习题四

第一题

对于如下方程组

$$\begin{pmatrix} 1 & -1 & 2 & 1 \\ -1 & 3 & 0 & -3 \\ 2 & 0 & 9 & -6 \\ 1 & -3 & -6 & 19 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 5 \\ 7 \end{pmatrix} \tag{1}$$

判断用Jacobi迭代、Gauss-Seidel 迭代、SOR迭代（分别取$\omega = 0.8, 1.2, 1.3, 1.6$）解上述方程组的收敛性。

若收敛，再用Jacobi迭代、Gauss-Seidel迭代、SOR迭代（分别取$\omega = 0.8, 1.2, 1.3, 1.6$）分别解上述方程组，若迭代终止条件为$\left\| b - Ax^{(n)} \right\|_2 \le 10^{-6}$，写出数值解。

比较上述各种迭代方法的收敛速度。

解：首先进行DLU分解，将$A = \{a_{ij}\}_{n \times n} \in \mathbb{R}^{n \times n}$分裂为$D - L - U$：

$$\begin{pmatrix} a_{11} & & & & \\ & a_{22} & & & \\ & & \ddots & & \\ & & & a_{n-1,n-1} & \\ & & & & a_{nn} \end{pmatrix} - \begin{pmatrix} 0 & & & & \\ -a_{21} & 0 & & & \\ \vdots & \vdots & \ddots & & \\ -a_{n-1,1} & a_{n-1,2} & \cdots & 0 & \\ -a_{n1} & -a_{n2} & \cdots & -a_{n,n-1} & 0 \end{pmatrix} - \begin{pmatrix} 0 & -a_{21} & \cdots & -a_{1,n-1} & -a_{1n} \\ & 0 & \cdots & -a_{2,n-1} & -a_{2n} \\ & & \ddots & \vdots & \vdots \\ & & & 0 & -a_{n-1,n} \\ & & & & 0 \end{pmatrix} \tag{2}$$

定义DLU分解函数

```matlab
function [D, L, U] = DLUDecomposition(A)

    % 名称：
    % 输入：
    %       A: 欲分解矩阵
    % 输出：
    %       D: 对角矩阵
    %       L: 下三角矩阵
    %       U: 上三角矩阵

    %% 函数

    order = size(A, 1);
    D = zeros(size(A));
    L = zeros(size(A));
    U = zeros(size(A));
    for i = 1: order
        D(i, i) = A(i, i);
        for j = 1: order
            if i > j
                L(i, j) = -A(i, j);
            elseif i < j
                U(i, j) = -A(i, j);
            end
        end
    end
```

```

end

```

Jacobi迭代：如果$\det D \neq 0$，那么

$$
\begin{aligned}
Ax = b \iff x = (I - D^{-1}A)x + D^{-1}b \iff x = B_J x + f_J \\
x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^{(k)}\right), \qquad 1 \le i \le n, k \in \mathbb{N}
\end{aligned} \tag{3}
$$

Gauss-Seidel迭代：如果$\det D \neq 0$，那么

$$
\begin{aligned}
Ax = b \iff x = (I - (D-L)^{-1}A)x + (D-L)^{-1}b \iff x = B_G x + f_G \\
x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \qquad 1 \le i \le n, k \in \mathbb{N}
\end{aligned} \tag{4}
$$

逐次超松弛迭代(SOR)迭代：选择松弛因子$w > 0$，那么

$$
\begin{aligned}
Ax = b \iff x = (I - w(D-wL)^{-1}A)x + w(D-wL)^{-1}b \iff x = B_w x + f_w \\
x_i^{(k+1)} = x_i^k + \frac{w}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i}^{n} a_{ij}x_j^{(k)}\right), \qquad 1 \le i \le n, k \in \mathbb{N}
\end{aligned} \tag{5}
$$

一阶线性定常迭代的基本定理：对于任意初始向量$x^{(0)}$，一阶线性定常迭代$x^{(n+1)} = Bx^{(n)} + f$收敛的充分必要条件为

$$
\lim_{n\to\infty} B^n = O \iff \rho(B) < 1 \iff \exists\|\cdot\|, \quad \|B\| < 1 \tag{6}
$$

分别定义迭代函数

```
function [judge, root] = JacobiIteration(A, b, x0, n)

    % 名称:        Jacobi迭代
    % 输入:
    %       A:      系数矩阵
    %       b:      右侧矩阵
    %       x0:     初始解
    %       n:      迭代次数
    % 输出:
    %       judge:  是否收敛
    %       root:   迭代解

    %% 函数

    % DLU分解
    D = DLUDecomposition(A);

    % Jacobi矩阵
    BJ = eye(size(A)) - D \ A;

    % 计算特征值
    eigenvalues = eig(BJ);

    % 判断是否收敛
    if max(abs(eigenvalues)) < 1
```

```
        judge = 1;
        root = x0;
        for k = 1: n
            root = BJ * root + D \ b;
        end
    else
        judge = 0;
        root = [];
    end

end

```

```
function [judge, root] = GaussSeidelIteration(A, b, x0, n)

    % 名称:       Gauss-Seidel迭代
    % 输入:
    %       A:      系数矩阵
    %       b:      右侧矩阵
    %       x0:     初始解
    %       n:      迭代次数
    % 输出:
    %       judge:  是否收敛
    %       root:   迭代解

    %% 函数

    % DLU分解
    [D, L, ~] = DLUDecomposition(A);

    % Gauss-Seidel矩阵
    BG = eye(size(A)) - (D - L) \ A;

    % 计算特征值
    eigenvalues = eig(BG);

    % 判断是否收敛
    if max(abs(eigenvalues)) < 1
        judge = 1;
        root = x0;
        for k = 1: n
            root = BG * root + (D - L) \ b;
        end
    else
        judge = 0;
        root = [];
    end

end

```

```
function [judge, root] = SORIteration(A, b, w, x0, n)

    % 名称:       SOR迭代
```

```
    % 输入：
    %       A:      系数矩阵
    %       b:      右侧矩阵
    %       w:      松弛因子
    %       x0:     初始解
    %       n:      迭代次数
    % 输出：
    %       judge: 是否收敛
    %       root:   迭代解

    %% 函数

    % DLU分解
    [D, L, ~] = DLUDecomposition(A);

    % 松弛矩阵
    Bw = eye(size(A)) - (D - w * L) \ A * w;

    % 计算特征值
    eigenvalues = eig(Bw);

    % 判断是否收敛
    if max(abs(eigenvalues)) < 1
        judge = 1;
        root = x0;
        for k = 1: n
            root = Bw * root + (D - w * L) \ b * w;
        end
    else
        judge = 0;
        root = [];
    end

end

```

定义主函数

```
clear; clc

% 定义系数矩阵与初始解
A = [1, -1, 2, 1;
    -1, 3, 0, -3;
    2, 0, 9, -6;
    1, -3, -6, 19];
b = [1; 3; 5; 7];
x0 = [0; 0; 0; 0];

% Jacobi迭代
JacobiRoot = x0;
JacobiNumber = 0;
while norm(b - A * JacobiRoot) > 1e-6
    JacobiNumber = JacobiNumber + 1;
    [JacobiJudge, JacobiRoot] = JacobiIteration(A, b, x0, JacobiNumber);
end
```

```
% Gauss-Seidel迭代
GaussSeidelRoot = x0;
GaussSeidelNumber = 0;
while norm(b - A * GaussSeidelRoot) > 1e-6
    GaussSeidelNumber = GaussSeidelNumber + 1;
    [GaussSeidelJudge, GaussSeidelRoot] = GaussSeidelIteration(A, b, x0, GaussSeidelNumber);
end

% SOR迭代
SORRootMatrix = [];
SORNumberMatrix = [];
SORJudgeMatrix = [];
for w = [0.8, 1.2, 1.3, 1.6]
    SORRoot = x0;
    SORNumber = 0;
    while norm(b - A * SORRoot) > 1e-6
        SORNumber = SORNumber + 1;
        [SORJudge, SORRoot] = SORIteration(A, b, w, x0, SORNumber);
    end
    SORRootMatrix = [SORRootMatrix, SORRoot];
    SORNumberMatrix = [SORNumberMatrix, SORNumber];
    SORJudgeMatrix = [SORJudgeMatrix, SORJudge];
end

%  创建表格
iterationName = {'Jacobi'; 'Gauss-Seidel'; 'SOR(w=0.8)'; 'SOR(w=1.2)'; 'SOR(w=1.3)'; 'SOR(w=1.6)'};
judge = [JacobiJudge; GaussSeidelJudge; SORJudgeMatrix'];
number = [JacobiNumber; GaussSeidelNumber; SORNumberMatrix'];
root = [JacobiRoot'; GaussSeidelRoot'; SORRootMatrix'];
variableNames = {'迭代方法', '是否收敛', '迭代次数', '迭代解'};
T = table(iterationName, int16(judge), int16(number), vpa(root, 3), 'VariableNames', variableNames);
% 显示表格
disp(T)
```

输出结果

```
         迭代方法           是否收敛     迭代次数                    迭代解
    ________________    ________    ________    ______________________________

    {'Jacobi'      }       1          417       -8.0    0.333    3.67    2.0
    {'Gauss-Seidel'}       1          204       -8.0    0.333    3.67    2.0
    {'SOR(w=0.8)'  }       1          309       -8.0    0.333    3.67    2.0
    {'SOR(w=1.2)'  }       1          136       -8.0    0.333    3.67    2.0
    {'SOR(w=1.3)'  }       1          110       -8.0    0.333    3.67    2.0
    {'SOR(w=1.6)'  }       1           35       -8.0    0.333    3.67    2.0
```

通过输出结果，我们可知这五种迭代方法均收敛，且数值解为

$$x_1 = -8, \qquad x_2 = 0.333, \qquad x_3 = 3.67, \qquad x_4 = 2 \tag{7}$$

迭代次数如结果所示，迭代次数越少，迭代速度越快。

第二题

用共轭梯度法求解方程组$Ax=b$，其中

$$A=\begin{pmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 99 & -1 \\ & & & -1 & 100 \end{pmatrix},\qquad b=\begin{pmatrix} 0 \\ 0 \\ 1 \\ 2 \\ \vdots \\ 96 \\ 97 \\ 99 \end{pmatrix} \tag{8}$$

若迭代终止条件为$\left\|b-Ax^{(n)}\right\|_2\leq 10^{-8}$，分别给出数值近似解，迭代步数和计算时间，并计算误差$\left\|x^{(n)}-x^*\right\|_2$，其中$x^*$为方程组的精确解

$$x^*=\begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \tag{9}$$

解：**共轭梯度法(CG方法)**：

$$\begin{cases} p^{(0)}=r^{(0)}=b-Ax^{(0)} \\ \rho^{(0)}=(r^{(0)},r^{(0)}) \\ \alpha_0=\frac{\rho^{(0)}}{(Ap^{(0)},p^{(0)})} \\ x^{(1)}=x^{(0)}+\alpha_0 p^{(0)} \end{cases},\qquad \begin{cases} r^{(n)}=b-Ax^{(n)} \\ \rho^{(n)}=(r^{(n)},r^{(n)}) \\ \beta_n=\frac{\rho^{(n)}}{\rho^{(n-1)}} \\ p^{(n)}=r^{(n)}+\beta_n p^{(n-1)} \\ \alpha_n=\frac{\rho^{(n)}}{(Ap^{(n)},p^{(n)})} \\ x^{(n+1)}=x^{(n)}+\alpha_n p^{(n)} \end{cases} \tag{10}$$

定义共轭梯度函数

```
function root = conjugateGradient(A, b, x0, n)

    % 名称:         共轭梯度算法
    % 输入:
    %       A:      系数矩阵
    %       b:      右侧矩阵
    %       x0:     初始解
    %       n:      迭代次数
    % 输出:
    %       root:   迭代解

    %% 函数

    p = b - A * x0;
    r = b - A * x0;
    rho = dot(r, r);
    alpha = rho / dot(A * p, p);
    root = x0 + alpha * p;
    if n >= 2
        for k = 2: n
```

```
            r = b - A * root;
            rho0 = rho;
            rho = dot(r, r);
            beta = rho / rho0;
            p = r + beta * p;
            alpha = rho / dot(A * p, p);
            root = root + alpha * p;
        end
    end

end

```

定义主函数

```
clear; clc

% 定义系数矩阵
A = zeros(100, 100);
b = transpose([0, 0, 1: 97, 99]);
for n = 1: 100
    A(n, n) = n;
    if n == 1
        A(n, n + 1) = -1;
    elseif n == 100
        A(n, n - 1) = -1;
    else
        A(n, n + 1) = -1;
        A(n, n - 1) = -1;
    end
end

% 精确根
exactRoot = A \ b;

% 迭代求解近似根
x0 = zeros(100, 1);    % 初始根
approximateRoot = x0; % 近似根
n = 0;
tic % 启动计时器
while norm(b - A * approximateRoot) > 1e-8
    n = n + 1;
    approximateRoot = conjugateGradient(A, b, x0, n);
end
runTime = toc; % 计算时间
error = norm(exactRoot - approximateRoot); % 计算误差

% 输出结果
disp('数值近似解为：')
disp(approximateRoot)
fprintf('迭代步数为：%d步\n', n);
fprintf('计算时间：%f秒\n', runTime)
fprintf('误差为：%e\n', error)

```

输出结果

```
数值近似解为：
    0.999999999999984
    1.000000000000031
    0.999999999999846
    1.000000000000559
    0.999999999998171
    1.000000000005283
    0.999999999986591
    1.000000000029690
    0.999999999943325
    1.000000000091597
    0.999999999878557
    1.000000000123857
    0.999999999919209
    1.000000000001305
    1.000000000068709
    0.999999999924879
    1.000000000011854
    1.000000000056110
    0.999999999946979
    0.999999999985556
    1.000000000055845
    0.999999999984342
    0.999999999956661
    1.000000000030604
    1.000000000029485
    0.999999999964283
    0.999999999980812
    1.000000000035984
    1.000000000013048
    0.999999999965696
    0.999999999989694
    1.000000000031962
    1.000000000010028
    0.999999999970667
    0.999999999988592
    1.000000000026371
    1.000000000013778
    0.999999999977110
    0.999999999983449
    1.000000000018731
    1.000000000019184
    0.999999999986142
    0.999999999978817
    1.000000000008394
    1.000000000022148
    0.999999999997375
    0.999999999978180
    0.999999999996951
    1.000000000020129
    1.000000000008188
    0.999999999982792
    0.999999999987597
```

```
    1.000000000013362
    1.000000000015433
    0.999999999990992
    0.999999999982809
    1.000000000004592
    1.000000000017776
    0.999999999999489
    0.999999999982565
    0.999999999997054
    1.000000000016509
    1.000000000005616
    0.999999999984629
    0.999999999992559
    1.000000000014369
    1.000000000008426
    0.999999999986209
    0.999999999991443
    1.000000000013831
    1.000000000007716
    0.999999999985467
    0.999999999994404
    1.000000000015639
    1.000000000001666
    0.999999999983732
    1.000000000004588
    1.000000000014370
    0.999999999987387
    0.999999999993515
    1.000000000017980
    0.999999999990537
    0.999999999991136
    1.000000000021119
    0.999999999978363
    1.000000000014988
    0.999999999992294
    1.000000000003009
    0.999999999999124
    1.000000000000177
    0.999999999999982
    0.999999999999999
    1.000000000000001
    1.000000000000000
    1.000000000000000
    1.000000000000000
    1.000000000000001
    1.000000000000001
    1.000000000000001
    0.999999999999996

迭代步数为：65步
计算时间：0.020306秒
误差为：3.004497e-10
```

第二题

已知方程

$$x^3 - 3x - 1 = 0 \tag{11}$$

分别用不动点迭代（取迭代函数为$\varphi(x) = \sqrt[3]{3x+1}$）、Steffensen迭代法（其中不动点迭代的迭代函数仍为$\varphi(x) = \sqrt[3]{3x+1}$)、Newton迭代法、Newton下山法求方程的根，其中除Newton下山法初值为$x_0 = 0.6$外，其余初值为$x_0 = 2$。迭代终止条件为$|x_{n+1} - x_n| < 10^{-6}$，并分别输出方程的近似根和每种迭代的次数。

解：**不动点迭代**：

$$x_{n+1} = \varphi(x_n) \tag{12}$$

Steffensen迭代：

$$y_n = \varphi(x_n), \qquad z_n = \varphi(y_n), \qquad x_{n+1} = x_n - \frac{(y_n - x_n)^2}{z_n - 2y_n + x_n} \tag{13}$$

Newton法：方程$f(x) = 0$的迭代

$$x_{n+1} = \varphi(x_n), \qquad \varphi(x) = x - \frac{f(x)}{f'(x)} \tag{14}$$

Newton下山法：方程$f(x) = 0$的迭代

$$x_{n+1} = x_n - \lambda_n \frac{f(x_n)}{f'(x_n)} \tag{15}$$

其中下山因子

$$\lambda_n = \max\left\{\frac{1}{2^r} : \left|f\left(x_n - \frac{f(x_n)}{2^r f'(x_n)}\right)\right| < |f(x_n)|, r \in \mathbb{N}\right\} \tag{16}$$

分别定义迭代函数

```matlab
function root = fixedPointIteration(phi, x0, n)

    % 名称:        不动点迭代
    % 输入:
    %       phi:    迭代函数
    %       x0:     初始解
    %       n:      迭代次数
    % 输出:
    %       root:   迭代解

    %% 函数
    root = x0;
    for k = 1: n
        root = phi(root);
    end

end

```

```matlab
function root = SteffensenIteration(phi, x0, n)

```

```

    % 名称:       Steffensen迭代
    % 输入:
    %       phi:    迭代函数
    %       x0:     初始解
    %       n:      迭代次数
    % 输出:
    %       root:   迭代解

    %% 函数
    root = x0;
    for k = 1: n
        y = phi(root);
        z = phi(y);
        root = root - (y - z)^2 / (z - 2 * y + root);
    end

end

```

```
function root = NewtonIteration(fun, x0, n)

    % 名称:       Newton迭代
    % 输入:
    %       fun:    函数
    %       x0:     初始解
    %       n:      迭代次数
    % 输出:
    %       root:   迭代解

    %% 函数
    syms x
    phi = matlabFunction(x - fun(x) ./ diff(fun(x)));
    root = x0;
    for k = 1: n
        root = phi(root);
    end

end

```

```
function root = NewtonDescentIteration(fun, x0, n)

    % 名称:       Newton下山迭代
    % 输入:
    %       fun:    函数
    %       x0:     初始解
    %       n:      迭代次数
    % 输出:
    %       root:   迭代解

    %% 函数
    syms x
    phi = matlabFunction(fun(x) ./ diff(fun(x)));
```

```
    root = x0;
    for k = 1: n
        lambda = 1;
        A = abs(fun(root - phi(root) / 2^lambda));
        B = abs(fun(root));
        while A > B
            lambda = lambda + 1;
            A = abs(fun(root - phi(root) / 2^lambda));
            B = abs(fun(root));
        end
        root = root - lambda * phi(root);
    end

end

```

定义主函数

```
clear; clc

% 不动点迭代
phi = @(x) (3 * x + 1) .^ (1 / 3);
x0 = 2;
fixedPointRoot = fixedPointIteration(phi, x0, 1);
fixedPointRootMatrix = [x0, fixedPointRoot];
fixedPointNumber = 1;
while abs(fixedPointRootMatrix(end) - fixedPointRootMatrix(end - 1)) >= 1e-6
    fixedPointNumber = fixedPointNumber + 1;
    fixedPointRoot = fixedPointIteration(phi, x0, fixedPointNumber);
    fixedPointRootMatrix = [fixedPointRootMatrix, fixedPointRoot];
end

% Steffensen迭代
phi = @(x) (3 * x + 1) .^ (1 / 3);
x0 = 2;
SteffensenRoot = SteffensenIteration(phi, x0, 1);
SteffensenRootMatrix = [x0, SteffensenRoot];
SteffensenNumber = 1;
while abs(SteffensenRootMatrix(end) - SteffensenRootMatrix(end - 1)) >= 1e-6
    SteffensenNumber = SteffensenNumber + 1;
    SteffensenRoot = SteffensenIteration(phi, x0, SteffensenNumber);
    SteffensenRootMatrix = [SteffensenRootMatrix, SteffensenRoot];
end

% Newton迭代
fun = @(x) x^3 - 3*x - 1;
x0 = 2;
NewtonRoot = NewtonIteration(fun, x0, 1);
NewtonRootMatrix = [x0, NewtonRoot];
NewtonNumber = 1;
while abs(NewtonRootMatrix(end) - NewtonRootMatrix(end - 1)) >= 1e-6
    NewtonNumber = NewtonNumber + 1;
    NewtonRoot = NewtonIteration(fun, x0, NewtonNumber);
    NewtonRootMatrix = [NewtonRootMatrix, NewtonRoot];
end
```

```
% Newton下山迭代
fun = @(x) x^3 - 3*x - 1;
x0 = 0.6;
NewtonDescentRoot = NewtonDescentIteration(fun, x0, 1);
NewtonDescentRootMatrix = [x0, NewtonDescentRoot];
NewtonDescentNumber = 1;
while abs(NewtonDescentRootMatrix(end) - NewtonDescentRootMatrix(end - 1)) >= 1e-6
    NewtonDescentNumber = NewtonDescentNumber + 1;
    NewtonDescentRoot = NewtonDescentIteration(fun, x0, NewtonDescentNumber);
    NewtonDescentRootMatrix = [NewtonDescentRootMatrix, NewtonDescentRoot];
end

% 精确解
root = roots([1, 0, -3, -1]);

% 输出结果
disp('精确解为：')
disp(root)
disp('----------------------------------------------')
disp(' ')
%  创建表格
iterationName = {'不动点迭代'; 'Steffensen迭代'; 'Newton迭代'; 'Newton下山迭代'};
number = [fixedPointNumber; SteffensenNumber; NewtonNumber; NewtonDescentNumber];
root = [fixedPointRoot; SteffensenRoot; NewtonRoot; NewtonDescentRoot];
variableNames = {'迭代方法', '迭代次数', '迭代解'};
T = table(iterationName, int16(number), vpa(root, 5), 'VariableNames', variableNames);
% 显示表格
disp(T)

```

输出结果

```
精确解为：
    1.8794
   -1.5321
   -0.3473

----------------------------------------------

         迭代方法           迭代次数      迭代解
    __________________    _______    _______

    {'不动点迭代'      }      10       1.8794
    {'Steffensen迭代'}     112       1.8794
    {'Newton迭代'     }       4       1.8794
    {'Newton下山迭代' }       6      -0.3473
```

第四题

已知$x^* = \sqrt{2}$为方程$x^4 - 4x^2 + 4 = 0$的二重根，分别用重根Newton迭代、求重根的含参数的Newton迭代、改进Newton迭代法求该方程的的近似值，其中初始解为$x_0 = 1.5$，迭代终止条件为$|x_{n+1} - x_n| < 10^{-6}$，给出几种方法的具体迭代步数。

解：**重根Newton法**：如果x^*为方程$f(x) = 0$的m重根，那么迭代

$$x_{n+1} = \varphi(x_n), \qquad \varphi(x) = x - \frac{f(x)}{f'(x)} \tag{17}$$

含参m的Newton迭代法：如果x^*为方程$f(x) = 0$的m重根，那么迭代

$$x_{n+1} = \varphi(x_n), \qquad \varphi(x) = x - m\frac{f(x)}{f'(x)} \tag{18}$$

改进Newton迭代法：如果x^*为方程$f(x) = 0$的m重根，那么迭代

$$x_{n+1} = \varphi(x_n), \qquad \varphi(x) = x - \frac{\mu(x)}{\mu'(x)}, \qquad \mu(x) = \frac{f(x)}{f'(x)} \tag{19}$$

分别定义迭代函数

```
function root = reRootsNewtonIteration(fun, x0, n)

    % 名称：        重根Newton迭代
    % 输入：
    %       fun:    函数
    %       x0:     初始解
    %       n:      迭代次数
    % 输出：
    %       root:   迭代解

    %% 函数
    syms x
    phi = matlabFunction(x - fun(x) ./ diff(fun(x)));
    root = x0;
    for k = 1: n
        root = phi(root);
    end

end

```

```
function order = orderOfRoot(fun, x0)

    % 名称：        求解函数零点的阶
    % 输入：
    %       fun:    函数
    %       x0:     初始解
    % 输出：
    %       order:  x0附近零点的阶

    %% 函数
    syms x
    % 找到最近的根
```

```
    roots = solve(fun, x);
    [~, index] = min(abs(roots - x0));
    exactRoot = roots(index);

    % 求解精确根的阶
    order = 1;
    Df = matlabFunction(diff(fun(x)));
    while abs(Df(exactRoot)) < 1e-3
        order = order + 1;
        Df = matlabFunction(diff(Df(x)));
    end

end

```

```
function root = NewtonIterationWithParameter(fun, x0, n)

    % 名称:        含参Newton迭代
    % 输入:
    %       fun:    函数
    %       x0:     初始解
    %       n:      迭代次数
    % 输出:
    %       root:   迭代解

    %% 函数
    syms x
    order = orderOfRoot(fun, x0);
    phi = matlabFunction(x - order .* fun(x) ./ diff(fun(x)));
    root = x0;
    for k = 1: n
        root = phi(root);
    end

end

```

```
function root = improvingNewtonIteration(fun, x0, n)

    % 名称:        改进Newton迭代
    % 输入:
    %       fun:    函数
    %       x0:     初始解
    %       n:      迭代次数
    % 输出:
    %       root:   迭代解

    %% 函数
    syms x
    mu = matlabFunction(fun(x) ./ diff(fun(x)));
    phi = matlabFunction(x - mu(x) ./ diff(mu(x)));
    root = x0;
    for k = 1: n
        root = phi(root);
```

```
    end

end

```

定义主函数

```
clear; clc

% 重根Newton迭代
fun = @(x) x^4 - 4*x^2 + 4;
x0 = 1.5;
reRootsNewtonRoot = reRootsNewtonIteration(fun, x0, 1);
reRootsNewtonRootMatrix = [x0, reRootsNewtonRoot];
reRootsNewtonNumber = 1;
while abs(reRootsNewtonRootMatrix(end) - reRootsNewtonRootMatrix(end - 1)) >= 1e-6
    reRootsNewtonNumber = reRootsNewtonNumber + 1;
    reRootsNewtonRoot = reRootsNewtonIteration(fun, x0, reRootsNewtonNumber);
    reRootsNewtonRootMatrix = [reRootsNewtonRootMatrix, reRootsNewtonRoot];
end

% 含参Newton迭代
fun = @(x) x^4 - 4*x^2 + 4;
x0 = 1.5;
NewtonWithParameterRoot = NewtonIterationWithParameter(fun, x0, 1);
NewtonWithParameterRootMatrix = [x0, NewtonWithParameterRoot];
NewtonWithParameterNumber = 1;
while abs(NewtonWithParameterRootMatrix(end) - NewtonWithParameterRootMatrix(end - 1)) >= 1e-6
    NewtonWithParameterNumber = NewtonWithParameterNumber + 1;
    NewtonWithParameterRoot = NewtonIterationWithParameter(fun, x0, NewtonWithParameterNumber);
    NewtonWithParameterRootMatrix = [NewtonWithParameterRootMatrix, NewtonWithParameterRoot];
end

% 改进Newton迭代
fun = @(x) x^4 - 4*x^2 + 4;
x0 = 1.5;
improvingNewtonRoot = improvingNewtonIteration(fun, x0, 1);
improvingNewtonRootMatrix = [x0, improvingNewtonRoot];
improvingNewtonNumber = 1;
while abs(improvingNewtonRootMatrix(end) - improvingNewtonRootMatrix(end - 1)) >= 1e-6
    improvingNewtonNumber = improvingNewtonNumber + 1;
    improvingNewtonRoot = improvingNewtonIteration(fun, x0, improvingNewtonNumber);
    improvingNewtonRootMatrix = [improvingNewtonRootMatrix, improvingNewtonRoot];
end

% 输出结果
%  创建表格
```

```
iterationName = {'重根Newton迭代'; '含参Newton迭代'; '改进Newton迭代'};
number = [reRootsNewtonNumber; NewtonWithParameterNumber;
improvingNewtonNumber];
root = [reRootsNewtonRoot; NewtonWithParameterRoot; improvingNewtonRoot];
variableNames = {'迭代方法', '迭代次数', '迭代解'};
T = table(iterationName, int16(number), vpa(root, 5), 'VariableNames',
variableNames);
% 显示表格
disp(T)

```

输出结果

```
         迭代方法            迭代次数      迭代解
    ________________    ________    _______

    {'重根Newton迭代'}        17        1.4142
    {'含参Newton迭代'}         8        1.4142
    {'改进Newton迭代'}         4        1.4142
```

第五题

用Euler公式、改进Euler公式、经典四阶Runge-Kutta 方法解下列初值问题

$$\begin{cases} y'(x) = \dfrac{2}{x}y + x^2\mathrm{e}^x, & 1 \le x \le 2 \\ y(1) = 0 \end{cases} \tag{20}$$

为使计算量相当，步长比为$1:2:4$，即三种方法的步长分别为$0.05, 0.1, 0.2$，计算在$x = 1.2, 1.4, 1.8, 2.0$点处的数值解，并与精确解比较误差，其中精确解为

$$y(x) = x^2(\mathrm{e}^x - \mathrm{e}) \tag{21}$$

解：**Euler公式**：

$$y_{n+1} = y_n + hf(x_n, y_n), \qquad x_n = x_0 + nh \tag{22}$$

改进Euler法：

$$y_{n+1} = y_n + \frac{h}{2}(f(x_n, y_n) + f(x_n + h, y_n + hf(x_n, y_n))) \tag{23}$$

经典四阶Runge-Kutta方法：

$$\begin{cases} y_{n+1} = y_n + \dfrac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\ K_1 = f(x_n, y_n) \\ K_2 = f\left(x_n + \dfrac{h}{2}, y_n + \dfrac{h}{2}K_1\right) \\ K_3 = f\left(x_n + \dfrac{h}{2}, y_n + \dfrac{h}{2}K_2\right) \\ K_4 = f\left(x_n + h, y_n + hK_3\right) \end{cases} \tag{24}$$

分别定义函数

```
function matrix = EulerFormula(fun, h, x0, xend, y0)

```

```
    % 名称:          Euler公式
    % 输入:
    %       fun:      函数
    %       h:        步长
    %       x0:       初始x值
    %       xend:     终止x值
    %       y0:       初始y值
    % 输出:
    %       matrix:   近似解

    %% 函数
    n = length(x0: h: xend);
    matrix = [x0: h: xend; y0, zeros(1, n-1)];
    for k = 1: n-1
        matrix(2, k+1) = matrix(2, k) + h * fun(matrix(1, k), matrix(2, k));
    end

end

```

```
function matrix = improvingEulerFormula(fun, h, x0, xend, y0)

    % 名称:          改进Euler公式
    % 输入:
    %       fun:      函数
    %       h:        步长
    %       x0:       初始x值
    %       xend:     终止x值
    %       y0:       初始y值
    % 输出:
    %       matrix:   近似解

    %% 函数
    n = length(x0: h: xend);
    matrix = [x0: h: xend; y0, zeros(1, n-1)];
    for k = 1: n-1
        matrix(2, k+1) = matrix(2, k) ...
            + h * fun(matrix(1, k), matrix(2, k)) / 2 ...
            + h * fun(matrix(1, k) + h, matrix(2, k) + h * fun(matrix(1, k), matrix(2, k))) / 2;
    end

end

```

```
function matrix = Classic4RungeKuttaMethod(fun, h, x0, xend, y0)

    % 名称:          经典四阶Runge-Kutta方法
    % 输入:
    %       fun:      函数
    %       h:        步长
    %       x0:       初始x值
    %       xend:     终止x值
    %       y0:       初始y值
```

```
    % 输出：
    %       matrix：  近似解

    %% 函数
    n = length(x0: h: xend);
    matrix = [x0: h: xend; y0, zeros(1, n-1)];
    for k = 1: n-1
        K1 = fun(matrix(1, k), matrix(2, k));
        K2 = fun(matrix(1, k) + h/2, matrix(2, k) + h*K1/2);
        K3 = fun(matrix(1, k) + h/2, matrix(2, k) + h*K2/2);
        K4 = fun(matrix(1, k) + h, matrix(2, k) + h*K3);
        matrix(2, k+1) = matrix(2, k) + h / 6 * (K1 + 2 * K2 + 2 * K3 + K4);
    end

end

```

定义主函数

```
clear; clc

% 定义函数
fun = @(x, y) 2 .* y ./ x + x .^ 2 .* exp(x);
x0 = 1;
xend = 2;
y0 = 0;

% Euler法
EulerMatrix05 = EulerFormula(fun, 0.05, x0, xend, y0);
EulerMatrix1 = EulerFormula(fun, 0.1, x0, xend, y0);
EulerMatrix2 = EulerFormula(fun, 0.2, x0, xend, y0);

% 改进Euler法
improvingEulerMatrix05 = improvingEulerFormula(fun, 0.05, x0, xend, y0);
improvingEulerMatrix1 = improvingEulerFormula(fun, 0.1, x0, xend, y0);
improvingEulerMatrix2 = improvingEulerFormula(fun, 0.2, x0, xend, y0);

% 经典四阶Runge-Kutta方法
RungeKuttaMatrix05 = Classic4RungeKuttaMethod(fun, 0.05, x0, xend, y0);
RungeKuttaMatrix1 = Classic4RungeKuttaMethod(fun, 0.1, x0, xend, y0);
RungeKuttaMatrix2 = Classic4RungeKuttaMethod(fun, 0.2, x0, xend, y0);

% 精确解
exactFunction = @(x) x .^ 2 .* (exp(x) - exp(1));

% 比较结果
matrix = [];
for x = [1.2, 1.4, 1.8, 2.0]
    matrix0 = [0.05, exactFunction(x), ...
    EulerMatrix05(2, EulerMatrix05(1, :) == x),...
    improvingEulerMatrix05(2, improvingEulerMatrix05(1, :) == x),...
    RungeKuttaMatrix05(2, RungeKuttaMatrix05(1, :) == x);
    0.1, exactFunction(x), ...
    EulerMatrix1(2, EulerMatrix1(1, :) == x),...
    improvingEulerMatrix1(2, improvingEulerMatrix1(1, :) == x),...
```

```
    RungeKuttaMatrix1(2, RungeKuttaMatrix1(1, :) == x);
    0.2, exactFunction(x), ...
    EulerMatrix2(2, EulerMatrix2(1, :) == x),...
    improvingEulerMatrix2(2, improvingEulerMatrix2(1, :) == x),...
    RungeKuttaMatrix2(2, RungeKuttaMatrix2(1, :) == x)];
    matrix = [matrix; matrix0];
end
matrix12 = matrix(1: 3, :);
matrix14 = matrix(4: 6, :);
matrix18 = matrix(7: 9, :);
matrix20 = matrix(10: 12, :);

% 输出结果

%  创建表格
variableNames = {'x', '步长', '精确解', 'Euler法', 'Euler法误差', '改进Euler法', '改进Euler法误差', '经典四阶Runge-Kutta方法', 'Runge-Kutta方法误差'};
num = 8;
X = [1.2; 1.2; 1.2; 1.4; 1.4; 1.4; 1.8; 1.8; 1.8; 2.0; 2.0; 2.0];
T = table(X, matrix(:, 1), vpa(matrix(:, 2), num), ...
    vpa(matrix(:, 3), num), vpa(abs(matrix(:, 3) - matrix(:, 2)), num), ...
    vpa(matrix(:, 4), num), vpa(abs(matrix(:, 4) - matrix(:, 2)), num), ...
    vpa(matrix(:, 5), num), vpa(abs(matrix(:, 5) - matrix(:, 2)), num), ...
    'VariableNames', variableNames);
% 显示表格
disp(T)

```

输出结果

```
     x      步长        精确解          Euler法        Euler法误差       改进Euler法      改进Euler法误差    经典四阶Runge-Kutta方法     Runge-Kutta方法误差
    ___    _____    __________    __________    ___________    __________    _______________    _____________________    __________________

    1.2     0.05    0.86664254    0.769696      0.096946536    0.86429069    0.0023518451          0.86664107            0.0000014660831
    1.2      0.1    0.86664254    0.68475558    0.18188696     0.85831454    0.0083279984          0.86662169            0.000020843031
    1.2      0.2    0.86664254    0.54365637    0.32298617     0.84053441    0.026108122           0.86637911            0.00026342379
    1.4     0.05    2.6203596     2.3402236     0.28013595     2.6141742      0.0061853358          2.6203562             0.0000033682149
    1.4      0.1    2.6203596     2.0935477     0.52681186     2.5982982      0.022061312           2.6203113             0.000048245364
    1.4      0.2    2.6203596     1.6810688     0.93929072     2.5502404      0.070119148           2.6197405             0.00061903077
    1.8     0.05    10.793625     9.7434894     1.0501353      10.774418      0.019206872           10.793616             0.0000084984631
    1.8      0.1    10.793625     8.8091197     1.984505       10.724467      0.0691576             10.793502             0.00012287684
    1.8      0.2    10.793625     7.2247183     3.5689063      10.569818      0.22380681            10.792018             0.001607063
      2     0.05    18.683097     16.949013     1.7340838      18.654245      0.028851759           18.683085             0.000011755683
```

```
       2     0.1    18.683097     15.398236     3.2848614      18.578882     0.10421463            18.682927            0.00017051423
       2     0.2    18.683097     12.750383     5.9327142      18.343834     0.33926303            18.680852            0.0022447174
```