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 $S_N = \frac{1}{2^2-1} + \frac{1}{3^2-1} + ... + \frac{1}{N^2-1}$，计算 S_N 的通用程序；
2. 编制按从小到大的顺序 $S_N = \frac{1}{N^2-1} + \frac{1}{(N-1)^2-1} + ... + \frac{1}{2^2-1}$，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 $S_N = \frac{1}{2^2-1} + \frac{1}{3^2-1} + ... + \frac{1}{N^2-1}$，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 $S_N = \frac{1}{N^2-1} + \frac{1}{(N-1)^2-1} + ... + \frac{1}{2^2-1}$，其结果应该更加接近真实值。

三、 程序

q1-1.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <math.h>

float f(int N) {
    float res = 0;
    res = float(1.0)/(pow(float(N), float(2)) - 1);
    return res;
}

float SN_1(int N) {
    float sum = 0;
    for (int i = 2; i <= N; i++)
    {
        sum += f(i);
```

```cpp
    }
    return sum;
}

float SN_2(int N) {
    float sum = 0;
    for (int i = N; i >= 2; i--)
    {
        sum += f(i);
    }
    return sum;
}

float SN_Real(int N) {
    float sum = 0;
    sum = 0.5*(1.5 - 1/N - 1/(N+1));
    return sum;
}

int main() {
    float data0 = 0;
    float data1 = 0;
    float data2 = 0;

    int N = 0;

    std::cout<<"请输入N:"<<std::endl;
    std::cin>>N;

    data0 = SN_Real(N);
    data1 = SN_1(N);
    data2 = SN_2(N);

    std::cout<<"N\t精确值\t\t从大到小\t误差1    \t从小到大\t误差2"<<std::endl;
    std::cout<<N<<"\t"<< std::fixed << std::setprecision(8)<<data0<<"\t"<<data1<<"\t"
        <<abs(data0-data1)<<"\t"<<data2<<"\t"<<abs(data0-data2)<<std::endl;

    return 0;
}
```

四、 算例

1. S_{10^2}

```
请输入 N: 100
准确值: 0.7399495244
正向求和: 0.7400494814, 误差: 0.0000999570
```

```
反向求和: 0.7400495410, 误差: 0.0001000166
```

2. S_{10^4}

```
请输入 N: 10000
准确值: 0.7498999834
正向求和: 0.7498521209, 误差: 0.0000478625
反向求和: 0.7498999834, 误差: 0.0000000000
```

3. S_{10^6}

```
请输入 N: 1000000
准确值: 0.7499989867
正向求和: 0.7498521209, 误差: 0.0001468658
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):

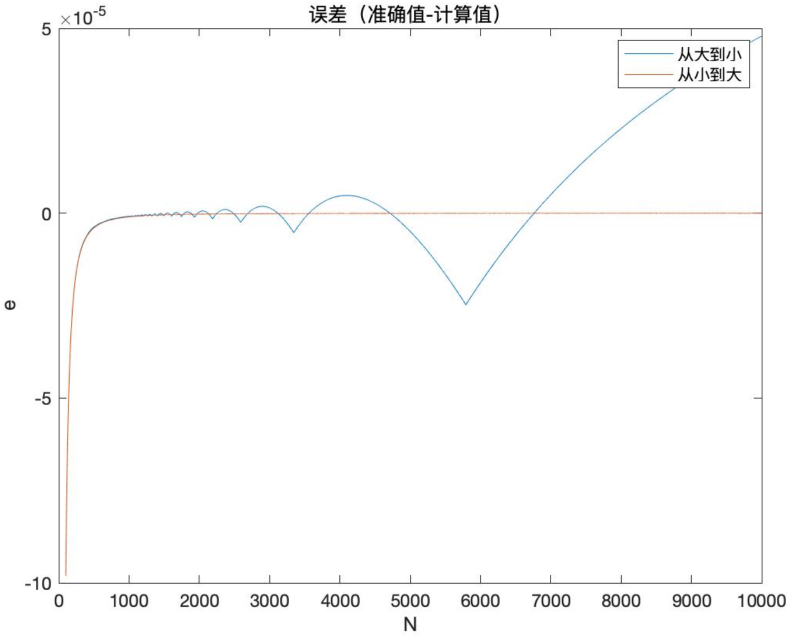

图 1.1　两种计算方法的误差对比, N = 10000

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

$$\big(a, f(a)\big),\ \big(b, f(b)\big)$$

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 $\big(a, f(a)\big)$ 与 $\big(b, f(b)\big)$ 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** $|f(c_n)|\le\epsilon$.

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 $\bar{x} = \dfrac{a+b}{2}$.
- 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

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

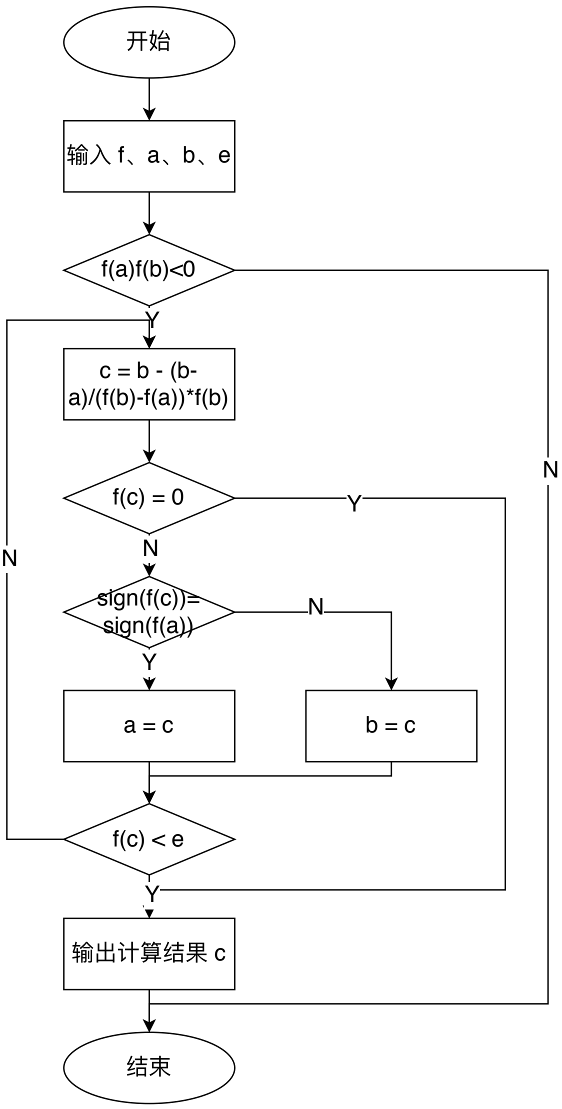

图 2.1　试值法流程图

```
"""
f = _func(expr)
fa_0 = f.value(a)
fb_0 = f.value(b)
res = 0
count = 0
if abs(fa_0) < e:
    res = a
elif abs(fb_0) < e:
    res = b
elif sympy.sign(fa_0) == sympy.sign(fb_0):
    print('f(a) and f(b) 同号')
    sys.exit()
else:
    while True:
```

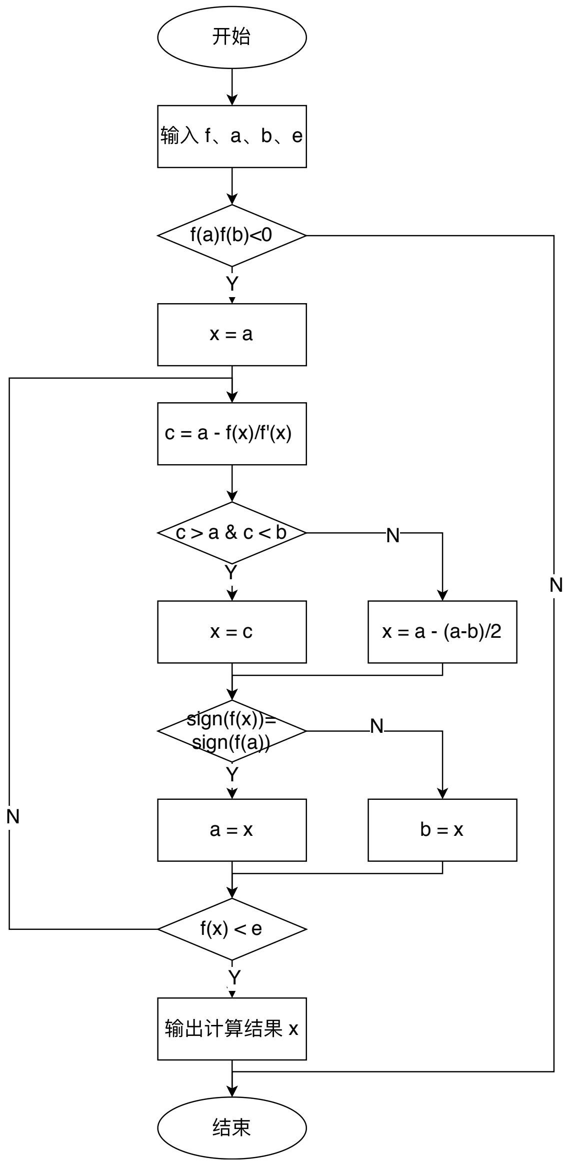

图 2.2　Newton 法同二分法结合流程

```
        count = count + 1
        fa = f.value(a)
        fb = f.value(b)
        c = b - ((b-a)/(fb - fa))*fb
        fc = f.value(c)

        # 更新有根区间
        if sympy.sign(fa) == sympy.sign(fc):
            a = c
        else:
            b = c

        # 判断计算结束
```

```python
            if abs(f.value(c)) < e:
                res = c
                break

    return res, count
```

Newton 法同二分法结合

```python
def Newton(expr, a, b, e):
    """
    牛顿法与二分法结合
    F@函数
    A@区间下限
    B@区间上限
    E@容忍误差限
    """
    f = _func(expr)
    fa_0 = f.value(a)
    fb_0 = f.value(b)
    res = 0
    count = 0
    if abs(fa_0) < e:
        res = a
    elif abs(fb_0) < e:
        res = b
    elif sympy.sign(fa_0) == sympy.sign(fb_0):
        print('f(a) and f(b) 同号')
        sys.exit()
    else:
        x = a
        while True:
            count = count + 1
            c = x - f.value(x)/f.diff_value(x)

            # Newton与二分法结合，找下一个点
            if (c > a) and (c < b):
                x = c
            else:
                x = a+(b-a)/2

            # 更新有根区间
            if sympy.sign(f.value(a)) == sympy.sign(f.value(x)):
                a = x
            else:
                b = x

            # 判断计算结束
            if abs(f.value(x)) < e:
```

```
            res = x
            break

    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}。

```
试值法 根：1.11416，迭代次数：3
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}。

```
试值法 根：2.23607，迭代次数：7
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}。

```
试值法 根：-2.00000，迭代次数：11
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。

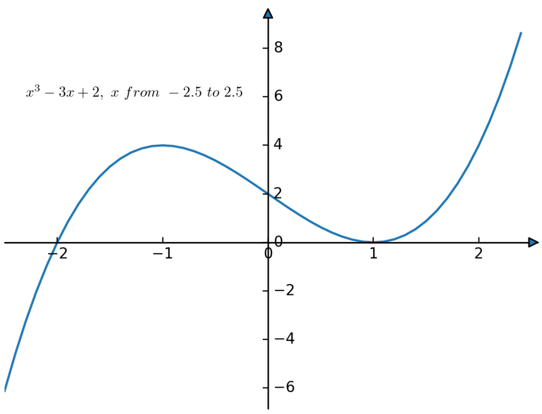

图 2.3　$f(x) = x^3 - 3x + 2$

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

```python
import sys
import numpy as np

A = [[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],
```

```python
    [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]]

b = [-15, 27, -23, 0, -20, 12, -7, 7, 10]

def find_shape(A, b):
    """
    获取 n
    """

    # n行 m列
    n1, m1 = A.shape
    n2, = b.shape

    # print(n1, m1)
    # print(n2)

    # 判断矩阵形状
    if n1 != m1 or n1 != n2:
        print('Error martix shape!')
        sys.exit()
    else:
        N = n1

    return N


def MGauss(A, b):
    """
    列主元 Gauss 消去 消元
    """

    # 判断矩阵形状
    N = find_shape(A, b)

    # 列主元高斯消元
    for k in range(0, N):
        p = k
        maxabs = abs(A[k, k])
        # 找列最大值
        for i in range(k+1, N):
            if abs(A[i, k]) > maxabs:
                p = i
                maxabs = abs(A[i, k])
        print('maxabs', maxabs)
```

```python
        # 最大值为 0
        if maxabs == 0:
            print('Singular')
            sys.exit()
        # 最大值不在对角线，则交换两行
        if p != k:
            A[[p,k],:] = A[[k,p],:]
            b[[p,k]] = b[[k,p]]
        print('exchange r{0} and r{1}:\r\n'.format(k, p), A)
        # 消元，将对角线以下变为 0
        for i in range(k+1, N):
            m_ik = A[i, k] / A[k, k]
            for j in range(0, N):
                A[i, j] -= A[k, j] * m_ik
            b[i] -= b[k] * m_ik
        print('After Elimination:\r\n', np.concatenate((A,np.asarray([b]).T), axis =
             1))

    if A[N-1, N-1] == 0:
        print('Singular')
        sys.exit()

    return A, b

def bring_back(A, b):
    """
    列主元 Gauss 消去 回带
    """

    # 判断矩阵形状
    N = find_shape(A, b)

    # 回带
    X = np.zeros(N)
    X[N-1] = b[N-1] / A[N-1, N-1]
    for i in range(0, N-1):
        k = N-2-i
        sigma = sum(A[k, j]*X[j] for j in range(k+1, N))
        # print('k:{0}, sigma:{1}'.format(k, sigma))
        X[k] = (b[k] - sigma) / A[k, k]

    return X

def main():
    """
    main
    """

```

```python
    A_np = np.asarray(A, dtype = float)
    b_np = np.asarray(b, dtype = float)

    # B_NP = B_NP.T

    print('A:\r\n', A_np)
    print('b:\r\n', b_np)

    A_G, b_G = MGauss(A_np, b_np)
    x_G = bring_back(A_G, b_G)

    print('A_G:b_G\r\n', np.concatenate((A_G,np.asarray([b_G]).T), axis = 1))
    print('x_G', x_G)


if __name__ == "__main__":
    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:

```
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:

```
>> A \ b'

ans =

```

```
-0.2892
 0.3454
-0.7128
-0.2206
-0.4304
 0.1543
-0.0578
 0.2011
 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 \ b` 求线性方程组解。

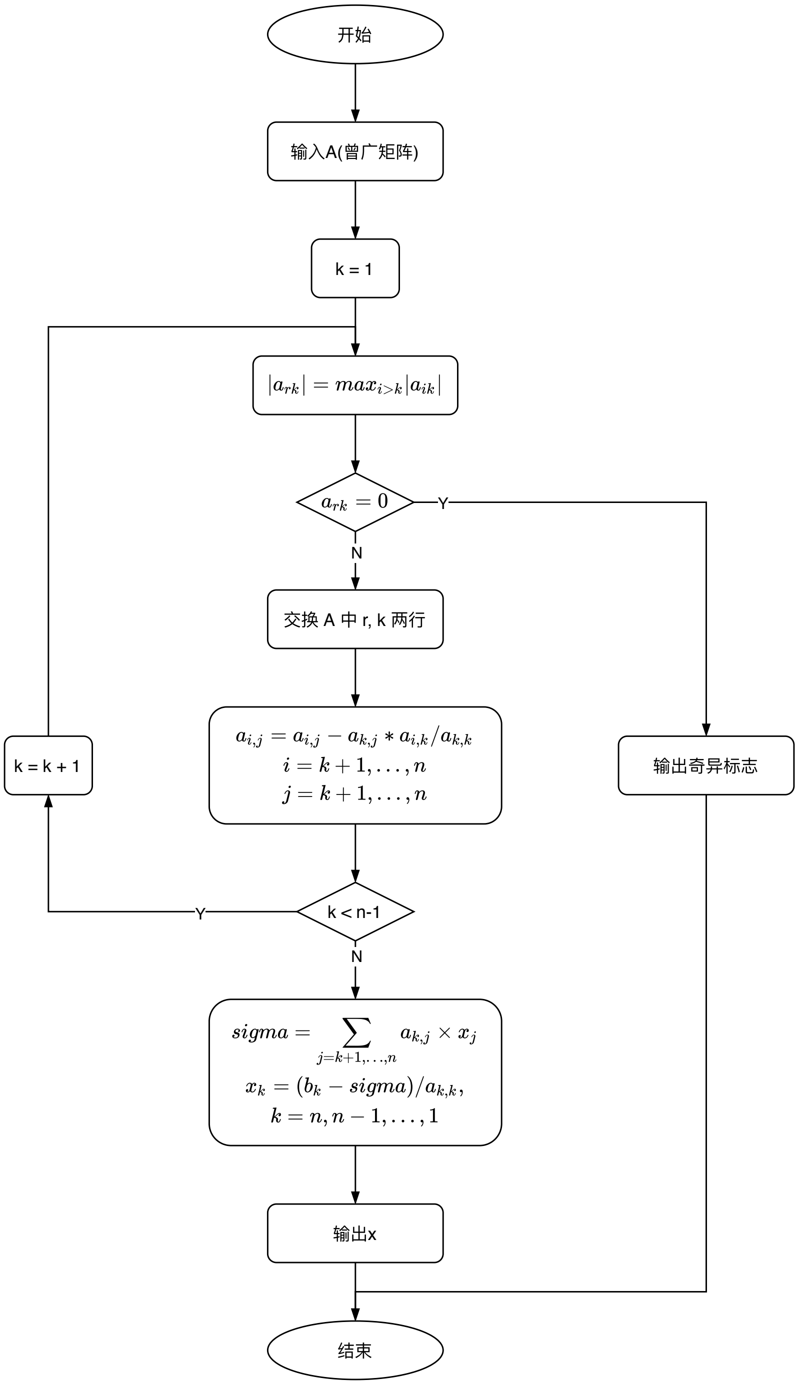

图 3.1　列主元 Gauss 消去算法流程

附录 A 第二章代码

q2-1.py

```
import sys
import sympy

def func_1_expr():
    """
    x*sin(x) - 1 = 0
    """
    x = sympy.symbols('x')
    return x*sympy.sin(x)-1.0

def func_2_expr():
    """
    x^2 - 5 = 0
    """
    x = sympy.symbols('x')
    return x**2.0 - 5.0

def func_3_expr():
    """
    x^3 -3x + 2 = 0
    """
    x = sympy.symbols('x')
    return x**3.0 - 3.0*x +2

class _func():
    """
    计算一元函数值及导数值
    """
    def __init__(self, expr, eff=15):
        """
        初始化计算表达式
        expr@表达式
        eff@有效数字位数
        """
        self._expr = expr()
        self._eff = eff
        self._x = list(self._expr.free_symbols)[0]
        self._diff_expr = sympy.diff(self._expr, self._x)

```

```
    def value(self, x):
        """
        计算 FUNC 值
        """
        expr = self._expr
        return expr.subs('x', x).evalf(self._eff)

    def diff_value(self, x):
        """
        计算导数值
        """
        expr = self._diff_expr
        return expr.subs('x', x).evalf(self._eff)



def TrailValue(expr, a, b, e):
    """
    试值法
    F@函数
    A@区间下限
    B@区间上限
    E@容忍误差限
    """
    f = _func(expr)
    fa_0 = f.value(a)
    fb_0 = f.value(b)
    res = 0
    count = 0
    if abs(fa_0) < e:
        res = a
    elif abs(fb_0) < e:
        res = b
    elif sympy.sign(fa_0) == sympy.sign(fb_0):
        print('f(a) and f(b) 同号')
        sys.exit()
    else:
        while True:
            count = count + 1
            fa = f.value(a)
            fb = f.value(b)
            c = b - ((b-a)/(fb - fa))*fb
            fc = f.value(c)

            # 更新有根区间
            if sympy.sign(fa) == sympy.sign(fc):
                a = c
            else:
```

```
                b = c

            # 判断计算结束
            if abs(f.value(c)) < e:
                res = c
                break

    return res, count

def Newton(expr, a, b, e):
    """
    牛顿法与二分法结合
    F@函数
    A@区间下限
    B@区间上限
    E@容忍误差限
    """
    f = _func(expr)
    fa_0 = f.value(a)
    fb_0 = f.value(b)
    res = 0
    count = 0
    if abs(fa_0) < e:
        res = a
    elif abs(fb_0) < e:
        res = b
    elif sympy.sign(fa_0) == sympy.sign(fb_0):
        print('f(a) and f(b) 同号')
        sys.exit()
    else:
        x = a
        while True:
            count = count + 1
            c = x - f.value(x)/f.diff_value(x)

            # Newton与二分法结合，找下一个点
            if (c > a) and (c < b):
                x = c
            else:
                x = a+(b-a)/2

            # 更新有根区间
            if sympy.sign(f.value(a)) == sympy.sign(f.value(x)):
                a = x
            else:
                b = x

            # 判断计算结束
```

```python
            if abs(f.value(x)) < e:
                res = x
                break

    return res, count

def main():
    """
    MAIN
    """
    res_t, count_t = TrailValue(func_1_expr, 1, 2, 1.0/sympy.Pow(10, 5))
    print('试值法 func 1 根: %.5f, 迭代次数: %d' % (res_t, count_t))

    res_n, count_n = Newton(func_1_expr, 1, 2, 1.0/sympy.Pow(10, 5))
    print('牛顿法 func 1 根: %.5f, 迭代次数: %d' % (res_n, count_n))

    res_t, count_t = TrailValue(func_2_expr, 2, 3, 1.0/sympy.Pow(10, 5))
    print('试值法 func 2 根: %.5f, 迭代次数: %d' % (res_t, count_t))

    res_n, count_n = Newton(func_2_expr, 2, 3, 1.0/sympy.Pow(10, 5))
    print('牛顿法 func 2 根: %.5f, 迭代次数: %d' % (res_n, count_n))

    res_t, count_t = TrailValue(func_3_expr, -2.5, -1.5, 1.0/sympy.Pow(10, 5))
    print('试值法 func 3 根: %.5f, 迭代次数: %d' % (res_t, count_t))

    res_n, count_n = Newton(func_3_expr, -2.5, -1.5, 1.0/sympy.Pow(10, 5))
    print('牛顿法 func 3 根: %.5f, 迭代次数: %d' % (res_n, count_n))

if __name__ == "__main__":
    main()
```