# 作业一

姓名 郑龙韬 学号 **PB18061352** 日期 **2021/5/2**

第一题 求解离散化线性系统 $Ax = b$。

**(a)** 分别使用 Jacobi 和 Gauss-Siedel 方法求解，误差大小和迭代次数的关系如图 1 所示。本题中，迭代次数 $N = 500$，由于题 (b) 的代码基于题 (a)，故本题代码一同展示在题 (b) 中。

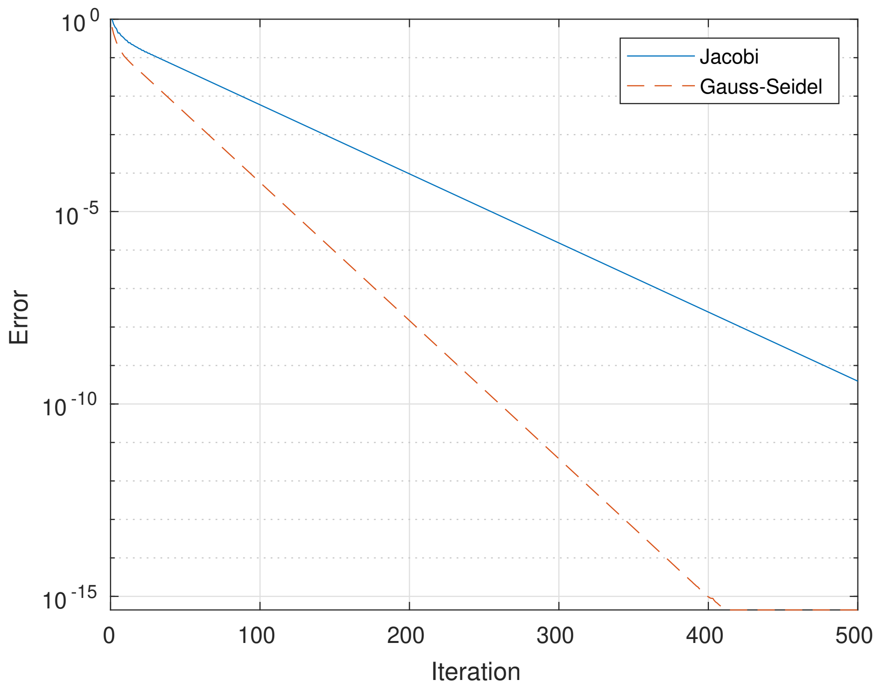

图 1: Jacobi 和 Gauss-Siedel 方法误差大小和迭代次数的关系

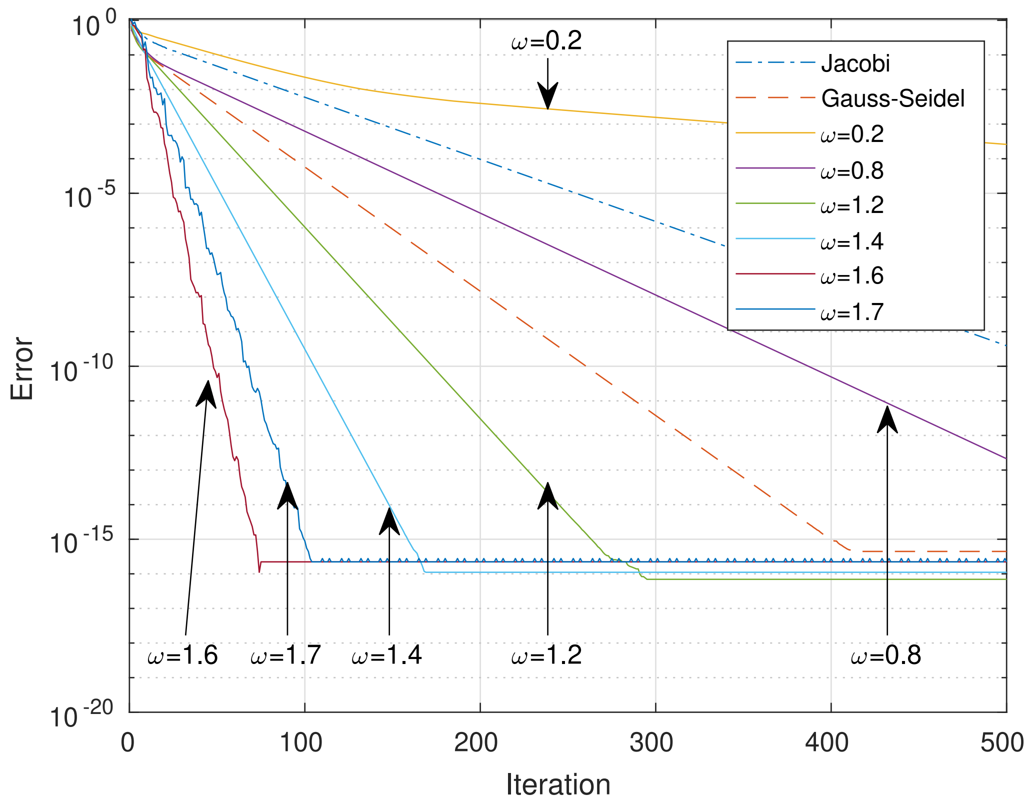

图 2: 不同的松弛因子 $\omega$ 的收敛曲线

**(b)** $\omega = 1.6$ 时，对应 $10^{-15}$ 的误差目标收敛速度最快，如图 2 所示。

以下为题 (a) 与题 (b) 的代码，按顺序包含了输入、求解、绘图代码以及 Jacobi 方法、Gauss-Seidel 方法、松弛迭代方法的 3 个函数。

```
clear , clc
SIZE = 10;   % input A, b
A = 2 * eye(SIZE);
for i = 1:SIZE
    if i ~= 1 && i ~= SIZE
        A(i, i - 1) = -1;
        A(i, i + 1) = -1;
    end
    if i == 1
        A(i, i + 1) = -1;
    end
    if i == SIZE
        A(i, i - 1) = -1;
    end
end
b = [2 -2 2 -1 0 0 1 -2 2 -2]';
N = 500;
global exact  % exact solution
exact = [1 0 1 0 0 0 0 -1 0 -1]';
[jacobi_solution, jacobi_error] = jacobi(A, b, N);
[gs_solution, gs_error] = gauss_seidel(A, b, N);

% plot Jacobi and Gauss-Seidel semilogy fig
x_range = 1:N;
semilogy(x_range,jacobi_error,'-.',x_range,gs_error,'--')
xlabel('Iteration')
ylabel('Error')
hold on

% plot SOR semilogy fig
omegas = [0.2, 0.8, 1.2, 1.4, 1.6, 1.7];
for i = 1:size(omegas, 2)
    [sor_solution, sor_error] = SOR(A, b, N, omegas(i));
    semilogy(x_range, sor_error)
    legend_str{i} = ['\omega=' num2str(omegas(i))];
end
legend(['Jacobi', 'Gauss-Seidel', legend_str])
annotation('textarrow',[0.5,0.5],[0.88,0.82], ...,
    'String','\omega=0.2')
annotation('textarrow',[0.8,0.8],[0.2,0.47], ...,
    'String','\omega=0.8')
annotation('textarrow',[0.27,0.27],[0.2,0.38], ...,
    'String','\omega=1.7')
annotation('textarrow',[0.36,0.36],[0.2,0.35], ...,
    'String','\omega=1.4')
annotation('textarrow',[0.18,0.2],[0.2,0.5], ...,
    'String','\omega=1.6')
annotation('textarrow',[0.5,0.5],[0.2,0.38], ...,
    'String','\omega=1.2')
grid on

% Jacobi method
function [solution, error] = jacobi(A, b, N)
    global exact
    error = zeros(N, 1);  % for logging error
    SIZE = size(b, 1);  % size of input matrix
    % compute matrice needed
    D = diag(diag(A));  % diagonal of A
    inversed_D = zeros(SIZE, SIZE);
    for i = 1:SIZE
        inversed_D(i, i) = 1 / D(i, i);  % D^{-1}
    end
    R = eye(SIZE) - inversed_D * A;
    g = inversed_D * b;
    solution = zeros(SIZE, 1);  % logging solution
    for iter = 1:N  % iteration
        solution = R * solution + g;
        % compute error
        e = norm(solution - exact, inf);
        error(iter, 1) = e;
    end
end

% Gauss-Seidel method
function [solution, error] = gauss_seidel(A, b, N)
    global exact
    error = zeros(N, 1);  % for logging error
    SIZE = size(b, 1);  % size of input matrix
    % compute matrice needed
    D = diag(diag(A));  % diagonal of A
    L = tril(A) - D;  % low triangle
    U = triu(A) - D;  % upper triangle
    inversed_DplusL = (D + L) \ eye(SIZE);
    S = -inversed_DplusL * U;  % iteration matrix
    f = inversed_DplusL * b;
    solution = zeros(SIZE, 1);  % logging solution
    for iter = 1:N  % iteration
        solution = S * solution + f;
        % compute error
        e = norm(solution - exact, inf);
        error(iter, 1) = e;
    end
end

% SOR method
function [solution, error] = SOR(A, b, N, omega)
    global exact
    error = zeros(N, 1);  % for logging error
    SIZE = size(b, 1);  % size of input matrix
    % compute matrice needed
    D = diag(diag(A));  % diagonal of A
    L = tril(A) - D;  % low triangle
    U = triu(A) - D;  % upper triangle
    inversed_D = D \ eye(SIZE);
    % iteration matrix for SOR
    inversed_DplusL = (eye(SIZE) + omega ...,
        * inversed_D * L) \ eye(SIZE);
    S = inversed_DplusL * ((1 - omega) * ...,
        eye(SIZE) - omega * inversed_D * U);
    f = omega * inversed_DplusL * inversed_D * b;
    solution = zeros(SIZE, 1);  % logging solution
    for iter = 1:N  % iteration
        solution = S * solution + f;
        % compute error
        e = norm(solution - exact, inf);
        error(iter, 1) = e;
    end
end
```

**(c)** 本题优化结果如表 1 所示，首先运行一次代码来忽略 Matlab 的 overhead，再重新运行计算 100 次总时间，左列为原始 3 种方法到达较为精确解 (tolerance 设置为 $10^{-15}$) 分别的总运行时间，右列为优化后总运行时间。

| Method | Elapsed Time (s) | |
|---|---|---|
| | Before Optimization | After Optimization |
| Jacobi | 0.085931 | 0.079422 |
| Gauss-Seidel | 0.052604 | 0.037022 |
| SOR | 0.013375 | 0.007728 |

表 1: 三种方法改进前后耗时对比

为了避免 Matlab 可能已有的矩阵计算优化对实验结果的影响，本题首先对前两题提供的代码迭代部分从 matrix-wise 改为课本上的使用两个 for 循环的 item-wise 算法，在对矩阵 $A$ 每行遍历时遍历所有列作为 baseline。以下为优化前的代码，在内层循环考虑每一列所有元素。

```
% Original Jacobi method
function [x2, error] = jacobi(A, b, N, tolerance)
    global exact
    error = zeros(N, 1);
    SIZE = size(b, 1);
    x1 = zeros(SIZE, 1);
    x2 = ones(SIZE, 1);
    for iter = 1:N  % iteration
        x1 = x2;
        for i = 1:SIZE
            s = 0;
            for j = 1:SIZE  % this loop can be optimized
                s = s + A(i, j) * x1(j);
            end
            x2(i) = (b(i) - s + A(i, i) * x1(i)) / A(i, i);
        end
        e = norm(x2 - exact, inf);  % compute error
        error(iter, 1) = e;
        if e < tolerance
            break
        end
    end
end
% Original Gauss-Seidel method
function [x2, error] = gauss_seidel(A, b, N, tolerance)
    global exact
    error = zeros(N, 1);
    SIZE = size(b, 1);
    x1 = zeros(SIZE, 1);
    x2 = ones(SIZE, 1);
    for iter = 1:N  % iteration
        x1 = x2;
        for i = 1:SIZE
            s = 0;
            for j = 1:SIZE  % this loop can be optimized
                s = s + A(i, j) * x2(j);
            end
            x2(i) = (b(i) - s + A(i, i) * x2(i)) / A(i, i);
        end
        e = norm(x2 - exact, inf);  % compute error
        error(iter, 1) = e;
        if e < tolerance
            break
        end
    end
end
% Original SOR method
function [x2, error] = SOR(A, b, N, omega, tolerance)
    global exact
    error = zeros(N, 1);
    SIZE = size(b, 1);
    x1 = zeros(SIZE, 1);
    x2 = ones(SIZE, 1);
    for iter = 1:N  % iteration
        x1 = x2;
        for i = 1:SIZE
            s = 0;
            for j = 1:SIZE  % this loop can be optimized
                if i ~= j
                    s = s + A(i, j) * x2(j);
                end
            end
            x2(i) = (omega * (b(i) - s) + (1 - omega) ...,
                * A(i, i) * x2(i)) / A(i, i);
        end
        e = norm(x2 - exact, inf);  % compute error
        error(iter, 1) = e;
        if e < tolerance
            break
        end
    end
end
```

以下为优化后的代码，题目要求省略和零元素相关的运算，观察得到，和零元素相关的运算发生在内层 for 循环 (对矩阵 $A$ 每个行向量的每个元素进行遍历时)，如果对矩阵 $A$ 的每行只需考虑 2 或 3 个不为零的元素 (当为首行与末行时只有 2 个元素)，只考虑列 index 下界 low 与列 index 上界 high 内的计算，相比于优化前考虑每一列所有元素，就可以使程序得到加速。

```
% Optimized Jacobi method
function [x2, error] = sparse_jacobi(A, b, N, tolerance)
    global exact
    error = zeros(N, 1);
    SIZE = size(b, 1);
    x1 = zeros(SIZE, 1);
    x2 = ones(SIZE, 1);
    for iter = 1:N  % iteration
        x1 = x2;
        for i = 1:SIZE
            s = 0;
            % optimization start here
            if i ~= 1 && i ~= SIZE  % only non-zero items
```

```
                low = i - 1;
                high = i + 1;
            else
                if i == 1  % consider 1st row
                    low = i;
                    high = i + 1;
                end
                if i == SIZE  % last row
                    low = i - 1;
                    high = i;
                end
            end
            for j = low:high  % main optimization
                s = s + A(i, j) * x1(j);
            end
            x2(i) = (b(i) - s + A(i, i) * x1(i)) / A(i, i);
        end
        e = norm(x2 - exact, inf);  % compute error
        error(iter, 1) = e;
        if e < tolerance
            break
        end
    end
end
% Optimized Gauss-Seidel method
function [x2, error] = sparse_gauss_seidel(A, b, ...,
    N, tolerance)
    global exact
    error = zeros(N, 1);
    SIZE = size(b, 1);
    x1 = zeros(SIZE, 1);
    x2 = ones(SIZE, 1);
    for iter = 1:N  % iteration
        x1 = x2;
        for i = 1:SIZE
```

```
            s = 0;
            % optimization start here
            if i ~= 1 && i ~= SIZE
                low = i - 1;
                high = i + 1;
            else
                if i == 1
                    low = i;
                    high = i + 1;
                end
                if i == SIZE
                    low = i - 1;
                    high = i;
                end
            end
            for j = low:high  % main optimization
                s = s + A(i, j) * x2(j);
            end
            x2(i) = (b(i) - s + A(i, i) * x2(i)) / A(i, i);
        end
        e = norm(x2 - exact, inf);  % compute error
        error(iter, 1) = e;
        if e < tolerance
            break
        end
    end
end
% Optimized SOR method
function [x2, error] = sparse_SOR(A, b, N, ...,
    omega, tolerance)
    global exact
    error = zeros(N, 1);
    SIZE = size(b, 1);
    x1 = zeros(SIZE, 1);
    x2 = ones(SIZE, 1);
    for iter = 1:N  % iteration
        x1 = x2;
        for i = 1:SIZE
            s = 0;
            % optimization start here
            if i ~= 1 && i ~= SIZE
                low = i - 1;
                high = i + 1;
            else
                if i == 1
                    low = i;
                    high = i + 1;
                end
                if i == SIZE
                    low = i - 1;
                    high = i;
                end
            end
            for j = low:high  % main optimization
                if i ~= j
                    s = s + A(i, j) * x2(j);
                end
            end
            x2(i) = (omega * (b(i) - s) + ...,
                (1 - omega) * A(i, i) * x2(i)) / A(i, i);
        end
        e = norm(x2 - exact, inf);  % compute error
        error(iter, 1) = e;
        if e < tolerance
            break
        end
    end
end
```

以下为计算耗时部分的代码，此处省略了前题中已展示的输入 $A, b$ 等部分的代码。

```
tolerance = 1e-15;
```

```
N = 2000;
omega = 1.6;
TEST_NUM = 100;
fprintf('Orgiginal Jacobi')
tic
for i = 1:TEST_NUM
    [jacobi_solution, jacobi_error] = jacobi(A, b, ...,
        N, tolerance);
end
toc
fprintf('Orgiginal Gauss-Seidel')
tic
for i = 1:TEST_NUM
    [gs_solution, gs_error] = gauss_seidel(A, b, N, ...,
        tolerance);
end
toc
fprintf('Orgiginal SOR')
tic
for i = 1:TEST_NUM
    [sor_solution, sor_error] = SOR(A, b, N, ...,
        omega, tolerance);
end
toc
fprintf('Optimized Jacobi')
tic
for i = 1:TEST_NUM
    [jacobi_solution, jacobi_error] = sparse_jacobi(A, ...,
        b, N, tolerance);
end
toc
fprintf('Optimized Gauss-Seidel')
tic
for i = 1:TEST_NUM
    [gs_solution, gs_error] = sparse_gauss_seidel(A, ...,
        b, N, tolerance);
end
toc
fprintf('Optimized SOR')
tic
for i = 1:TEST_NUM
    [sor_solution, sor_error] = sparse_SOR(A, b, N, ...,
        omega, tolerance);
end
toc
```

程序输出如下。

```
Orgiginal Jacobi历时 0.085931 秒。
Orgiginal Gauss-Seidel历时 0.052604 秒。
Orgiginal SOR历时 0.013375 秒。
Optimized Jacobi历时 0.079422 秒。
Optimized Gauss-Seidel历时 0.037022 秒。
Optimized SOR历时 0.007728 秒。
```

第二题 本题利用求解方程

$$x^3 - 3x^2 + 2 = 0 \tag{1}$$

的根来深入关于 Newton 方法的收敛速度的讨论。容易验证 (2) 的三个根分别位于 $[-3, 0]$、$[0, 2]$、$[2, 4]$ 三个区间内。我们依从左向右的顺序分别称这三个根为 $x_l$、$x_m$、$x_r$。

**(a)** 对三个区间分别选取迭代初始点 $-2.5, 0.5, 2.5$，用 Newton 法分别求解这三个根，每一步迭代的新的近似值与大概的收敛阶数展示于以下方框，即程序的输出结果，其中 iter. 为迭代次数，x 与 order 分别表示每一步迭代的新的近似值，收敛阶数的估计从第三次迭代开始。

```
initial: x = -2.500000
iter.   x         order
------------------------------
1    -2.500000
2    -1.540741
3    -1.004316   1.613361
4    -0.779063   1.857559
```

```
5    -0.733851    1.981932
6    -0.732054    1.999626
7    -0.732051    2.000000
------------------------------
True solution: x = -0.732051

initial: x = 0.500000
iter.   x        order
------------------------------
1    0.500000
2    1.111111
3    0.999074     3.002595
4    1.000000     3.000000
------------------------------
True solution: x = 1.000000

initial: x = 2.500000
iter.   x        order
------------------------------
1    2.500000
2    2.800000
3    2.735714     1.971720
4    2.732062     1.999148
5    2.732051     1.999999
------------------------------
True solution: x = 2.732051
```

Newton 法求解题中方程的代码如下。

```
clear, clc
syms x  % input f(x), compute diff
f(x) = x^3 - 3*x^2 + 2;
f_prime(x) = diff(f(x));
g(x) = x - f(x) / f_prime(x);
% initial settings
x_0 = [-2.5, 0.5, 2.5];
tolerance = 1e-15;
```

```
N = 300;
% compute 3 roots
x_l = newton(x_0(1), g, tolerance, N);
x_m = newton(x_0(2), g, tolerance, N);
x_r = newton(x_0(3), g, tolerance, N);
% compute orders of convergence
order_l = newton_with_order(x_0(1), x_l, g, tolerance, N);
order_m = newton_with_order(x_0(2), x_m, g, tolerance, N);
order_r = newton_with_order(x_0(3), x_r, g, tolerance, N);

% Newton method
function root = newton(x_0, g, tolerance, N)
    for iter = 1:N  % iteration
        x_1 = vpa(g(x_0));
        if abs(x_1 - x_0) < tolerance
            root = vpa(x_1);  % return solution
            break
        end
        x_0 = x_1;
    end
end

% Newton method with estimating order of convergence
function orders = newton_with_order(x_0, ...,
    x_exact, g, tolerance, N)
    fprintf('initial: x = %f\n', x_0);  % output format
    fprintf('iter.\tx\t\torder\n');
    dashString = repmat('-', 1, 30);
    fprintf('%s\n', dashString);
    x_1 = vpa(g(x_0));
    e_kp1 = abs(x_1 - x_exact);  % error_{k+1}
    e_k = abs(x_1 - x_exact);  % error_{k}
    e_km1 = abs(x_1 - x_exact);  % error_{k-1}
    for iter = 1:N  % iteration
        if abs(x_1 - x_0) < tolerance
```

```
            root = vpa(x_1);  % return solution
            fprintf('%s\n', dashString);
            fprintf('True solution: x = %f\n\n', root)
            break
        end
        if iter >= 3  % start estimating order ...,
            at 3rd iteration
            orders(iter, 1) = converge_order(e_kp1, ...,
                e_k, e_km1);
            fprintf('%d\t%f\t%f\n', iter, x_0, ...,
                orders(iter, 1));
        else
            fprintf('%d\t%f\n', iter, x_0);
        end
        x_0 = x_1;  % update solution and errors
        e_km1 = e_k;
        e_k = e_kp1;
        x_1 = vpa(g(x_0));
        e_kp1 = abs(x_1 - x_exact);
    end
end

% order of convergence, provided 3 successive errors
function order = converge_order(e_kp1, e_k, e_km1)
    order = log(abs(e_kp1 / e_k)) / log(abs(e_k / e_km1));
end
```

**(b)** 每一步迭代大致的收敛阶数已展示于题 (a)，此处估计收敛阶数 (order of convergence) 的方法为：记第 $n$ 次迭代与精确解的绝对误差为 $e_n = |x_n - x^*|$，由收敛速度 (rate of convergence)

$$\mu = \lim_{n\to\infty} \frac{e_{n+1}}{e_n^q} = \lim_{n\to\infty} \frac{e_n}{e_{n-1}^q} \tag{2}$$

右侧等号两边同时取对数，则可估计如下

$$\log e_{n+1} - \log e_n = q(\log e_n - \log e_{n-1}) \tag{3}$$

其中 $q$ 为收敛阶数

$$q = \frac{\log e_{n+1} - \log e_n}{\log e_n - \log e_{n-1}} = \frac{\log |\frac{e_{n+1}}{e_n}|}{\log |\frac{e_n}{e_{n-1}}|} \tag{4}$$

就可以估计收敛阶数。

**(c)** 在求解根 $x_m$ 时观察到了比二阶收敛更快的现象。原因解释如下：

记 $\varphi(x) = x - \frac{f(x)}{f'(x)}$，分析 Newton 迭代法收敛的阶如下：

$$\begin{aligned} x_{n+1} - x^* &= \varphi(x_n) - \varphi(x^*) \\ &= \varphi(x^*) + (x_n - x^*)\varphi'(x^*) + \frac{(x_n - x^*)^2}{2!}\varphi''(x^*) + \frac{(x_n - x^*)^3}{3!}\varphi'''(\xi) - \varphi(x^*) \\ &= (x_n - x^*)\varphi'(x^*) + \frac{(x_n - x^*)^2}{2!}\varphi''(x^*) + \frac{(x_n - x^*)^3}{3!}\varphi'''(\xi) \end{aligned} \tag{5}$$

在根 $x_m = 1.0$ 附近，$f(x_m) = 0$，$f'(x_m) = -3$，$f''(x_m) = 0$，$\varphi(x_m) = 1$，而且有

$$\varphi'(x_m) = 1 - \frac{(f'(x_m))^2 - f(x_m)f''(x_m)}{(f'(x_m))^2} = 0 \tag{6}$$

$$\varphi''(x_m) = \frac{(f'(x_m))^2(2f''(x_m) - f'(x_m)f''(x_m) - f(x_m)f^{(3)}(x_m)) - 0}{(f'(x_m))^4} = 0 \tag{7}$$

所以

$$x_{n+1} - x_m = \frac{(x_n - x_m)^3}{3!}\varphi'''(\xi) \tag{8}$$

$$\lim_{n\to\infty} \frac{|x_{n+1} - x_m|}{|x_n - x_m|^3} = \lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^3} = \varphi'''(x_m) \tag{9}$$

因此在根 $x_m$ 附近收敛阶数为 3 阶。

## 第三题 幂法求解特征值问题

**(a)** 以下伪代码呈现的算法能够求解

(i) 存在一个绝对值意义下的最大特征值

(ii) 存在最大的两个大小相同但符号相反特征值

两种情况。

使用幂法，判定在规范运算中迭代序列的几种情况：$\{q^{(k)}\}$ 收敛或 $\{q^{(2k+1)}\}$ 与 $\{q^{(2k)}\}$ 分别收敛于互为反号的向量，则为情况 (i)，在伪代码中的具体位置为代码的前两个 if 语句块；$\{q^{(2k+1)}\}$ 与 $\{q^{(2k)}\}$ 分别收敛于两个不同的向量，则为情况 (ii)，具体位置为代码的第三个 if 语句块。

**Algorithm 1** Power-Method$(A, m, \epsilon)$

```
1: $q_{old} = (1, 1, .., 1)^T$    ▷ initialization
2: $\overline{q}_{old} = q_{old}/\|q_{old}\|_\infty$
3: $\overline{q}_{old\_gap} = \overline{q}_{old}$
4: $\overline{q}_{new\_gap} = \overline{q}_{old}$
5: for $k = 1 : m$ do
6:     $q_{new} = A * \overline{q}_{old}$
7:     $\lambda = \|q_{new}\|_\infty$    ▷ eigenvalue
8:     $\overline{q}_{new} = q_{new}/\lambda$    ▷ eigenvector
9:     if $\|\overline{q}_{old} - \overline{q}_{new}\|_\infty < \epsilon$ then    ▷ dominant eigenvalue is positive
10:        return $\lambda, \overline{q}_{new}$
11:    end if
12:    if $\|\overline{q}_{old} + \overline{q}_{new}\|_\infty < \epsilon$ then    ▷ dominant eigenvalue is negative
13:        return $-\lambda, \overline{q}_{new}$
14:    end if
15:    if $k > 3$ then    ▷ judge whether two opposite dominant eigenvalues exist
16:        if $\|\overline{q}_{old\_gap} - \overline{q}_{old}\|_\infty < \epsilon$ and $\|\overline{q}_{new\_gap} - \overline{q}_{new}\|_\infty < \epsilon$ then
17:            $q_{old} = q_{new}$
18:            $q_{new} = A * q_{new}$    ▷ consider $A^2$, and yield $\lambda^2$
19:            $\lambda_1 = \sqrt{q_{new}(1)/\overline{q}_{old}(1)}$
20:            $\lambda_2 = -\lambda_1$    ▷ eigenvalues
21:            $\overline{q}_{new1} = q_{new} + \lambda_1 q_{old}$
22:            $\overline{q}_{new2} = q_{new} + \lambda_2 q_{old}$    ▷ eigenvectors
23:            return $\lambda_1, \lambda_2, \overline{q}_{new1}, \overline{q}_{new2}$
24:        end if
25:    end if
26:    $\overline{q}_{old\_gap} = \overline{q}_{new\_gap}$    ▷ update, the sequence order is: $\overline{q}_{old\_gap}, \overline{q}_{new\_gap}, \overline{q}_{old}, \overline{q}_{new}$
27:    $\overline{q}_{new\_gap} = \overline{q}_{old}$
28:    $\overline{q}_{old} = \overline{q}_{new}$
29: end for
```

**(b)** 本题 (b)(c) 中，tolerance 均设置为 $10^{-15}$，迭代次数 $N = 500$。计算结果如下，矩阵 $A$ 的模最大的特征值与特征向量如下框最后部分。通过在代码中添加输出语句 (贴出的代码省略了输出语句)，前半部分展示了迭代过程中的特征值 (为了节约篇幅，对输出进行了重新排版，省略了中间重复的 8.0000，下同)。

```
lambda = 1316
lambda = 7.4757
lambda = 7.0053
lambda = 7.3720
lambda = 7.6926
lambda = 7.8670
lambda = 7.9460
lambda = 7.9788
lambda = 7.9918
lambda = 7.9969
lambda = 7.9988
lambda = 7.9996
lambda = 7.9998
lambda = 7.9999
lambda = 8.0000
...
lambda = 8.0000
positive dominant eigenvalue:
lambda =
    8.0000
q_new_bar =
   -0.3103
    1.0000
   -0.7931
    0.1379
```

矩阵 $-A$ 的模最大的特征值与特征向量如下框最后部分，程序对负特征值也有效 (由于 $-A$ 迭代过程输出的 $\lambda$ 与 $A$ 迭代过程相同，为节约篇幅在此省略)。

```
negative dominant eigenvalue
lambda =
   -8.0000
```

```
q_new_bar =
    0.3103
   -1.0000
    0.7931
   -0.1379
```

题 (a) 中算法的具体代码实现如下。

```
clear, clc
% A = [-148 -105 -83 -67; 488 343 269 216;
        -382 -268 -210 -170; 50 38 32 29];
% A = -A; % switch input for different questions
A = [222 580 584 786; -82 -211 -208 -288;
        37 98 101 132; -30 -82 -88 -109];
tolerance = 1e-15;
N = 1000;
power_m(A, N, tolerance);

function power_m(A, N, tolerance) % power method
  q_old = ones(size(A, 1), 1);
  q_old_bar = q_old / norm(q_old, inf);
  q_old_bar_gap = q_old_bar;
  q_new_bar_gap = q_old_bar;
  for iter = 1:N
    q_new = A * q_old_bar;
    lambda = norm(q_new, inf); % eigenvalue
    q_new_bar = q_new / lambda; % eigenvector

    % only one dominant eigenvalue
    if norm(q_old_bar - q_new_bar, inf) < tolerance
      fprintf('positive dominant eigenvalue:');
      lambda
      q_new_bar
      break
    end
    if norm(q_old_bar + q_new_bar, inf) < tolerance
      fprintf('negative dominant eigenvalue');
      lambda = -lambda
      q_new_bar
      break
    end

    % two opposite dominant eigenvalues
    if iter > 3
      if (norm(q_old_bar_gap - q_old_bar, inf) < ...,
        tolerance) && (norm(q_new_bar_gap - ...,
            q_new_bar, inf) < tolerance)
        fprintf('two opposite dominant eigenvalues');
        % consider A^2, yield \lambda^2
        q_old = q_new;
        q_new = A * q_new;
        lambda1 = sqrt(q_new(1) / q_old_bar(1))
        lambda2 = -lambda1
        q_new_bar1 = q_new + lambda1 * q_old;
        q_new_bar2 = q_new + lambda2 * q_old;
        q_new_bar1 = q_new_bar1 / norm(q_new_bar1, inf)
        q_new_bar2 = q_new_bar2 / norm(q_new_bar2, inf)
        break
      end
    end
    % update recent 4 q-vectors
    q_old_bar_gap = q_new_bar_gap;
    q_new_bar_gap = q_old_bar;
    q_old_bar = q_new_bar;
  end
end
```

**(c)** 同样使用题 (b) 实现的代码，计算得到本题中矩阵的模最大的特征值和特征向量如下，存在两个大小相同但符号相反的最大特征值，程序能够分别输出特征值与对应的特征向量 (由于 $\lambda_2 = -\lambda_1$，此处仅输出迭代过程中的 $\lambda_1$)。

```
lambda1 = 4.8359
lambda1 = 5.1199
lambda1 = 4.9706
```

```
lambda1 = 5.0190
lambda1 = 4.9952
lambda1 = 5.0031
lambda1 = 4.9992
lambda1 = 5.0005
lambda1 = 4.9999
lambda1 = 5.0001
lambda1 = 5.0000
...
lambda1 = 5.0000
two opposite dominant eigenvalues
lambda1 =
    5.0000
lambda2 =
   -5.0000
q_new_bar1 =
    1.0000
   -0.5000
    0.1250
   -0.0000
q_new_bar2 =
    1.0000
   -0.3333
    0.1667
   -0.1667
```

**(d)** 注意，由于复向量的无穷范数为一个实数，故在复数域中使用无穷范数进行规范化无法收敛，而 max 操作按顺序比较幅度与相位，返回一个复数，命令行中测试如下，故该题使用 max 进行规范化。

```
>>> norm([2+i,1+2i],inf)
ans = 2.2361
>>> max([2+i,1+2i])
ans = 1.0000 + 2.0000i
```

对一个 $100 \times 100$ 的随机矩阵，编程求解离 $0.8-0.6i$ 最近的特征值和特征向量，通过反幂法结合位移实现，代码如下。

```
clear, clc
rng(2);
A = rand(100, 100);
target = 0.8 - 0.6i;
tolerance = 1e-13;
N = 1000;
inv_shift_power_m(A, N, target, tolerance)

function inv_shift_power_m(A, N, p, tolerance)
    A = A - p * eye(size(A, 1));  % shift
    q_old = ones(size(A, 1), 1);
    q_old_bar = q_old / max(q_old);
    q_old_bar_gap = q_old_bar;
    q_new_bar_gap = q_old_bar;
    for iter = 1:N
        q_new = A \ q_old_bar;  % inverse power method
        mu = max(q_new);
        q_new_bar = q_new / mu;
        % only one dominant eigenvalue
        if norm(q_old_bar - q_new_bar, inf) < tolerance
            lambda = p + 1 / mu
            q_new_bar
            break
        end
        if norm(q_old_bar + q_new_bar, inf) < tolerance
            lambda = -lambda
            q_new_bar
            break
        end
        % two opposite dominant eigenvalues
        if iter > 3
            if (norm(q_old_bar_gap - q_old_bar, inf) ...,
                < tolerance) && (norm(q_new_bar_gap ...,
                    - q_new_bar, inf) < tolerance)
                q_old = q_new;
                q_new = A \ q_new;
                mu = sqrt(q_new(1) / q_old_bar(1));
                lambda1 = p + 1 / mu
                lambda2 = -lambda1
                q_new_bar1 = q_new + lambda1 * q_old;
                q_new_bar2 = q_new + lambda2 * q_old;
                q_new_bar1 = q_new_bar1 / max(q_new_bar1)
                q_new_bar2 = q_new_bar2 / max(q_new_bar2)
                break
            end
        end
        % update recent 4 q-vectors
        q_old_bar_gap = q_new_bar_gap;
        q_new_bar_gap = q_old_bar;
        q_old_bar = q_new_bar;
    end
end
```

程序输出的迭代过程中的特征值、离 0.8−0.6$i$ 最近的特征值和对应的特征向量展示如下 (特征向量过于冗长，只取头尾 5 个元素进行展示)。

```
lambda = 2.8656 - 0.7462i
lambda = 0.6834 - 0.4931i
lambda = 0.8534 - 0.6635i
lambda = 0.8536 - 0.6644i
lambda = 0.8547 - 0.6618i
lambda = 0.8545 - 0.6622i
lambda = 0.8545 - 0.6621i
...
lambda = 0.8545 - 0.6621i


lambda =
   0.8545 - 0.6621i
q_new_bar =
   0.1969 + 0.1324i
  -0.7869 + 0.1151i
  -0.3421 - 0.0554i
```

```
 0.2603 - 0.1368i
-0.1977 - 0.1863i
...
-0.5745 - 0.2636i
 0.0578 - 0.1252i
 0.3992 + 0.2100i
 0.9710 + 0.0458i
-0.1525 - 0.0026i
```