[1][1];
43. //分解A=L\*D\*LT
44. **for**(**int** i=2; i<=n; ++i){
45. //(1)
46. **for**(**int** j=1; j<=i-1; ++j){
47. **double** sum=0; //累加 c[i][k]\*l[j][k]
49. c[i][j]=h[i][j];
50. **for**(**int** k=1; k<=j-1; ++k)
51. sum+=c[i][k]\*l[j][k];
53. c[i][j]=h[i][j]-sum;
54. l[i][j]=c[i][j]/d[j];
55. }
57. //(2)
58. d[i]=h[i][i];
59. **for**(**int** k=1; k<=i-1; ++k)
60. d[i]-=c[i][k]\*l[i][k];
61. }
63. //解 h\*x=b
64. //(1)解 L\*y=b
65. y[1]=b[1];
66. **for**(**int** i=2; i<=n; ++i){
67. **double** sum=0;   //累加 l[i][k]\*y[k]
68. **for**(**int** k=1; k<=i-1; ++k)
69. sum+=l[i][k]\*y[k];
71. y[i]=b[i]-sum;
72. }
74. //(2)解 LT\*x=D^-1\*y
75. x\_[n]=y[n]/d[n];
76. **for**(**int** i=n-1; i>=1; --i){
77. **double** sum=0;   //累加 l[k][i]\*x[k]
78. **for**(**int** k=i+1; k<=n; ++k)
79. sum+=l[k][i]\*x[k];
81. x\_[i]=y[i]/d[i]-sum;
82. }
84. //rn=bn-Hn\*x\_
85. **for**(**int** i=1; i<=n; ++i){
86. **double** sum=0;   //存贮 Hi\*x
87. **for**(**int** j=1; j<=n; ++j)
88. sum+=h[i][j]\*x\_[i];
90. r[i]=b[i]-sum;
91. }
93. //dx=x\_-x
94. **for**(**int** i=1; i<=n; ++i)
95. dx[i]=x\_[i]-x[i];
97. //输出
98. //n x\_
99. cout<<"n: "<<n<<"\tx\_T={";
100. **for**(**int** i=1; i<=n-1; ++i)
101. cout<<x\_[i]<<",";
102. cout<<x\_[n]<<"}"<<endl;
104. //r
105. cout<<endl<<"\trT={";
106. **for**(**int** i=1; i<=n-1; ++i)
107. cout<<r[i]<<",";
108. cout<<r[n]<<"}"<<endl;
110. //dx
111. cout<<endl<<"\tdx={";
112. **for**(**int** i=1; i<=n-1; ++i)
113. cout<<dx[i]<<",";
114. cout<<dx[n]<<"}"<<endl;
116. cout<<endl;
118. //判断条件 x\_有没有有效数字
119. **bool** judge=1;
120. **for**(**int** i=1; i<=n; ++i){
121. **if**(fabs(x\_[i]-x[i])>0.5){
122. judge=0;
123. **break**;
124. }
125. }
126. **if**(judge==1)
127. ++n;
128. **else**{
129. cout<<"当n="<<n<<"时，x\_连一位有效数字也没有";
130. **break**;
131. }
132. }
133. **return** 0;
134. }

## 四、运行输出结果：





当n增加时，x\_的有效位数会降低，当n等于14时，一位有效数字都没有了。

## 五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训：

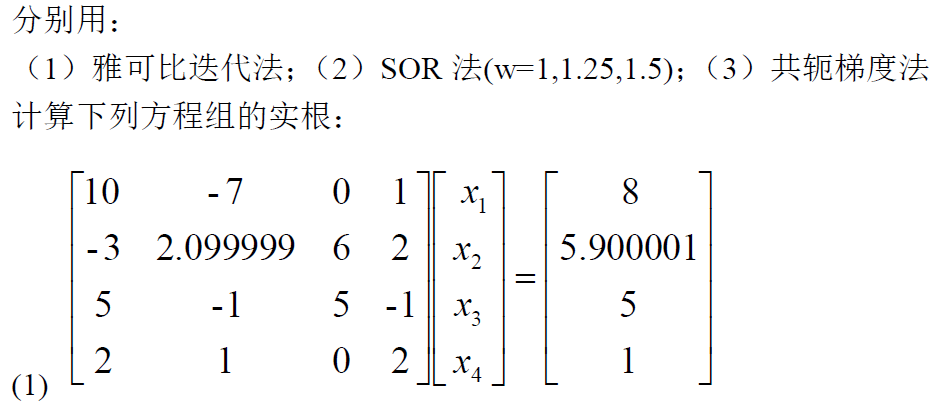
产生问题：输出结果一开始只到n=3的时候，并且X\_的值偏差很大。

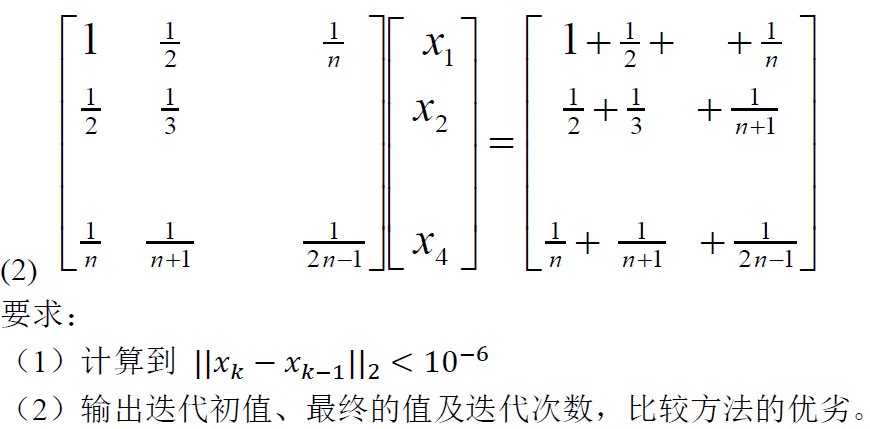
采取措施：后来发现是在每次循环的时候对b没有清0，于是加入了对b清0的操作

吸取教训：这个小问题卡了我很久很久，以后在码代码的时候要尽可能考虑周全。

# 实验三、线性方程组的迭代解法

## 一、实验内容：





## 二、算法基本思想及复杂度分析：

（一）雅可比迭代法

对于非奇异且对角线不为0的矩阵A，要解Ax=b，就将A分解为A=D-L-U（其中D=diag(a11,a22,……ann)），令M=D，N=L+U，可得雅可比迭代法：



也就是将方程组分解为，然后不断迭代。

（二）SOR法

在G-S迭代法（在雅可比迭代法的基础上，后面的式子采用前面的式子算出来的值进行计算）的基础上引入松弛因子w

（三）共轭梯度法

当前点xk，下降速度最大的方向是rk，前一个搜索方向是pk-1，在过xk，上述两方向张成的平面上找函数的最小值。

PS：关于收敛性判断，我们选取绝对不可能为解的值进行判断。本题中我判断如果计算得出的x1~xn的绝对值都大于1000则不收敛。这样判断收敛性会更加简单但对于某些特殊的运算可能不是很好确定这个绝对不可能为解的值。

## 三、源程序及注释：

（1）

（一）雅可比迭代法

1. //实验三(1)（一） 雅可比
3. #include <iostream>
4. #include <iomanip>
5. #include <cmath>
6. **using** **namespace** std;
8. //判断精度 ||x[k]-x[k-1]||2 < 1e-6
9. **bool** judge(**double** x[], **double** y[]) {
10. **double** x1, x2, x3, x4, tmp;
11. x1=y[0]-x[0];
12. x2=y[1]-x[1];
13. x3=y[2]-x[2];
14. x4=y[3]-x[3];
15. tmp=sqrt(x1\*x1+x2\*x2+x3\*x3+x4\*x4);
17. **if**(tmp>=1e-6)
18. **return** 1;
19. **else**
20. **return** 0;
21. }
23. **int** main() {
24. **double** a[4][4];
25. **double** b[4];
27. a[0][0]=10; a[0][1]=-7;         a[0][2]=0;  a[0][3]=1;  b[0]=8;
28. a[1][0]=-3; a[1][1]=2.099999;   a[1][2]=6;  a[1][3]=2;  b[1]=5.900001;
29. a[2][0]=5;  a[2][1]=-1;         a[2][2]=5;  a[2][3]=-1; b[2]=5;
30. a[3][0]=2;  a[3][1]=1;          a[3][2]=0;  a[3][3]=2;  b[3]=1;
32. **double** x[4]={0,0,0,0};  //xk 并赋初值
33. **double** y[4]={1,1,1,1};  //xk-1 保存前一个xk 赋合适的初值使得精度不满足判断条件
35. **int** n=0;    //记录迭代次数
37. **while** (judge(x,y)) {
38. //y存储上一个x的值
39. **for** (**int** i=0; i<4; ++i)
40. y[i] = x[i];
42. **for** (**int** i=0; i<4; ++i){
43. **double** sum=0;
44. **for**(**int** j=0; j<4; ++j)
45. **if**(j!=i)
46. sum+=a[i][j]\*y[j];
48. x[i]=1.0\*(b[i]-sum)/a[i][i];
49. }
51. n++;
53. //判断收敛性
54. **if**(fabs(x[0])>1000&&fabs(x[1])>1000&&fabs(x[2])>1000&&fabs(x[3])>1000)
55. **break**;
56. }
57. //输出
58. **if**(fabs(x[0])>1000&&fabs(x[1])>1000&&fabs(x[2])>1000&&fabs(x[3])>1000)
59. cout<<"不收敛"<<endl;
60. **else**{
61. cout<<"迭代初值：xT={0,0,0,0}"<<endl;
62. cout<<"迭代终值：xT={"<<x[0]<<","<<x[1]<<","<<x[2]<<","<<x[3]<<"}"<<endl;
63. cout<<"迭代次数："<<n<<endl;
64. cout<<endl;
65. }
66. **return** 0;
67. }

（二）SOR法

1. //实验三(1)（二） SOR
3. #include <iostream>
4. #include <iomanip>
5. #include <cmath>
6. **using** **namespace** std;
8. //判断精度 ||x[k]-x[k-1]||2 < 1e-6
9. **bool** judge(**double** x[], **double** y[]) {
10. **double** x1, x2, x3, x4, tmp;
11. x1=y[0]-x[0];
12. x2=y[1]-x[1];
13. x3=y[2]-x[2];
14. x4=y[3]-x[3];
15. tmp=sqrt(x1\*x1+x2\*x2+x3\*x3+x4\*x4);
17. **if**(tmp>=1e-6)
18. **return** 1;
19. **else**
20. **return** 0;
21. }
23. **int** main() {
24. **double** a[4][4];
25. **double** b[4];
26. **double** w[3]={1,1.25,1.5};
28. a[0][0]=10; a[0][1]=-7;         a[0][2]=0;  a[0][3]=1;  b[0]=8;
29. a[1][0]=-3; a[1][1]=2.099999;   a[1][2]=6;  a[1][3]=2;  b[1]=5.900001;
30. a[2][0]=5;  a[2][1]=-1;         a[2][2]=5;  a[2][3]=-1; b[2]=5;
31. a[3][0]=2;  a[3][1]=1;          a[3][2]=0;  a[3][3]=2;  b[3]=1;
33. **int** k=0; //对w的三个给定值分别求解
34. **while**(k!=3){
35. cout<<"w="<<w[k]<<endl;
36. k++;
38. **double** x[4]={0,0,0,0};  //xk 并赋初值
39. **double** y[4]={1,1,1,1};  //xk-1 保存前一个xk 赋合适的初值使得精度不满足判断条件
41. **int** n=0;    //记录迭代次数
43. **while** (judge(x,y)) {
44. //y存储上一个x的值
45. **for** (**int** i=0; i<4; ++i)
46. y[i] = x[i];
48. **for** (**int** i=0; i<4; ++i){
49. **double** sum=0;
50. **for**(**int** j=0; j<4; ++j)
51. **if**(j!=i)
52. sum+=a[i][j]\*x[j];
54. x[i]=(1-w[k-1])\*x[i]+1.0\*w[k-1]\*(b[i]-sum)/a[i][i];
55. }
57. n++;
59. //判断收敛性
60. **if**(fabs(x[0])>1000&&fabs(x[1])>1000&&fabs(x[2])>1000&&fabs(x[3])>1000)
61. **break**;
62. }
63. //输出
64. **if**(fabs(x[0])>1000&&fabs(x[1])>1000&&fabs(x[2])>1000&&fabs(x[3])>1000){
65. cout<<"不收敛"<<endl;
66. cout<<endl;
67. }
68. **else**{
69. cout<<"迭代初值：xT={0,0,0,0}"<<endl;
70. cout<<"迭代终值：xT={"<<x[0]<<","<<x[1]<<","<<x[2]<<","<<x[3]<<"}"<<endl;
71. cout<<"迭代次数："<<n<<endl;
72. cout<<endl;
73. }
74. }
75. **return** 0;
76. }

（三）共轭梯度法

1. //实验三(1)（三） 共轭梯度法
3. #include <iostream>
4. #include <iomanip>
5. #include <cmath>
6. **using** **namespace** std;
8. //判断精度 ||x[k]-x[k-1]||2 < 1e-6
9. **bool** judge(**double** x[], **double** y[]) {
10. **double** x1, x2, x3, x4, tmp;
11. x1=y[0]-x[0];
12. x2=y[1]-x[1];
13. x3=y[2]-x[2];
14. x4=y[3]-x[3];
15. tmp=sqrt(x1\*x1+x2\*x2+x3\*x3+x4\*x4);
17. **if**(tmp>=1e-6)
18. **return** 1;
19. **else**
20. **return** 0;
21. }
23. **int** main()
24. {
25. **double** a[4][4];
26. **double** b[4];
28. a[0][0]=10; a[0][1]=-7;         a[0][2]=0;  a[0][3]=1;  b[0]=8;
29. a[1][0]=-3; a[1][1]=2.099999;   a[1][2]=6;  a[1][3]=2;  b[1]=5.900001;
30. a[2][0]=5;  a[2][1]=-1;         a[2][2]=5;  a[2][3]=-1; b[2]=5;
31. a[3][0]=2;  a[3][1]=1;          a[3][2]=0;  a[3][3]=2;  b[3]=1;
33. **double** x[4]={0,0,0,0};  //xk 并赋初值
34. **double** y[4]={1,1,1,1};  //xk-1 保存前一个xk 赋合适的初值使得精度不满足判断条件
36. **double** r[4]={0};
37. **double** p[4]={0};
38. **double** alpha=0;
39. **double** r2[4]={0};
40. **double** beta=0;
42. //r=b-Ax  p=r
43. **for**(**int** i=0; i<4; ++i){
44. r[i]=b[i]-0;
45. p[i]=r[i];
46. }
48. **int** n=0;
50. **while**(judge(x,y)){
51. //y存储上一个x的值
52. **for**(**int** i=0; i<4; ++i)
53. y[i]=x[i];
55. //计算alpha （计算搜索步长）  alpha=(rT\*r)/(pT\*A\*p)
56. **double** sum1=0;
57. **double** sum2=0;
58. **double** sum3=0;
59. **for**(**int** i=0; i<4; ++i){
60. sum1+=r[i]\*r[i];
61. sum2=0;
62. **for**(**int** j=0; j<4; ++j)
63. sum2+=p[j]\*a[j][i];
64. sum3+=sum2\*p[i];
65. }
66. alpha=1.0\*sum1/sum3;
68. //计算x (更新解) x=x+alpha\*p
69. **for**(**int** i=0; i<4; ++i)
70. x[i]=x[i]+alpha\*p[i];
72. //计算r2  r2=r
73. **for**(**int** i=0; i<4; ++i)
74. r2[i]=r[i];
76. //计算r（更新残差向量）   r=r-alpha\*A\*p
77. **for**(**int** i=0; i<4; ++i){
78. **double** sum=0;
79. **for**(**int** j=0; j<4; ++j)
80. sum+=a[i][j]\*p[j];
81. r[i]=r[i]-alpha\*sum;
82. }
84. //计算beta    beta=(rT\*r)/(r2T\*r2)
85. **double** sum4=0;
86. **double** sum5=0;
87. **for**(**int** i=0; i<4; ++i){
88. sum4+=r[i]\*r[i];
89. sum5+=r2[i]\*r2[i];
90. }
91. beta=1.0\*sum4/sum5;
93. //计算p（新的搜索方向）   p=r+beta\*p
94. **for**(**int** i=0; i<4; ++i)
95. p[i]=r[i]+beta\*p[i];
97. n++;
99. //判断收敛性
100. **if**(fabs(x[0])>1000&&fabs(x[1])>1000&&fabs(x[2])>1000&&fabs(x[3])>1000)
101. **break**;
102. }
103. //输出
104. **if**(fabs(x[0])>1000&&fabs(x[1])>1000&&fabs(x[2])>1000&&fabs(x[3])>1000)
105. cout<<"不收敛"<<endl;
106. **else**{
107. cout<<"迭代初值：xT={0,0,0,0}"<<endl;
108. cout<<"迭代终值：xT={"<<x[0]<<","<<x[1]<<","<<x[2]<<","<<x[3]<<"}"<<endl;
109. cout<<"迭代次数："<<n<<endl;
110. cout<<endl;
111. }
112. **return** 0;
113. }

（2）

（一）雅可比迭代法

1. //实验三(2)（一） 雅可比
3. #include <iostream>
4. #include <iomanip>
5. #include <cmath>
6. **using** **namespace** std;
8. //判断精度 ||x[k]-x[k-1]||2 < 1e-6
9. **bool** judge(**int** m, **double** x[], **double** y[]) {
10. **double** dx[100], tmp, sum=0;
11. **for**(**int** i=0; i<m; ++i){
12. dx[i]=x[i]-y[i];
13. sum+=dx[i]\*dx[i];
14. }
15. tmp=sqrt(sum);
17. **if**(tmp>=1e-6)
18. **return** 1;
19. **else**
20. **return** 0;
21. }
23. **int** main(){
24. //阶数
25. **for**(**int** m=2; m<=10; ++m){
26. cout<<"方程阶数："<<m<<endl;
28. **double** a[100][100];
29. **double** b[100]={0};
31. **for**(**int** i=0; i<m; ++i)
32. **for**(**int** j=0; j<m; ++j){
33. a[i][j]=1.0/(i+j+1);
34. b[i]+=a[i][j];
35. }
37. **double** x[100]={0};  //xk 并赋初值
38. **double** y[100]={0};  //xk-1 保存前一个xk 赋合适的初值使得精度不满足判断条件
39. **for**(**int** i=0; i<m; ++i)
40. y[i]=1;
42. **int** n=0;    //记录迭代次数
43. **bool** jd2=0;  //用于判断收敛性
45. **while** (judge(m,x,y)) {
46. //y存储上一个x的值
47. **for** (**int** i=0; i<m; ++i)
48. y[i] = x[i];
50. **for** (**int** i=0; i<m; ++i){
51. **double** sum=0;
52. **for**(**int** j=0; j<m; ++j)
53. **if**(j!=i)
54. sum+=a[i][j]\*y[j];
56. x[i]=1.0\*(b[i]-sum)/a[i][i];
57. }
59. n++;
61. //判断收敛性
62. **bool** jd=1;
63. **for**(**int** i=0; i<m; ++i)
64. **if**(fabs(x[i])<1000)
65. jd=0;
66. **if**(jd){
67. jd2=1;
68. **break**;
69. }
70. }
71. //输出
72. **if**(jd2){
73. cout<<"不收敛"<<endl;
74. cout<<endl;
75. }
76. **else**{
77. cout<<"迭代初值：xT={0";
78. **for**(**int** i=1; i<m; ++i)
79. cout<<",0";
80. cout<<"}"<<endl;
81. cout<<"迭代终值：xT={"<<x[0];
82. **for**(**int** i=1; i<m; ++i)
83. cout<<","<<x[i];
84. cout<<"}"<<endl;
85. cout<<"迭代次数："<<n<<endl;
86. cout<<endl;
87. }
88. }
89. **return** 0;
90. }

（二）SOR法

1. //实验三(2)（二） SOR
3. #include <iostream>
4. #include <iomanip>
5. #include <cmath>
6. **using** **namespace** std;
8. //判断精度 ||x[k]-x[k-1]||2 < 1e-6
9. **bool** judge(**int** m, **double** x[], **double** y[]) {
10. **double** dx[100], tmp, sum=0;
11. **for**(**int** i=0; i<m; ++i){
12. dx[i]=x[i]-y[i];
13. sum+=dx[i]\*dx[i];
14. }
15. tmp=sqrt(sum);
17. **if**(tmp>=1e-6)
18. **return** 1;
19. **else**
20. **return** 0;
21. }
23. **int** main() {
24. //阶数
25. **for**(**int** m=2; m<=10; ++m){
26. cout<<"方程阶数："<<m<<endl;
28. **double** a[100][100];
29. **double** b[100]={0};
30. **double** w[3]={1,1.25,1.5};
32. **for**(**int** i=0; i<m; ++i)
33. **for**(**int** j=0; j<m; ++j){
34. a[i][j]=1.0/(i+j+1);
35. b[i]+=a[i][j];
36. }
38. **int** k=0; //对w的三个给定值分别求解
39. **while**(k!=3){
40. cout<<"w="<<w[k]<<endl;
41. k++;
43. **double** x[100]={0};  //xk 并赋初值
44. **double** y[100]={0};  //xk-1 保存前一个xk 赋合适的初值使得精度不满足判断条件
45. **for**(**int** i=0; i<m; ++i)
46. y[i]=1;
48. **int** n=0;    //记录迭代次数
49. **bool** jd2=0;  //用于判断收敛性
51. **while** (judge(m,x,y)) {
52. //y存储上一个x的值
53. **for** (**int** i=0; i<m; ++i)
54. y[i] = x[i];
56. **for** (**int** i=0; i<m; ++i){
57. **double** sum=0;
58. **for**(**int** j=0; j<m; ++j)
59. **if**(j!=i)
60. sum+=a[i][j]\*x[j];
62. x[i]=(1-w[k-1])\*x[i]+w[k-1]\*(b[i]-sum)/a[i][i];
63. }
65. n++;
67. //判断收敛性
68. **bool** jd=1;
69. **for**(**int** i=0; i<m; ++i)
70. **if**(fabs(x[i])<1000)
71. jd=0;
72. **if**(jd){
73. jd2=1;
74. **break**;
75. }
76. }
77. //输出
78. **if**(jd2){
79. cout<<"不收敛"<<endl;
80. cout<<endl;
81. }
82. **else**{
83. cout<<"迭代初值：xT={0";
84. **for**(**int** i=1; i<m; ++i)
85. cout<<",0";
86. cout<<"}"<<endl;
87. cout<<"迭代终值：xT={"<<x[0];
88. **for**(**int** i=1; i<m; ++i)
89. cout<<","<<x[i];
90. cout<<"}"<<endl;
91. cout<<"迭代次数："<<n<<endl;
92. cout<<endl;
93. }
94. }
95. }
96. **return** 0;
97. }

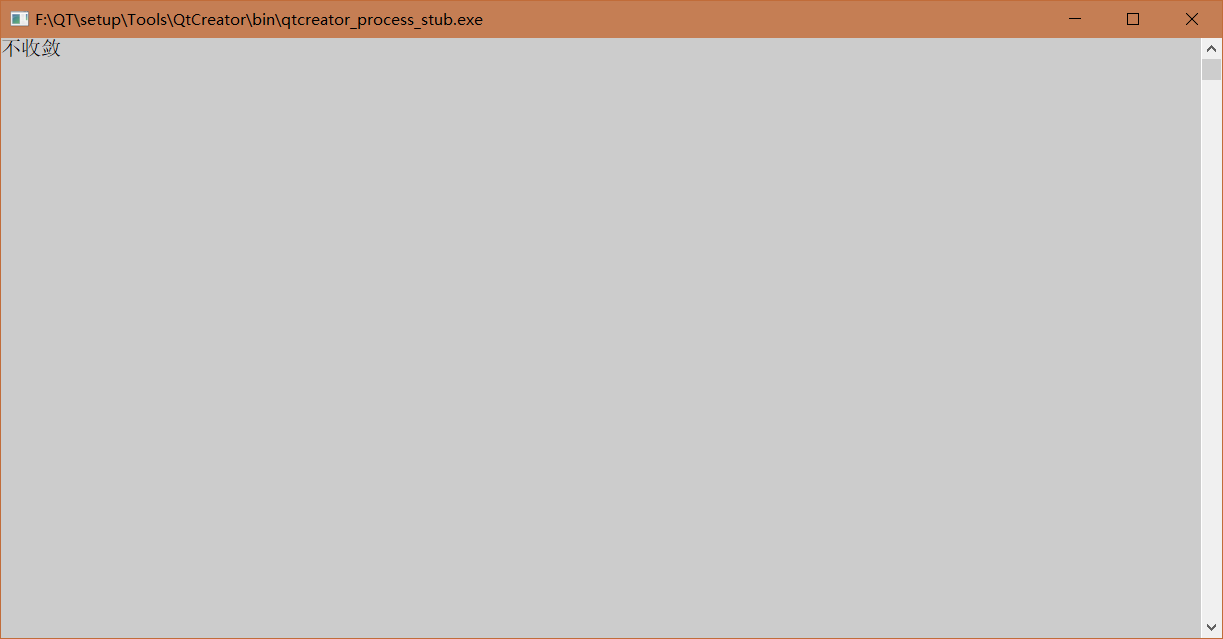
（三）共轭梯度法

1. //实验三(2)（三） 共轭梯度法
3. #include <iostream>
4. #include <iomanip>
5. #include <cmath>
6. **using** **namespace** std;
8. //判断精度 ||x[k]-x[k-1]||2 < 1e-6
9. **bool** judge(**int** m, **double** x[], **double** y[]) {
10. **double** dx[100], tmp, sum=0;
11. **for**(**int** i=0; i<m; ++i){
12. dx[i]=x[i]-y[i];
13. sum+=dx[i]\*dx[i];
14. }
15. tmp=sqrt(sum);
17. **if**(tmp>=1e-6)
18. **return** 1;
19. **else**
20. **return** 0;
21. }
23. **int** main()
24. {
25. //阶数
26. **for**(**int** m=2; m<=10; ++m){
27. cout<<"方程阶数："<<m<<endl;
29. **double** a[100][100];
30. **double** b[100]={0};
32. **for**(**int** i=0; i<m; ++i)
33. **for**(**int** j=0; j<m; ++j){
34. a[i][j]=1.0/(i+j+1);
35. b[i]+=a[i][j];
36. }
38. **double** x[100]={0};  //xk 并赋初值
39. **double** y[100]={0};  //xk-1 保存前一个xk 赋合适的初值使得精度不满足判断条件
40. **for**(**int** i=0; i<m; ++i)
41. y[i]=1;
43. **double** r[100]={0};
44. **double** p[100]={0};
45. **double** alpha=0;
46. **double** r2[100]={0};
47. **double** beta=0;
49. //r=b-Ax  p=r
50. **for**(**int** i=0; i<m; ++i){
51. r[i]=b[i]-0;
52. p[i]=r[i];
53. }
55. **int** n=0;
56. **bool** jd2=0;  //用于判断收敛性
58. **while**(judge(m,x,y)){
59. //y存储上一个x的值
60. **for**(**int** i=0; i<m; ++i)
61. y[i]=x[i];
63. //计算alpha （计算搜索步长）  alpha=(rT\*r)/(pT\*A\*p)
64. **double** sum1=0;
65. **double** sum2=0;
66. **double** sum3=0;
67. **for**(**int** i=0; i<m; ++i){
68. sum1+=r[i]\*r[i];
69. sum2=0;
70. **for**(**int** j=0; j<m; ++j)
71. sum2+=p[j]\*a[j][i];
72. sum3+=sum2\*p[i];
73. }
74. alpha=1.0\*sum1/sum3;
76. //计算x (更新解) x=x+alpha\*p
77. **for**(**int** i=0; i<m; ++i)
78. x[i]=x[i]+alpha\*p[i];
80. //计算r2  r2=r
81. **for**(**int** i=0; i<m; ++i)
82. r2[i]=r[i];
84. //计算r（更新残差向量）   r=r-alpha\*A\*p
85. **for**(**int** i=0; i<m; ++i){
86. **double** sum=0;
87. **for**(**int** j=0; j<m; ++j)
88. sum+=a[i][j]\*p[j];
89. r[i]=r[i]-alpha\*sum;
90. }
92. //计算beta    beta=(rT\*r)/(r2T\*r2)
93. **double** sum4=0;
94. **double** sum5=0;
95. **for**(**int** i=0; i<m; ++i){
96. sum4+=r[i]\*r[i];
97. sum5+=r2[i]\*r2[i];
98. }
99. beta=1.0\*sum4/sum5;
101. //计算p（新的搜索方向）   p=r+beta\*p
102. **for**(**int** i=0; i<m; ++i)
103. p[i]=r[i]+beta\*p[i];
105. n++;
107. //判断收敛性
108. **bool** jd=1;
109. **for**(**int** i=0; i<m; ++i)
110. **if**(fabs(x[i])<1000)
111. jd=0;
112. **if**(jd){
113. jd2=1;
114. **break**;
115. }
116. }
117. //输出
118. **if**(jd2){
119. cout<<"不收敛"<<endl;
120. cout<<endl;
121. }
122. **else**{
123. cout<<"迭代初值：xT={0";
124. **for**(**int** i=1; i<m; ++i)
125. cout<<",0";
126. cout<<"}"<<endl;
127. cout<<"迭代终值：xT={"<<x[0];
128. **for**(**int** i=1; i<m; ++i)
129. cout<<","<<x[i];
130. cout<<"}"<<endl;
131. cout<<"迭代次数："<<n<<endl;
132. cout<<endl;
133. }
134. }
135. **return** 0;
136. }

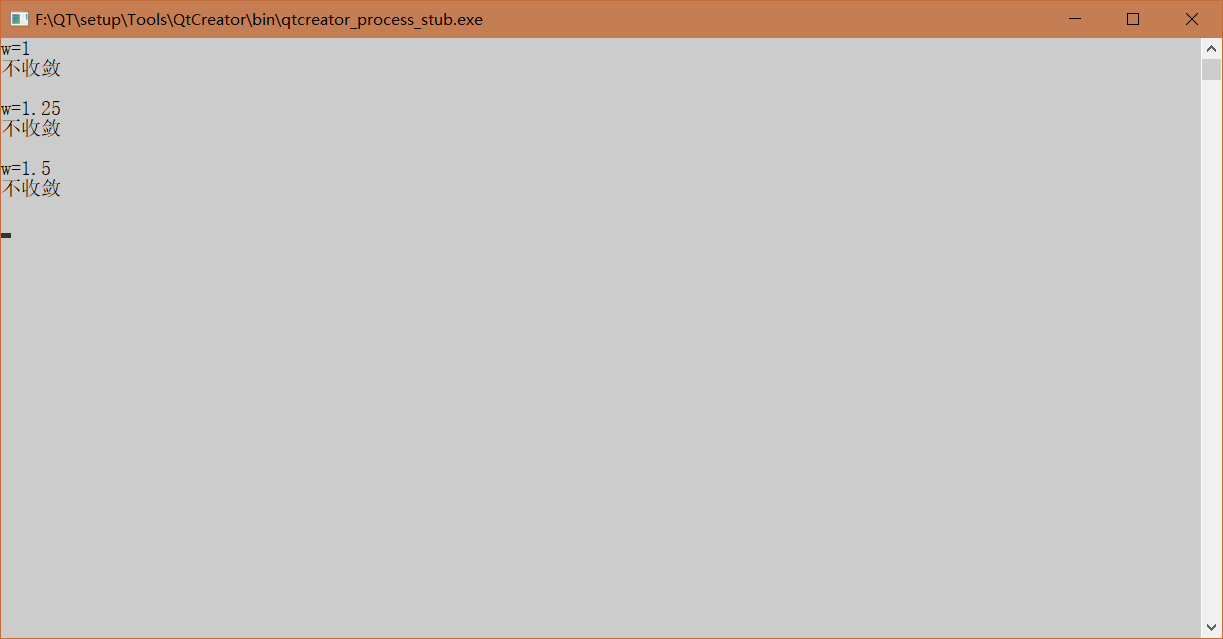
## 四、运行输出结果：

（1）

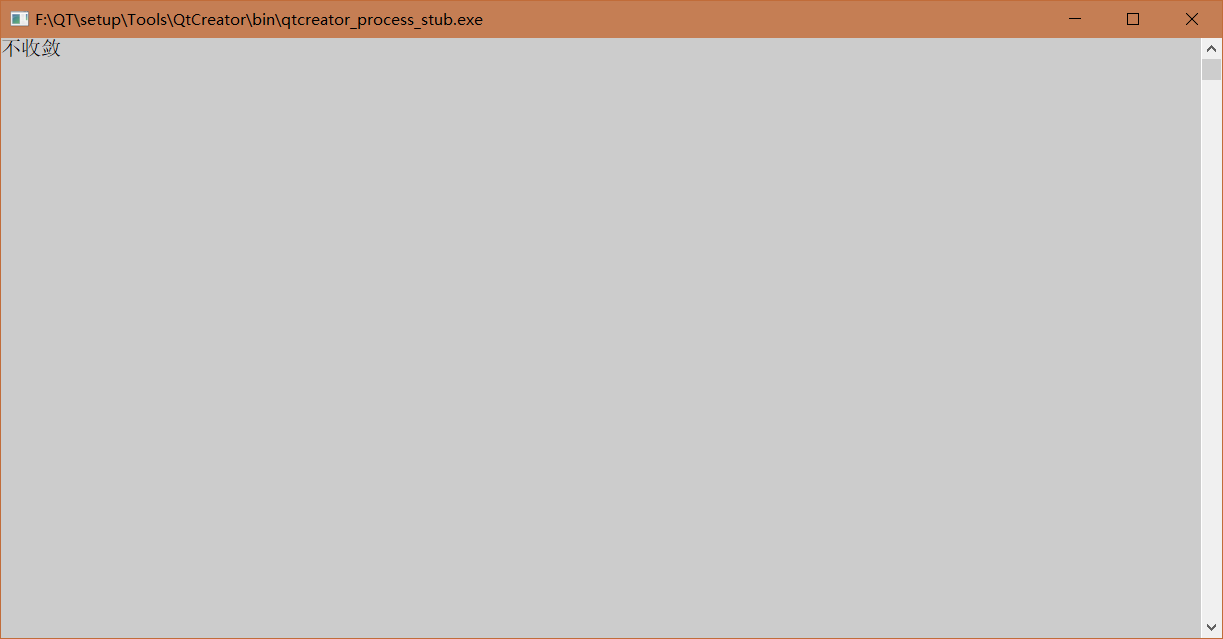
（一）雅可比迭代法



（二）SOR法

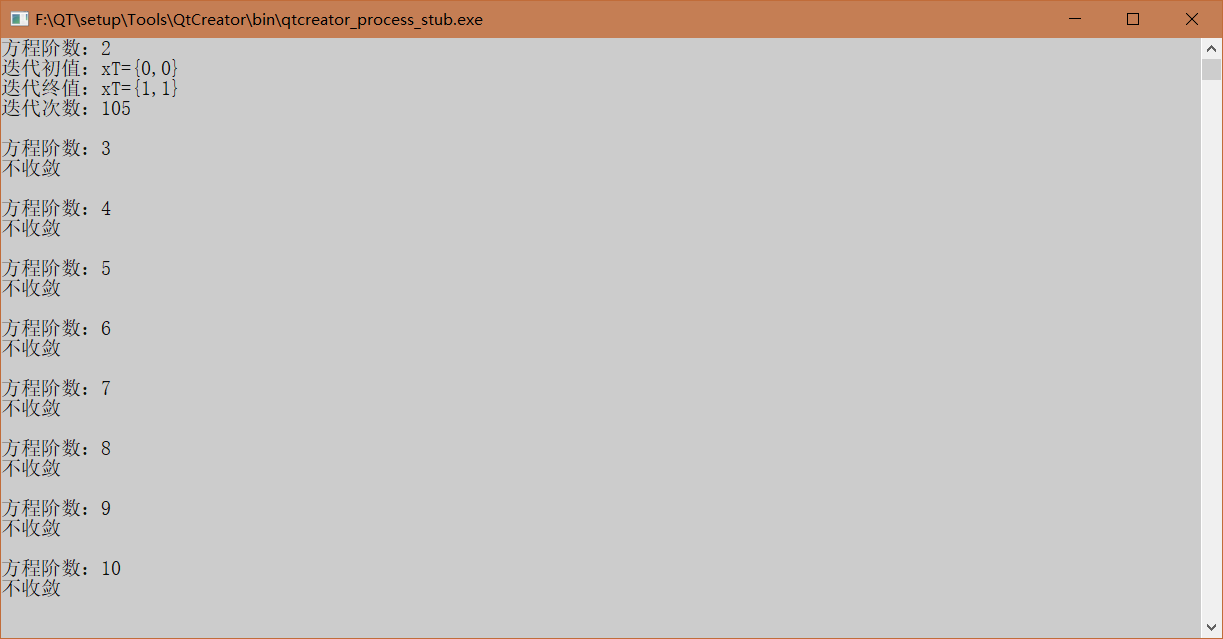


（三）共轭梯度法

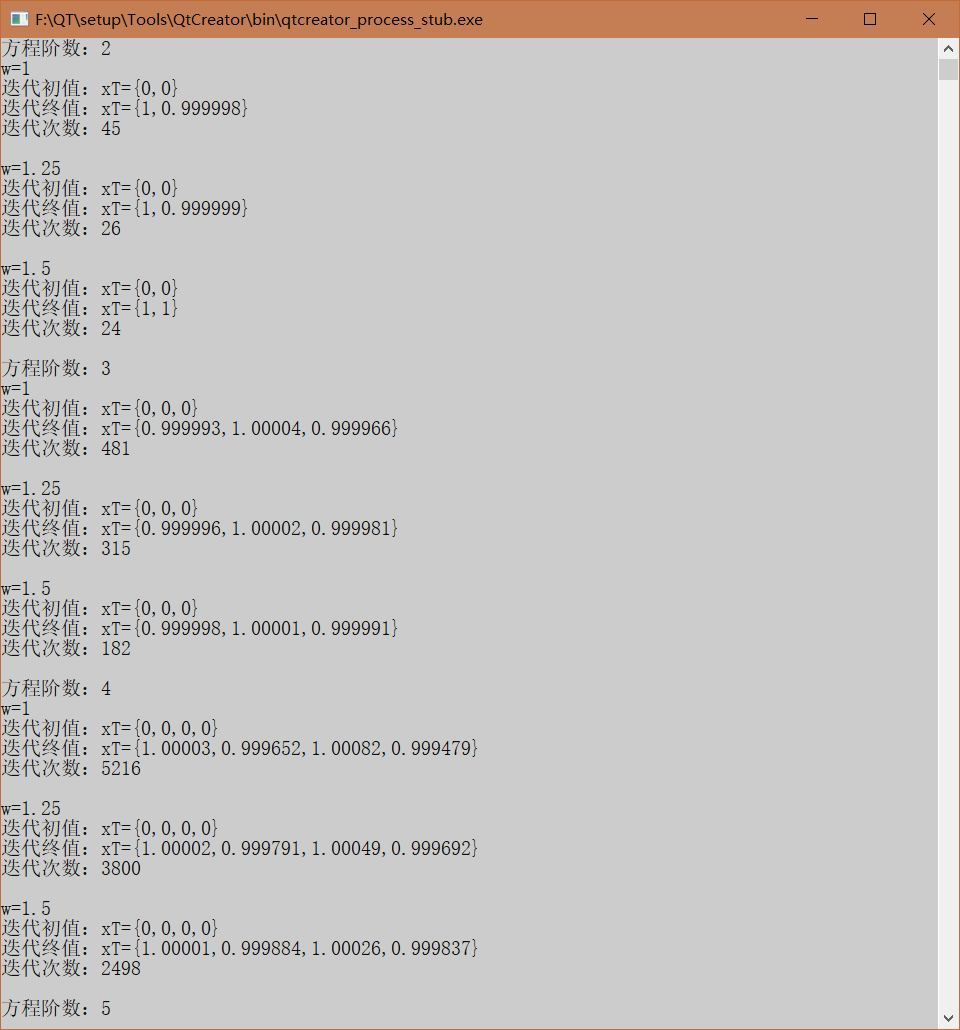


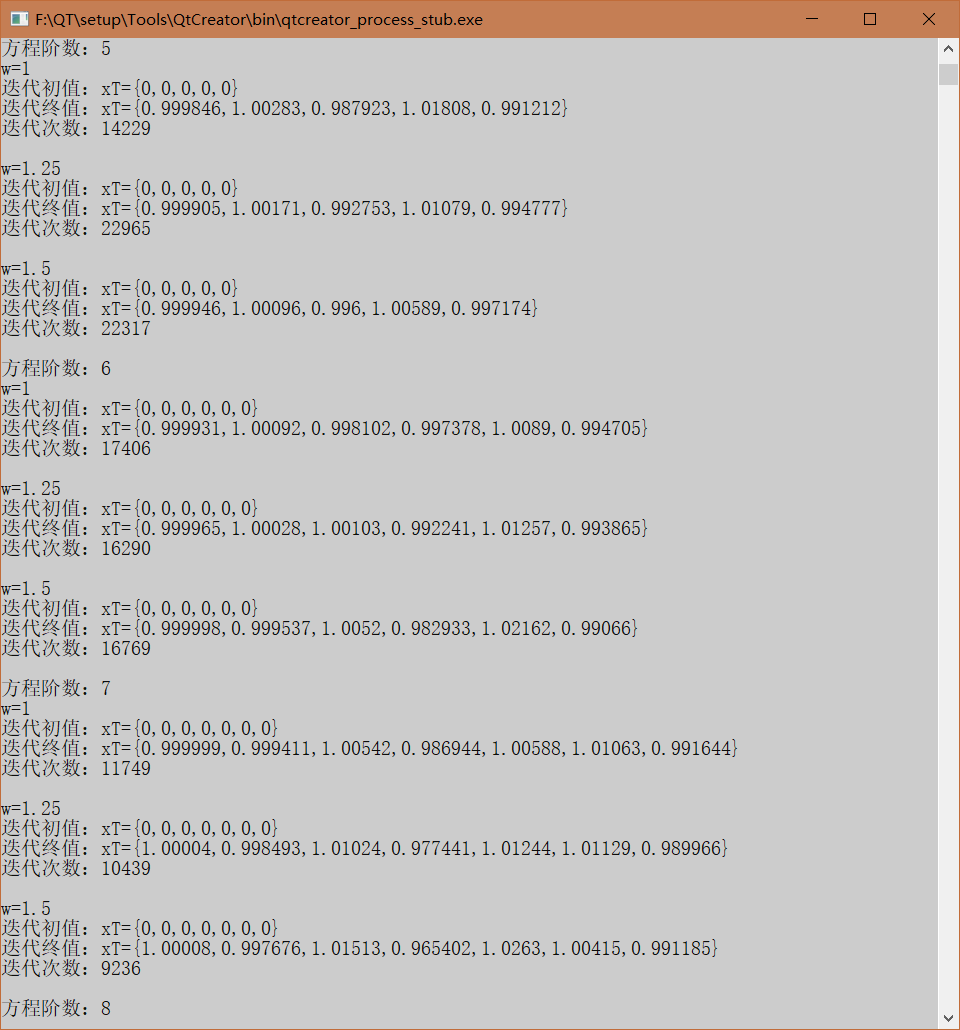
（2）

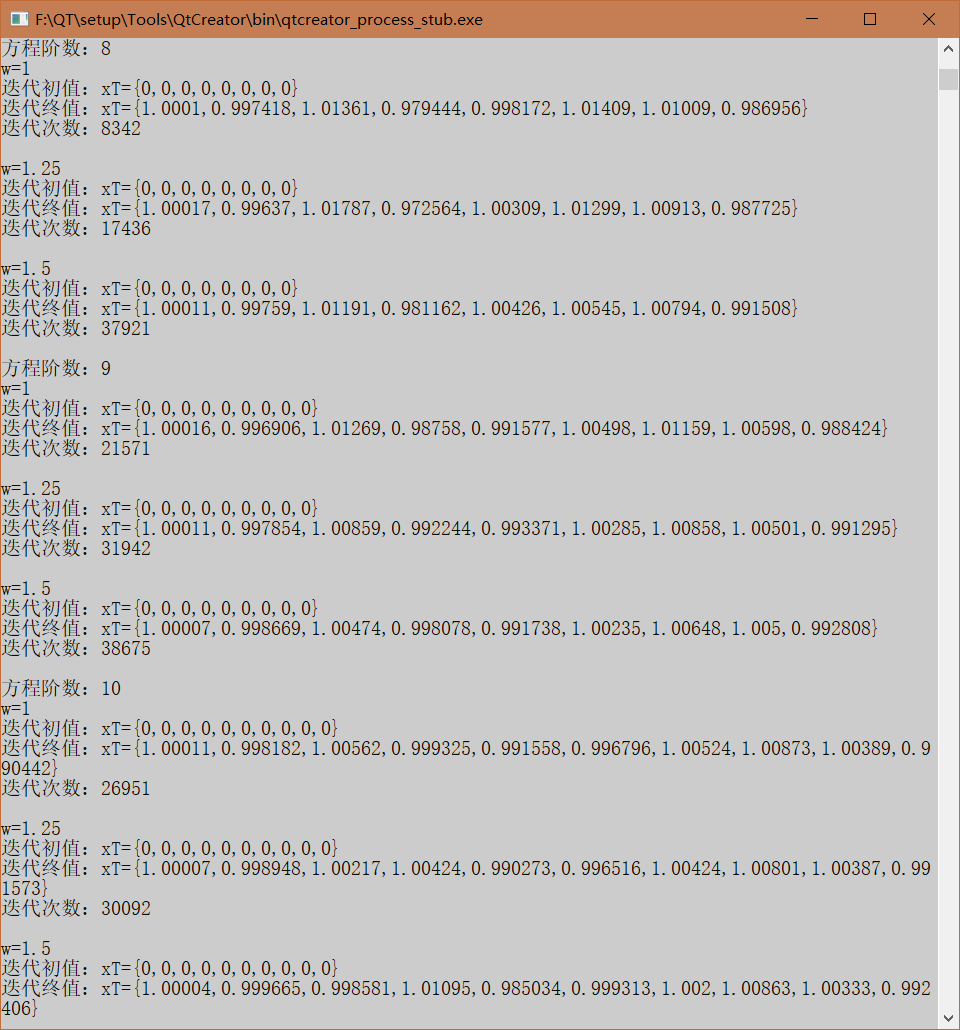
（一）雅可比迭代法



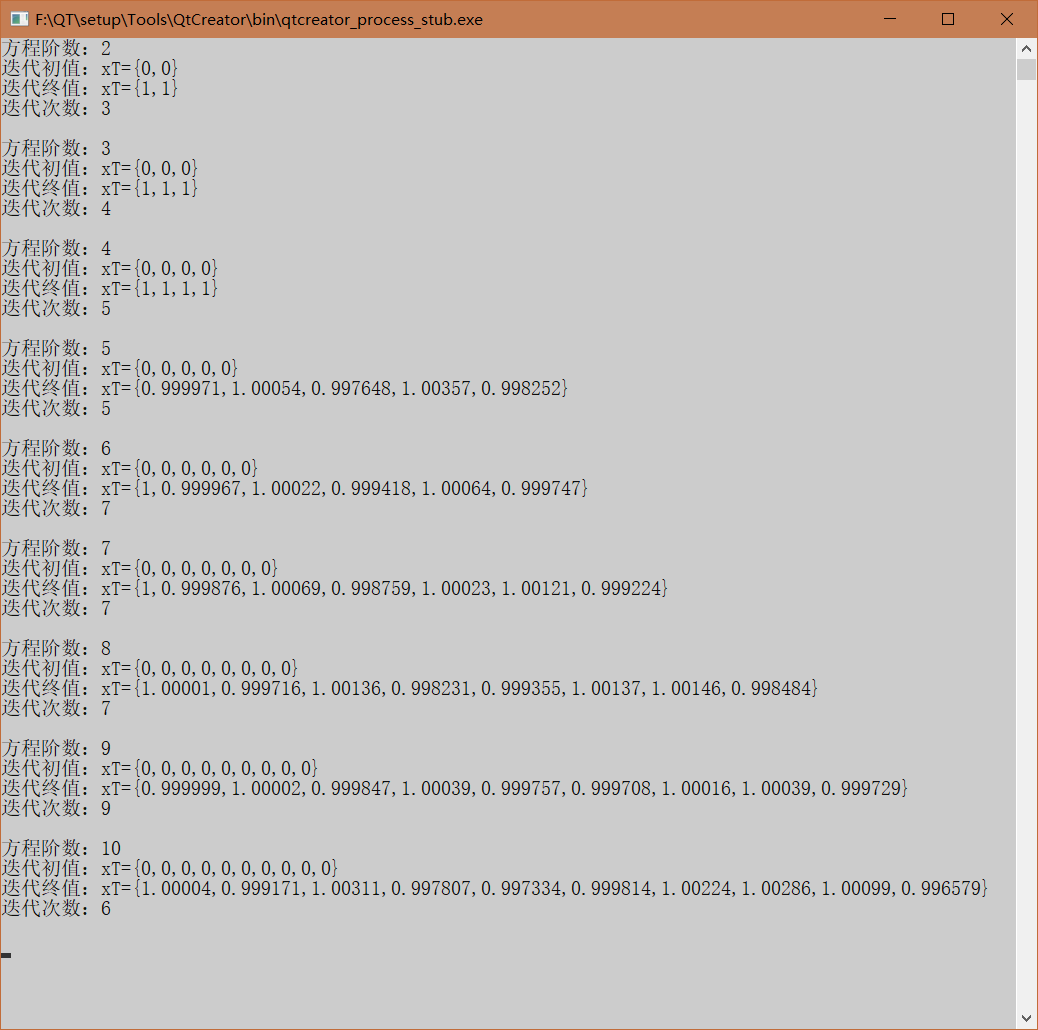
（二）SOR法







（三）共轭梯度法



可见，雅可比方法的收敛性以及收敛速度没有保障，而SOR法由于加入了松弛因子让收敛速度有了一定的提升。而共轭梯度法的速度是最快的。

## 五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训：

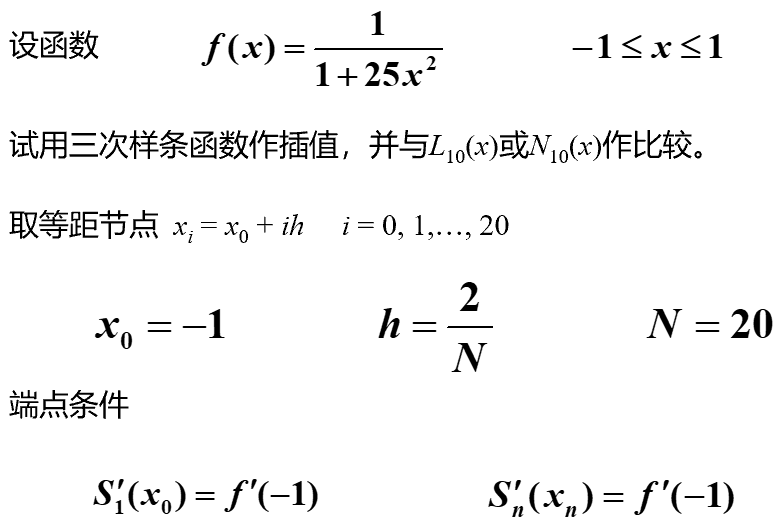
产生问题：输出为Inf。

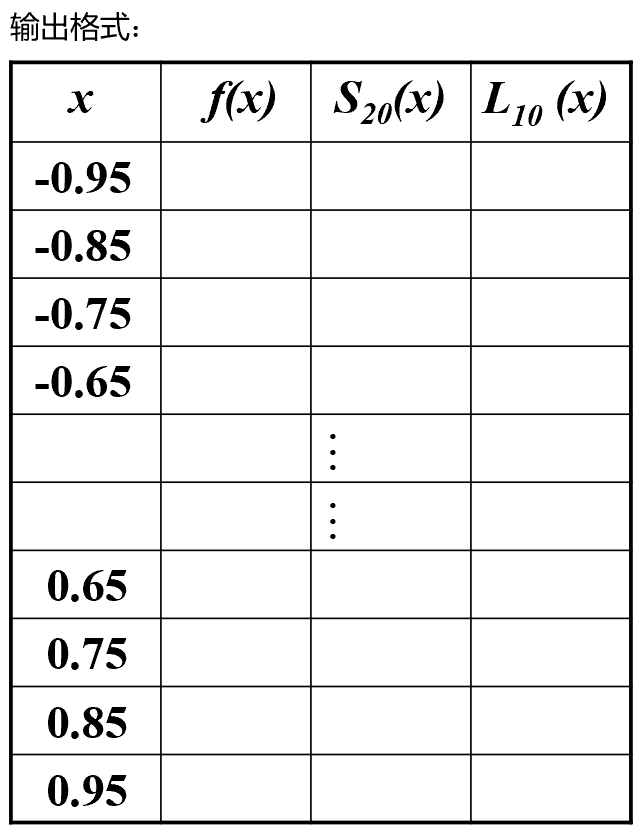
采取措施：加入了收敛判断，做法见基本思想。

吸取教训：考虑要全面。

# 实验四、函数插值

## 一、实验内容：





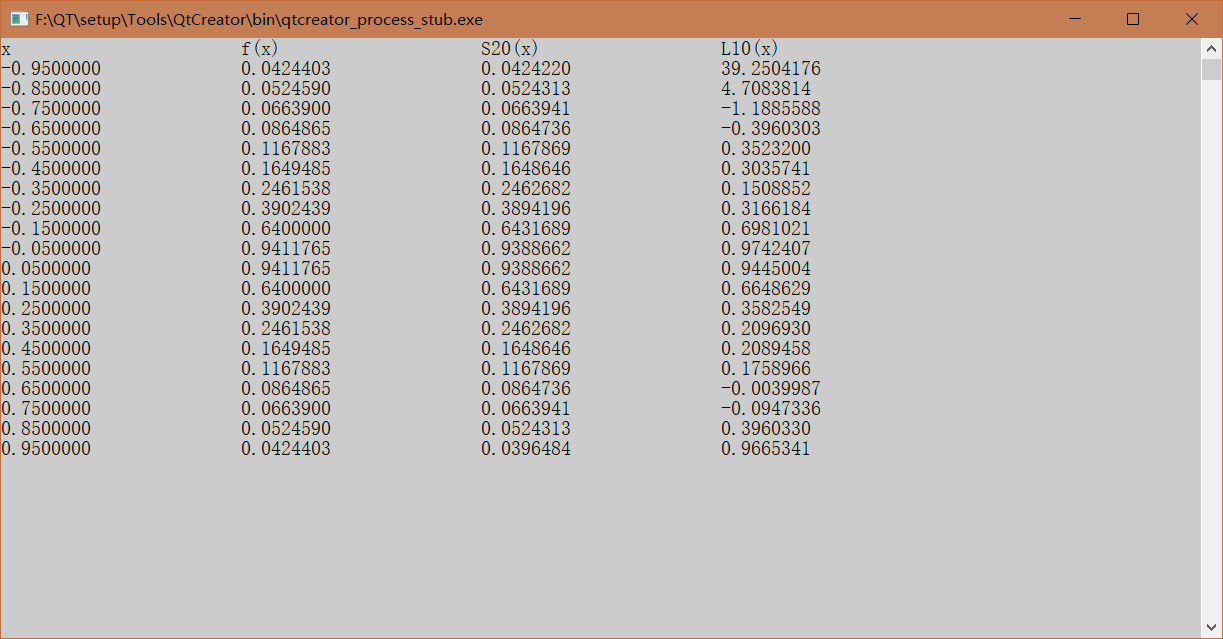
## 二、算法基本思想及复杂度分析：

所谓三次样条插值，就是给定插值节点以及函数值，求定义在[a, b]上的插值函数S(x)。首先构造出一个以m为未知数的三对角方程组，之后代入端点条件求出方程组的第一个和最后一个方程（本题可求d0和dn），之后用追赶法解方程组从而计算出S(x)即可。

## 三、源程序及注释：

1. //实验四 三次样条插值
3. #include <iostream>
4. #include <cmath>
5. #include <iomanip>
6. **using** **namespace** std;
8. //f(x)
9. **double** f(**double** x){
10. **return** 1.0/(1+25\*x\*x);
11. }
13. //f'(x)
14. **double** f\_(**double** x){
15. **return** -50\*x/((1+25\*x\*x)\*(1+25\*x\*x));
16. }
18. //三次样条插值函数 S(x)
19. //i表示 si(x)
20. //x\_表示 -0.95~0.95的插值节点
21. **double** s(**int** i, **double** x[], **double** m[], **double** y[], **double** h, **double** x\_){
22. **return** m[i-1]\*(x[i]-x\_)\*(x[i]-x\_)\*(x[i]-x\_)/(6\*h)
23. +m[i]\*(x\_-x[i-1])\*(x\_-x[i-1])\*(x\_-x[i-1])/(6\*h)
24. +(y[i-1]-m[i-1]/6\*h\*h)\*(x[i]-x\_)/h
25. +(y[i]-m[i]/6\*h\*h)\*(x\_-x[i-1])/h;
26. }
28. //Lagrange插值法 l(x)
29. //n表示 ln(x)
30. //x\_表示 -0.95~0.95的插值节点
31. **double** l(**int** n, **double** x[], **double** x\_){
32. **double** sum=0;
33. **for**(**int** k=0; k<n; ++k){
34. **double** product=1;
35. **for**(**int** i=0; i<n; ++i){
36. **if**(i!=k)
37. product\*=(x\_-x[i])/(x[k]-x[i]);
38. }
39. sum+=f(x[k])\*product;
40. }
41. **return** sum;
42. }
44. **int** main(){
45. **double** x[21]={0};   //x 等距节点
46. **double** y[21]={0};
47. **for**(**int** i=0; i<=20; ++i){
48. x[i]=-1+0.1\*i;
49. y[i]=f(x[i]);
50. }
52. **double** h=0.1;   //h=2/N N=20 则 h=0.1
53. **double** lambda=0.5;  //由于等距节点 lambda=h/(h+h)=1/2
54. **double** mu=0.5;      //mu=1-lambda=1/2
55. **int** ci=2;
57. **double** d[21]={0};
58. //由于等距节点 求 d[1]-d[19] 的值
59. **for**(**int** i=1; i<=19; ++i){
60. d[i]=3.0/(h\*h)\*(y[i+1]-2\*y[i]+y[i-1]);
61. }
63. //解 n-1阶 三对角方程组
64. //采用 追赶法    第三章1-64   a=mu b=c c=lambda x=m f=d
65. **double** a[21]={0};
66. **double** b[21]={0};
67. **double** c[21]={0};
69. **for**(**int** i=1; i<=19; ++i){
70. a[i]=mu;
71. b[i]=ci;
72. c[i]=lambda;
73. }
74. a[1]=0;
75. c[19]=0;
77. //计算 beta[i] 的递推公式
78. **double** beta[21]={0};
79. beta[1]=c[1]/b[1];
80. **for**(**int** i=2; i<=18; ++i)
81. beta[i]=c[i]/(b[i]-a[i]\*beta[i-1]);
83. //解 Ly=f
84. **double** yi[21]={0};
85. yi[1]=d[1]/b[1];
86. **for**(**int** i=2; i<=19; ++i)
87. yi[i]=(d[i]-a[i]\*yi[i-1])/(b[i]-a[i]\*beta[i-1]);
89. //解 Ux=y
90. **double** m[21]={0};   //这里的 m 即 追赶法中要求解的 x
91. m[19]=yi[19];
92. **for**(**int** i=18; i>=1; --i)
93. m[i]=yi[i]-beta[i]\*m[i+1];
95. //带入端点条件计算端点值
96. m[0]=(6\*((y[1]-y[0])/h-f\_(-1))/h-m[1])/2;
97. m[20]=(6\*(f\_(-1)-(y[20]-y[19])/h)/h-m[19])/2;
99. **double** x\_[21]={0};
100. **for**(**int** i=1; i<=20; ++i)
101. x\_[i]=-1.05+0.1\*i;
103. **double** s20[21]={0};
104. **for**(**int** i=1; i<=20; ++i)
105. s20[i]=s(i,x,m,y,h,x\_[i]);
107. **double** x2[11];
108. **for**(**int** i=0; i<=10; ++i)
109. x2[i]=1-0.2\*i;
111. //Lagrange插值法 第五章1-16
112. **double** l10[21]={0};
113. **for**(**int** n=1; n<=20; ++n)
114. l10[n]=l(10,x2,x\_[n]);
116. //输出
117. cout<<"x\t\t\tf(x)\t\t\tS20(x)\t\t\tL10(x)\n";
118. **for**(**int** i=1; i<=20; ++i)
119. cout<<fixed<<setprecision(7)<<x\_[i]<<"\t\t"<<f(x\_[i])<<"\t\t"<<s20[i]<<"\t\t"<<l10[i]<<"\n";
121. **return** 0;
122. }

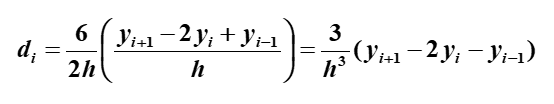
## 四、运行输出结果：



## 五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训：

发现问题：结果始终有很大偏差

采取措施：反复看PPT原理部分，最终发现PPT上有一处错误（5-3 的14页）

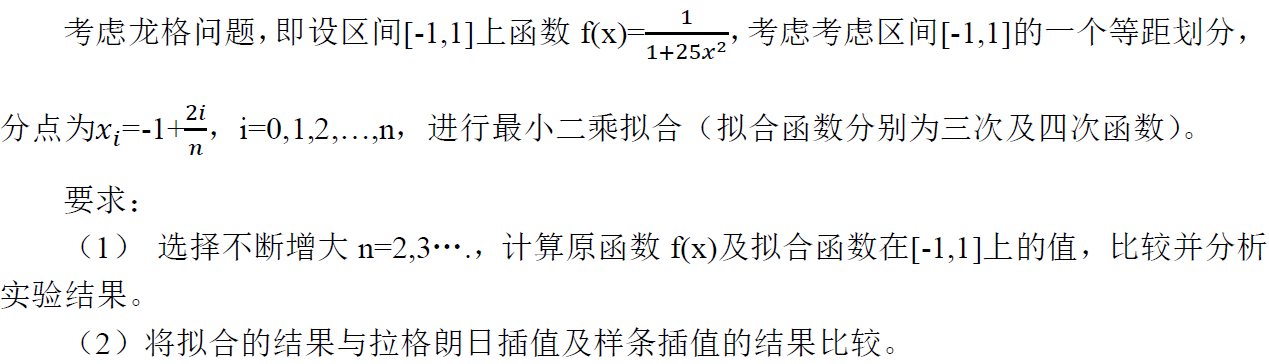


我最开始没有在意直接用了后面的式子，后来发现错了。

吸取教训：对于PPT上的每一步，需要自己再验算一遍。

# 实验五、最小二乘拟合

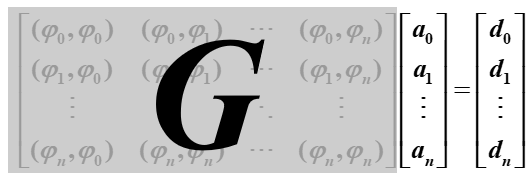
## 一、实验内容：



## 二、算法基本思想及复杂度分析：

找一个S(x)，使S(x)-f(x)总体上尽可能小，最小二乘数法就是使最小，也就是让每个点到p(x)的距离的平方和尽可能最小。我们通过假设一个拟合函数S(x)（本题中为三次及四次函数），来让最小，也就是让它的导数等于0。

具体做法则是求解如下多项式



利用共轭梯度法求得a[0]~a[n]后

带入得到插值函数

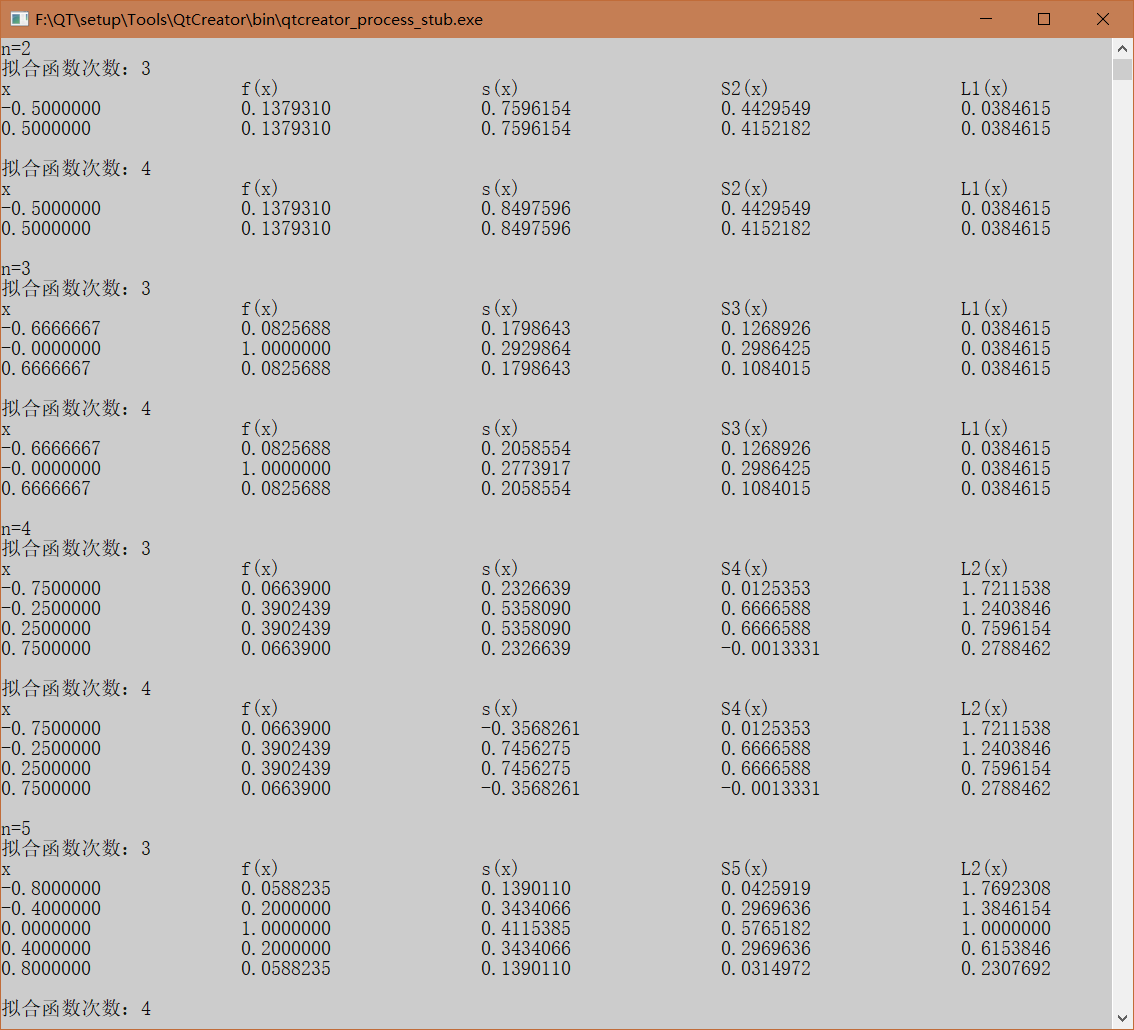
## 三、源程序及注释：

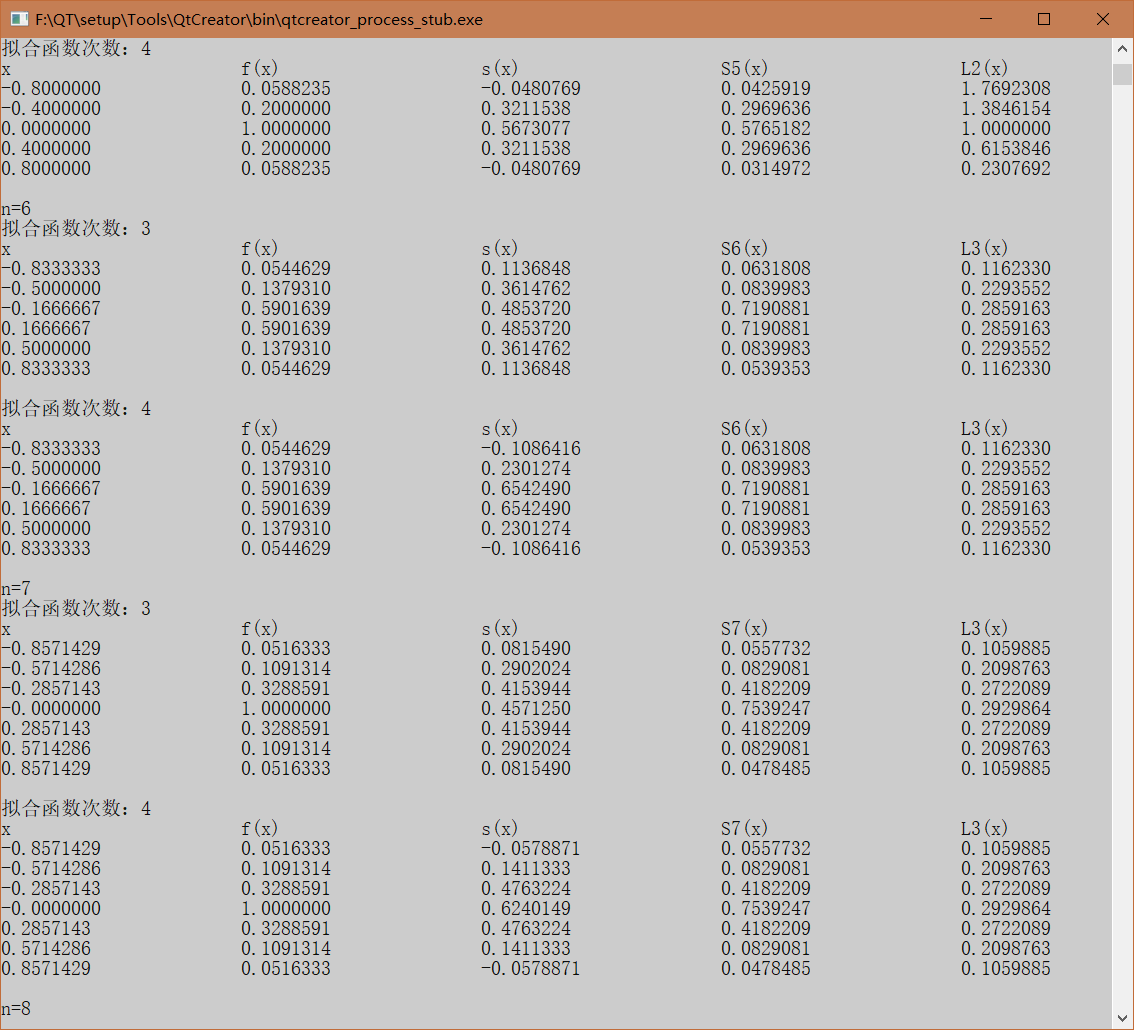
1. //实验五 最小二乘数法
3. #include <iostream>
4. #include <cmath>
5. #include <iomanip>
6. **using** **namespace** std;
8. //f(x)
9. **double** f(**double** x){
10. **return** 1.0/(1+25\*x\*x);
11. }
13. //选取基函数
14. **double** phi(**int** i, **double** x){
15. **return** pow(x,i);
16. }
18. //计算 S(x)
19. //x: 要求的节点  m:拟合函数次数
20. **double** s(**int** m, **double** a[], **double** x){
21. **double** sum=0;
22. **for**(**int** i=0; i<=m; ++i)
23. sum+=a[i]\*phi(i,x);
24. **return** sum;
25. }

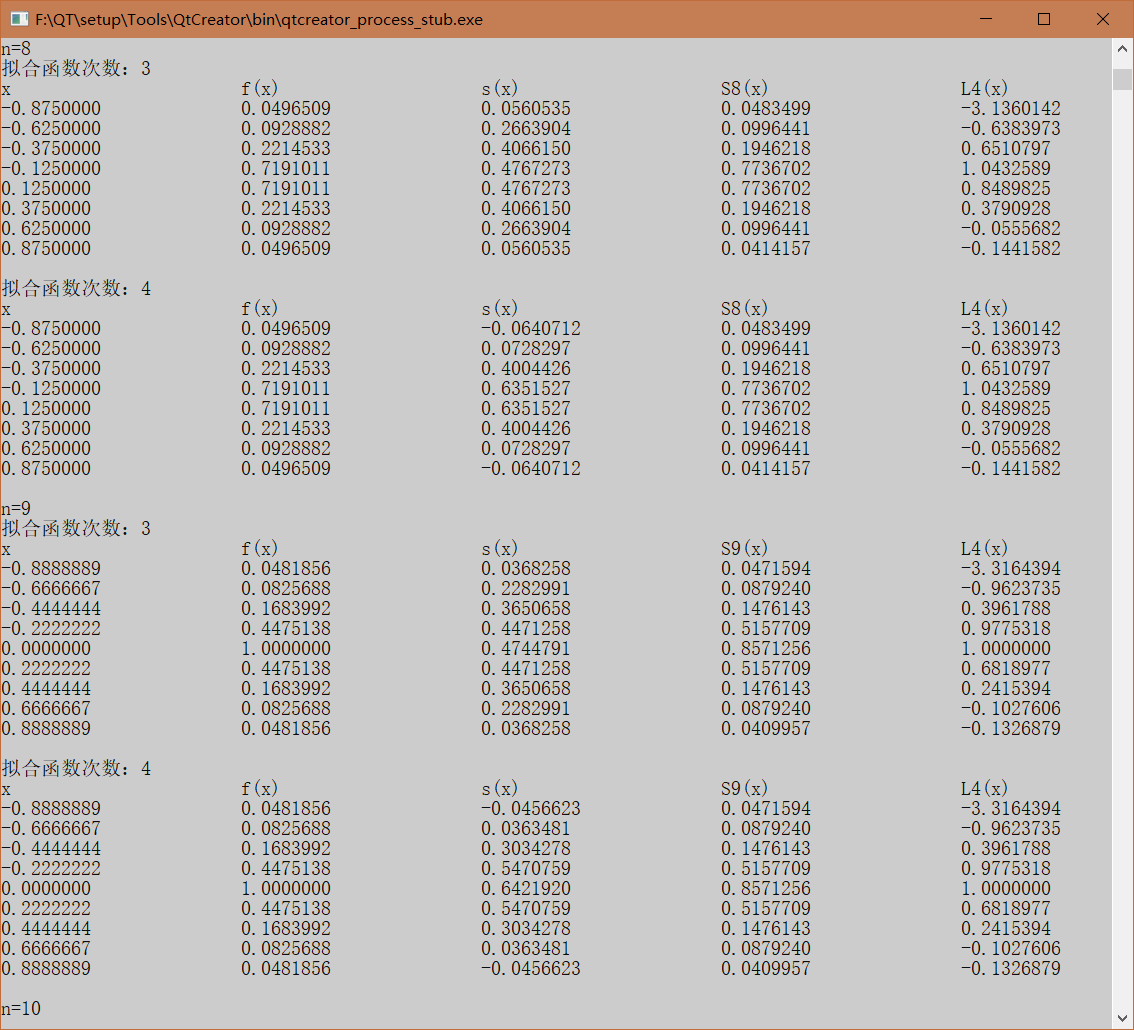
28. //对比用，代码在实验四基础上修改
29. ////////////////////////////////////////////////////////////////////////
30. //f'(x)
31. **double** f\_(**double** x){
32. **return** -50\*x/((1+25\*x\*x)\*(1+25\*x\*x));
33. }
35. //三次样条插值函数 S(x)
36. //i表示 si(x)
37. //x\_表示 -0.95~0.95的插值节点
38. **double** s2(**int** i, **double** x[], **double** m[], **double** y[], **double** h, **double** x\_){
39. **return** m[i-1]\*(x[i]-x\_)\*(x[i]-x\_)\*(x[i]-x\_)/(6\*h)
40. +m[i]\*(x\_-x[i-1])\*(x\_-x[i-1])\*(x\_-x[i-1])/(6\*h)
41. +(y[i-1]-m[i-1]/6\*h\*h)\*(x[i]-x\_)/h
42. +(y[i]-m[i]/6\*h\*h)\*(x\_-x[i-1])/h;
43. }
45. //Lagrange插值法 l(x)
46. //n表示 ln(x)
47. //x\_表示插值节点
48. **double** l(**int** n, **double** x[], **double** x\_){
49. **double** sum=0;
50. **for**(**int** k=0; k<n; ++k){
51. **double** product=1;
52. **for**(**int** i=0; i<n; ++i){
53. **if**(i!=k)
54. product\*=(x\_-x[i])/(x[k]-x[i]);
55. }
56. sum+=f(x[k])\*product;
57. }
58. **return** sum;
59. }
60. //////////////////////////////////////////////////////////////////////////
62. //计算法方程 共轭梯度法
63. /////////////////////////////////////////////////////////////////////////
64. //判断精度 ||x[k]-x[k-1]||2 < 1e-6
65. **bool** judge(**int** m, **double** x[], **double** y[]) {
66. **double** dx[100], tmp, sum=0;
67. **for**(**int** i=0; i<=m; ++i){
68. dx[i]=x[i]-y[i];
69. sum+=dx[i]\*dx[i];
70. }
71. tmp=sqrt(sum);
73. **if**(tmp>=1e-6)
74. **return** 1;
75. **else**
76. **return** 0;
77. }
78. ////////////////////////////////////////////////////////////////////////////
80. **int** main(){
81. //节点个数
82. **for**(**int** n=2; n<=20; ++n){
83. cout<<"n="<<n<<endl;
85. **double** x[100]={0};  //x 等距节点
86. **double** x\_[100]={0}; //x\_ 要求的节点
87. **double** y[100]={0};  //y 节点值
88. **for**(**int** i=0; i<=n; ++i){
89. x[i]=-1+2.0\*i/n;
90. y[i]=f(x[i]);
91. }
93. **for**(**int** i=0; i<n; ++i)
94. x\_[i]=(x[i]+x[i+1])/2;
96. //法方程
97. **double** a[100][100];
98. **double** b[100]={0};
99. **double** af[100]={0};
101. **int** m[2]={3,4}; //拟合函数次数
102. **for**(**int** mi=0; mi<2; ++mi){
103. cout<<"拟合函数次数："<<m[mi]<<endl;
105. //a, b, af 清0
106. **for**(**int** i=0; i<=m[mi]; ++i){
107. **for**(**int** j=0; j<=m[mi]; ++j)
108. a[i][j]=0;
109. b[i]=0;
110. af[i]=0;
111. }
113. **for**(**int** i=0; i<=m[mi]; ++i){
114. **for**(**int** j=0; j<=m[mi]; ++j){
115. **for**(**int** k=0; k<=n; ++k)
116. a[i][j]+=phi(i,x[k])\*phi(j,x[k]);
117. }
118. **for**(**int** k=0; k<=n; ++k)
119. b[i]+=y[k]\*phi(i,x[k]);
120. }
122. //计算法方程 共轭梯度法
123. /////////////////////////////////////////////////////////////////////////
124. **double** xf[100]={0};  //xk 并赋初值
125. **double** yf[100]={0};  //xk-1 保存前一个xk 赋合适的初值使得精度不满足判断条件
126. **for**(**int** i=0; i<=m[mi]; ++i)
127. yf[i]=1;
129. **double** r[100]={0};
130. **double** p[100]={0};
131. **double** alpha=0;
132. **double** r2[100]={0};
133. **double** betaf=0;
135. //r=b-Ax  p=r
136. **for**(**int** i=0; i<=m[mi]; ++i){
137. r[i]=b[i]-0;
138. p[i]=r[i];
139. }
141. **bool** jd2=0;  //用于判断收敛性
143. **while**(judge(m[mi],xf,yf)){
144. //yf存储上一个x的值
145. **for**(**int** i=0; i<=m[mi]; ++i)
146. yf[i]=xf[i];
148. //计算alpha （计算搜索步长）  alpha=(rT\*r)/(pT\*A\*p)
149. **double** sum1=0;
150. **double** sum2=0;
151. **double** sum3=0;
152. **for**(**int** i=0; i<=m[mi]; ++i){
153. sum1+=r[i]\*r[i];
154. sum2=0;
155. **for**(**int** j=0; j<=m[mi]; ++j)
156. sum2+=p[j]\*a[j][i];
157. sum3+=sum2\*p[i];
158. }
159. alpha=1.0\*sum1/sum3;
161. //计算x (更新解) x=x+alpha\*p
162. **for**(**int** i=0; i<=m[mi]; ++i)
163. xf[i]=xf[i]+alpha\*p[i];
165. //计算r2  r2=r
166. **for**(**int** i=0; i<=m[mi]; ++i)
167. r2[i]=r[i];
169. //计算r（更新残差向量）   r=r-alpha\*A\*p
170. **for**(**int** i=0; i<=m[mi]; ++i){
171. **double** sum=0;
172. **for**(**int** j=0; j<=m[mi]; ++j)
173. sum+=a[i][j]\*p[j];
174. r[i]=r[i]-alpha\*sum;
175. }
177. //计算beta    beta=(rT\*r)/(r2T\*r2)
178. **double** sum4=0;
179. **double** sum5=0;
180. **for**(**int** i=0; i<=m[mi]; ++i){
181. sum4+=r[i]\*r[i];
182. sum5+=r2[i]\*r2[i];
183. }
184. betaf=1.0\*sum4/sum5;
186. //计算p（新的搜索方向）   p=r+beta\*p
187. **for**(**int** i=0; i<=m[mi]; ++i)
188. p[i]=r[i]+betaf\*p[i];
190. //判断收敛性
191. **bool** jd=1;
192. **for**(**int** i=0; i<=m[mi]; ++i)
193. **if**(fabs(xf[i])<1000)
194. jd=0;
195. **if**(jd){
196. jd2=1;
197. **break**;
198. }
199. }
200. //输出
201. **if**(jd2){
202. cout<<"法方程不收敛"<<endl;
203. }
204. **else**{
205. **for**(**int** i=0; i<=m[mi]; ++i){
206. af[i]=xf[i];
207. }
208. }
209. /////////////////////////////////////////////////////////////////////////

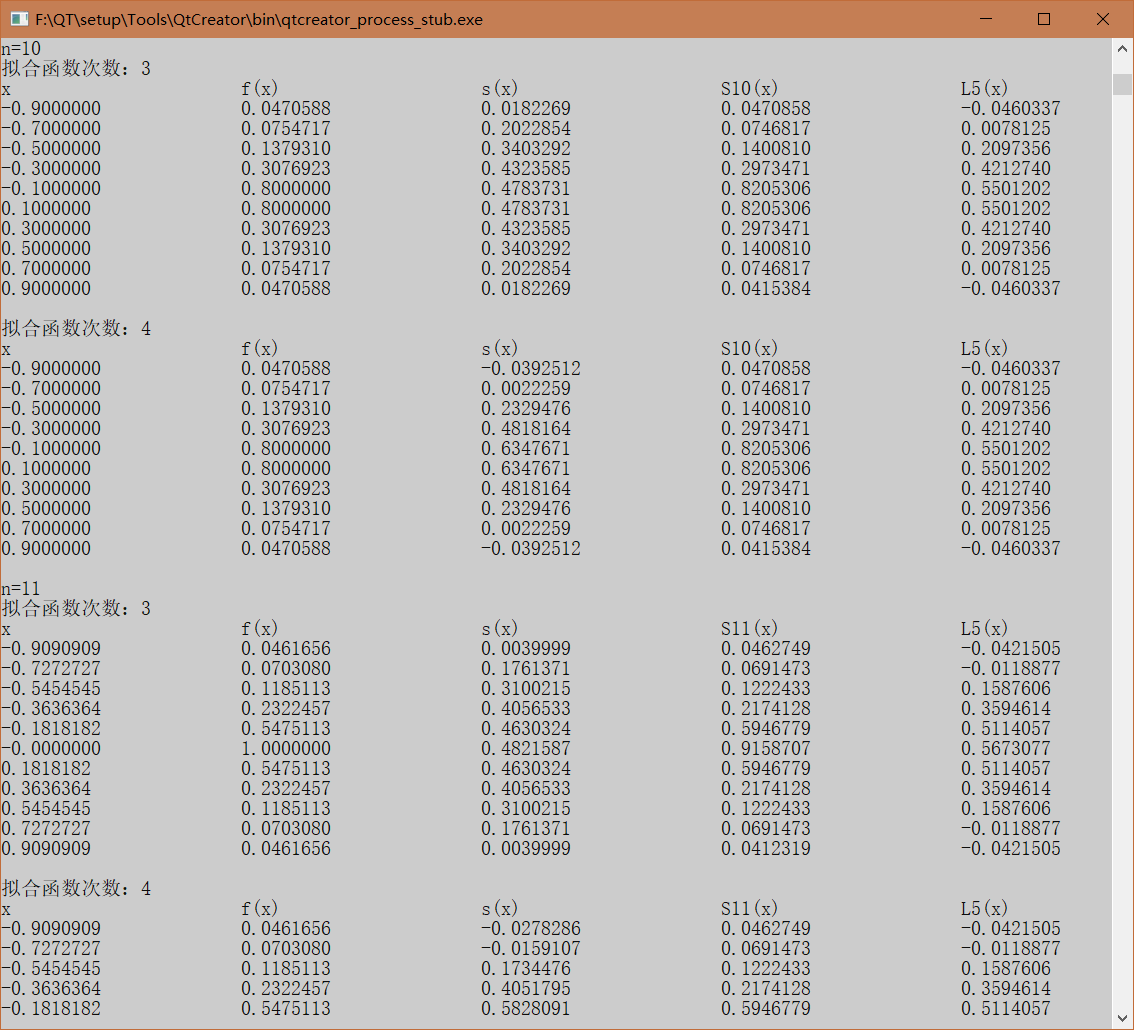
212. //(与拉格朗日插值法与三次样条插值法)对比用，代码在实验四基础上修改
213. ///////////////////////////////////////////////////////////////////////
214. //三次样条插值
215. **double** h=2.0/n;   //h=2/N
216. **double** lambda=0.5;  //由于等距节点 lambda=h/(h+h)=1/2
217. **double** mu=0.5;      //mu=1-lambda=1/2
218. **int** ci=2;
220. **double** d[100]={0};
221. //由于等距节点 求 d[1]-d[19] 的值
222. **for**(**int** i=1; i<n; ++i)
223. d[i]=3.0/(h\*h)\*(y[i+1]-2\*y[i]+y[i-1]);
225. //解 n-1阶 三对角方程组
226. //采用 追赶法    第三章1-64   a=mu b=c c=lambda x=m f=d
227. **double** a2[100]={0};
228. **double** b2[100]={0};
229. **double** c[100]={0};
231. **for**(**int** i=1; i<=n-1; ++i){
232. a2[i]=mu;
233. b2[i]=ci;
234. c[i]=lambda;
235. }
236. a2[1]=0;
237. c[n-1]=0;
239. //计算 beta[i] 的递推公式
240. **double** beta[100]={0};
241. beta[1]=c[1]/b2[1];
242. **for**(**int** i=2; i<=n-2; ++i)
243. beta[i]=c[i]/(b2[i]-a2[i]\*beta[i-1]);
245. //解 Ly=f
246. **double** yi[100]={0};
247. yi[1]=d[1]/b2[1];
248. **for**(**int** i=2; i<=n-1; ++i)
249. yi[i]=(d[i]-a2[i]\*yi[i-1])/(b2[i]-a2[i]\*beta[i-1]);
251. //解 Ux=y
252. **double** m2[100]={0};   //这里的 m 即 追赶法中要求解的 x
253. m2[n-1]=yi[n-1];
254. **for**(**int** i=n-2; i>=1; --i)
255. m2[i]=yi[i]-beta[i]\*m2[i+1];
257. //带入端点条件计算端点值
258. m2[0]=(6\*((y[1]-y[0])/h-f\_(-1))/h-m2[1])/2;
259. m2[n]=(6\*(f\_(-1)-(y[n]-y[n-1])/h)/h-m2[n-1])/2;
261. **double** sn[100]={0};
262. **for**(**int** i=1; i<=n; ++i)
263. sn[i-1]=s2(i,x,m2,y,h,x\_[i-1]);
265. **double** x2[100];
266. **for**(**int** i=0; i<=n/2; ++i)
267. x2[i]=1-2.0/(n/2)\*i;
269. //Lagrange插值法 第五章1-16
270. **double** ln\_[100]={0};
271. **int** n\_=n/2; //计算 Ln\_
272. **for**(**int** i=1; i<=n; ++i)
273. ln\_[i-1]=l(n\_,x2,x\_[i-1]);
274. ///////////////////////////////////////////////////////////////////////
276. //输出
277. cout<<"x\t\t\tf(x)\t\t\ts(x)\t\t\tS"<<n<<"(x)\t\t\tL"<<n/2<<"(x)\n";
278. **for**(**int** i=0; i<n; ++i)
279. cout<<fixed<<setprecision(7)<<x\_[i]<<"\t\t"<<f(x\_[i])<<"\t\t"
280. <<s(m[mi],af,x\_[i])<<"\t\t"<<sn[i]<<"\t\t"<<ln\_[i]<<endl;
281. cout<<endl;
282. }
284. }
285. **return** 0;
286. }

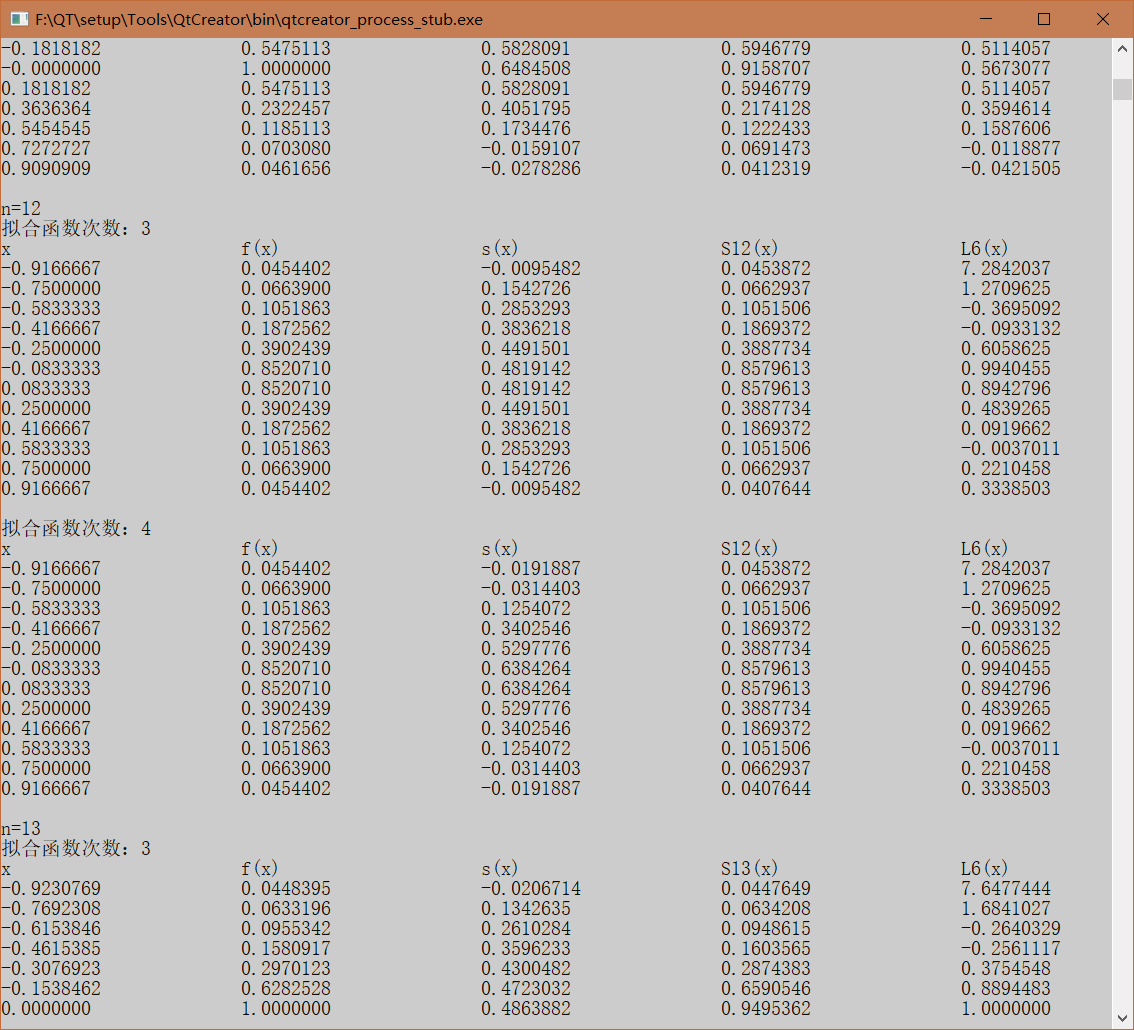
## 四、运行输出结果：

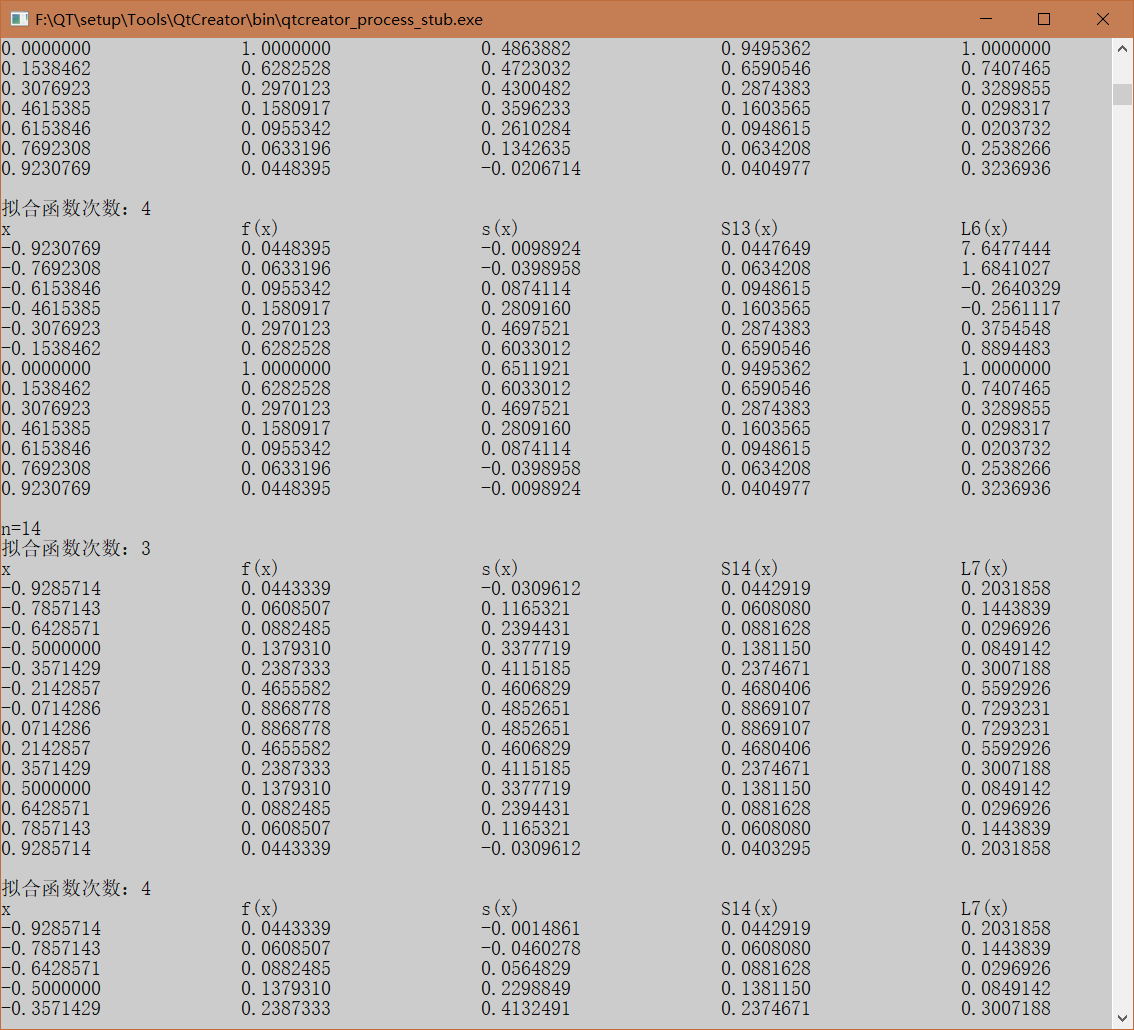


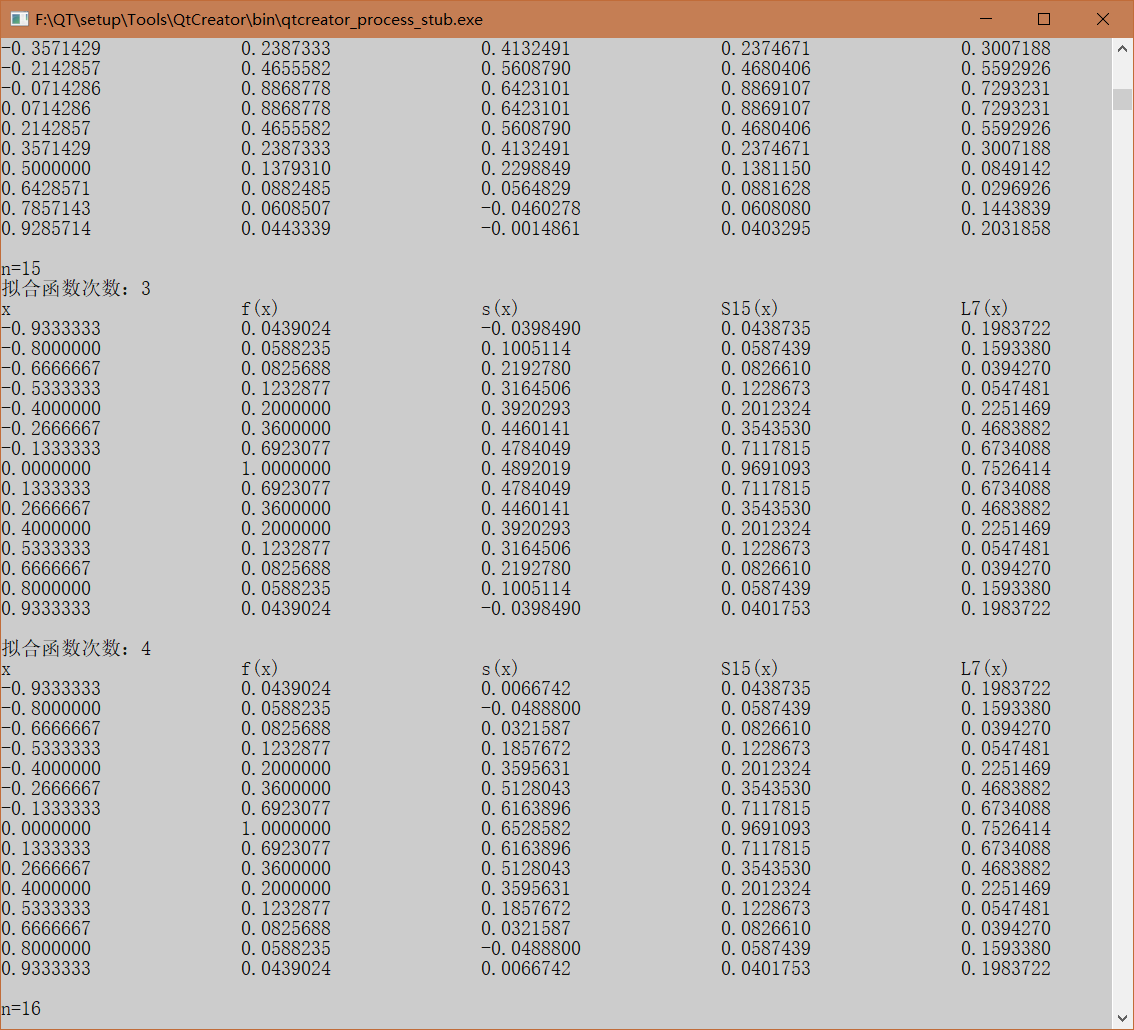


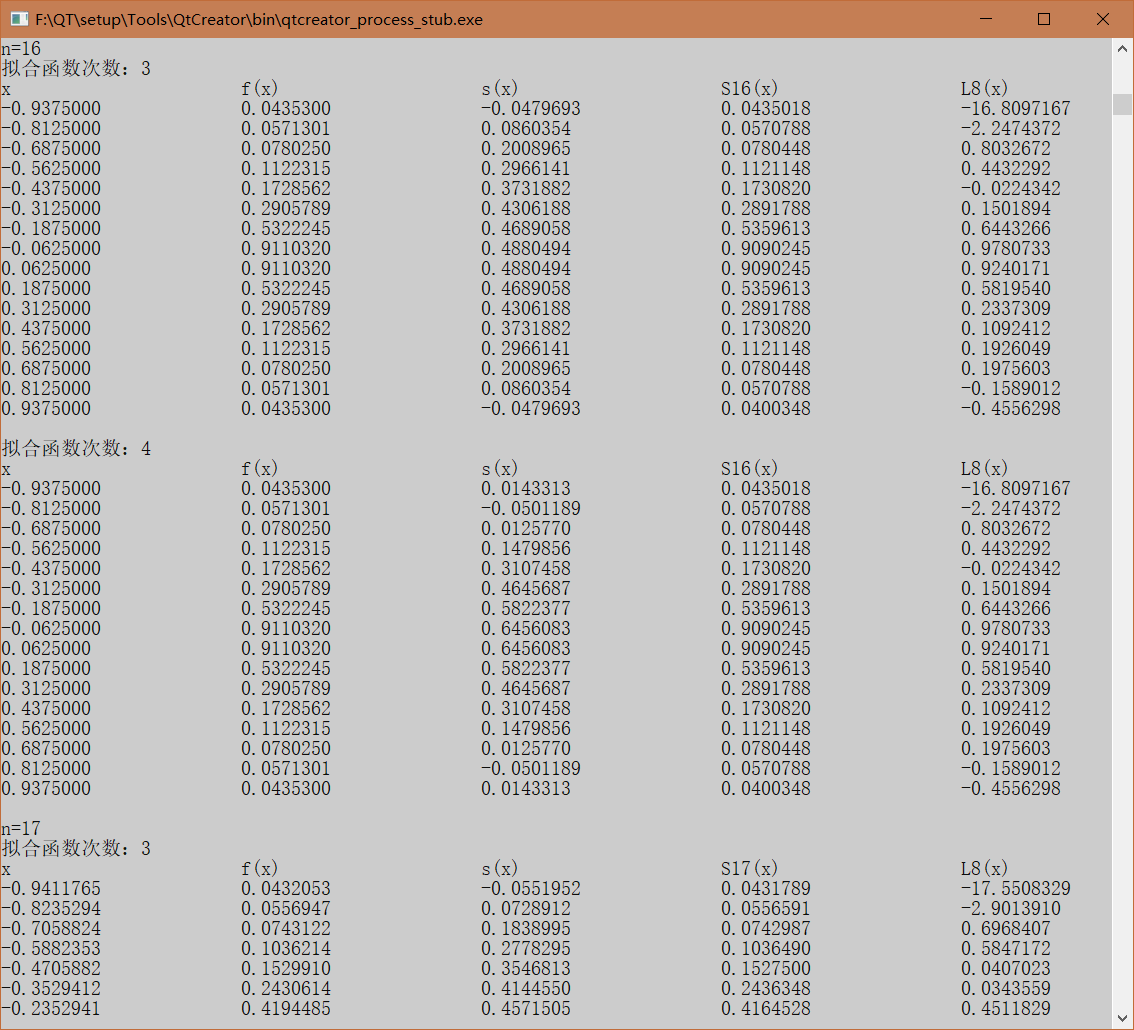


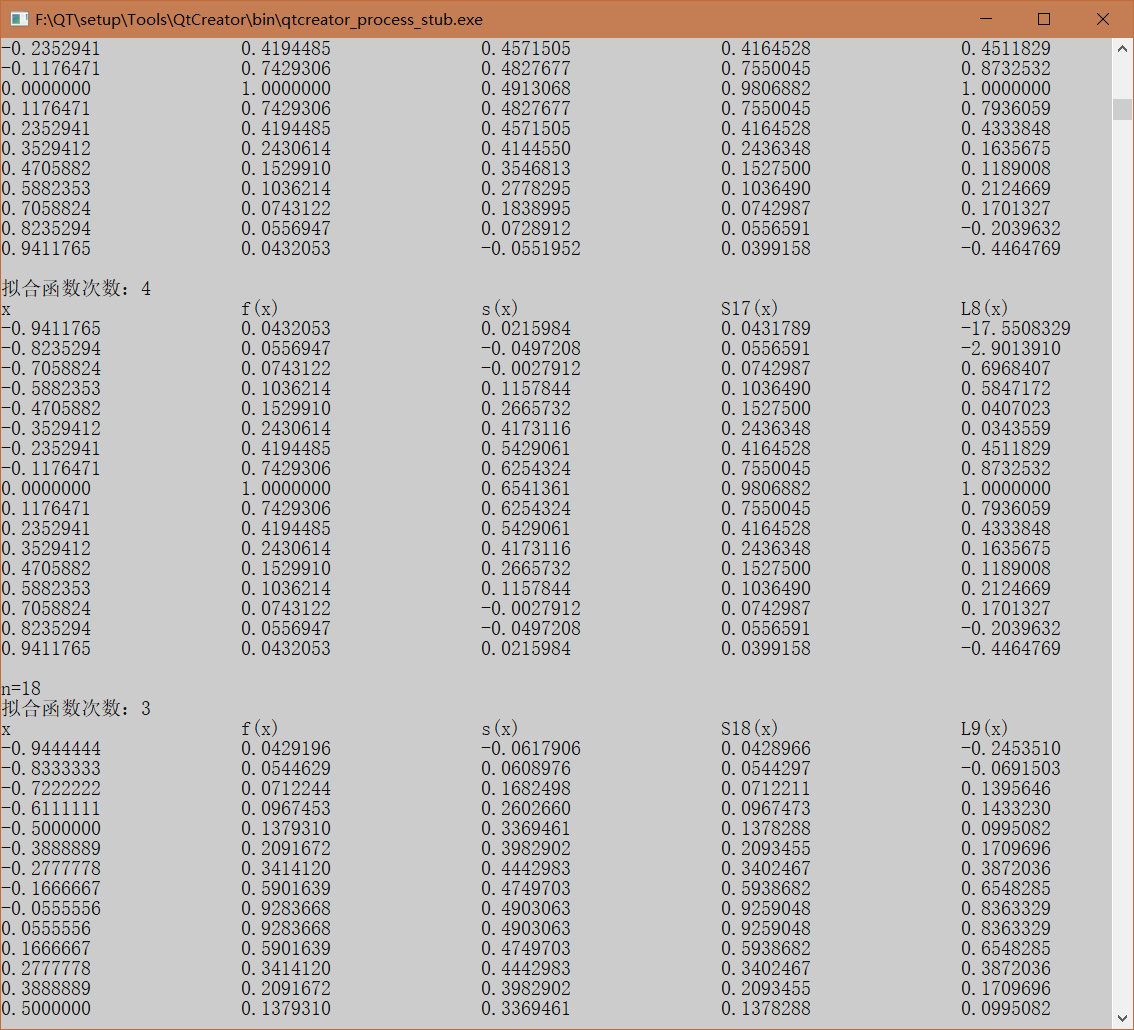


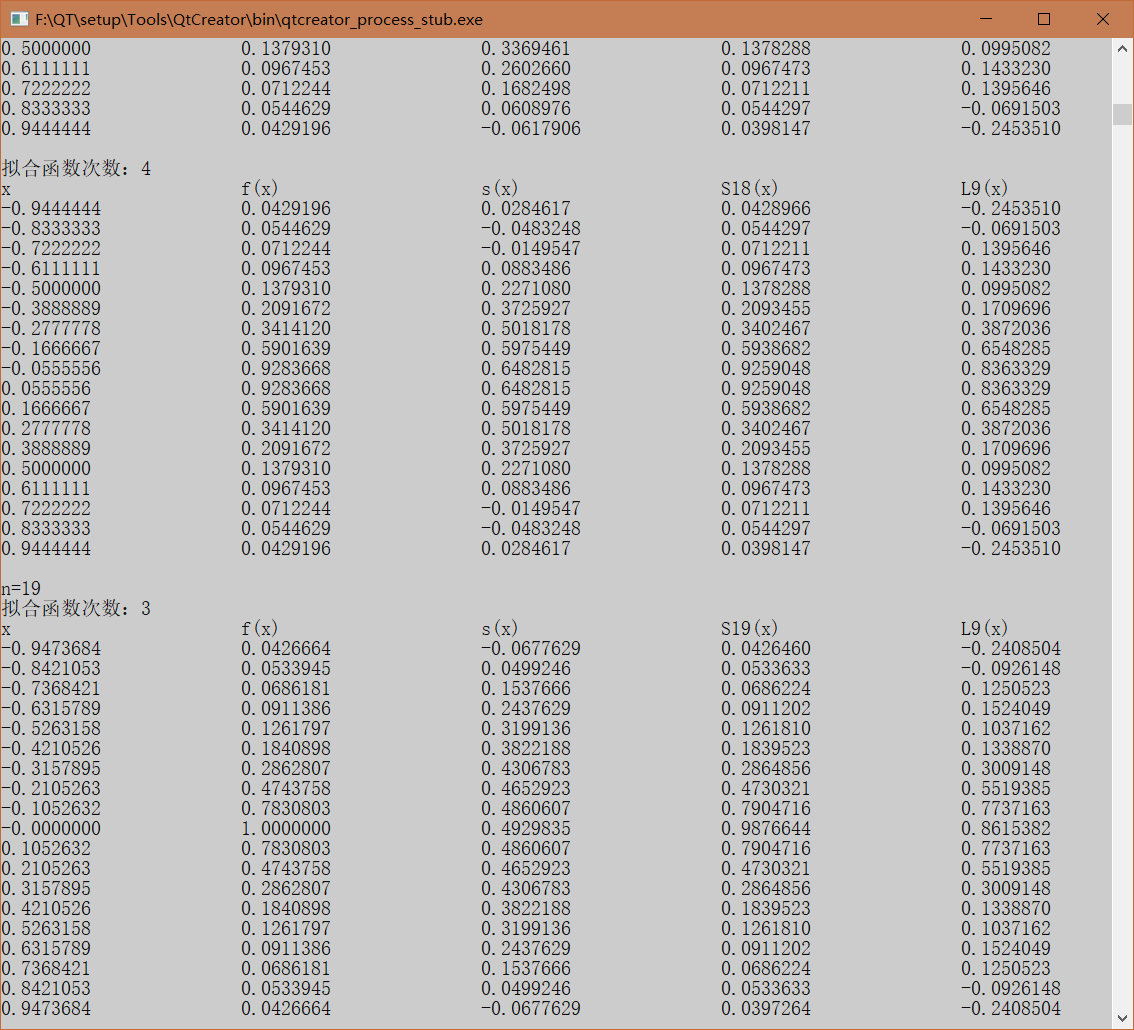


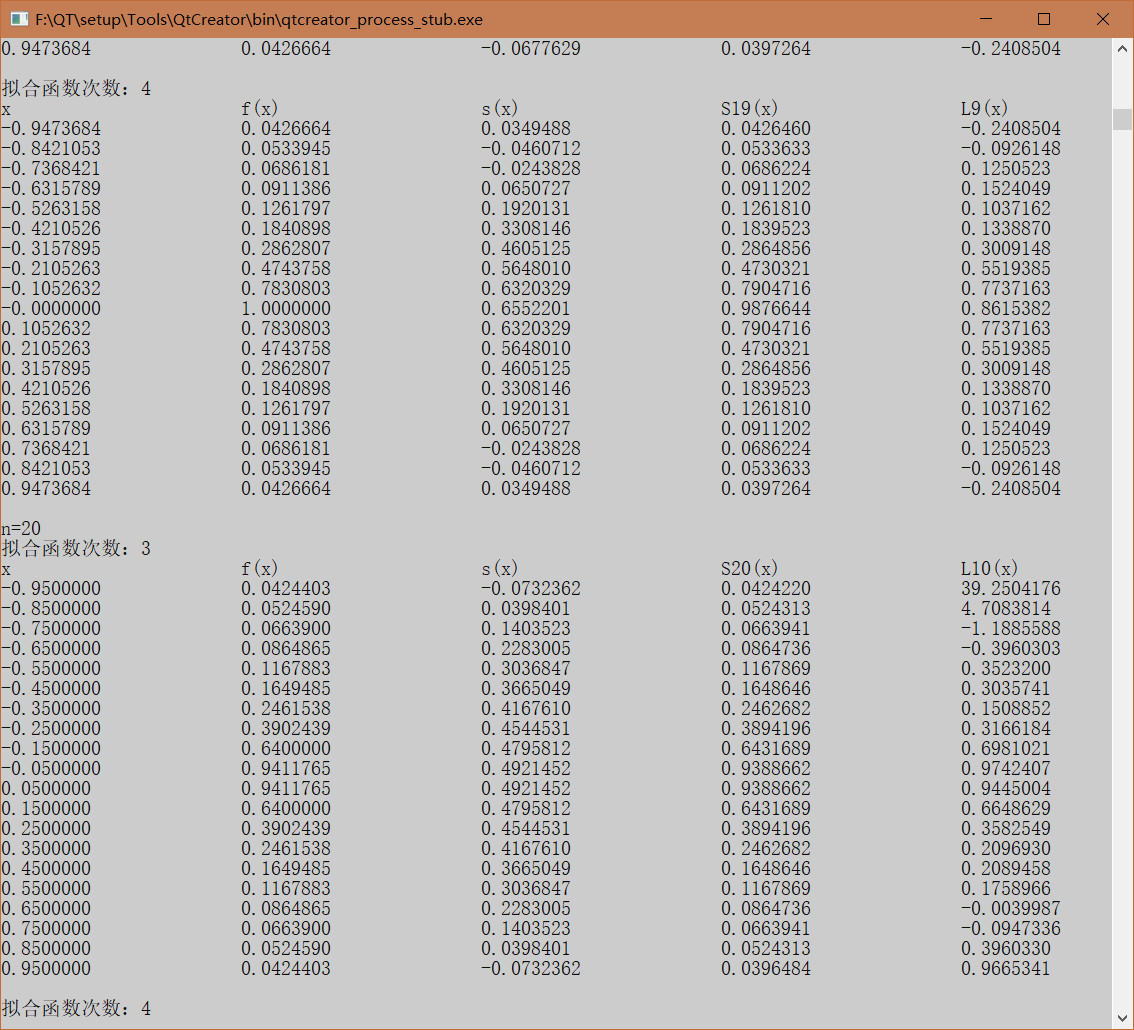


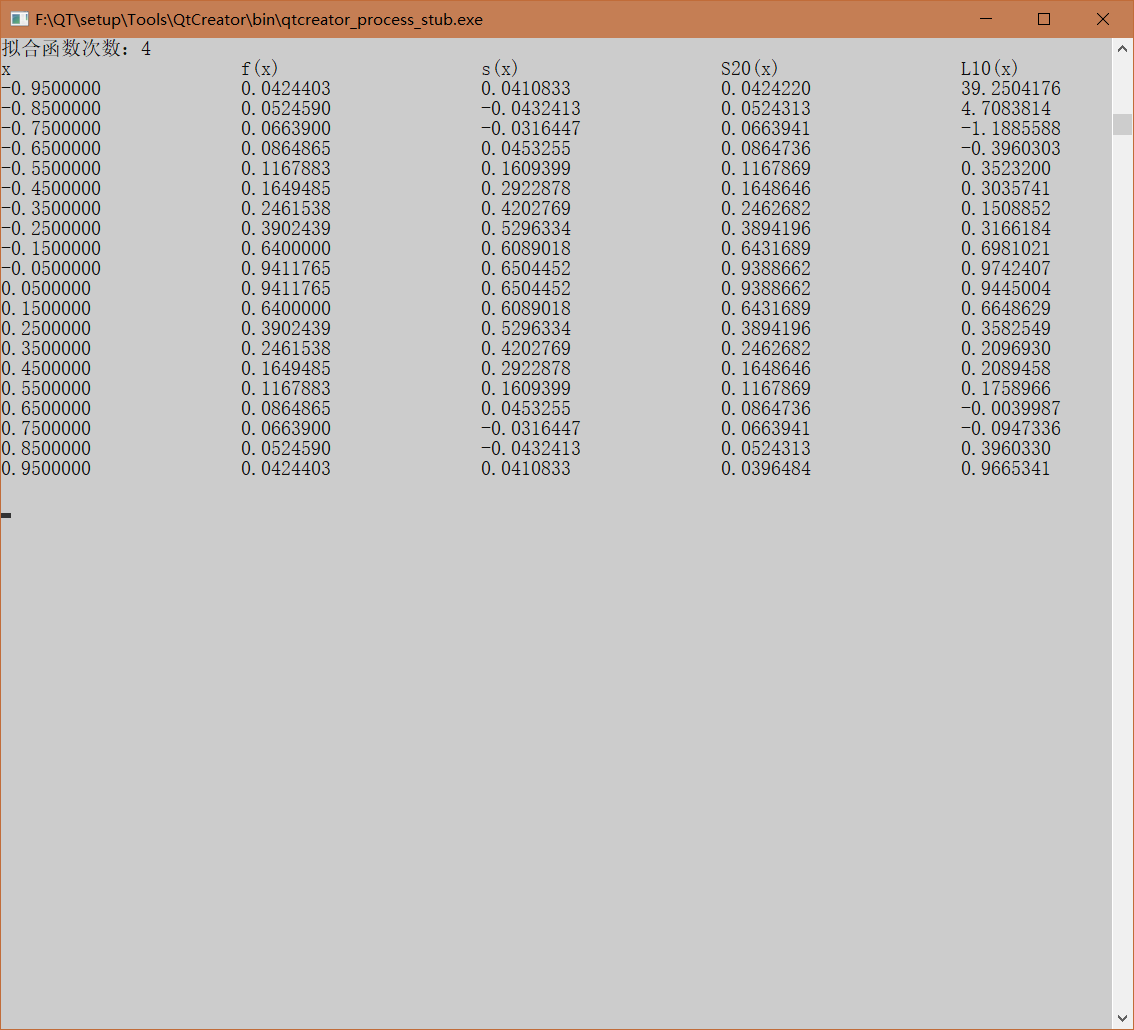












由结果可见，当n不断增大的时候，最小二乘拟合精。有了提升，而三次样条插值方法由于是对每一段上求拟合函数，所以对精度的影响不是很大。而拉格朗日插值法出现了明显的龙格现象。

## 五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训：

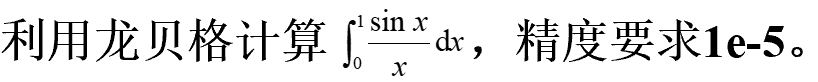
产生问题：法方程用共轭梯度法不收敛

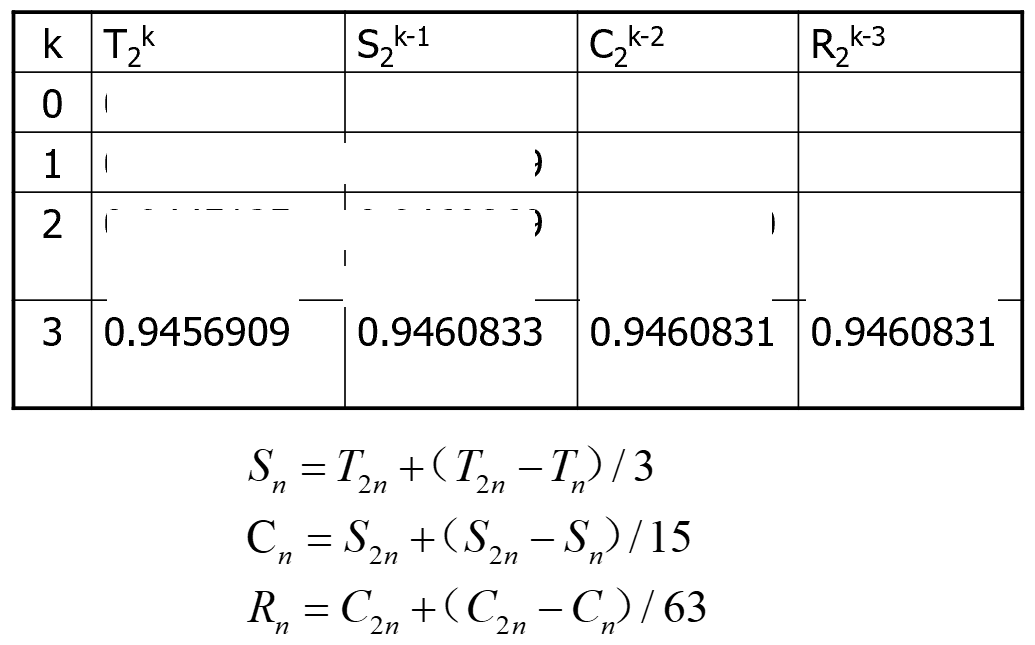
采取措施：后来发现是因为在最小二乘数拟合中定义了数组x，而后面的用于对比的三次样条插样中重复定义了数组x，覆盖掉了最小二乘数中的x。所以将最小二乘数中的数组x更改了名字

吸取教训：在引用其他代码（本题是引用实验四和三的代码）时，要格外消息有没有覆盖的情况。

# 实验六、数值积分

## 一、实验内容：





## 二、算法基本思想及复杂度分析：

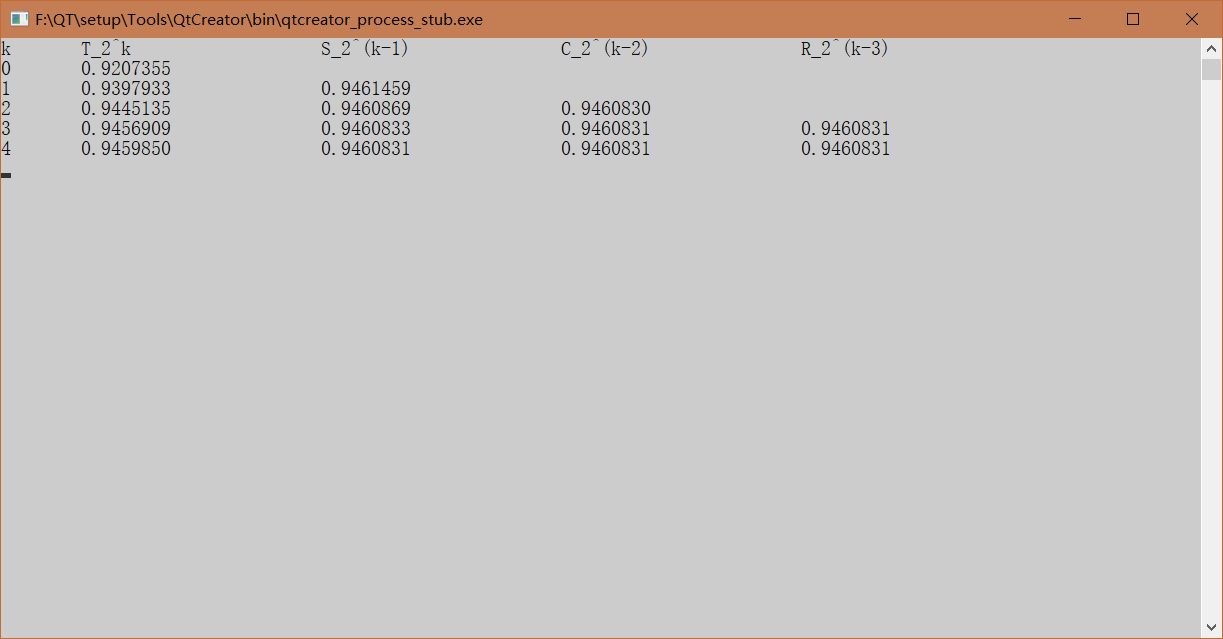
在梯形公式、辛普森公式和科特斯公式之间的关系的基础上，构造出的一种加速计算积分的方法。实际就是将区间不断对半分来进行快速地计算。可定义数组保存Tn, Sn, Cn, Rn（Sn, Cn, Rn计算公式如上），预处理到T16，之后计算|Rn-Rn-1|<误差限，不满足则重复计算，满足则输出结果。

其中Tn计算公式如下：

## 三、源程序及注释：

1. //实验六 龙贝格
3. #include <iostream>
4. #include <cmath>
5. #include <iomanip>
6. **using** **namespace** std;
8. //f(x)
9. **double** f(**double** x){
10. **if**(x==0.0)
11. **return** 1;
12. **else**
13. **return** sin(x)/x;
14. }
16. //Tn
17. //Tn=(b-a)/2n\*(f(a)+2\*sum[j=1~n-1]f(xj)+f(b))
18. **double** Tn(**int** i){
19. //a=0, b=1, 则 b-a=1
20. **double** h=1.0/pow(2,i);
21. **double** sum=0;
22. **for**(**int** j=1; j<pow(2,i); ++j)
23. sum+=2\*f(h\*j);
24. sum+=f(0)+f(1);
25. **return** h/2.0\*sum;
26. }
28. //Sn
29. **double** Sn(**int** n, **double** T[]){
30. **return** T[n+1]+(T[n+1]-T[n])/3.0;
31. }
33. //Cn
34. **double** Cn(**int** n, **double** S[]){
35. **return** S[n+1]+(S[n+1]-S[n])/15.0;
36. }
38. //Rn
39. **double** Rn(**int** n, **double** C[]){
40. **return** C[n+1]+(C[n+1]-C[n])/63.0;
41. }
43. **int** main(){
44. **double** T[100]={0};
45. **double** S[100]={0};
46. **double** C[100]={0};
47. **double** R[100]={0};
49. //Tn
50. //T[0]=T\_1 T[1]=T\_2 T[2]=T\_4 T[3]=T\_8 T[4]=T\_16
51. //预处理到 T\_16
52. **for**(**int** i=0; i<5; ++i){   //
53. T[i]=Tn(i);
54. }
56. //Sn
57. //S[0]=S\_1 S[1]=S\_2 S[2]=S\_4 S[3]=S\_8
58. //预处理到 S\_8
59. **for**(**int** n=0; n<4; ++n)
60. S[n]=Sn(n,T);
62. //Cn
63. //C[0]=C\_1 C[1]=C\_2 C[2]=C\_4
64. //预处理到 C\_4
65. **for**(**int** n=0; n<3; ++n)
66. C[n]=Cn(n,S);
68. //Rn
69. //R[0]=R\_1 R[1]=R\_2
70. //预处理到 R\_2
71. **for**(**int** n=0; n<2; ++n)
72. R[n]=Rn(n,C);
74. **int** n=1;    //记录R[n]的长度 可推 C[n]长度为n+1 S[n]长度为n+2 T[n]长度为n+3
75. **while**(fabs(R[n]-R[n-1])>=1e-5){
76. n++;
77. T[n+3]=Tn(n+3);
78. S[n+2]=Sn(n+2,T);
79. C[n+1]=Cn(n+1,S);
80. R[n]=Rn(n,C);
81. }
83. //输出
84. **int** k[100]={0};
85. **for**(**int** i=0; i<100; ++i)
86. k[i]=i;
87. cout<<"k\tT\_2^k\t\t\tS\_2^(k-1)\t\tC\_2^(k-2)\t\tR\_2^(k-3)\n";
88. cout<<fixed<<setprecision(7);
89. cout<<k[0]<<"\t"<<T[0]<<endl;
90. cout<<k[1]<<"\t"<<T[1]<<"\t\t"<<S[0]<<endl;
91. cout<<k[2]<<"\t"<<T[2]<<"\t\t"<<S[1]<<"\t\t"<<C[0]<<endl;
92. cout<<k[3]<<"\t"<<T[3]<<"\t\t"<<S[2]<<"\t\t"<<C[1]<<"\t\t"<<R[0]<<endl;
93. **for**(**int** i=4; i<=n+3; ++i)
94. cout<<k[i]<<"\t"<<T[i]<<"\t\t"<<S[i-1]<<"\t\t"<<C[i-2]<<"\t\t"<<R[i-3]<<endl;
96. **return** 0;
97. }

## 四、运行输出结果：



## 五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训：

出现问题：如果用课件上的迭代计算，循环次数的确定有些麻烦。

采取措施：通过百度，发现了Tn的一般算法，放弃了Tn迭代计算的方法，这样既不用计算T1，也省略了相对复杂的迭代方法。

吸取教训：对于相对循环次数少的方法，可以尝试用其他方法进行计算（说是其他方法，其实还是龙贝格算法，只不过换成了其他形式）。

# 心得体会

通过这次实验，我加深了对这些算法的理解，同时也提升了自己码代码的能力。

那些在课上可能不是很理解的东西，通过作业和这次实验，为了解决给定的问题，我会反反复复地去解读这些东西，最终达到一个较好的理解水平。最印象深刻的就是每次对着PPT上的伪代码打完，几乎每个实验运行完的结果都会有各种各样的问题，每当这种时候我甚至感觉自己在写的不是算法，而是bug。然后老老实实地从PPT关于该算法的第一页开始，重新慢慢理解。比如说实验三，这个耗了我最长时间的一个实验，一开始总是输出好多好多数字，然后最后变成Inf，看了PPT，我知道原来还有不收敛的情况，于是加入了收敛判断。最开始的收敛判断是如果循环次数大于100，就输出不收敛。但是随着第二个式子的进行，我发现这样貌似不妥，有的时候循环次数非常大，但它还是收敛的。于是，我又更改了收敛判断。如果发散的话，那么|x|就会趋向无穷，那么只要最后计算出来的数大于一个足够大的数（大到不可能是结果），那么不就算是收敛了吗？于是新的收敛判断方式诞生了。看着输出的结果，会有强烈的满足感。

而有一种情况就会比较难受，就是反复看PPT、看原理、看代码，感觉都没问题，但是答案却有很大的问题。比如实验五，由于实验五需要用到实验三和实验四的代码，于是我进行了移植。但是变量重名的情况却比较麻烦，我也确实因为一个变量重名修改后，下面运用这个变量的一个地方没改而耗费了很长时间。经过漫长时间地debug之后，发现了原来是这样子的错误总是让我有一种很微妙的感觉。这也让我明白了一些写代码的规则，首先是变量的命名方式，用x,y,z,a,b,c这种确实很方便，但是如果那一天要引用的话就会很麻烦；其次，main函数里面最好不要放算法，而是写在类或者一个大的函数里面，然后在main函数里面调用它（虽然我这次不是这么干的，但不这么做的后果真的让我很痛苦，之后写代码我会把能不放在main函数里面的东西拿出来放在类或函数里面）。

我很庆幸自己有写注释的习惯，让我找错误的时候方便了很多。有的时候我会选择先往下做，然后再回过去继续debug，这时候确实会有很多遗忘，注释的重要性就显现出来了。同时，我也很满意自己的输出和代码格式，我个人很讨厌凌乱的输出和排版，所以每次写代码都力求简洁美观（除了实验五可能乱了些，之后再遇到这种情况我一样会把引用的东西放到main函数外面去的，如上段所言），这不光可以让自己或者要看我的代码的人感觉很舒服，也实际上方便了修改和维护代码。

总的来说，这次实验我最大的收获就是1、对这些算法有了更深的理解；2、提高了自己解决问题的能力；3、对代码的认识提高了；4、满满的成就感和满足感（哈哈）。