数值实验报告
单步位移的显式 QR 算法

梁子龙

2018 年 4 月 8 日

介绍

本文按照教材第 1.1 节的内容，利用 MATLAB 实现了单步位移的显式 QR 算法，并且进行了如下四个数值实验：

1. 比较利用 Householder 变换和 Givens 旋转进行上 Hessenberg 约化所需的运行时间.
2. 对某些矩阵对比显式 QR 算法与 MATLAB 内建命令 `eig` 求得的矩阵特征值，验证算法实现的正确性.
3. 对比选取不同位移（不带位移、Rayleigh 商位移和 Wilkinson 位移）的情况下显式 QR 算法的运行时间.
4. 考察矩阵特征值分离程度对所需 QR 迭代步数的关系.

为考察算法在不同种类的矩阵上的数值性质，在实验中我们选取以下几种具有代表性的矩阵.

- 随机生成的矩阵. 如利用 MATLAB 内建命令 `rand` 或 `randn` 生成的矩阵.
- 块三对角矩阵. 这里选取的是解 Poisson 方程时利用五点差分格式得到的系数矩阵[①]，形如

$$A = \left[\begin{array}{ccc:ccc:c} 4 & -1 & & -1 & & & \\ -1 & 4 & -1 & & -1 & & \\ & -1 & 4 & & & -1 & \\ \hdashline -1 & & & 4 & -1 & & \\ & -1 & & -1 & 4 & -1 & \\ & & -1 & & -1 & 4 & \\ \hdashline & & & & & & \ddots \end{array}\right] \tag{1}$$

在实验中，这类矩阵使用一个辅助函数生成（见附录中代码片段 7）.

- 对称三对角矩阵. 这里选取的是 MATLAB 的内建命令 `wilkinson` 生成的一种 Wilkinson 矩阵[②]. 它定义为

$$W_{2n+1} = \begin{bmatrix} -n & 1 & & & & \\ 1 & -n+1 & 1 & & & \\ & 1 & -n+2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & n & 1 \end{bmatrix} \tag{2}$$

[①] 见北京大学《数值线性代数》第二版第 4.3.2 节.

[②] 最常用的 Wilkinson 矩阵是 W_{21}. 关于 Wilkinson 矩阵的更多细节，详见 `https://blogs.mathworks.com/cleve/2013/04/15/wilkinsons-matrices-2/`.

Wilkinson 矩阵有一些特殊的性质：

- 除去一个特征值外，所有的特征值都是正数．
- 正特征值成对出现，一对特征值彼此靠近但不相等．并且特征值越大，两个成对出现的特征值距离越小．

作为例子，我们在图 1 中绘出了 W_{21} 的所有特征值．

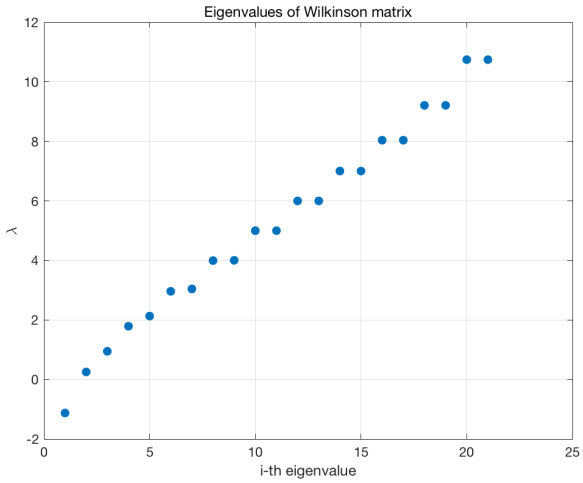


图 1: 用于实验的一种 Wilkinson 矩阵 W_{21} 的全部特征值．可以看出，它的特征值成对出现，并且特征值越大，成对的特征值越靠近．

实验一

实验项目

比较利用 Householder 变换和 Givens 旋转进行上 Hessenberg 约化所需的运行时间.

实验原理

计算酉相似的上 Hessenberg 约化可以通过 Householder 变换或 Givens 旋转来实现. 通过 Householder 变换计算上 Hessenberg 约化的算法由教材中的算法 1.1.3 给出, 该算法所需的运算量为 $10n^3/3$, 而如需累计酉矩阵 $U = H_1H_2\cdots H_{n-2}$, 所增加的运算量为 $4n^3/3$.

类似地, 利用 Givens 旋转的性质, 我们可以得到利用 Givens 旋转实现的上 Hessenberg 约化.

算法 1 (利用 Givens 旋转计算上 Hessenberg 约化) 给定一个矩阵 $A \in \mathscr{C}^{n\times n}$, 下面的算法利用 *Givens* 旋转计算一个酉矩阵 $U \in \mathscr{C}^{n\times n}$ 和一个上 *Hessenberg* 矩阵, 使得 $U^*AU = H$.

$H = A$; $U = \mathbf{eye}(n, n)$
for $j = 1 : n-2$
  for $j = n : -1 : j+2$
    if $H(i, j) = 0$
      continue
    end
    $[c, s, \eta] = \mathbf{givens}(H(i-1, j), H(i, j))$
    $G = \left[\begin{smallmatrix} c & s \\ -\bar{s} & \bar{c} \end{smallmatrix}\right]$
    $H(i-1 : i, j) = \left[\begin{smallmatrix} \eta \\ 0 \end{smallmatrix}\right]$
    $H(i-1 : i, j+1 : n) = GH(i-1 : i, j+1 : n)$
    $H(1 : n, i-1 : i) = H(1 : n, i-1 : i)G^*$
    $U(1 : n, i-1 : i) = U(1 : n, i-1 : i)G^*$
  end
end

该算法没有考察特定的矩阵结构来调整 *Givens* 旋转的次序, 是通用的上 *Hessenberg* 约化算法, 计算量为 $O(n^3)$. 但由于循环中对零元的判断, 可以猜想, 该算法在 A 具有较多零元时, 可以适当减少运算量.

实验代码

我们选取了四种矩阵分别进行实验, 计算了 1～100 阶矩阵进行上 Hessenberg 分解所需的运行时间. 为消除时间统计的误差以及随机生成矩阵的随机性, 每一阶的实验重复了 10 次. 最终, 我们绘制了约化过程的运行时间随矩阵阶数的变化曲线图, 并对 10～100 阶的结果进行了 loglog 意义下的线性拟合.

代码片段 1 下面的代码片段实现了上述实验过程. 其中的关键步骤添加了中文注释.

```
%% Experiment 1: Upper Hessenberg reduction using Householder or Givens
%
% This experiment performs a comparison between Householder transformation
% and Givens rotation on upper Hessenberg decomposition process.
```

```
%  ------------------------------------------------
%  Experiments on Matrix Computations -- Spring 2018
%  Author: Liang Zilong
%  Date:   2018-04-01
%  ------------------------------------------------

clear; close all; clc;


n_max = 100;  % 矩阵阶数为 1～n_max
repeat = 10;  % 每一阶的实验重复若干次以消除时间统计的误差
matrices = {'general', 'block-tridiagonal', 'sparse', 'tridiagonal'};


for k = 1:4  % 对四种矩阵进行实验
    times_house = zeros(n_max, 1);
    times_givens = zeros(n_max, 1);

    for n = 1:n_max
        for j = 1:repeat
            switch k
                case 1  % 元素服从正态分布的随机矩阵
                    A = randn(n, n);
                case 2  % 块三对角阵
                    A = blocktridiag(n / 3);
                case 3  % 特殊的矩阵: 仅第二个次对角线有稀疏的非零元
                    A = triu(randn(n, n));
                    A = A + diag(randn(n-1, 1).*binornd(1, 0.4, [n-1, 1]), -1);
                    A = A + diag(randn(n-2, 1).*binornd(1, 0.4, [n-2, 1]), -2);
                case 4  % Wilkinson 矩阵
                    A = wilkinson(n);
            end

            t0_householder = clock;
            hessenberg(A);
            times_house(n) = times_house(n) + etime(clock, t0_householder);

            t0_givens = clock;
            hessenberg_givens(A);
            times_givens(n) = times_givens(n) + etime(clock, t0_givens);
        end

        times_house(n) = times_house(n) / repeat;
        times_givens(n) = times_givens(n) / repeat;
    end

    % 对 10～n_max 阶的实验结果进行 loglog 意义下的线性拟合
    xx = zeros(n_max-9, 2);
    xx(:, 1) = log(10:n_max);
    xx(:, 2) = 1;
```

```
    yy_house = log(times_house(10:end));
    yy_givens = log(times_givens(10:end));
    coef_house = xx \ yy_house;
    coef_givens = xx \ yy_givens;

    % 绘图
    figure(k);
    loglog(times_house, '.-'); grid on; hold on;
    loglog(times_givens, '.-');
    loglog(exp(xx(:, 1)), exp(coef_house(2) + coef_house(1) .* xx(:, 1)));
    text(0.7*exp(xx(1, 1)), exp(coef_house(2) + coef_house(1) .* xx(1, 1)), ...
         sprintf('k_{1} = %.3f', coef_house(1)));
    loglog(exp(xx(:, 1)), exp(coef_givens(2) + coef_givens(1) .* xx(:, 1)));
    text(0.7*exp(xx(1, 1)), exp(coef_givens(2) + coef_givens(1) .* xx(1, 1)), ...
         sprintf('k_{2} = %.3f', coef_givens(1)));
    xlabel('n');
    ylabel('time (s)');
    legend('Householder', 'Givens');
    title(sprintf('Experiment : Upper Hessenberg reduction (%s)', ...
          matrices{k}));
end
```

其中，实现各项功能的函数见附录的代码片段 5（*house.m*），6（*givens.m*），8（*hessenberg.m*），9（*hessenberg_givens.m*）.

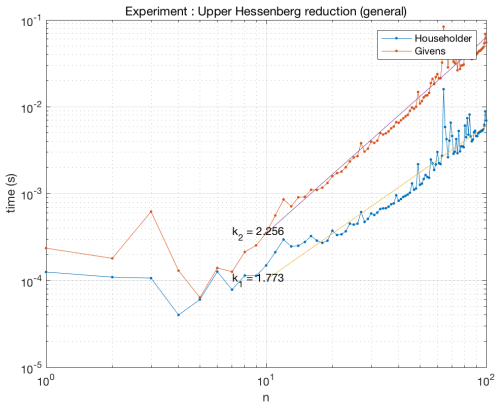
实验结果及讨论

我们首先选取了稠密的矩阵（由 `randn` 生成的矩阵）进行实验，结果如图 2a 所示（见第 6 页）. 可以看到，在 loglog 的意义下，两种上 Hessenberg 约化的增长性质均趋于线性，Givens 旋转所需的运行时间比 Householder 变换高一个数量级，两者的增长量级大约相差 $k_2/k_1 = 1.272$. 这里与教材中所述“Givens 旋转一般所需要的运算量大约是算法 1.1.3 的两倍”相符情况不佳，怀疑是由于我们的算法 1 增加了一些判断的成本和内层循环无法向量化造成的.

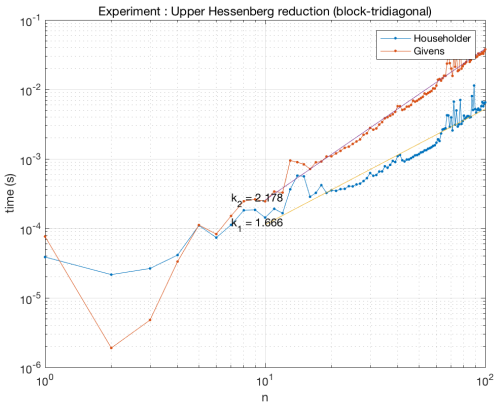
我们接下来考察了零元较多的块三对角矩阵进行实验，结果如图 2b 所示. 由于在 Givens 旋转的相似变换中会逐步破坏矩阵的稀疏结构，导致即便这种块三对角阵零元较多，Givens 旋转所需的时间仍比 Householder 变换高出微小的量级.

这种结果让我们思考：在什么情况下，我们实现的 Givens 旋转的性能可能高过 Householder 变换？我们首先选取了一种特殊的矩阵：它的上三角元素由 `randn` 随机生成，而第一、第二个次对角元添加上一些稀疏的非零元，其余的元素均为 0. 这样的矩阵在 Givens 旋转时不会大范围破坏矩阵的稀疏结构，仅需不多的旋转便可约化为一个上 Hessenberg 矩阵. 这个实验的结果如图 2c 所示. 可以看出，此时 Givens 旋转的运行时间即使在内层循环没有做到向量化的前提下快于了 Householder 变换，与我们的预期相符. 这也印证了，如果矩阵具有比较特殊的稀疏结构，利用 Givens 旋转进行上 Hessenberg 变换的速度可能更快.

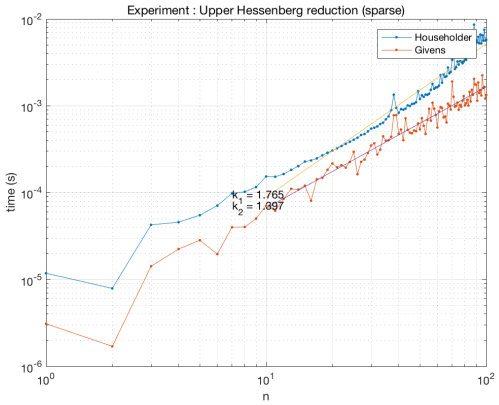
最后，我们选取了一种极端情况：当矩阵已是上 Hessenberg 形式，实验结果会如何？图 2d 展示了对 Wilkinson 矩阵做上 Hessenberg 约化的实验结果. 可以看出，当矩阵已是上 Hessenberg 形式时 Givens 旋转仅需一系列判断，而不需进行任何运算即可完成约化过程.



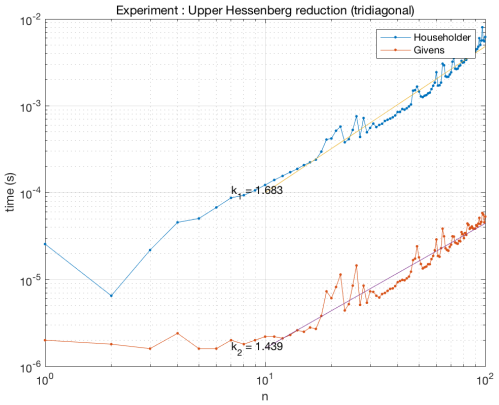
(a) **randn** 生成的随机矩阵



(b) 块三对角矩阵



(c) 更加稀疏的矩阵



(d) Wilkinson 矩阵

图 2: 对比通过 Householder 变换和 Givens 旋转进行上 Hessenberg 约化的实验结果.

实验二

实验项目

对某些矩阵对比显式 QR 算法与 MATLAB 内建命令 `eig` 求得的矩阵特征值，验证算法实现的正确性.

实验原理及代码

这是一个简单的验证实验，我们使用 MATLAB 实现了带 Wilkinson 位移的显式 QR 算法（见附录的代码片段 10，11，12），计算了 25 阶的三种矩阵的特征值，并与 MATLAB 内建命令 `eig` 进行了对比.

代码片段 2 下面的代码片段实现了上述实验过程. 其中的关键步骤添加了中文注释.

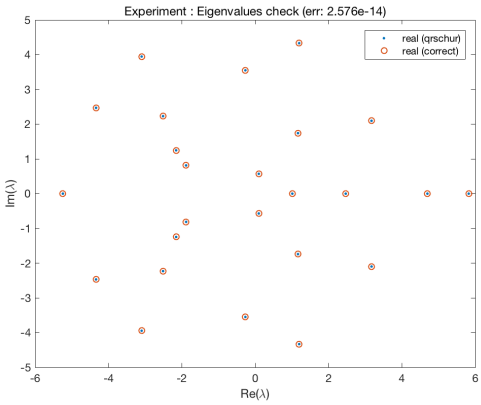
```
%% Experiment 2: Eigenvalues checking
%
% This experiment simply checks the wrote-up function 'qrschur.m' works well.
%
%  ------------------------------------------------
%  Experiments on Matrix Computations -- Spring 2018
%  Author: Liang Zilong
%  Date:   2018-04-01
%  ------------------------------------------------

clear; close all; clc;


n = 25;
matrices = {'real', 'real-symmetric', 'complex'};

for k = 1:3
    switch k
        case 1  % 元素服从正态分布的随机实矩阵
            A = randn(n);
            xlabelstr = 'Re(\lambda)';
            ylabelstr = 'Im(\lambda)';
        case 2  % Wilkinson 矩阵
            A = wilkinson(n);
            xlabelstr = 'i-th eigenvalue';
            ylabelstr = '\lambda';
        case 3  % 元素服从正态分布的随机复矩阵
            A = randn(n) + 1i*randn(n);
            xlabelstr = 'Re(\lambda)';
            ylabelstr = 'Im(\lambda)';
    end

    T = qrschur(A);
    eigs_qrschur = diag(T);
    eigs_correct = eig(A);
```

```
36      err = norm(sort(abs(eigs_qrschur)) - sort(abs(eigs_correct)), Inf);
37
38      % Judge real or complex, and sort if real
39      if isreal(eigs_correct)
40          eigs_qrschur = sort(real(eigs_qrschur));
41          eigs_correct = sort(eigs_correct);
42          xlabelstr = 'i-th eigenvalue';
43          ylabelstr = '\lambda';
44      end
45
46      figure(k);
47      plot(eigs_qrschur, '.'); hold on;
48      plot(eigs_correct, 'o');
49      xlabel(xlabelstr);
50      ylabel(ylabelstr);
51      legend(sprintf('%s (qrschur)', matrices{k}), ...
52             sprintf('%s (correct)', matrices{k}));
53      % 将最大模误差标注在图表标题栏
54      title(sprintf('Experiment : Eigenvalues check (err: %.3e)', err));
55  end
```

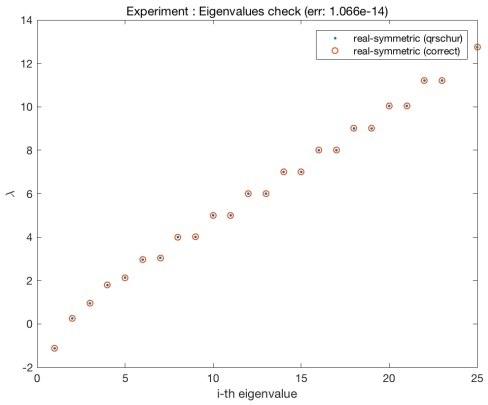
其中，实现各项功能的函数见附录的代码片段 *11*（*qriteration.m*），*12*（*qrschur.m*）.

实验结果及讨论

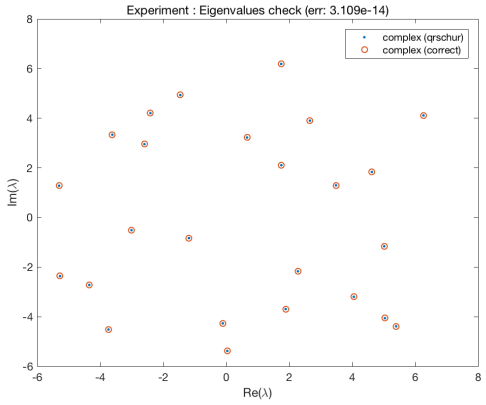
图 3（见第 9 页）展示了利用显式 QR 算法和 MATLAB 内建命令 `eig` 求矩阵特征值的实验结果，其误差最大模标注在图标标题栏中. 作为一个验证实验，该实验通过计算具有复特征值的实矩阵、特征值为实数的实对称正定矩阵、复矩阵三种矩阵的特征值，误差均在 10^{-14} 量级，基本验证了我们编写的算法代码的正确性，为我们进行后续实验提供了支持.



(a) `randn` 生成的随机实矩阵



(b) Wilkinson 矩阵



(c) `randn` 生成的随机复矩阵

图 3: 利用显式 QR 算法和 MATLAB 内建命令 `eig` 求矩阵特征值

实验三

实验项目

对比选取不同位移（不带位移、Rayleigh 商位移和 Wilkinson 位移）的情况下显式 QR 算法的运行时间.

实验原理

为了加快 QR 迭代的收敛速度，我们引进了位移. 利用 Rayleigh 商位移和 Wilkinson 位移，可以使复矩阵特征值的渐进收敛速度提高到局部二次收敛. 本实验通过对不带位移、Rayleigh 商位移和 Wilkinson 位移三种情况记录运算时间，以对比它们的收敛速度.

实验代码

我们选取了三种位移选项进行了实验，使用显式 QR 算法计算了 1～50 阶随机生成的复矩阵的特征值. 为消除时间统计的误差以及随机生成矩阵的随机性，每一阶的实验重复了 3 次. 最终，我们绘制了求 Schur 标准型所需的运行时间随矩阵阶数的变化曲线图，并对实验结果进行了 loglog 意义下的线性拟合.

代码片段 3 下面的代码片段实现了上述实验过程. 其中的关键步骤添加了中文注释.

```
%% Experiment 3: Comparison among different types of shift
%
% This experiment performs a comparison among different shifts on single-shifted
% QR algorithm. We choose non-shifted method, Rayleigh quotient shift and
% Wilkinson's shift to perform the comparison, and A is a random complex matrix.
%
% Caution: Performance of this experiment could be extremely low, thanks to the
%          non-shifted method.
%
%  ------------------------------------------------
%  Experiments on Matrix Computations -- Spring 2018
%  Author: Liang Zilong
%  Date:   2018-04-01
%  ------------------------------------------------

clear; close all; clc;


n_max = 50;  % 矩阵阶数为 1～n_max
repeat = 3;  % 每一阶的实验重复若干次以消除时间统计的误差
shifts = {'none', 'rayleigh', 'wilkinson'};
legendcmd = 'legend(';

for k = 1:3  % 对三种位移种类进行实验
    times = zeros(n_max, 1);
    for n = 1:n_max
        for j = 1:repeat
```

```
            sprintf('k = %d, n = %d, repeat = %d', k, n, j)
            A = randn(n) + 1i * randn(n);  % 选取随机生成的复矩阵

            t0 = clock;
            qrschur(A, shifts{k}, 1e-5);
            times(n) = times(n) + etime(clock, t0);
        end
        times(n) = times(n) / repeat;
    end

    % 对实验结果进行 loglog 意义下的线性拟合
    xx = zeros(n_max-1, 2);
    xx(:, 1) = log(2:n_max);
    xx(:, 2) = 1;
    yy = log(times(2:end));
    coef = xx \ yy;

    % 绘图
    loglog(2:n_max, times(2:end), ...
           '.-', 'MarkerSize', 13, 'LineWidth', 1.5); grid on; hold on;
    loglog(exp(xx(:, 1)), exp(coef(2) + coef(1) .* xx(:, 1)));
    text(0.7*exp(xx(1, 1)), exp(coef(2) + coef(1) .* xx(1, 1)), ...
         sprintf('k_{%d} = %.3f', k, coef(1)));
    legendcmd = strcat(legendcmd, sprintf('''%s'',''%s (fit)'',', ...
                                          shifts{k}, shifts{k}));
end

% 补充绘图信息
xlabel('dim');
ylabel('average time');
legendcmd = strcat(legendcmd(1:end-1), ');');
eval(legendcmd);
title('Experiment : Comparison among different types of shift');
```

其中，实现各项功能的函数见附录的代码片段 *10*（*wilkinsonshift.m*），*11*（*qriteration.m*），*12*（*qrschur.m*）.

实验结果及讨论

图 4（见第 14 页）显示了选取不同位移的情况下使用显式 QR 算法计算 Schur 标准型所需的运行时间. 可以看出，如果忽略计算 Wilkinson 位移所需要的运算量（每一次计算仅需要常数级别的运算量），对复矩阵选取 Rayleigh 商位移和 Wilkinson 位移的渐近收敛速度基本相同. 而不带位移的 QR 算法相比带位移的算法要慢，并且收敛速度不稳定. 如果对三者的实验结果进行 loglog 意义下的线性拟合，可以发现选取 Rayleigh 商位移和 Wilkinson 位移的 QR 算法的增长量级也是基本相同的，而不带位移的算法则大约是带位移增长量级的 $k_1/k_2 = 1.89$ 倍. 这也验证了，如果不带位移的 QR 算法中特征值的渐近收敛速度是线性收敛的话，加上位移可以使收敛速度提高到大约二次收敛.

实验四

实验项目

考察矩阵特征值分离程度对所需 QR 迭代步数的关系.

实验代码

我们选取了四种 25 阶矩阵进行了实验，使用带 Wilkinson 位移的显式 QR 算法按分离出的先后顺序得到了矩阵的所有特征值和分离出该特征值所需的 QR 迭代步数. 对于每一种矩阵的实验，我们绘制了特征值模长和对应迭代步数的双轴图，并计算出了平均迭代步数.

代码片段 4 下面的代码片段实现了上述实验过程. 其中的关键步骤添加了中文注释.

```
%% Experiment 4: Average iteration steps
%
% This experiment performs an observation on average iterations steps of single-
% shifted QR algorithm. Also, we choose three kind of matrices (random, block-
% tridiagonal and Wilkinson) to compare, to show the relationships between
% iterations steps and distributions of eigenvalues.
%
%  ------------------------------------------------
%  Experiments on Matrix Computations -- Spring 2018
%  Author: Liang Zilong
%  Date:   2018-04-01
%  ------------------------------------------------

clear; close all; clc;


n = 25;  % 选取 25 阶矩阵进行实验
xx = 1:n;

matrices = {'rand', 'randn', 'blocktridiag', 'wilkinson'};

for k = 1:4  % 对四种矩阵进行实验
    switch k
        case 1  % 元素服从均匀分布的矩阵
            A = rand(n, n);
        case 2  % 元素服从正态分布的矩阵
            A = randn(n, n);
        case 3  % 块三对角矩阵
            A = blocktridiag(n / 3);
        case 4  % Wilkinson 矩阵
            A = wilkinson(n);
    end

    [eigs_qrschur, iters] = qrschur_observation(A);
    steps = iters - [iters(2:end); 0];  % 得到分离各个特征值的迭代步数
    avg = mean(steps);  % 得到分离一个特征值所需的平均迭代步数
```

```
37
38      % 绘图
39      figure(k);
40      [AX, H1, H2] = plotyy(xx, steps(end:-1:1), ...
41                            xx, abs(eigs_qrschur(end:-1:1))); grid on; hold on;
42      plot([0, n], [avg, avg], 'b-', 'LineWidth', 1.5);
43      text(0.5, 1.05*avg, sprintf('AVG: %.3f', avg), 'FontWeight', 'bold');
44      set(H1, 'Marker', '.', 'MarkerSize', 10, 'LineStyle', '--', 'Color', 'b');
45      set(H2, 'Marker', '.', 'MarkerSize', 15, 'LineStyle', 'none');
46      set(get(AX(1), 'ylabel'), 'string', 'iteration steps');
47      set(get(AX(2), 'ylabel'), 'string', 'abs(\lambda)');
48      xlabel('i-th eigenvalue');
49      title(sprintf('Experiment: Average iteration steps (%s)', ...
50            matrices{k}));
51  end
```

其中，实现各项功能的函数见附录的代码片段 *13*（*qrschur_observation.m*）.

实验结果及讨论

图 5（见第 14 页）显示了不同矩阵使用带 Wilkinson 位移的 QR 算法所求得的特征值与迭代步数的关系. 每一张实验结果图中，橙色轴与橙色散点表示第 i 个被分离出的特征值的模长，蓝色轴与蓝色虚线代表该特征值被分离出来所需要的迭代步数，深蓝色实线表示 QR 算法的平均迭代步数.

需要注意的是，我们考察待分离的右下角的特征值

$$H = \begin{bmatrix} \ddots & \ddots & \\ \ddots & \delta_{11} & \delta_{12} \\ & \varepsilon & \delta_{22} \end{bmatrix} \quad \to \quad \tilde{H} = \begin{bmatrix} \ddots & \ddots & \\ \ddots & \lambda_1 & \tilde{\delta}_{12} \\ & 0 & \lambda_2 \end{bmatrix}, \tag{3}$$

那么 ε 置零所需的迭代步数，反映的是 λ_1 与 λ_2 的分离程度. 所以，我们在观察实验结果图时，第 i 个特征值被分离出来所需的迭代步数反映的是与第 i 和 $i+1$ 个特征值分离程度的关系.

我们首先对元素服从均匀分布的随机矩阵进行了实验，结果如图 5a 所示. 在计算 25 阶矩阵的 Schur 标准型过程中，QR 迭代所需的平均步数为 3.400. 如图所示，前 24 个特征值模长极靠近，QR 算法得到一个特征值基本上需要 3～5 步迭代. 但最后一个特征值模长与前面的特征值相差极大，反映到迭代步数上可以看到，第 24 个特征值仅需 0～1 步即可分离出来.

接下来，我们考察了元素服从正态分布的随机矩阵，结果如图 5b 所示. 该矩阵的特征值模长集中在 $[1, 5]$ 之间，因此迭代所需步数与特征值分布的相关性在图中显示不明显. 但值得注意的是两种随机生成矩阵进行 QR 迭代所需的平均步数均为 3.4 左右，与之后块三对角矩阵与 Wilkinson 矩阵所需较少的平均迭代步数（分别为 2.040 与 1.880）形成对比，说明矩阵的特殊结构导致的特征值分布特性可能使 QR 算法更容易得到 Schur 标准型的计算结果.

对于块三对角矩阵（见图 5c），它在 QR 算法中依序分离出的特征值有较为特殊的结构：某些模长靠近的特征值会被连续分离出，而某些则会出现跳跃. 我们考察第 8、10、12、15、18 个特征

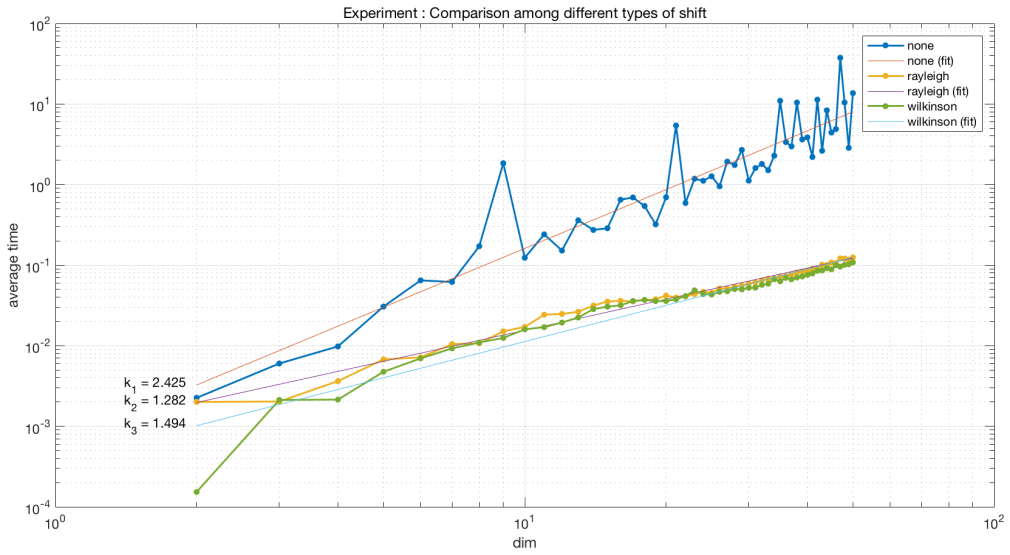
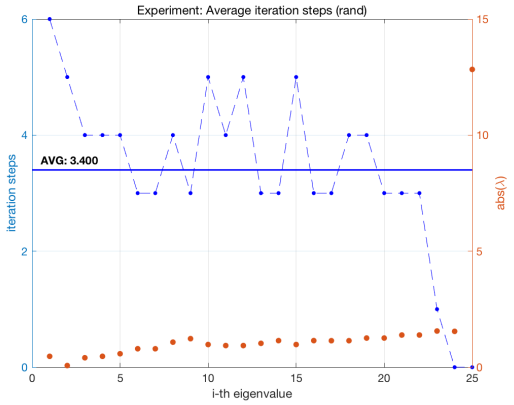
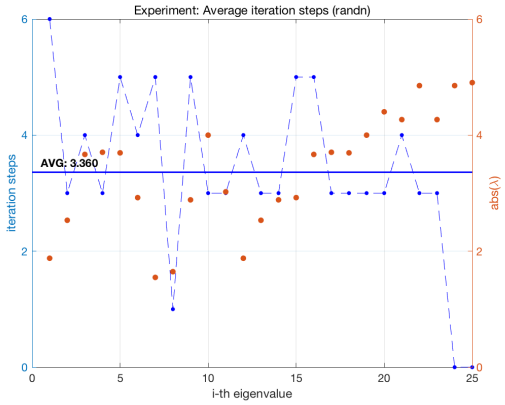


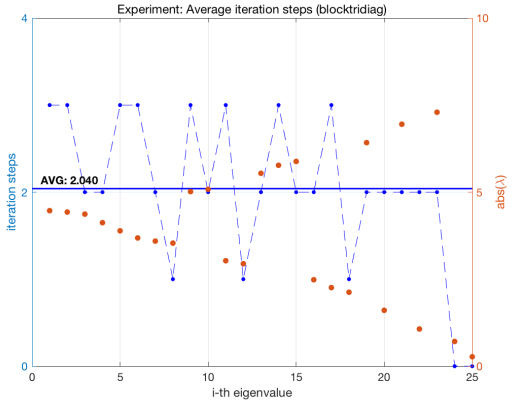
图 4: 选取不同位移的情况下使用显式 QR 算法计算 Schur 标准型所需的运行时间.



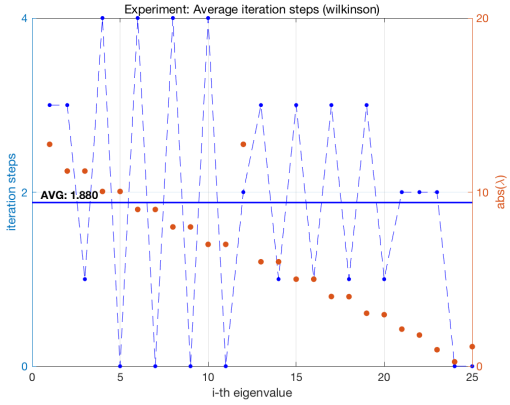
(a) `rand` 生成的随机矩阵



(b) `randn` 生成的随机矩阵



(c) 块三对角矩阵



(d) Wilkinson 矩阵

图 5: 不同矩阵使用带 Wilkinson 位移的 QR 算法所求得的特征值与迭代步数的关系.

值，它们均与后一个特征值呈现跳跃的特性．而对应的，分离出这些特征值的步数从一般的 3 步减少到了 2 步或 1 步，基本验证了特征值模长越分离收敛速度越快的理论．

值得注意的是，这种现象在 Wilkinson 这种具有特殊性质的矩阵上显示得尤为明显．图 5d 显示了在矩阵 W_{25} 上的实验结果．在图中可以明显地发现，将成对出现、相互靠近的特征值分离开基本需要 3～4 步 QR 迭代，而将不同对的特征值分离仅需要 0～1 步 QR 迭代．这说明了特征值模长越分离、收敛速度越快的性质，也正是由于矩阵特征值的这种特殊的分布特性，使得块三对角和 Wilkinson 矩阵的 QR 算法所需得平均迭代步数得以减少．

附录：实验中用到的 MATLAB 函数

代码片段 5 (计算 Householder 变换) 下面的函数按照教材中算法 *1.1.1* 实现了计算 *Householder* 变换的功能.

```
function [w, gamma] = house(a)
% HOUSE    Compute Householder transformation
%
% A Householder matrix: H = I-ww' denotes a unitray transformation. This
% function computes the vector 'w' and a scalar 'gamma', given a vector 'a',
% so that Ha = gamma*e_1. Note that norm(w, 2) = sqrt(2).
%
% argin:
%   a - A column vector to transform (real or complex)
%
% argout:
%   w     - The vector in the Householder transformation H = I-ww'
%   gamma - The scalar in the formula Ha = gamma*e_1
%
% ------------------------------------------------
% Experiments on Matrix Computations -- Spring 2018
% Author: Liang Zilong
% Date:   2018-03-31
% ------------------------------------------------

w = a;
gamma = norm(a, 2);

if gamma == 0
    w(1) = sqrt(2);
    return
end

if w(1) ~= 0
    delta = conj(w(1)) / abs(w(1));
else
    delta = 1;
end

w = (delta / gamma) * w;
w(1) = w(1) + 1;

w = w / sqrt(w(1));
gamma = - conj(delta) * gamma;
```

代码片段 6 (计算 Givens 旋转) 下面的函数按照教材中算法 *1.1.2* 实现了计算 *Givens* 旋转的功能.

```
function [c, s, eta] = givens(alpha, beta)
```

```
% GIVENS    Compute Givens rotation
%
% The Givens rotation is an unitray transformation, from [a b]' to [eta 0]'.
% This function computes the sine, cosine value and the scalar eta.
%
% argin:
%   alpha, beta - Two scalars in the vector to rotate (real or complex)
%
% argout:
%   c, s - The cosine and sine values (scalars) in the Givens rotation
%   eta  - The eta value (scalar) in the rotated result
%
% ------------------------------------------------
% Experiments on Matrix Computations -- Spring 2018
% Author: Liang Zilong
% Date:   2018-03-31
% ------------------------------------------------

if beta == 0
    c = 1;
    s = 0;
    eta = alpha;
    return
end

if alpha == 0
    c = 0;
    s = 1;
    eta = beta;
    return
end

abs_alpha = abs(alpha);  % Prevent to compute for three times.
mu = alpha / abs_alpha;
tau = abs_alpha + abs(beta);
delta = tau * sqrt(abs(alpha/tau)^2 + abs(beta/tau)^2);

c = abs_alpha / delta;
s = mu * conj(beta) / delta;
eta = delta * mu;
```

代码片段 7 (生成块三对角矩阵) 下面的辅助函数可以生成由解 *Poisson* 方程的五点差分格式得到的系数矩阵，它是一个块三对角、对角占优的矩阵.

```
function A = blocktridiag(n)
% ------------------------------------------------
% Helper function: Generate 3*n-D block-tridiagonal
%                  matrix of five-point finite
%                  difference method.
```

```
% ------------------------------------------------

A = - diag(ones(3*n-3,1), 3) - diag(ones(3*n-3,1), -3);
D = [4, -1, 0; -1, 4, -1; 0, -1, 4];
for k = 0:n-1
    A(3*k+1:3*k+3, 3*k+1:3*k+3) = D;
end
```

代码片段 8 (利用 Householder 变换计算上 Hessenberg 约化) 下面的函数基本按照教材中算法 *1.1.3* 实现了利用 *Householder* 变换计算上 *Hessenberg* 约化的功能. 这里, 酉矩阵仅在需要时进行计算.

```
function [H, U] = hessenberg(A)
% HESSENBERG    Compute upper Hessenberg reduction
%
% This function compues the upper Hessenberg reduction of a matrix A, so that
% U'AU = H. Note that A should be a square matrix.
%
% argin:
%   A - A matrix to perform upper Hessenberg reduction (real or complex)
%
% argout:
%   H - The upper Hessenberg matrix which is unitray similar with A
%   U - the unitray matrix transforming A to H, so that U'AU = H
%
% ------------------------------------------------
% Experiments on Matrix Computations -- Spring 2018
% Author: Liang Zilong
% Date:   2018-03-31
% ------------------------------------------------

n = size(A);
if n(1) ~= n(2)
    error('Fatal Error: The input is not a square matrix!');
end
n = n(1);

H = A;
if nargout == 2
    U = zeros(n, n);
end


for k = 1:n-2
    w = house(H(k+1:n, k));
    v = w' * H(k+1:n, k:n);
    H(k+1:n, k:n) = H(k+1:n, k:n) - w * v;
    v = H(1:n, k+1:n) * w;
    H(1:n, k+1:n) = H(1:n, k+1:n) - v * w';
```

```
    H(k+2:n, k) = 0;
    if nargout == 2
        U(k+1:n,k) = w;
    end
end

if nargout == 2
    U(:, n) = zeros(n, 1); U(n, n) = 1;
    U(:, n-1) = zeros(n, 1); U(n-1, n-1) = 1;
    for k = n-2:-1:1
        w = U(k+1:n, k);
        v = w' * U(k+1:n, k+1:n);
        U(k+1:n, k+1:n) = U(k+1:n, k+1:n) - w * v;
        U(:, k) = zeros(n, 1); U(k, k) = 1;
    end
end
```

代码片段 9 (利用 Givens 旋转计算上 Hessenberg 约化) 下面的函数按照算法 *1* 实现了利用 *Givens* 旋转计算上 *Hessenberg* 约化的功能. 这里, 酉矩阵仅在需要时进行计算.

```
function [H, U] = hessenberg_givens(A)
% HESSENBERG_GIVENS   Compute upper Hessenberg reduction using Givens rotations
%
% This function compues the upper Hessenberg reduction of a matrix A using
% Givens rotations, so that U'AU = H. Note that A should be a square matrix.
%
% argin:
%   A - A matrix to perform upper Hessenberg reduction (real or complex)
%
% argout:
%   H - The upper Hessenberg matrix which is unitray similar with A
%   U - the unitray matrix transforming A to H, so that U'AU = H
%
% ------------------------------------------------
% Experiments on Matrix Computations -- Spring 2018
% Author: Liang Zilong
% Date:   2018-04-01
% ------------------------------------------------

n = size(A);
if n(1) ~= n(2)
    error('Fatal Error: The input is not a sqaure matrix!');
end
n = n(1);

H = A;
if nargout == 2
    U = eye(n, n);
end
```

```
30
31 for j = 1:n-2
32     for i = n:-1:j+2
33         if H(i, j) == 0 || abs(H(i, j)) < 2 * eps
34             continue
35         end
36         [c, s, eta] = givens(H(i-1, j), H(i, j));
37         G = [c, s; -conj(s), conj(c)];
38         H(i-1:i, j) = [eta; 0];
39         H(i-1:i, j+1:n) = G * H(i-1:i, j+1:n);
40         H(:, i-1:i) = H(:, i-1:i) * G';
41         if nargout == 2
42             U(:, i-1:i) = U(:, i-1:i) * G';
43         end
44     end
45 end
```

代码片段 10 (计算 Wilkinson 位移) 下面的函数按照教材中算法 *1.1.4* 实现了计算 *Wilkinson* 位移的功能.

```
1  function mu = wilkinsonshift(alpha, beta, gamma, delta)
2  % WILKINSONSHIFT    Compute Wilkinson's shift
3  %
4  % For a 2x2 matrix [alpha, beta; gamma, delta], Wilkinson's shift means the
5  % eigenvalue of this matrix which is closer to 'delta'.
6  %
7  % argin:
8  %   alpha, beta, gamma, delta - Elements of the 2x2 matrix to compute the shift
9  %
10 % argout:
11 %   mu - Wilkinson's shift of this matrix
12 %
13 % ------------------------------------------------
14 % Experiments on Matrix Computations -- Spring 2018
15 % Author: Liang Zilong
16 % Date:   2018-03-31
17 % ------------------------------------------------
18
19 mu = delta;
20 s = sum(abs([alpha, beta, gamma, delta]));
21
22 if s == 0
23     return
24 end
25
26 q = (beta / s) * (gamma / s);
27
28 if q ~= 0
29     p = 0.5 * (alpha / s - delta / s);
```

```
    r = sqrt(p^2 + q);
    if real(p)*real(r) + imag(p)*imag(r) < 0
        r = -r;
    end
    mu = mu - s*(q/(p+r));
end
```

代码片段 11 (带位移的显式 QR 迭代) 下面的函数基本按照教材中算法 *1.1.5* 进行一次带位移的显式 *QR* 迭代. 这里, 可以选取不带位移、*Rayleigh* 商位移和 *Wilkinson* 位移三种选项; 积累的 *Givens* 旋转直接传出以供 *QR* 算法使用, 而不计算出酉矩阵.

```
function [H, G] = qriteration(H, shift)
% QRITERATION    Perform a step of QR iteration with shift
%
% This version of QR itertaion performs on an upper Hessenberg matrix 'H' with
% single-shifted algorithm.
%
% argin:
%   H     - An upper Hessenberg matrix to iterate
%   shift - A string of shift type (options: 'none', 'rayleigh' or 'wilkinson';
%           default:'wilkinson')
%
% argout:
%   H - Iteration result of H
%   G - The unitray matrices of the iteration H = P'HP, where
%       P = prod[G(:, :, k)]
%
% ------------------------------------------------
% Experiments on Matrix Computations -- Spring 2018
% Author: Liang Zilong
% Date:   2018-03-31
% ------------------------------------------------

if nargin == 1
    shift = 'wilkinson';
end

n = length(H);

switch shift
    case 'none'
        mu = 0;
    case 'rayleigh'
        mu = H(n, n);
    case 'wilkinson'
        mu = wilkinsonshift(H(n-1, n-1), H(n-1, n), H(n, n-1), H(n, n));
end

H(1, 1) = H(1, 1) - mu;
```

```
39
40 c = zeros(n-1, 1);
41 s = zeros(n-1, 1);
42 G = zeros(2, 2, n-1);
43
44 for k = 1:n-1
45     [c(k), s(k), eta] = givens(H(k, k), H(k+1, k));
46     G(:, :, k) = [c(k), s(k); -conj(s(k)), conj(c(k))];
47     H(k+1, k+1) = H(k+1, k+1) - mu;
48     H(k:k+1, k) = [eta; 0];
49     H(k:k+1, k+1:n) = G(:, :, k) * H(k:k+1, k+1:n);
50 end
51 for k = 1:n-1
52     H(1:k+1, k:k+1) = H(1:k+1, k:k+1) * G(:, :, k)';
53     H(k, k) = H(k, k) + mu;
54 end
55 H(n, n) = H(n, n) + mu;
```

代码片段 12 (带位移的显式 QR 算法) 下面的函数基本按照教材中算法 *1.1.6* 实现了实用的单步位移 QR 算法. 这里, 酉矩阵仅在需要时进行计算, 并且为了避免 $O(n^3)$ 量级的矩阵乘法, 计算时利用 QR 迭代得到的 *Givens* 旋转进行逐步计算.

```
1  function [H, Q] = qrschur(A, shift, tol)
2  % QRSCHUR    Compute Schur decomposition of a square matrix
3  %
4  % Given a square matrix A, this function computes its Schur decomposition using
5  % single-shifted QR algorithm. Also, this function can computes all
6  % eigenvectors of the matrix.
7  %
8  % argin:
9  %   A     - A square matrix
10 %   shift - A string of shift type (options: 'none', 'rayleigh' or 'wilkinson';
11 %           default:'wilkinson')
12 %   tol   - Tolerance of the precision (default: eps)
13 %
14 % argout:
15 %   T - The upper triangular matrix of Schur decomposition Q'AQ = T
16 %   Q - The unitray matrix of Schur decomposition Q'AQ = T
17 %
18 % ------------------------------------------------
19 % Experiments on Matrix Computations -- Spring 2018
20 % Author: Liang Zilong
21 % Date:   2018-03-31
22 % ------------------------------------------------
23
24 if nargin == 1
25     shift = 'wilkinson';
26     tol = eps;
27 elseif nargin == 2
```

```
28      tol = eps;
29  end
30
31  n = size(A);
32  if n(1) ~= n(2)
33      error('Fatal Error: The input is not a sqaure matrix!');
34  end
35  n = n(1);
36  if n == 1
37      H = A;
38      Q = 1;
39      return
40  end
41
42  % Step 1: Upper Hessenberg reduction
43  [H, U] = hessenberg(A);
44  if nargout == 2
45      Q = U;
46  end
47
48  m = 0;
49  while true
50      % Step 2: Judge convergence of an eigenvalue
51      for k = 1:n-1
52          if abs(H(k+1, k)) <= tol * (abs(H(k, k)) + abs(H(k+1, k+1)))
53              H(k+1, k) = 0;
54          end
55      end
56      for m = m:n-2
57          if H(n-m, n-m-1) ~=0
58              break
59          end
60      end
61      if m == n-2 && H(n-m, n-m-1) == 0  % Final condition
62          break
63      end
64      l = 0;
65      for k = n-m-2:-1:1
66          if H(k+1, k) == 0
67              l = k;
68              break
69          end
70      end
71
72      % Step 3: QR iteration
73      [H22, G] = qriteration(H(l+1:n-m, l+1:n-m), shift);
74      H(l+1:n-m, l+1:n-m) = H22;
75      for k = l+1:n-m-1
76          H(1:l, k:k+1) = H(1:l, k:k+1) * G(:, :, k-l)';
77          H(k:k+1, n-m+1:n) = G(:, :, k-l) * H(k:k+1, n-m+1:n);
```

```
        if nargout == 2
            Q(:, k:k+1) = Q(:, k:k+1) * G(:, :, k-l)';
        end
    end
end
```

代码片段 13 (可观察到迭代步数的 QR 算法) 下面的辅助函数是代码片段 *12* 的修改，使得它可以返回每一个特征值，以及分离出该特征值所需的迭代步数.

```
function [eigs_qrschur, iters] = qrschur_observation(A)
% ------------------------------------------------
% Helper function: Single-shifted QR algorithm
%                  with observation of convergense
%                  properties.
%                  (Adapted from 'qrschur.m')
% ------------------------------------------------

n = size(A);
if n(1) ~= n(2)
    error('Fatal Error: The input is not a sqaure matrix!');
end
n = n(1);

eigs_qrschur = zeros(n, 1);
iters = zeros(n, 1);

% Step 1: Upper Hessenberg reduction
H = hessenberg(A);

l = 0;
m = 0;
m_next = 0;
iter = 0;
while true
    % Step 2: Judge convergence of an eigenvalue
    for k = 1:n-1
        if abs(H(k+1, k)) <= eps * (abs(H(k, k)) + abs(H(k+1, k+1)))
            H(k+1, k) = 0;
        end
    end
    for m_next = m:n-2
        if H(n-m_next, n-m_next-1) ~=0
            break
        end
    end
    if m_next == n-2 && H(n-m, n-m-1) == 0
        m_next = n;
    end
    % 接下来的两行将新得到的特征值及收敛步数对应地存储起来
```

```
    eigs_qrschur(n-m_next+1:n-m) = diag(H(n-m_next+1:n-m, n-m_next+1:n-m));
    iters(n-m_next+1:n-m) = iter;
    if m_next == n  % Final condition
        break
    end
    m = m_next;

    % Step 3: QR iteration
    [H22, G] = qriteration(H(l+1:n-m, l+1:n-m));
    H(l+1:n-m, l+1:n-m) = H22;
    for k = l+1:n-m-1
        H(1:l, k:k+1) = H(1:l, k:k+1) * G(:, :, k-l)';
        H(k:k+1, n-m+1:n) = G(:, :, k-l) * H(k:k+1, n-m+1:n);
    end
    iter = iter + 1;
end
```