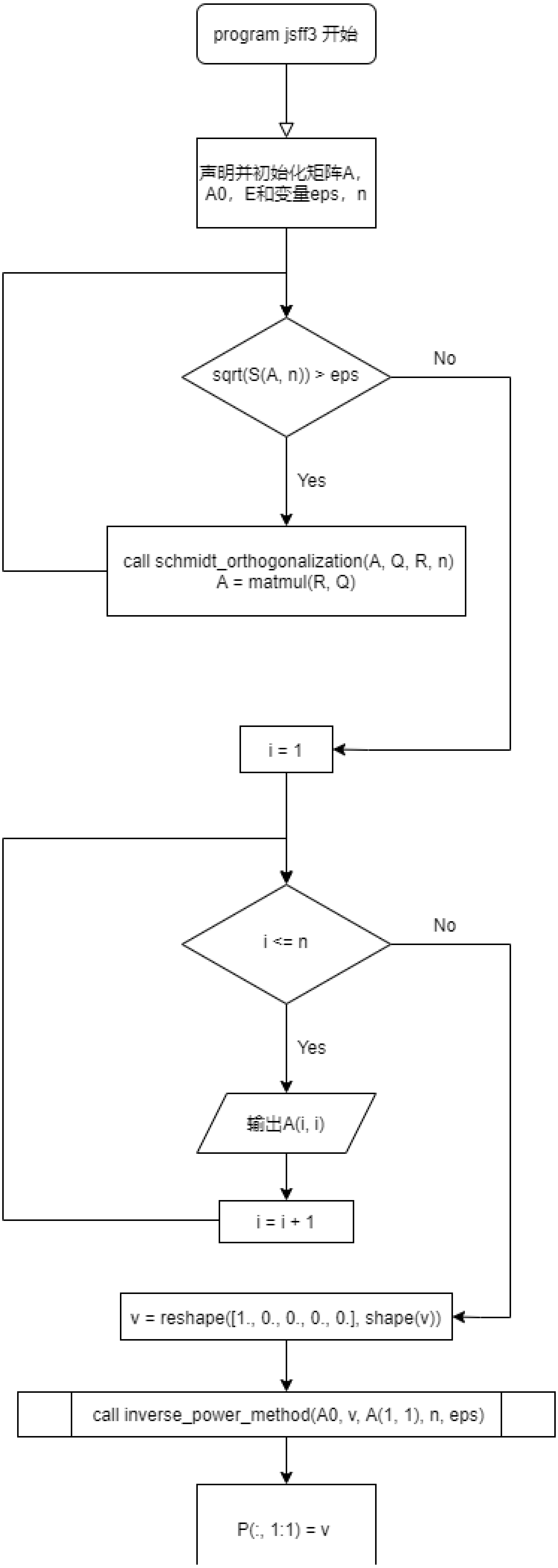
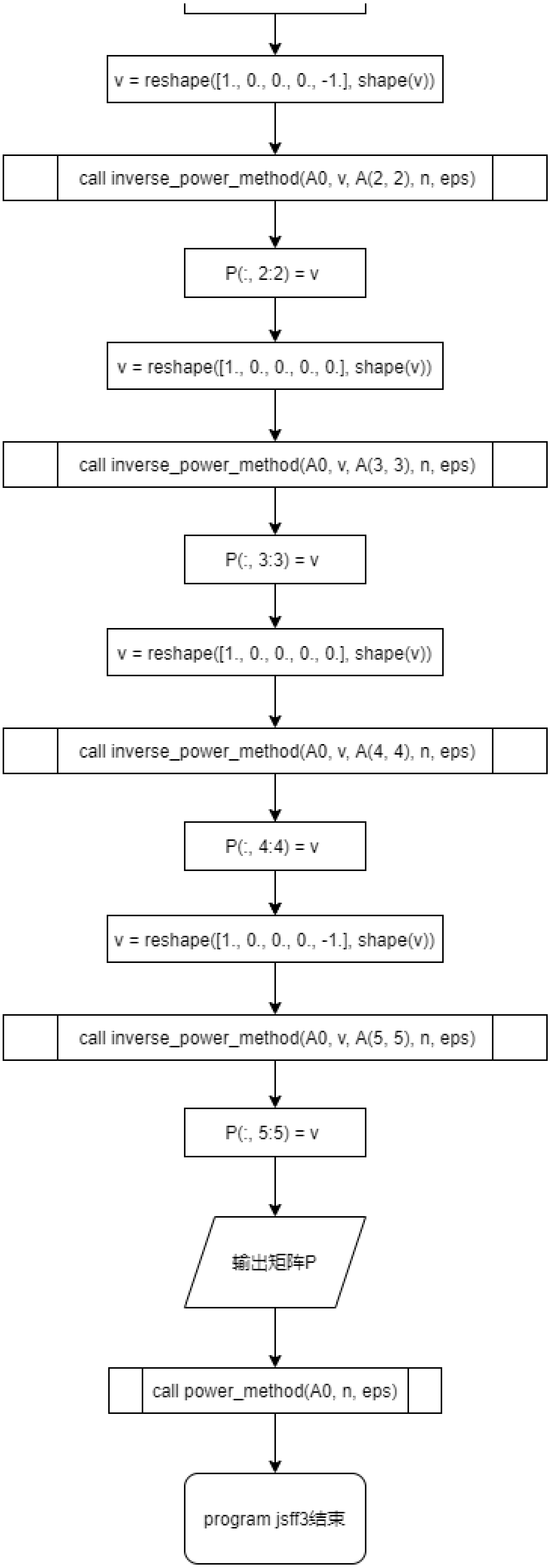
计算方法上机实习三 实习报告

2019级 大气科学学院 赵志宇

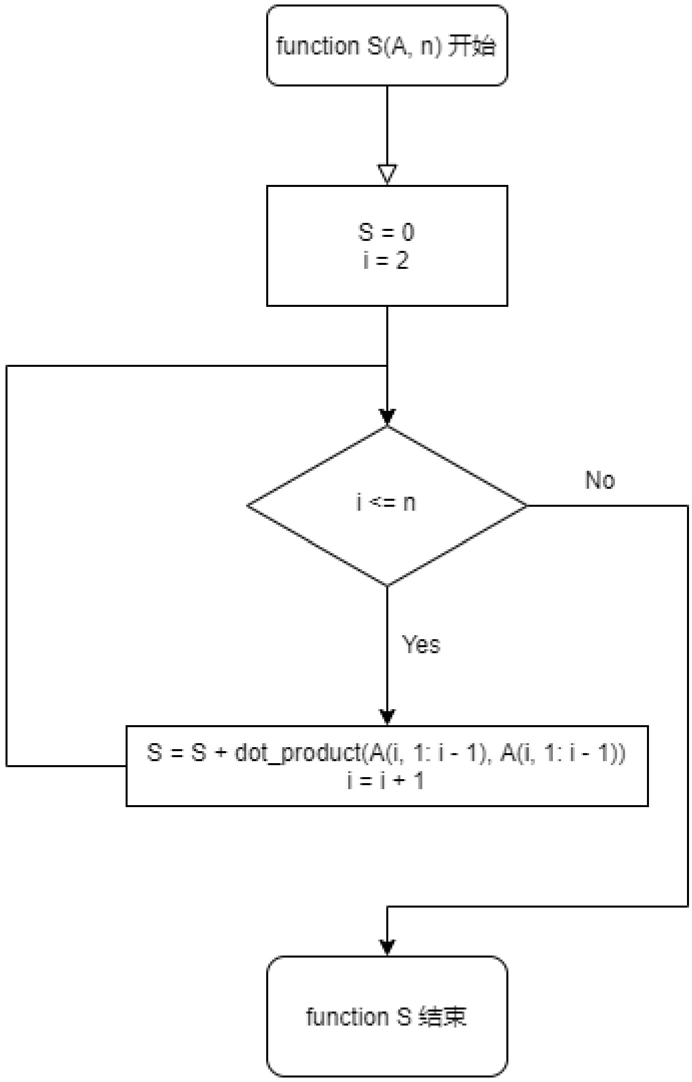
学号：191830227

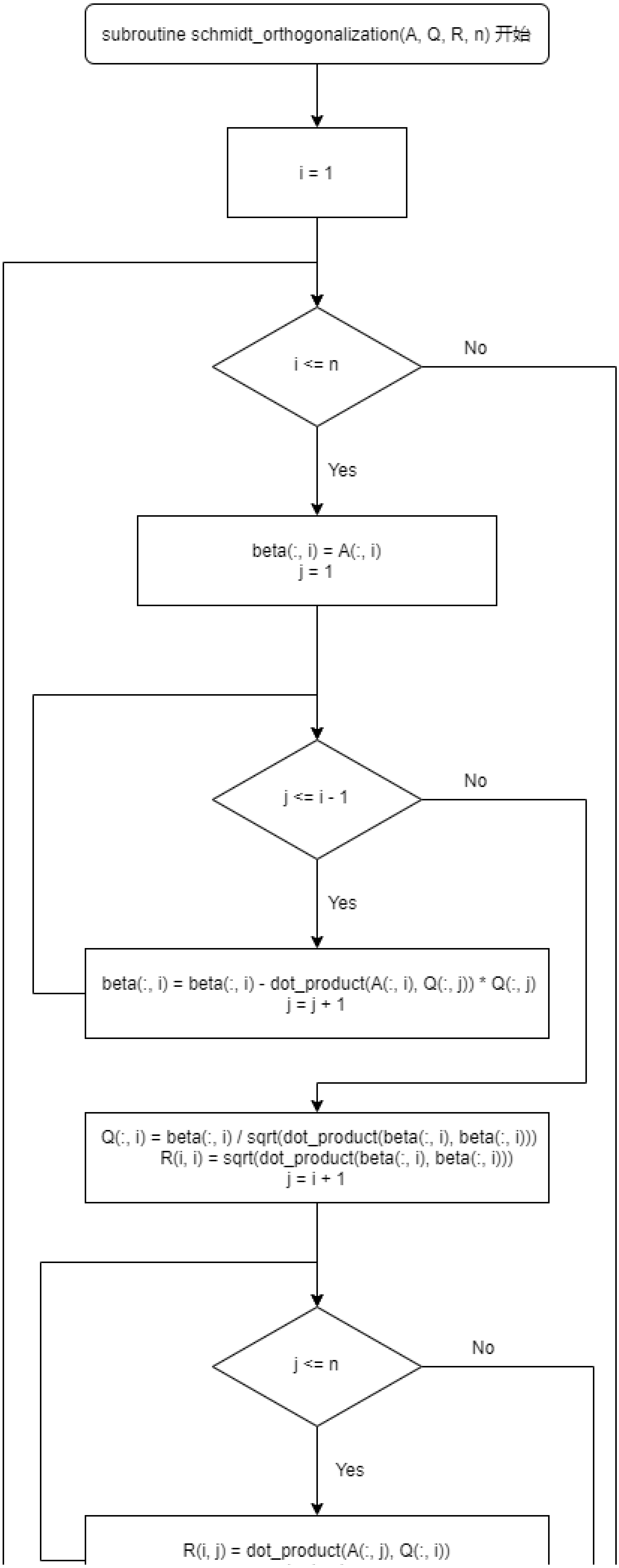
一、编程流程图

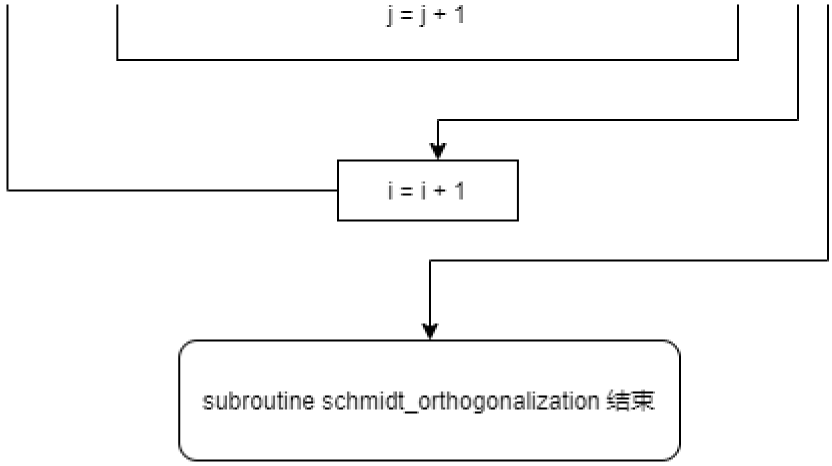

program jsff3 开始
声明并初始化矩阵A，A0，E和变量eps，n
sqrt(S(A, n)) > eps
No
Yes
call schmidt_orthogonalization(A, Q, R, n)
A = matmul(R, Q)
i = 1
i <= n
No
Yes
输出A(i, i)
i = i + 1
v = reshape([1., 0., 0., 0., 0.], shape(v))
call inverse_power_method(A0, v, A(1, 1), n, eps)
P(:, 1:1) = v

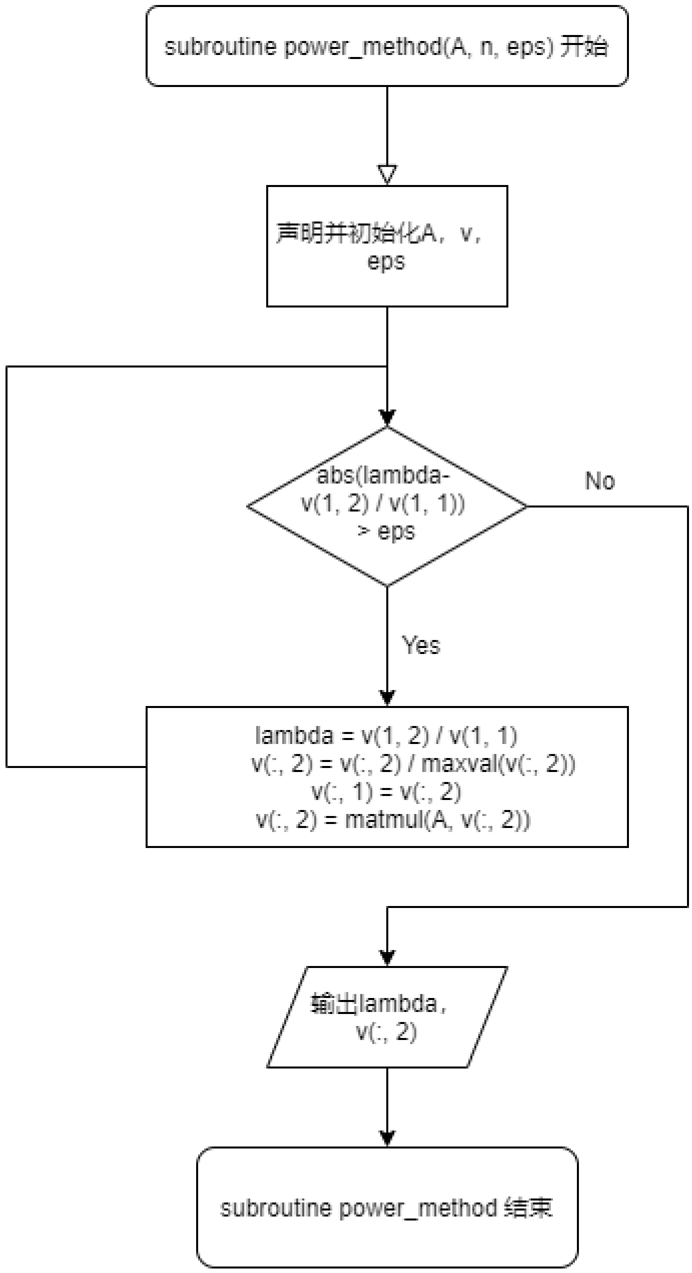


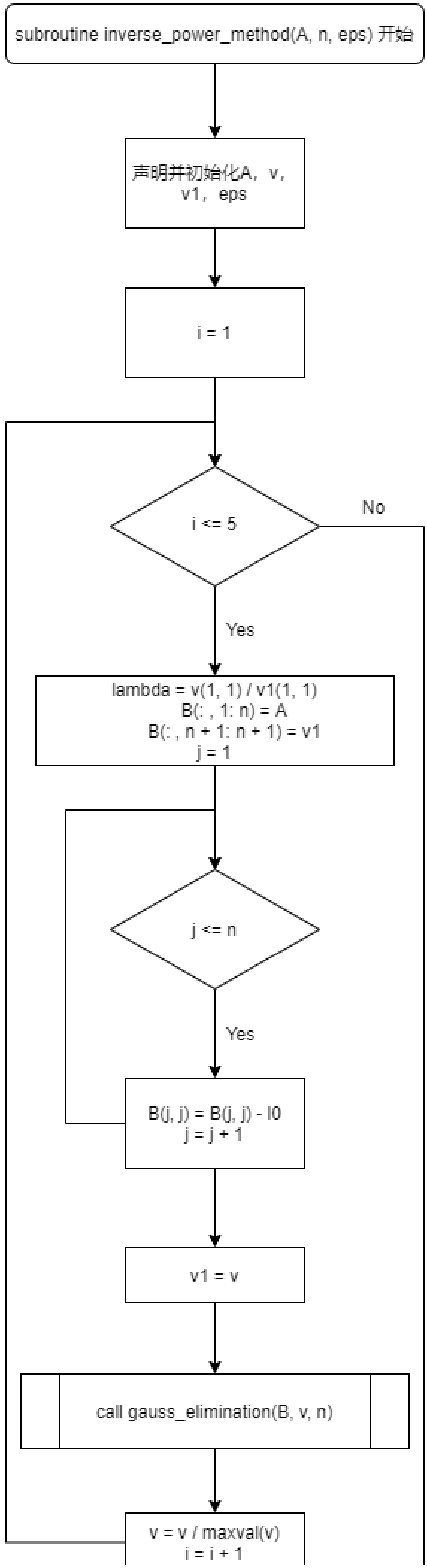
v = reshape([1., 0., 0., 0., -1.], shape(v))
call inverse_power_method(A0, v, A(2, 2), n, eps)
P(:, 2:2) = v
v = reshape([1., 0., 0., 0., 0.], shape(v))
call inverse_power_method(A0, v, A(3, 3), n, eps)
P(:, 3:3) = v
v = reshape([1., 0., 0., 0., 0.], shape(v))
call inverse_power_method(A0, v, A(4, 4), n, eps)
P(:, 4:4) = v
v = reshape([1., 0., 0., 0., -1.], shape(v))
call inverse_power_method(A0, v, A(5, 5), n, eps)
P(:, 5:5) = v
输出矩阵P
call power_method(A0, n, eps)
program jsff3结束


function S(A, n) 开始
S = 0
i = 2
i <= n
No
Yes
S = S + dot_product(A(i, 1: i - 1), A(i, 1: i - 1))
i = i + 1
function S 结束

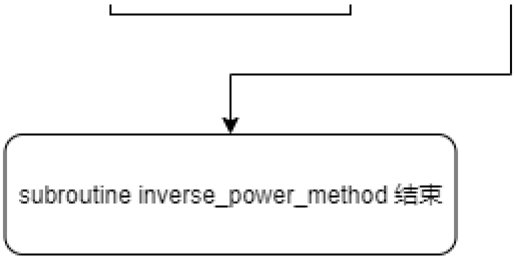

subroutine schmidt_orthogonalization(A, Q, R, n) 开始
i = 1
i <= n
No
Yes
beta(:, i) = A(:, i)
j = 1
j <= i - 1
No
Yes
beta(:, i) = beta(:, i) - dot_product(A(:, i), Q(:, j)) * Q(:, j)
j = j + 1
Q(:, i) = beta(:, i) / sqrt(dot_product(beta(:, i), beta(:, i)))
R(i, i) = sqrt(dot_product(beta(:, i), beta(:, i)))
j = i + 1
j <= n
No
Yes
R(i, j) = dot_product(A(:, j), Q(:, i))


j = j + 1
i = i + 1
subroutine schmidt_orthogonalization 结束


subroutine power_method(A, n, eps) 开始
声明并初始化A，v，eps
abs(lambda- v(1, 2) / v(1, 1)) > eps
No
Yes
lambda = v(1, 2) / v(1, 1)
v(:, 2) = v(:, 2) / maxval(v(:, 2))
v(:, 1) = v(:, 2)
v(:, 2) = matmul(A, v(:, 2))
输出lambda，v(:, 2)
subroutine power_method 结束



subroutine inverse_power_method(A, n, eps) 开始
声明并初始化A，v，v1，eps
i = 1
i <= 5
No
Yes
lambda = v(1, 1) / v1(1, 1)
B(: , 1: n) = A
B(: , n + 1: n + 1) = v1
j = 1
j <= n
Yes
B(j, j) = B(j, j) - l0
j = j + 1
v1 = v
call gauss_elimination(B, v, n)
v = v / maxval(v)
i = i + 1



二、源代码

```fortran
program jsff3
    ! homework3 of Numerical Methods
    ! arthor : zzy

    implicit none
    ! A0 : record the primary value of A
    ! A : the matrix to be iterated in QR method
    ! P : the rows of P are eigenvectors
    ! Q : a orthogonal matrix
    ! R : a upper triangular matrix
    ! E : identity matrix
    real(8), dimension(5, 5) :: A0, A, P, Q, R, E
    ! v : initial vector of inverse power method
    real(8), dimension(5, 1) :: v
    ! eps : calculation precision
    real(8), parameter :: eps = 1e-8
    ! S : quadratic sum of A(i, j), where 1 <= i <= n, j < i
    real(8) :: S
    integer(4) :: i, j, n = 5

    ! initialize A, A0, E
    A = reshape([11, -3,  -8,  1,   8, &
                 -6,  5,  12,  6, -18, &
                  4, -2,  -3, -2,   8, &
                -10,  4,  12,  3, -14, &
                 -4,  1,   4, -1,  -1] &
                 , shape(A))
    A0 = A
    do i = 1, n
        do j = 1, n
            if (i == j) then
                E(i, j) = 1.
            else
                E(i, j) = 0.
            end if
        end do
    end do

    ! QR method
    do while(sqrt(S(A, n)) > eps)
        call schmidt_orthogonalization(A, Q, R, n)
        A = matmul(R, Q)
    end do

    ! output eigenvalues
```

```
    print *, 'eigenvalues:'
    do i = 1, n
        print "(f8.5)", A(i, i)
    end do

    ! calculate eigenvectors by inverse power method
    v = reshape([1., 0., 0., 0., 0.], shape(v))
    call inverse_power_method(A0, v, A(1, 1), n, eps)
    P(:, 1:1) = v

    v = reshape([1., 0., 0., 0., -1.], shape(v))
    call inverse_power_method(A0, v, A(2, 2), n, eps)
    P(:, 2:2) = v

    v = reshape([1., 0., 0., 0., 0.], shape(v))
    call inverse_power_method(A0, v, A(3, 3), n, eps)
    P(:, 3:3) = v

    v = reshape([1., 0., 0., 0., 0.], shape(v))
    call inverse_power_method(A0, v, A(4, 4), n, eps)
    P(:, 4:4) = v

    v = reshape([1., 0., 0., 0., -1.], shape(v))
    call inverse_power_method(A0, v, A(5, 5), n, eps)
    P(:, 5:5) = v

    ! print *, 'eigenvectors:'
    ! call print_matrix(P, n, n)
    ! print *, ' '
    ! call print_matrix(matmul(A0, P), n, n)

    call power_method(A0, n, eps)

end program jsff3

function S(A, n)
    ! calculate quadratic sum of A(i, j), where 1 <= i <= n, j < i
    ! parameters : A : input matrix
    !              n : shape of A is (n, n)
    implicit none
    real(8), dimension(5, 5), intent(in out) :: A
    real(8) :: S
    integer :: i, n

    S = 0
    do i = 2, n
        S = S + dot_product(A(i, 1: i - 1), A(i, 1: i - 1))
    end do

end function S

subroutine schmidt_orthogonalization(A, Q, R, n)
    ! apply schmidt orthogonalization to A
    ! parameters: A : the matrix to be orthogonalize
    !             Q : orthogonal matrix
    !             R : upper triangular matrix
    !             n : shape of A, Q, R is (n, n)
    implicit none
```

```fortran
    real(8), dimension(5, 5), intent(in out) :: A, Q, R
    ! beta : orthogonal vectors
    real(8), dimension(5, 5) :: beta
    integer :: i, j, n

    do i = 1, n
        beta(:, i) = A(:, i)
        do j = 1, i - 1
            beta(:, i) = beta(:, i) - dot_product(A(:, i), Q(:, j)) * Q(:, j)
        end do
        Q(:, i) = beta(:, i) / sqrt(dot_product(beta(:, i), beta(:, i)))
        R(i, i) = sqrt(dot_product(beta(:, i), beta(:, i)))
        do j = i + 1, n
            R(i, j) = dot_product(A(:, j), Q(:, i))
        end do
    end do

end subroutine

subroutine power_method(A, n, eps)
    ! apply power method to calculate the largest eigenvalue and corresponding eigenvector
    ! parameters: A : the matrix to be calculated
    !             n : shape of A is (n, n)
    !             eps : precision
    implicit none
    real(8), dimension(n, n) :: A
    ! v : iteration vector
    real(8), dimension(n, 2) :: v
    ! lambda : initial eigenvalue
    real(8) :: lambda = -1e8, lam_temp, eps
    integer(4) :: n

    v = reshape([1., 2., 3., 4., 5., 1., 2., 3., 4., 5.], shape(v))

    ! v = reshape([-1.00000004e+00,  3.33333333e-01,  1.00000000e+00, -4.21368997e-08, -9.99999958e-01, &
                !  -1.00000004e+00,  3.33333333e-01,  1.00000000e+00, -4.21368997e-08, -9.99999958e-01], shape(v))

    ! v = reshape([1., 0., 0., 12.00000994, -1., 1., 0., 0., 12.00000994, -1.], shape(v))

    print *, 'v0 =', v(:, 1)
    do while(abs(lambda - lam_temp) > eps)
        lambda = lam_temp
        v(:, 2) = v(:, 2) / maxval(v(:, 2))
        v(:, 1) = v(:, 2)
        v(:, 2) = matmul(A, v(:, 2))
        lam_temp = dot_product(v(:, 2), matmul(A, v(:, 2))) / dot_product(v(:, 2), v(:, 2))
    end do
    print *, 'eigenvalue:'
    print *, lambda
    print *, 'eigenvector:'
    print *, v(:, 2)
    ! print *, matmul(A, v(:, 2))
```

```fortran
end subroutine power_method

subroutine inverse_power_method(A, v, l0, n, eps)
    ! apply inverse power method to calculate the eigenvector of given eigenvalue l0
    ! parameters: A : the matrix to be calculated
    !             v : the iteration vector
    !             l0 : the given eigenvalue
    !             n : shape of A is (n, n)
    !             eps : precision
    implicit none
    real(8), dimension(n, n), intent(in) :: A
    real(8), dimension(n, n + 1) :: B
    ! v1, v : the iteration vectors
    real(8), dimension(n, 1) :: v1, v
    ! lambda : initial eigenvalue
    real(8) :: l0, eps
    integer(4) :: n, i, j

    v1 = v

    do i = 1, 5
        B(: , 1: n) = A
        B(: , n + 1: n + 1) = v1
        do j = 1, n
            B(j, j) = B(j, j) - l0
        end do
        ! print *, lambda
        v1 = v
        call gauss_elimination(B, v, n)
        v = v / maxval(v)

    end do
    ! print *, lambda
    ! print *, v
end subroutine inverse_power_method

subroutine gauss_elimination(A, theta, n)
    ! apply gauss elimination algorithm
    ! parameters: B : agumented matrix
    !             theta : solution of linear equations
    !             n : the length of theta is (n + 1)
    ! author: zzy

    implicit none
    integer(4), intent(in) :: n
    real(8), intent(in out), dimension(n, n + 1) :: A
    real(8), intent(in out), dimension(n) :: theta
    integer(4) :: i, j, k

    ! use elementary transformation to transform B into upper triangular matrix
    do i = 1, n ! ii : rows
        do j = i + 1, n + 1 ! j : columns
            A(i, j) = A(i, j) / A(i, i)
        end do
        A(i, i) = 1
        do j = i + 1, n ! j : rows
```

```fortran
            do k = i + 1, n + 1 ! k : columns
                A(j, k) = A(j, k) - A(j, i) * A(i, k)
            end do
            A(j, i) = 0
        end do
    end do

    ! solve theta by transform B(1:n+1, 1:n+1) into diagonal matrix
    do i = n, 1, -1
        do j = i + 1, n
            A(i, n + 1) = A(i, n + 1) - theta(j) * A(i, j)
        end do
        theta(i) = A(i, n + 1);
    end do

end subroutine gauss_elimination

subroutine print_matrix(A, m, n)
    ! debug function, print a matrix
    ! parameters: A : matrix to be printed
    !              (m, n) : shape of matrix
    ! author: zzy

    implicit none
    integer(4) :: m, n, i
    real(8), dimension(m, n) :: A

    do i = 1, m
        print *, A(i, :)
    end do

end subroutine print_matrix
```

三、运行结果

编译指令（在jsff3.f90所在的目录内）：

```
gfortran jsff3.f90 -o jsff3 && ./jsff3
```

```
shenye@shenye-virtual-machine:~/FortranPrograms$ ./jsff3
 eigenvalues:
 5.00000
 5.00000
 3.00000
 1.00000
 1.00000
 eigenvectors:
  -1.0000000421368995        1.0000000000000000      -0.50000000000000033       0.59485530546623855      -0.53958201393286720
  0.33333333333333326      -0.28571431084187260       0.25000000000000006      -0.19614147909967852        3.9582013932867664E-002
   1.0000000000000000      -0.85714293252561813        1.0000000000000000      -0.79742765273311889        1.0000000000000000
  -4.2136899716304952E-008  0.14285706747438218       0.24999999999999931       -6.4308681672020068E-003  0.42083597213426494
 -0.99999995786310003       0.71428586505123581      -0.99999999999999989        1.0000000000000000       -1.4604179860671316
```

```
shenye@shenye-virtual-machine:~/FortranPrograms$ gfortran jsff3.f90 -o jsff3 && ./jsff3
 (1)
 v0 =   1.0000000000000000        2.0000000000000000        3.0000000000000000        4.0000000000000000        5.0000000000000000
 eigenvalue:
   4.9999999760906588
 eigenvector:
  -5.4166666084933279        1.6666666530516268        4.9999999702944589      -0.41666664933844821       -4.5833333098164308
shenye@shenye-virtual-machine:~/FortranPrograms$ gfortran jsff3.f90 -o jsff3 && ./jsff3
 (2)
 v0 =   1.0000000000000000        0.0000000000000000        0.0000000000000000       -2.0000000000000000       -1.0000000000000000
 eigenvalue:
   5.0000000032768046
 eigenvector:
   4.4736842139302055       -1.5789473695919893       -4.7368421087759671      -0.26315789510445786        5.0000000036217305
shenye@shenye-virtual-machine:~/FortranPrograms$ gfortran jsff3.f90 -o jsff3 && ./jsff3
 (3)
 v0 =  -1.0000000000000000       0.33333334326744080        1.0000000000000000       -4.2136900191280802E-008 -0.99999994039535522
 eigenvalue:
   4.9999999846318586
 eigenvector:
  -5.0000001474405646        1.6666666581854299        4.9999999814895020       -1.7290729592556175E-007  -4.9999998016338685
```

四、分析报告

1.问题分析

给定实方阵

$$A=\begin{bmatrix}11 & -6 & 4 & -10 & -4\\ -3 & 5 & -2 & 4 & 1\\ -8 & 12 & -3 & 12 & 4\\ 1 & 6 & -2 & 3 & -1\\ 8 & -18 & 8 & -14 & -1\end{bmatrix}$$

(1) 用施密特正交变换的 QR 法计算 A 的全部特征值和相应的特征向量，按特征值的绝对值从大到小排列.

(2) 按以下的方法取三组不同初始向量，分别用幂法求 A 的按模最大特征值.

  a) 任意非零初始向量;

  b) 与（1）求出的最大特征向量正交的初始向量;

  c) 与（1）求出的最大特征向量相近的初始向量;

2.算法细节

（1）QR法的实现

不断对矩阵A进行进行schmit正交化，将A分解为$A=QR$. 其中Q为正交阵，满足$Q^TQ=E$；R为上三角阵.

随着迭代过程的进行，A逐渐变成上三角阵. A趋近于上三角阵的程度由函数$S(A)$决定，其中S接受矩阵$A\in\mathbb{R}^{n\times n}$作为输入，以实数$S(A)\in\mathbb{R}$作为输出，即$S:\mathbb{R}^{n\times n}\to\mathbb{R}$.

函数S被定义为$S(A)=\sum_{i=2}^{n}\sum_{j=1}^{i-1}A_{ij}^2$，即矩阵A非下三角元素的平方和. S(A)越接近0，说明A越接近于上三角矩阵. 当S(A)小于预先设置的精度eps时，A被认为已经足够接近上三角矩阵，迭代停止，此时矩阵A的对角线元素即为特征值的数值解.

程序在第39行用循环实现了QR法.

（2）特征向量的求解

QR法仅能求解A的所有特征值，不能求解特征向量. 问题转化为已知特征值求特征向量，使用原点平移的反幂法求解.

反幂法能够求出矩阵A绝对值最小的特征向量，假设已知矩阵A的某一个特征值λ_i，对应的特征向量为ξ_i，则矩阵(λ_iE-A)具有特征值0，对应的特征向量为ξ_i. 所以对矩阵(λ_iE-A)使用反幂法即可得到λ_i对应的特征向量.

反幂法的迭代过程如下：

$$
\begin{cases}
v_k = A^{-1}u_{k-1} \Leftrightarrow Av_k = u_{k-1} \\
\mu_k = max(v_k) \\
u_k = \dfrac{v_k}{\mu_k}
\end{cases}
\qquad (k = 1, 2, \cdots, n)
$$

每次迭代需要用高斯消元法解线性方程组$Av_k = u_{k-1}$来得到v_k.

由于重特征值可能对应多个线性无关的特征向量，因此需要多次调用反幂法并更改初始向量u_k来求得所有特征向量.

以本题为例，特征值5为二重特征值，对应了两个线性无关的特征向量. 先带入任意的初始向量u_0，计算出一个特征向量ξ_1，然后将u_0设置为与ξ_1正交的向量，再执行一次反幂法，即可得到与ξ_1线性无关的特征向量ξ_2.

反幂法由子程序inverse_power_method实现.

（3）幂法的实现

幂法的思路与反幂法类似，通过不断迭代求出矩阵A特征值绝对值最大的特征向量.

幂法在程序中由子程序power_method实现.

选取初始向量v0，将其不断左乘矩阵A，向量v会不断接近矩阵A特征值最大的特征向量. 程序声明了一个n*2维的矩阵v，v的第一列v(:, 1)是迭代过程中的中间变量，v的第二列v(:, 2)存储最终计算出来的特征向量.

在迭代完成后，设最终得到的特征向量为v，使用公式$\lambda = v^TAv/v^Tv$来计算特征值，相当于对v的每一个元素以某种方式取平均后再计算特征值，在程序中通过如下语句实现：

```
lambda = dot_product(v(:, 2), matmul(A, v(:, 2))) / dot_product(v(:, 2), v(:, 2))
```

3.编程思路

主要函数/子程序：

function S(A, n) 计算矩阵A的非上三角元素($A_{ij}, 2 \le i \le n, 1 < j < i$)的平方和.

subroutine schmidt_orthogonalization(A, Q, R, n) 施密特正交化.

subroutine power_method(A, n, eps) 幂法.

subroutine inverse_power_method(A, v, l0, n, eps) 反幂法.

subroutine gauss_elimination(A, theta, n) 高斯消去法.

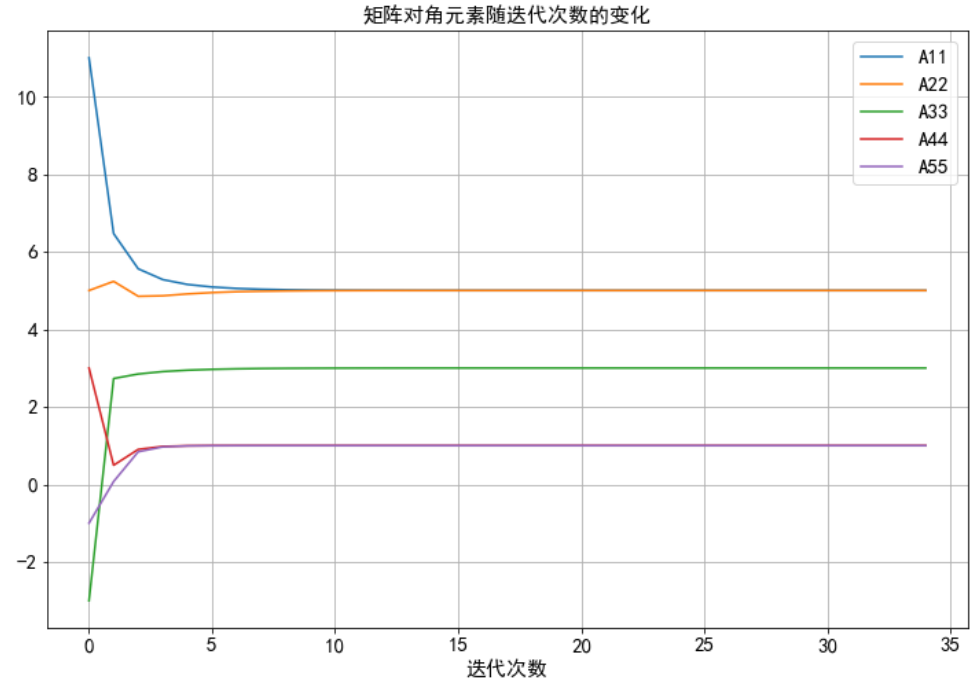
subroutine print_matrix(A, m, n) 调试函数，输出矩阵A.

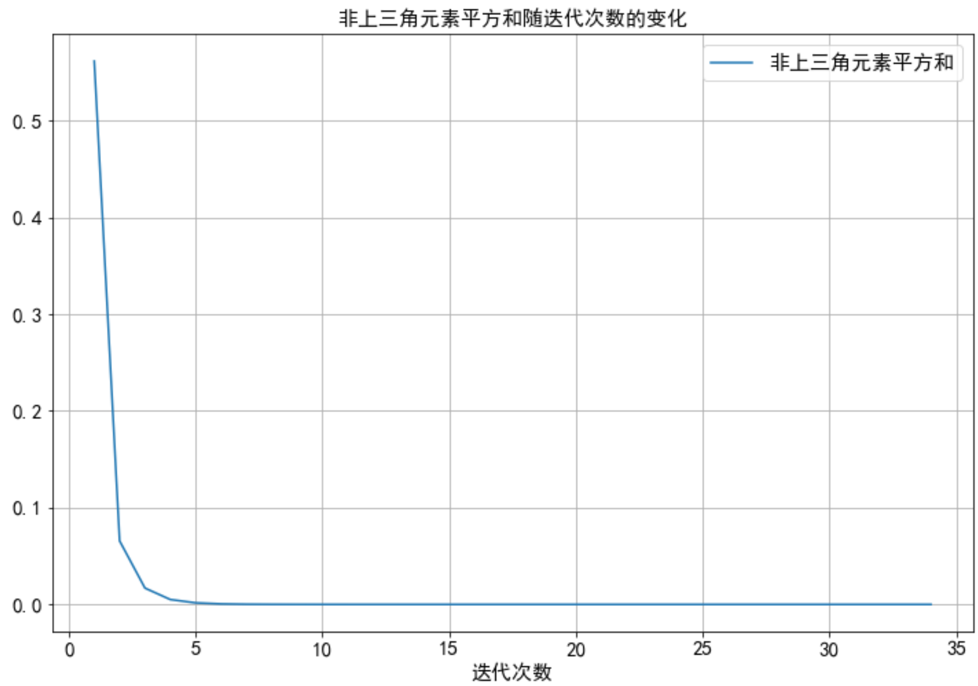
4.运行结果分析

（1）QR法的收敛速度

设定精度eps=1e-8，当S(A) < 1e-8时迭代结束.

随着迭代次数的增加，对角线元素和非上三角元素的平方和的变化如下：



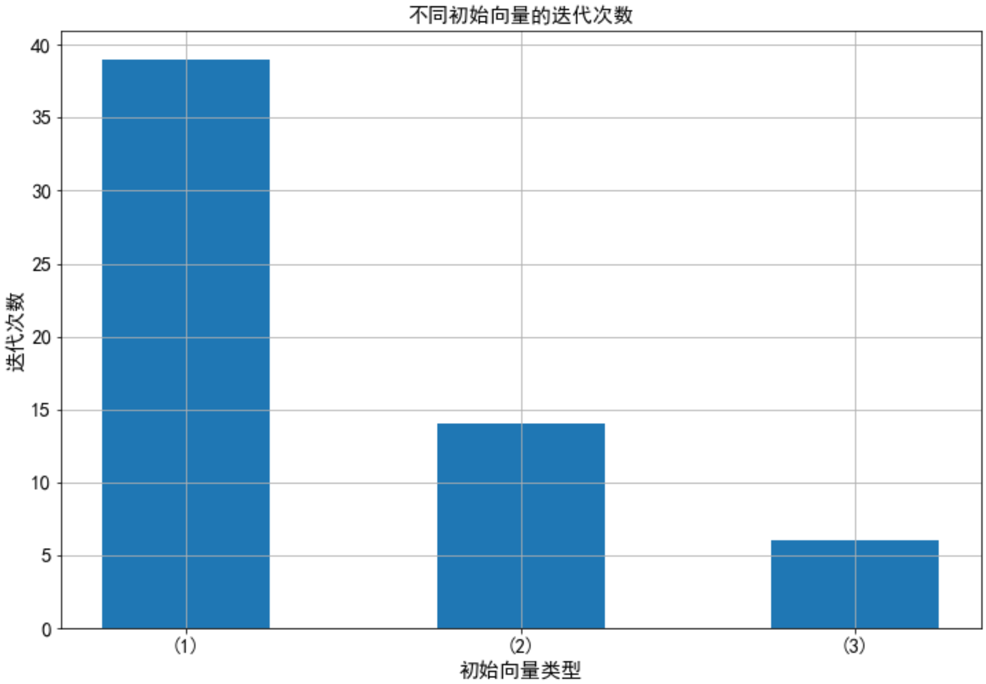


从图中可以看出QR法的收敛速度较快.

（2）（2）计算的结果与（1）对比，是否三种给定的初始值都能收敛到最大特征值？收敛速度有何差异？

设定精度eps=1e-8，当两次迭代产生的特征值之差的绝对值 < 1e-8 时，迭代停止.

三种初始值都能收敛到最大特征值（分别收敛到4.999999976，5.000000003，4.999999985），收敛速度如下图所示：



其中（1）代表任意初始向量，（2）代表与最大特征向量正交的初始向量，（3）代表与最大特征向量相近的初始向量.

由于最大特征值5对应了两个线性无关的特征向量，故选取与这两个特征向量都正交的初始向量.

从图中可以看出，收敛速度（3）>（2）>（1）.

（3）疑问

（3）接近于特征向量，而（1）为随机向量，所以（3）的收敛速度大于（1）.

对于（2），由于选取的v0与特征值5对应的所有特征向量都正交，所以在v0的线性表示中5对应特征向量的分量为0，按理说应该不能收敛到5对应的特征向量，但是为什么实际运行结果是能收敛，且速度比随机初始向量还要快?