



東南大學

数值分析上机报告

院(系)名称: 微电子学院

学生姓名: 周玉乾

学号: 220205764

二〇二〇年十一月

第一章

一、 问题

舍入误差与有效数字

设 $S_N = \sum_{j=2}^N \frac{1}{j^2-1}$ ，其精确值为 $\frac{1}{2}(\frac{3}{2} - \frac{1}{N} - \frac{1}{N+1})$ 。

1. 编制按从大到小的顺序 $S_N = \frac{1}{2^2-1} + \frac{1}{3^2-1} + \dots + \frac{1}{N^2-1}$ ，计算 S_N 的通用程序；
2. 编制按从小到大的顺序 $S_N = \frac{1}{N^2-1} + \frac{1}{(N-1)^2-1} + \dots + \frac{1}{2^2-1}$ ，计算 S_N 的通用程序；
3. 按两种顺序分别计算 S_{10^2} 、 S_{10^4} 、 S_{10^6} ，并指出其有效位数 (编程时用单精度)；
4. 通过本上机题你明白了什么？

二、 分析

对于 $\frac{1}{N^2-1}$ ，当 N 很大时， $\frac{1}{N^2-1}$ 接近 0，因此如果按从大到小的顺序计算 $S_N = \frac{1}{2^2-1} + \frac{1}{3^2-1} + \dots + \frac{1}{N^2-1}$ ，由于计算机的舍入误差，会出现**大数吃小数**的情况，从而比真实结果略小；而如果按从小到大的顺序计算 $S_N = \frac{1}{N^2-1} + \frac{1}{(N-1)^2-1} + \dots + \frac{1}{2^2-1}$ ，其结果应该更加接近真实值。

三、 程序

q1-1.cpp

```
1 #include <iostream>
2 #include <iomanip>
3 #include <math.h>
4
5 float f(int N) {
6     float res = 0;
7     res = float(1.0)/(pow(float(N), float(2)) - 1);
8     return res;
9 }
10
11 float SN_1(int N) {
12     float sum = 0;
13     for (int i = 2; i <= N; i++)
14     {
15         sum += f(i);
```

```

16     }
17     return sum;
18 }
19
20 float SN_2(int N) {
21     float sum = 0;
22     for (int i = N; i >= 2; i--)
23     {
24         sum += f(i);
25     }
26     return sum;
27 }
28
29 float SN_Real(int N) {
30     float sum = 0;
31     sum = 0.5*(1.5 - 1/N - 1/(N+1));
32     return sum;
33 }
34
35 int main() {
36     float data0 = 0;
37     float data1 = 0;
38     float data2 = 0;
39
40     int N = 0;
41
42     std::cout<<"请输入N:"<<std::endl;
43     std::cin>>N;
44
45     data0 = SN_Real(N);
46     data1 = SN_1(N);
47     data2 = SN_2(N);
48
49     std::cout<<"N\t精确值\t\t从大到小\t误差1\t\t从小到大\t误差2"<<std::endl;
50     std::cout<<N<<"\t"<< std::fixed << std::setprecision(8)<<data0<<"\t"<<data1<<"\t"
        <<abs(data0-data1)<<"\t"<<data2<<"\t"<<abs(data0-data2)<<std::endl;
51
52     return 0;
53 }

```

四、算例

1. S_{10^2}

-
- 1 请输入 N: 100
 - 2 准确值: 0.7399495244
 - 3 正向求和: 0.7400494814, 误差: 0.0000999570

4 反向求和: 0.7400495410, 误差: 0.0001000166

2. S_{10^4}

- 1 请输入 N: 10000
 - 2 准确值: 0.7498999834
 - 3 正向求和: 0.7498521209, 误差: 0.0000478625
 - 4 反向求和: 0.7498999834, 误差: 0.0000000000
-

3. S_{10^6}

- 1 请输入 N: 1000000
 - 2 准确值: 0.7499989867
 - 3 正向求和: 0.7498521209, 误差: 0.0001468658
 - 4 反向求和: 0.7499990463, 误差: 0.0000000596
-

4. 从 $10^2 \sim N$ 将两种方式计算的误差曲线绘制出来 (为了观察变化趋势, 误差没有取绝对值):

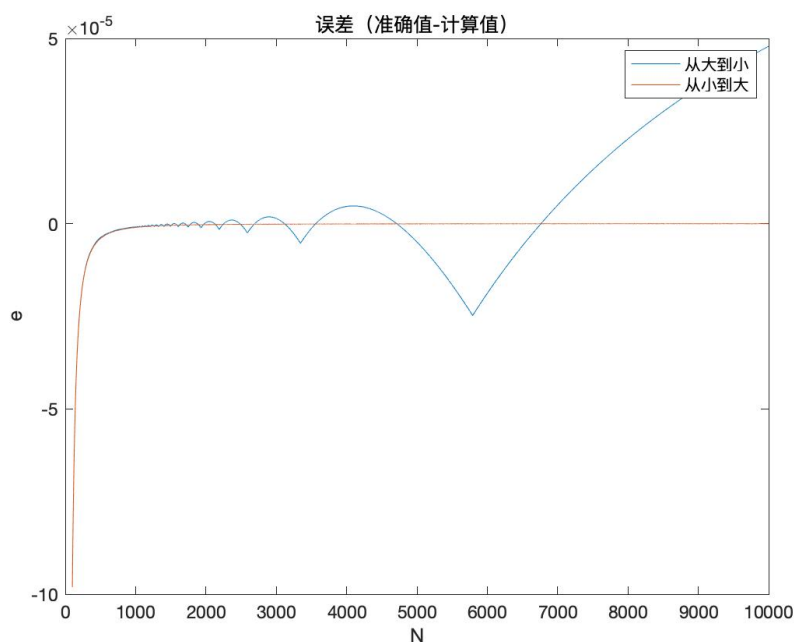


图 1.1 两种计算方法的误差对比, $N = 10000$

五、 结论

1. 编程证明了之前的分析, 及从大到小求和时, 会出现大数吃小数的现象, 导致误差偏大。从大到小求和时, 由于舍入误差的影响, 导致结果不稳定; 而从小到大求和结果则比较稳定, 可以看到随着 N 的增大, 误差逐渐趋于 0;

表 1.1 有效位数

N	10^2	10^4	10^6
从大到小	3	4	3
从小到大	3	7	6

2. 再次证明了数学上的等价并不意味着数值上的等价，在实际的运算中，舍入误差的影响不可低估，在计算中选择一种好的算法可以使结果更加精确。

第二章

一、 问题

试值法或者 Newton 法同二分法结合

问题 1

3.2.4 试值法(The Method of False Position)

由于二分法的收敛速度相对较慢,因此有些方法**尝试对它进行改进**.

二分法选择区间的中点进行下一次迭代,而所谓试值法选择

$$(a, f(a)), (b, f(b))$$

的连线同 x 轴的交点的横坐标作为下一个迭代点.

理论分析

假设 $f(a) \cdot f(b) < 0$. 经过点 $(a, f(a))$ 与 $(b, f(b))$ 的直线方程为:

$$y = \frac{f(b) - f(a)}{b - a}(x - b) + f(b),$$

令 $y = 0$, 求出

$$c = b - \frac{b - a}{f(b) - f(a)} \cdot f(b).$$

接下来有三种可能性:

- $f(c)$ 与 $f(a)$ 符号相反,则下一个有根区间为 $[a, c]$.
- $f(c)$ 与 $f(b)$ 符号相反,则下一个有根区间为 $[c, b]$.
- $f(c) = 0$, 计算结束.

第三种情况直接得结果, 否则根区间得到压缩. 同二分法类似, 可以构造一个 $\{[a_n, b_n]\}$ 的序列, 其中每个区间都包含零点, 零点 x^* 的近似值选为:

$$c_n = b_n - \frac{b_n - a_n}{f(b_n) - f(a_n)} \cdot f(b_n).$$

如果 $f(x)$ 是连续函数, 可以证明这个算法一定收敛. 若 $f(x)$ 是线性的, 该方法一步就得到根. 但是, 有时候该方法的收敛速度甚至比二分法还要慢. 二分法的有根区间的长度 $b_n - a_n$ 趋近于 0, 试值法里 $b_n - a_n$ 会越来越小, 但可能不趋近于 0. 因此, **该方法的终止判据应选择 $|f(c_n)| \leq \epsilon$.**

问题 2

Newton法同二分法的结合

为了提升Newton法的稳定性,减少初值对它的影响,可以把它与二分法相结合.具体的策略是:

- 假设 $a < b$, $f(a)f(b) < 0$,从 $x = a$ 或者 $x = b$ 开始迭代.

- 如果

$$\bar{x} = x - \frac{f(x)}{f'(x)} \in (a, b),$$

接受它,否则取 $\bar{x} = \frac{a+b}{2}$.

- 根据函数值的正负号,选取 $[a, \bar{x}]$ 或者 $[\bar{x}, b]$ 作为新的有根区间.
- 重复前面的过程,并在 $|f(\bar{x})|$ 足够小时终止迭代.

二、 分析

试值法流程

图2.1 为试值法算法流程图。

Newton 法同二分法结合流程

图2.2 为试值法算法流程图。

三、 程序

试值法

```
1 def TrailValue(expr, a, b, e):
2     """
3     试值法
4     F@函数
5     A@区间下限
6     B@区间上限
7     E@容忍误差限
```

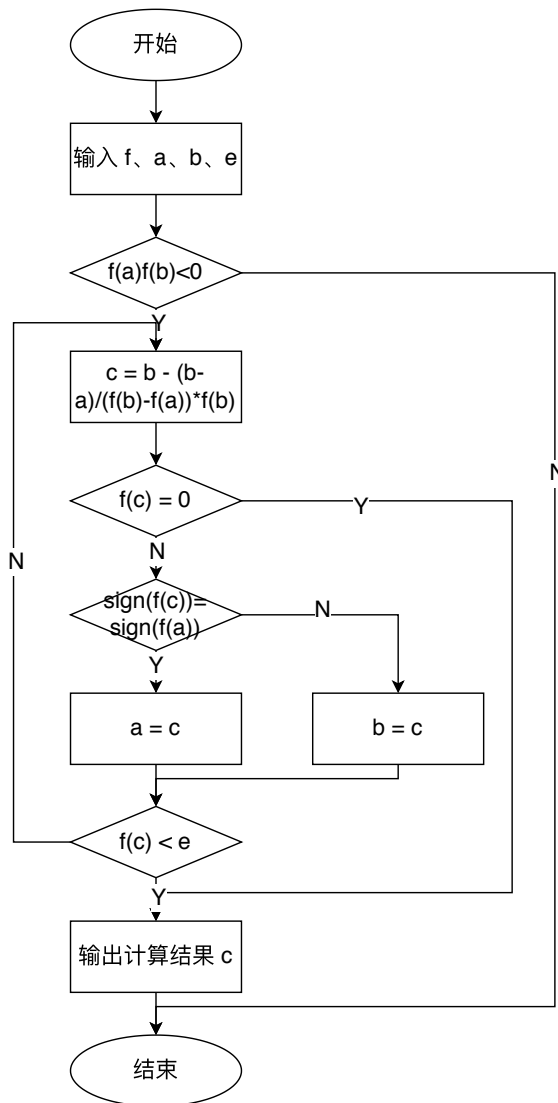


图 2.1 试值法流程图

```

8      """
9      f = _func(expr)
10     fa_0 = f.value(a)
11     fb_0 = f.value(b)
12     res = 0
13     count = 0
14     if abs(fa_0) < e:
15         res = a
16     elif abs(fb_0) < e:
17         res = b
18     elif sympy.sign(fa_0) == sympy.sign(fb_0):
19         print('f(a) and f(b) 同号')
20         sys.exit()
21     else:
22         while True:

```

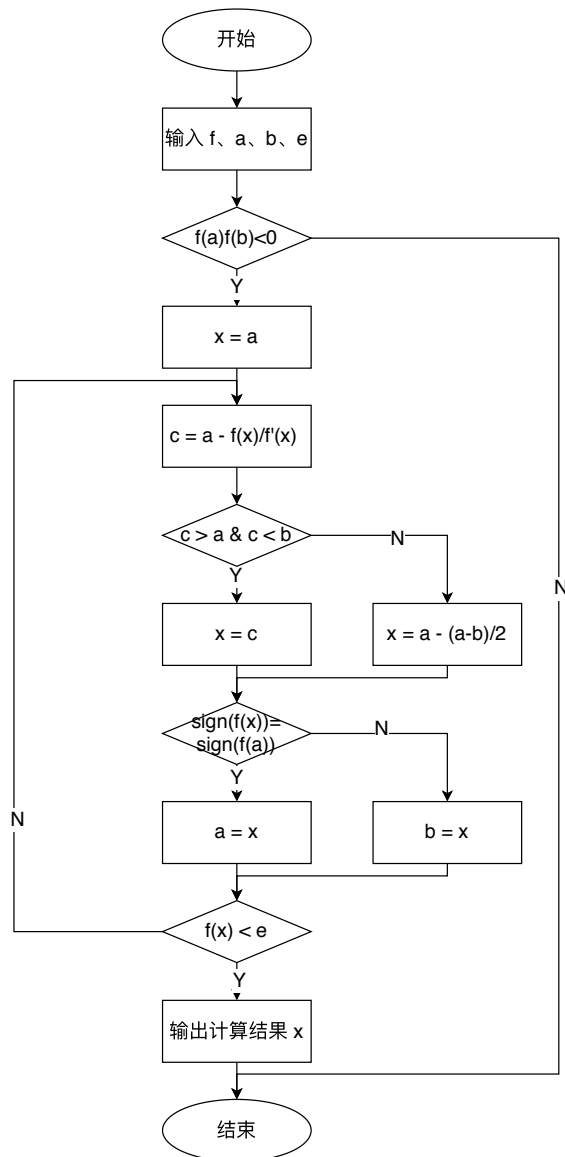



图 2.2 Newton 法同二分法结合流程

```

23     count = count + 1
24     fa = f.value(a)
25     fb = f.value(b)
26     c = b - ((b-a)/(fb - fa))*fb
27     fc = f.value(c)
28
29     # 更新有根区间
30     if sympy.sign(fa) == sympy.sign(fc):
31         a = c
32     else:
33         b = c
34
35     # 判断计算结束

```

```

36         if abs(f.value(c)) < e:
37             res = c
38             break
39
40     return res, count

```

Newton 法同二分法结合

```

1 def Newton(expr, a, b, e):
2     """
3     牛顿法与二分法结合
4     F@函数
5     A@区间下限
6     B@区间上限
7     E@容忍误差限
8     """
9     f = _func(expr)
10    fa_0 = f.value(a)
11    fb_0 = f.value(b)
12    res = 0
13    count = 0
14    if abs(fa_0) < e:
15        res = a
16    elif abs(fb_0) < e:
17        res = b
18    elif sympy.sign(fa_0) == sympy.sign(fb_0):
19        print('f(a) and f(b) 同号')
20        sys.exit()
21    else:
22        x = a
23        while True:
24            count = count + 1
25            c = x - f.value(x)/f.diff_value(x)
26
27            # NEWTON与二分法结合, 找下一个点
28            if (c > a) and (c < b):
29                x = c
30            else:
31                x = a+(b-a)/2
32
33            # 更新有根区间
34            if sympy.sign(f.value(a)) == sympy.sign(f.value(x)):
35                a = x
36            else:
37                b = x
38
39            # 判断计算结束
40            if abs(f.value(x)) < e:

```

```

41         res = x
42         break
43
44     return res, count

```

四、算例

1. $x \times \sin(x) - 1 = 0$, 有根区间为 $(1, 2)$, 误差限为 1×10^{-5} 。

1	试值法	根: 1.11416, 迭代次数: 3
2	牛顿法	根: 1.11416, 迭代次数: 2

2. $x^2 - 5 = 0$, 有根区间为 $(2, 3)$, 误差限为 1×10^{-5} 。

1	试值法	根: 2.23607, 迭代次数: 7
2	牛顿法	根: 2.23607, 迭代次数: 3

3. $x^3 - 3x + 2 = 0$, 有根区间为 $(-2.5, -1.5)$, 误差限为 1×10^{-5} 。

1	试值法	根: -2.00000, 迭代次数: 11
2	牛顿法	根: -2.00000, 迭代次数: 4

五、结论

1. 相比于试值法, Newton 与二分法结合的算法收敛速度更快;
2. Newton 与二分法结合提升了原始 Newton 法的稳定性;
3. 无论是试值法还是 Newton 与二分法结合的算法, 都只能求解函数穿过 x 轴的根, 不能求解函数与 x 轴相切的根, 例如在算例 3 中, 无法求解 $x = 1.0$ 的根。

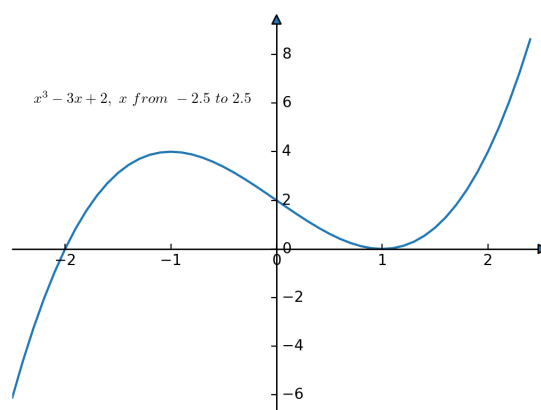


图 2.3 $f(x) = x^3 - 3x + 2$

第三章

一、 问题

列主元 Gauss 消去

对于某电路的分析，归结于求解线性方程组 $RI = V$ ，其中

$$R = \begin{bmatrix} 31 & -13 & 0 & 0 & 0 & -10 & 0 & 0 & 0 \\ -13 & 35 & -9 & 0 & -11 & 0 & 0 & 0 & 0 \\ 0 & -9 & 31 & -10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -10 & 79 & -30 & 0 & 0 & 0 & -9 \\ 0 & 0 & 0 & -30 & 57 & -7 & 0 & -5 & 0 \\ 0 & 0 & 0 & 0 & -7 & 47 & -30 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -30 & 41 & 0 & 0 \\ 0 & 0 & 0 & 0 & -5 & 0 & 0 & 27 & -2 \\ 0 & 0 & 0 & -9 & 0 & 0 & 0 & -2 & 29 \end{bmatrix}$$

$$V^T = (-15, 27, -23, 0, -20, 12, -7, 7, 10)^T$$

1. 编制解 n 阶线性方程组 $Ax = b$ 的列主元 Gauss 消去的通用程序；
2. 用所编程序解线性方程组 $RI = V$ ，并打印出解向量，保留 5 位有效数字；
3. 本题编程之中，你提高了哪些能力？

二、 分析

列主元 Gauss 消去的算法流程图如图 3.1 所示。

三、 程序

```
1 import sys
2 import numpy as np
3
4 A = [[31, -13, 0, 0, 0, -10, 0, 0, 0],
5      [-13, 35, -9, 0, -11, 0, 0, 0, 0],
6      [0, -9, 31, -10, 0, 0, 0, 0, 0],
7      [0, 0, -10, 79, -30, 0, 0, 0, -9],
```

```

8     [0, 0, 0, -30, 57, -7, 0, -5, 0],
9     [0, 0, 0, 0, -7, 47, -30, 0, 0],
10    [0, 0, 0, 0, 0, -30, 41, 0, 0],
11    [0, 0, 0, 0, -5, 0, 0, 27, -2],
12    [0, 0, 0, -9, 0, 0, 0, -2, 29]]
13
14 b = [-15, 27, -23, 0, -20, 12, -7, 7, 10]
15
16 def find_shape(A, b):
17     """
18     获取 N
19     """
20
21     # N行 M列
22     n1, m1 = A.shape
23     n2, = b.shape
24
25     # PRINT(N1, M1)
26     # PRINT(N2)
27
28     # 判断矩阵形状
29     if n1 != m1 or n1 != n2:
30         print('Error martix shape!')
31         sys.exit()
32     else:
33         N = n1
34
35     return N
36
37
38 def MGauss(A, b):
39     """
40     列主元 GAUSS 消去 消元
41     """
42
43     # 判断矩阵形状
44     N = find_shape(A, b)
45
46     # 列主元高斯消元
47     for k in range(0, N):
48         p = k
49         maxabs = abs(A[k, k])
50         # 找列最大值
51         for i in range(k+1, N):
52             if abs(A[i, k]) > maxabs:
53                 p = i
54                 maxabs = abs(A[i, k])
55         print('maxabs', maxabs)

```

```

56     # 最大值为 0
57     if maxabs == 0:
58         print('Singular')
59         sys.exit()
60     # 最大值不在对角线, 则交换两行
61     if p != k:
62         A[[p,k],:] = A[[k,p],:]
63         b[[p,k]] = b[[k,p]]
64     print('exchange r{0} and r{1}:\r\n'.format(k, p), A)
65     # 消元, 将对角线以下变为 0
66     for i in range(k+1, N):
67         m_ik = A[i, k] / A[k, k]
68         for j in range(0, N):
69             A[i, j] -= A[k, j] * m_ik
70             b[i] -= b[k] * m_ik
71     print('After Elimination:\r\n', np.concatenate((A,np.asarray([b]).T), axis =
72             1))
73
74     if A[N-1, N-1] == 0:
75         print('Singular')
76         sys.exit()
77
78     return A, b
79
80 def bring_back(A, b):
81     """
82     列主元 GAUSS 消去 回带
83     """
84
85     # 判断矩阵形状
86     N = find_shape(A, b)
87
88     # 回带
89     X = np.zeros(N)
90     X[N-1] = b[N-1] / A[N-1, N-1]
91     for i in range(0, N-1):
92         k = N-2-i
93         sigma = sum(A[k, j]*X[j] for j in range(k+1, N))
94         # PRINT('K:{0}, SIGMA:{1}'.format(k, sigma))
95         X[k] = (b[k] - sigma) / A[k, k]
96
97     return X
98
99 def main():
100     """
101     MAIN
102     """

```

```

103     A_np = np.asarray(A, dtype = float)
104     b_np = np.asarray(b, dtype = float)
105
106     # B_NP = B_NP.T
107
108     print('A:\r\n', A_np)
109     print('b:\r\n', b_np)
110
111     A_G, b_G = MGauss(A_np, b_np)
112     x_G = bring_back(A_G, b_G)
113
114     print('A_G:b_G\r\n', np.concatenate((A_G,np.asarray([b_G]).T), axis = 1))
115     print('x_G', x_G)
116
117
118 if __name__ == "__main__":
119     main()

```

四、算例

$$A = \begin{bmatrix} 31 & -13 & 0 & 0 & 0 & -10 & 0 & 0 & 0 \\ -13 & 35 & -9 & 0 & -11 & 0 & 0 & 0 & 0 \\ 0 & -9 & 31 & -10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -10 & 79 & -30 & 0 & 0 & 0 & -9 \\ 0 & 0 & 0 & -30 & 57 & -7 & 0 & -5 & 0 \\ 0 & 0 & 0 & 0 & -7 & 47 & -30 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -30 & 41 & 0 & 0 \\ 0 & 0 & 0 & 0 & -5 & 0 & 0 & 27 & -2 \\ 0 & 0 & 0 & -9 & 0 & 0 & 0 & -2 & 29 \end{bmatrix}$$

$$b^T = (-15, 27, -23, 0, -20, 12, -7, 7, 10)^T$$

运算结果:

```

1 x_G [-0.28923382  0.34543572 -0.71281173 -0.22060851 -0.43040043  0.15430874
      -0.05782287  0.20105389  0.29022866]

```

使用 MATLAB 自带的求解线性方程组的方法求解，验证编写算法的正确性:

```

1 >> A \ b'
2
3 ans =
4

```

5	-0.2892
6	0.3454
7	-0.7128
8	-0.2206
9	-0.4304
10	0.1543
11	-0.0578
12	0.2011
13	0.2902

可以看到结果是一致的。

五、 结论

1. 列主元 Gauss 消去法避免了小数作除数，因此一般能保证舍入误差不增大，这个方法基本上是稳定的；
2. MATLAB 中可用 $A \setminus b$ 求线性方程组解。

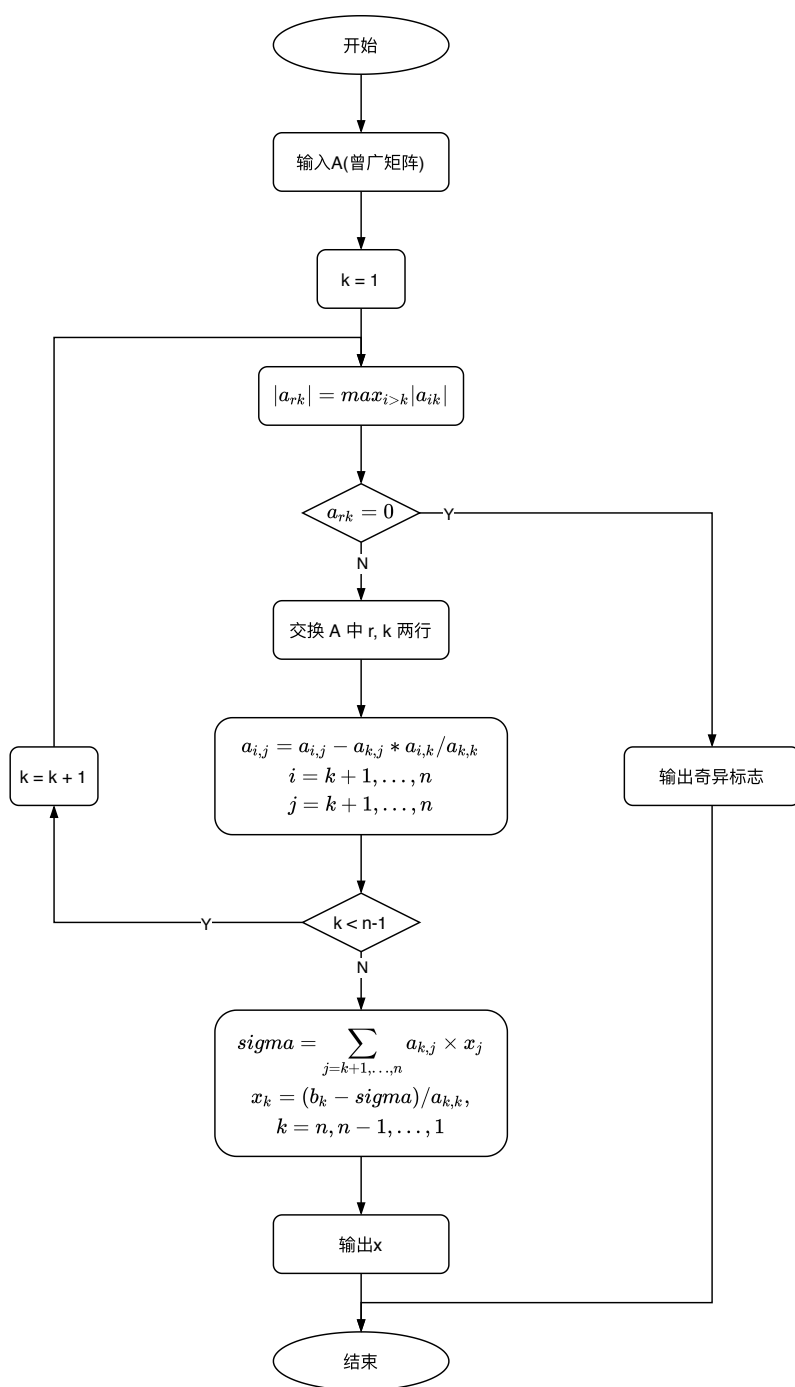


图 3.1 列主元 Gauss 消去算法流程

附录 A 第二章代码

q2-1.py

```
1 import sys
2 import sympy
3
4 def func_1_expr():
5     """
6      $x \sin(x) - 1 = 0$ 
7     """
8     x = sympy.symbols('x')
9     return x*sympy.sin(x)-1.0
10
11 def func_2_expr():
12     """
13      $x^2 - 5 = 0$ 
14     """
15     x = sympy.symbols('x')
16     return x**2.0 - 5.0
17
18 def func_3_expr():
19     """
20      $x^3 - 3x + 2 = 0$ 
21     """
22     x = sympy.symbols('x')
23     return x**3.0 - 3.0*x +2
24
25 class _func():
26     """
27     计算一元函数值及导数值
28     """
29     def __init__(self, expr, eff=15):
30         """
31         初始化计算表达式
32         EXPR@表达式
33         EFF@有效数字位数
34         """
35         self._expr = expr()
36         self._eff = eff
37         self._x = list(self._expr.free_symbols)[0]
38         self._diff_expr = sympy.diff(self._expr, self._x)
39
```

```

40     def value(self, x):
41         """
42         计算 FUNC 值
43         """
44         expr = self._expr
45         return expr.subs('x', x).evalf(self._eff)
46
47     def diff_value(self, x):
48         """
49         计算导数值
50         """
51         expr = self._diff_expr
52         return expr.subs('x', x).evalf(self._eff)
53
54
55
56 def TrailValue(expr, a, b, e):
57     """
58     试值法
59     F@函数
60     A@区间下限
61     B@区间上限
62     E@容忍误差限
63     """
64     f = _func(expr)
65     fa_0 = f.value(a)
66     fb_0 = f.value(b)
67     res = 0
68     count = 0
69     if abs(fa_0) < e:
70         res = a
71     elif abs(fb_0) < e:
72         res = b
73     elif sympy.sign(fa_0) == sympy.sign(fb_0):
74         print('f(a) and f(b) 同号')
75         sys.exit()
76     else:
77         while True:
78             count = count + 1
79             fa = f.value(a)
80             fb = f.value(b)
81             c = b - ((b-a)/(fb - fa))*fb
82             fc = f.value(c)
83
84             # 更新有根区间
85             if sympy.sign(fa) == sympy.sign(fc):
86                 a = c
87             else:

```

```

88         b = c
89
90         # 判断计算结束
91         if abs(f.value(c)) < e:
92             res = c
93             break
94
95     return res, count
96
97 def Newton(expr, a, b, e):
98     """
99     牛顿法与二分法结合
100    F@函数
101    A@区间下限
102    B@区间上限
103    E@容忍误差限
104    """
105    f = _func(expr)
106    fa_0 = f.value(a)
107    fb_0 = f.value(b)
108    res = 0
109    count = 0
110    if abs(fa_0) < e:
111        res = a
112    elif abs(fb_0) < e:
113        res = b
114    elif sympy.sign(fa_0) == sympy.sign(fb_0):
115        print('f(a) and f(b) 同号')
116        sys.exit()
117    else:
118        x = a
119        while True:
120            count = count + 1
121            c = x - f.value(x)/f.diff_value(x)
122
123            # NEWTON与二分法结合，找下一个点
124            if (c > a) and (c < b):
125                x = c
126            else:
127                x = a+(b-a)/2
128
129            # 更新有根区间
130            if sympy.sign(f.value(a)) == sympy.sign(f.value(x)):
131                a = x
132            else:
133                b = x
134
135        # 判断计算结束

```

```

136         if abs(f.value(x)) < e:
137             res = x
138             break
139
140     return res, count
141
142 def main():
143     """
144     MAIN
145     """
146     res_t, count_t = TrailValue(func_1_expr, 1, 2, 1.0/sympy.Pow(10, 5))
147     print('试值法 func 1 根: %.5f, 迭代次数: %d' % (res_t, count_t))
148
149     res_n, count_n = Newton(func_1_expr, 1, 2, 1.0/sympy.Pow(10, 5))
150     print('牛顿法 func 1 根: %.5f, 迭代次数: %d' % (res_n, count_n))
151
152     res_t, count_t = TrailValue(func_2_expr, 2, 3, 1.0/sympy.Pow(10, 5))
153     print('试值法 func 2 根: %.5f, 迭代次数: %d' % (res_t, count_t))
154
155     res_n, count_n = Newton(func_2_expr, 2, 3, 1.0/sympy.Pow(10, 5))
156     print('牛顿法 func 2 根: %.5f, 迭代次数: %d' % (res_n, count_n))
157
158     res_t, count_t = TrailValue(func_3_expr, -2.5, -1.5, 1.0/sympy.Pow(10, 5))
159     print('试值法 func 3 根: %.5f, 迭代次数: %d' % (res_t, count_t))
160
161     res_n, count_n = Newton(func_3_expr, -2.5, -1.5, 1.0/sympy.Pow(10, 5))
162     print('牛顿法 func 3 根: %.5f, 迭代次数: %d' % (res_n, count_n))
163
164 if __name__ == "__main__":
165     main()

```
