一、实验一:给定常微分方程二阶边值问题

1.题目:

$$\begin{cases} y'' + 4y = (4-\pi^2)\left[2\cos(\pi x) + 3\sin(\pi x)\right], 0 \le x \le 1, \\ y(0) = 2, y(1) = -2. \end{cases} \tag{1}$$

取计算步长 h=0.01 , 应用有限差分格式将上述边值问题离散成三对角型线性方程组。然后, 分别利用 Doolitle 分解法、改进的Cholesky 分解法、 QR 分解法及追赶法求解该线性方程组，获得原边值问题在网格内点 $x_i(i = 1, 2, \ldots, 99)$ 的四组逼近解 (保留 5 位有效数字)。此外, 要求给出四组逼近解的计算误差: $err = \max_{1\le i\le 99} |y(x_i) - y_i|$ 及计算时间, 以此比较其算法的计算效率。

解:

STEP ONE:应用有限差分格式将上述边值问题离散成三对角型线性方程组

当取计算步长为 0.01 时原方程有有限差分格式为:

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = -4y_i + r_i, i = 0, 1, \ldots, 100 \tag{2}$$

其中$x_i = ih, y_i \approx y(x_i), r_i = (4-\pi^2)\left[2\cos(\pi x_i) + 3\sin(\pi x_i)\right], y_0 = 2, y_{100} = -2$

所以该差分格式等价于如下三对角型线性方程组:

$$\begin{pmatrix} 2-4h^2 & -1 & & & & \\ -1 & 2-4h^2 & -1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & -1 & 2-4h^2 & -1 & \\ & & & -1 & 2-4h^2 & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{98} \\ y_{99} \end{pmatrix} = \begin{pmatrix} -h^2r_1 + 2 \\ -h^2r_2 \\ \vdots \\ -h^2r_{98} \\ -h^2r_{99} - 2 \end{pmatrix} \tag{3}$$

生成该线性方程组的代码如下:

```
h = 0.01;                                          %步长为0.01
N = 100;                                           %划分为100份
n = 99;
e = ones(n, 1);
A = spdiags([-e, 1.9996*e, -e], -1:1, n, n);   %用于生成三对角型矩阵
A = full(A)                                        %转化为常规形式
% 生成b()
B = zeros(n, 1);
k = 1;
for i = 0.01:0.01:0.99
    B(k) = -h*h*(2*cos(pi*i) + 3*sin(pi*i))*(4 - pi*pi);
    k = k+1;
end
B(1) = B(1) + 2;
B(99) = B(99) - 2;
```

STEP TWO: 使用四种不同的方法求解方程组

将对应的h带入后方程组如下所示：

$$\begin{pmatrix} 1.9996 & -1 & & & & \\ -1 & 1.9996 & -1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & -1 & 1.9996 & -1 & \\ & & & -1 & 1.9996 & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{98} \\ y_{99} \end{pmatrix} = \begin{pmatrix} 2.0012 \\ 0.0013 \\ \vdots \\ -0.0011 \\ -2.0011 \end{pmatrix} \tag{4}$$

下设A, B分别为方程组的系数矩阵和等号右端列向量，而x满足$Ax = B$

(1)Doolitle分解法

首先计算A的各阶顺序主子式，其计算代码如下：

%计算A的各阶顺序主子式，为Doolitle分解做准备

```
dett = zeros(99, 1);
for i = 1:99
  dett(i) = det(A(1:i, 1:i));
end
min(abs(dett))
```

计算结果为ans = 1.9996

因此A的各阶顺序主子式皆不为零，可以使用Doolitle分解法求解，从而得解如下表所示：

1	2	3	4	5	6	7	8	9	10
2.0933	2.1845	2.2735	2.3603	2.4448	2.5268	2.6064	2.6834	2.7577	2.8293
2.8981	2.9641	3.0271	3.0872	3.1442	3.1981	3.2489	3.2964	3.3407	3.3817
3.4193	3.4536	3.4845	3.5119	3.5359	3.5563	3.5733	3.5867	3.5966	3.603
3.6058	3.605	3.6007	3.5928	3.5814	3.5664	3.548	3.526	3.5005	3.4716
3.4393	3.4035	3.3645	3.322	3.2764	3.2274	3.1753	3.1201	3.0618	3.0004
2.9361	2.8689	2.7989	2.7261	2.6506	2.5725	2.4919	2.4088	2.3233	2.2355
2.1455	2.0534	1.9593	1.8633	1.7654	1.6657	1.5645	1.4616	1.3574	1.2518
1.1449	1.037	0.92798	0.81808	0.70736	0.59595	0.48395	0.37147	0.25863	0.14553
0.032284	-0.08099	-0.19419	-0.30719	-0.41989	-0.53217	-0.64393	-0.75506	-0.86544	-0.97496
-1.0835	-1.191	-1.2973	-1.4024	-1.506	-1.6082	-1.7088	-1.8077	-1.9048	

其中第i行第k列对应着第x_{ik}个值

Doolitle分解法代码如下：

```
function x = Doolitle(A, B)
 [n, n] = size(A);
 L = zeros(n);
 U = zeros(n);
 x = zeros(n, 1);
 y = zeros(n, 1);
 for r = 1:n
     for i = r:n
         U(r, i) = A(r, i) - sum(L(r, 1:r-1).*U(1:r-1, i)');
         L(i, r) = (A(i, r)- sum(L(i, 1:r-1).*U(1:r-1, r)'))/U(r, r);
```

```
    end
  end;
  L;, U;                                %即为所得的L与U
  for i= 1:n
      y(i) = B(i) - sum(L(i, 1:i-1).*y(1:i-1)');     %直接除的方式要慢
  end
  for j = n:-1:1
      x(j) = (y(j) - sum(U(j, j+1:n).*x(j+1:n)'))/U(j, j);
  end
end
```

(2)改进的cholesky分解法

由于对于改进的cholesky分解法，从定理证明过程可看出其适用条件为系数阵对称，且任意阶顺序主子式不为零，这两点显然满足，于是可以使用改进的cholesky分解法，得到的解如下表所示：

1	2	3	4	5	6	7	8	9	10
2.0933	2.1845	2.2735	2.3603	2.4448	2.5268	2.6064	2.6834	2.7577	2.8293
2.8981	2.9641	3.0271	3.0872	3.1442	3.1981	3.2489	3.2964	3.3407	3.3817
3.4193	3.4536	3.4845	3.5119	3.5359	3.5563	3.5733	3.5867	3.5966	3.603
3.6058	3.605	3.6007	3.5928	3.5814	3.5664	3.548	3.526	3.5005	3.4716
3.4393	3.4035	3.3645	3.322	3.2764	3.2274	3.1753	3.1201	3.0618	3.0004
2.9361	2.8689	2.7989	2.7261	2.6506	2.5725	2.4919	2.4088	2.3233	2.2355
2.1455	2.0534	1.9593	1.8633	1.7654	1.6657	1.5645	1.4616	1.3574	1.2518
1.1449	1.037	0.92798	0.81808	0.70736	0.59595	0.48395	0.37147	0.25863	0.14553
0.032284	-0.08099	-0.19419	-0.30719	-0.41989	-0.53217	-0.64393	-0.75506	-0.86544	-0.97496
-1.0835	-1.191	-1.2973	-1.4024	-1.506	-1.6082	-1.7088	-1.8077	-1.9048	

其中第i行第k列对应着第x_{ik}个值

改进的cholesky分解法代码如下图：

```
function x = Cholesky(A, B)     %高阶情形快于LU分解
n = length(B);
v = zeros(n);
x = zeros(n, 1);
y = zeros(n, 1);
for j = 1:n
    for i = 1:j-1
        v(j, i) = A(j, i)*A(i, i);
    end
    A(j, j) = A(j, j) - A(j, 1:j-1)*v(j, 1:j-1)';
    %因为在后续计算中a这个矩阵前面的数字已经没有用了，所以为了简化计算时间，直接将L和D矩
      阵的值填到a中
    A(j+1:n, j) = (A(j+1:n, j) - A(j+1:n, 1:j-1)*v(j, 1:j-1)')/A(j, j);
    %A(j, j)代表了d(j), A(j+1:n, j)代表了L（i, j）
end
L = tril(A, -1) + eye(n);, U = diag(diag(A))*L';
```

```
    for i = 1:n
        y(i) = B(i) - L(i, 1:i-1)*y(1: i-1);
    end
    for j = n:-1:1
        x(j) = (y(j) - U(j, j+1:n)*x(j+1:n))/U(j, j);
    end
end
```

(3)QR分解法

由于已经知道系数阵的各阶顺序主子式不为零，故可知系数阵非奇异，满足QR分解法的适用条件，得到的解如下表所示：

1	2	3	4	5	6	7	8	9	10
2.0933	2.1845	2.2735	2.3603	2.4448	2.5268	2.6064	2.6834	2.7577	2.8293
2.8981	2.9641	3.0271	3.0872	3.1442	3.1981	3.2489	3.2964	3.3407	3.3817
3.4193	3.4536	3.4845	3.5119	3.5359	3.5563	3.5733	3.5867	3.5966	3.603
3.6058	3.605	3.6007	3.5928	3.5814	3.5664	3.548	3.526	3.5005	3.4716
3.4393	3.4035	3.3645	3.322	3.2764	3.2274	3.1753	3.1201	3.0618	3.0004
2.9361	2.8689	2.7989	2.7261	2.6506	2.5725	2.4919	2.4088	2.3233	2.2355
2.1455	2.0534	1.9593	1.8633	1.7654	1.6657	1.5645	1.4616	1.3574	1.2518
1.1449	1.037	0.92798	0.81808	0.70736	0.59595	0.48395	0.37147	0.25863	0.14553
0.032284	-0.08099	-0.19419	-0.30719	-0.41989	-0.53217	-0.64393	-0.75506	-0.86544	-0.97496
-1.0835	-1.191	-1.2973	-1.4024	-1.506	-1.6082	-1.7088	-1.8077	-1.9048	

其中第i行第k列对应着第x_{ik}个值

QR分解法代码如下图：

```
function x = qrfact(A, B)
n =length(A);
Q = eye(n);    %生成单位矩阵
R = zeros(n);
for j = 1:n
    if j < n
        [v, beta] = Householder(A(j:n, j));    %不断的对A矩阵做householder变换，但不断向右下角缩小
    else                                       %这里是第一次缩短计算时间，因为将v(1)化1
        v = 1;
        beta = 2 - 2*mod(n, 2);
    end
    A(j:n, j:n) = (eye(n -j +1) - beta*v*v')*A(j:n, j:n);
    %储存R，将R储存在A的右上角,并且由于H矩阵的性质（左上角1：j-1为单位阵）只用储存j:n的部分即可，其余部分在后续计算中不改变
    d(j) = beta;
    if j < n
        A(j + 1:n, j) = v(2: n - j + 1);
    %储存Q,通过储存v的方式，由于v对应的向量第一位一直为1，因此只储存后面的，刚好储存在A的左下角（不算对角线）
    end
```

```
    end
    R = triu(A);     %从储存的A中获取R
    for k = 1:n
        H = eye(n);
        H1 = eye(n - k +1) - d(k)*[1, A(k+1:n, k)']'*[1, A(k+1:n, k)'];
        %计算Q 因为Q = H1*H2...*Hn，因此首先计算每个H，再乘到一起
        H(k:n, k:n)=H1;
        %每个H1只是n*n矩阵的右下角的一部分（详细见Householder变换）
        Q = Q*H;
    end
    x = zeros(n, 1);
    % Hn*...H2*H1*A = R, 所以Q = H1*H2...*Hn（H为正交矩阵）
    B = Q'*B;
    for j = n:-1:1
      x(j) = (B(j) - sum(R(j, j+1:n).*x(j+1:n)'))'/R(j, j);
    end
end
```

其中所使用的householder变换代码如下：

```
function [v, beta] = Householder(x)
n=length(x);
eta = norm(x, inf);
x = x/eta;
sigma = x(2: n)'*x(2: n);
v = x;
v(1) = 1;
if sigma == 0
    beta = 0;
else
      alpha = sqrt(x(1)^2 + sigma);
    if x(1) ≤ 0
        v(1) = x(1) - alpha;
    else
        v(1) = -sigma/(x(1)+ alpha);
    end
      beta = 2*v(1)^2/(sigma + v(1)^2);
      v = v/v(1);
end
end
```

(4)追赶法

由于系数阵为三对角稀疏型方阵，故满足追赶发的适用条件，得到的解如下表所示：

1	2	3	4	5	6	7	8	9	10
2.0933	2.1845	2.2735	2.3603	2.4448	2.5268	2.6064	2.6834	2.7577	2.8293
2.8981	2.9641	3.0271	3.0872	3.1442	3.1981	3.2489	3.2964	3.3407	3.3817
3.4193	3.4536	3.4845	3.5119	3.5359	3.5563	3.5733	3.5867	3.5966	3.603
3.6058	3.605	3.6007	3.5928	3.5814	3.5664	3.548	3.526	3.5005	3.4716
3.4393	3.4035	3.3645	3.322	3.2764	3.2274	3.1753	3.1201	3.0618	3.0004
2.9361	2.8689	2.7989	2.7261	2.6506	2.5725	2.4919	2.4088	2.3233	2.2355
2.1455	2.0534	1.9593	1.8633	1.7654	1.6657	1.5645	1.4616	1.3574	1.2518
1.1449	1.037	0.92798	0.81808	0.70736	0.59595	0.48395	0.37147	0.25863	0.14553
0.032284	-0.08099	-0.19419	-0.30719	-0.41989	-0.53217	-0.64393	-0.75506	-0.86544	-0.97496
-1.0835	-1.191	-1.2973	-1.4024	-1.506	-1.6082	-1.7088	-1.8077	-1.9048	

其中第i行第k列对应着第x_{ik}个值
追赶法代码如下图：

```
function x = Thomas(a, b, c, d)
n = length(b);
f(1) = c(1)/b(1);
g(1) = d(1)/b(1);
for i = 2:n-1          %由于f只到n-1，所以在这里范围到n-1， g（n）单独算
    h(i) = b(i) -f(i-1)*a(i - 1);
    f(i) = c(i)/h(i);
    g(i) = (d(i) - g(i-1)*a(i-1))/h(i);
end
g(n) = (d(n) - g(n-1)*a(n-1))/(b(n) - f(n-1)*a(n-1));
x(n) = g(n);
for i = n-1:-1:1
    x(i) = g(i) - f(i)*x(i+1);
end
end
```

STEP THREE:四组逼近解的计算误差
由常微分的知识我们可以知道该边值问题的精确解为：$y(x) = 2\cos(\pi x) + 3\sin(\pi x)$
下面我们设$yreal = (y(x_1), \ldots, y(x_{99}))^{\mathrm{T}}$，　err $= \max_{1\le i\le 99} |y(x_i) - x_i|$
则计算误差的代码为

```
err = max(abs(yreal - x))
```

四种计算方法的误差如下表所示：

表 1: 四种计算方法的误差

	Doolitle	改进的cholesky	QR分解法	追赶法
误差值	0.0004	0.0004	0.0004	0.0004

STEP FOUR:四组计算方法的计算时间

以下四种方法的计算时间皆取五次计算时间的平均值，其代码如下所示(只展示Doolitle分解法，其余同理)：

```
tic
x = Doolitle(A, B);
toc
```

四种计算方法的计算时间如下表所示：

表 2: 四种计算方法的计算时间

	Doolitle	改进的cholesky	QR分解法	追赶法
计算时间	0.0276	0.0049	0.0219	0.0034

STEP FRIVE:比较计算效率:

首先，根据STEP THREE中所得四组逼近解的计算误差，由于小数点再往后的部分对于误差估计并无帮助，而在误差估计中起主要作用的部分四者完全相同，所以我们有充分的理由认为这四种算法在本题中精度一致。

其次，在计算时间上追赶法最快，改进的cholesky分解法其次，Doolitle分解法最慢，这是由于本题所解的线性方程组为三对角型，因此追赶法最具有优势。

因此我们可以认为在本题中追赶法效率最高，改进的cholesky分解法次之，Doolitle分解法效率最低。

二、实验二：迭代法求解线性方程组

1.题目：

$$\begin{cases} x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 = 55 \\ -2x_1 + 3x_2 + 4x_3 + 5x_4 + 6x_5 = 66 \\ -3x_1 - 4x_2 + 5x_3 + 6x_4 + 7x_5 = 63 \\ -4x_1 - 5x_2 - 6x_3 + 7x_4 + 8x_5 = 36 \\ -5x_1 - 6x_2 - 7x_3 - 8x_4 + 9x_5 = -25 \end{cases} \tag{5}$$

试以 $\left\|X^{(k+1)} - X^{(k)}\right\|_\infty < 10^{-5}$ 作为终止准则, 分别利用 Jacobi 迭代法、Gauss - Seidel 迭代法及具最佳松弛因子的 JOR 迭代法和 SOR 迭代法求解该方程组, 并比较这些方法的计算精度和计算时间。

解：

STEP ONE: Jacobi迭代法与JOR迭代法

由于JOR迭代法为Jacobi迭代法的优化情况，两者具有较高的相似性，因此本文将两个迭代算法放于同一个matlab函数代码中。

首先是Jacobi迭代法，将对应的矩阵带入程序中我们发现其输出结果为：'Jacobi方法不

收敛’ 因此Jacobi迭代法对应的迭代矩阵的谱半径大于一，所以不收敛，因此Jacobi迭代法不适用。再进行正则化后仍不能使用，因为谱半径的值为1.8604，大于1

下面我们再使用JOR迭代法，将对应的矩阵带入程序中我们发现其输出结果为：’请使用正交化对矩阵进行处理’，因此该矩阵存在不为实数的特征值，我们需要使用如下等式进行处理，使其具有实特征值：

$$A^{\mathrm{T}}AX = A^{\mathrm{T}}b \tag{6}$$

易知此时的Jacobi迭代阵为对称阵。我们再次带入程序中我们发现其输出结果为:’不存在最佳松弛因子，请输入松弛因子使JOR方法收敛’，这说明迭代矩阵特征值大于一，因此不存在最佳松弛因子。

由于我们已知迭代矩阵的谱半径越小收敛速度越快，因此我们在选择松弛因子时应当使迭代矩阵谱半径尽可能的小，因此在保留小数点后四位的条件下，我们选择松弛因子的值为ω 为0.6864(该值为理论最佳松弛因子的值)，其结果如下：

$$x = [0.9999900234, 2.0000028318, 3.0000002639, 4.0000004433, 4.9999966643]^T$$

迭代次数为：323次，误差值为1.4502×10^{-5}

Jacobi迭代法和JOR迭代法对应的代码如下所示：

```
function [x, n, err] = Jacobi(A, b, p, k, N, wp, r)
%n 为迭代次数， x 为最终的迭代值 err为误差值
%A 为线性方程组，b为方程式右端值， p为收敛条件
%k为迭代方法的选择，如果k=1则使用Jacobi方法，若k = 2则使用JOR迭代法
%N为矩阵范数的选择，输入1，2，Inf分别代表1范数，2范数与无穷范数
%r为精确解，用来计算误差,如果没有精确值则不输入
%wp为如果没有最佳松弛因子时自己选择输入的松弛因子
%初始解始终设为  x1 = 0
  U = triu(A, 1);      %提出上三角
  L = tril(A, -1);      %提出下三角
  d = diag(A);      %提取对角线
  D = diag(d);      %组成矩阵
  n = length(A);
  B = -D\(L + U);     %求出迭代矩阵B
  TB = max(abs(eig(B)));    %求出谱半径
  if k == 1
      disp('使用Jacobi方法')
      if TB < 1
         disp('Jacobi方法收敛')
         x1 = zeros(n,1);      %构造初始值
         x2 = zeros(n, 1) + 0.5;
         n = 0;                %记录迭代次数
         if exist('r', 'var')                %根据是否有精确解来计算误差，如果没有则根据后验误差计算
            while norm((r - x1), N) ≥ p    %判断收敛条件
               x2 = x1;
```

```
                x1 = B*x2 + D\b;
                n = n + 1;
            end
            err = norm(x1 - r);      %计算误差值，统一采用矩阵二范数计算
        else
            while norm((x2 - x1), N) ≥ p    %判断收敛条件
                x2 = x1;
                x1 = B*x2 + D\b;
                n = n + 1;
            end
            err = norm(x1 - x2);      %计算误差值，统一采用矩阵二范数计算
        end
        x = x1;
    else
        disp('Jacobi方法不收敛')
    end
elseif k == 2
    disp('使用JOR方法')
    if isreal(eig(B)) && TB < 1
        disp('JOR方法存在最佳松弛因子')
        MB = max(eig(B));      %计算松弛因子
        mB = min(eig(B));
        w = 2/(2 - MB - mB);
        BJ = eye(n) - w*(D\A);   %计算JOR迭代中的B矩阵
        if max(abs(eig(BJ))) < 1    %判断JOR方法是否收敛
            disp('JOR迭代法收敛')
            x1 = zeros(n,1);      %构造初始值
            x2 = zeros(n, 1) + 1;
            n = 0;                %记录迭代次数
            if exist('r', 'var')              %根据是否有精确解来计算误差，如果
                没有则根据后验误差计算
                while norm((r - x1), N) ≥ p    %判断收敛条件
                    x2 = x1;
                    x1 = BJ*x2 + w*(D\b);
                    n = n + 1;
                end
                err = norm(x1 - r);      %计算误差值，统一采用矩阵二范数计算
            else
                while norm((x2 - x1), N) ≥ p
                    x2 = x1;
                    x1 = BJ*x2 + w*(D\b);
                    n = n + 1;
                end
                err = norm(x1 - x2);
            end
            x = x1;
        end
    elseif isreal(eig(B)) == 0
        disp('请使用正交化对矩阵进行处理')
    elseif  isreal(eig(B)) && TB > 1            %需要注意的是对于使用A进行迭代
```

```
            还是A'*A进行迭代迭代次数会有显著的变化
75          disp('不存在最佳松弛因子，请输入松弛因子使JOR方法收敛')
76          if all(eig(A)>0) && all(eig(2*D/wp-A)>0)
77              disp('松弛因子选择合理，JOR方法收敛')
78              BJ = eye(n) - wp*(D\A);   %计算无最佳松弛因子的JOR迭代中的迭代矩阵
79              x1 = zeros(n,1);      %构造初始值
80              x2 = zeros(n, 1) + 1;
81              n = 0;                %记录迭代次数
82              if exist('r', 'var')              %根据是否有精确解来计算误差，如果
                    没有则根据后验误差计算
83                  while norm((r - x1), N) ≥ p    %判断收敛条件
84                      x2 = x1;
85                      x1 = BJ*x2 + wp*(D\b);
86                      n = n + 1;
87                  end
88                  err = norm(x1 - r);      %计算误差值，统一采用矩阵二范数计算
89              else
90                  while norm((x2 - x1), N) ≥ p
91                      x2 = x1;
92                      x1 = BJ*x2 + wp*(D\b);
93                      n = n + 1;
94                  end
95                  err = norm(x1 - x2);
96              end
97              x = x1;
98          else
99              disp('松弛因子选择不合理，JOR方法不收敛,请从新选择')
100             return
101         end
102     end
103   end
104 end
```

STEP TWO: Gauss-Seidel迭代法于SOR迭代法

由于SOR迭代法为Gauss-Seidel迭代法的优化情况，两者具有较高的相似性，因此本文将两个迭代算法放于同一个matlab函数代码中。

首先是Gauss-Seidel迭代法，将对应的矩阵带入程序中我们发现其输出结果为：'Gauss方法不收敛' 因此Gauss-Seidel迭代法对应的迭代矩阵的谱半径大于一，所以不收敛，因此Gauss-Seidel迭代法不适用。

但如果对矩阵进行正则化后，Gauss-Seidel迭代法可以使用,其迭代矩阵的谱半径为0.8882<1,其迭代结果为：

$$x = [0.9999226117542, 2.000012545663, 3.000003400859, 4.000005781968, 4.999971529132]^T$$

迭代次数为：97次，误差值为1.0537×10^{-5}

下面我们再使用SOR迭代法，将对应的矩阵带入程序中我们发现其输出结果为：'请使用正交化对矩阵进行处理'，因此该矩阵存在不为实数的特征值，我们需要使用如下等式进行处理，使其具有实特征值：

$$A^{\mathrm{T}}AX = A^{\mathrm{T}}b \tag{7}$$

易知此时的Jacobi迭代阵为对称阵。我们再次带入程序中我们发现其输出结果为:'不存在最佳松弛因子，请输入松弛因子使SOR方法收敛'，这说明迭代矩阵特征值大于一，因此不存在最佳松弛因子。并且最佳松弛因子也无法计算，因此SOR方法无法完成。

但如果我们不使用最佳松弛因子!,由于我们已知迭代矩阵的谱半径越小收敛速度越快，因此我们在选择松弛因子时应当使迭代矩阵谱半径尽可能的小，因此在保留小数点后四位的条件下，我们选择松弛因子的值为ω 为1.3056，其结果如下：

$$x = [0.9999900064, 2.0000027946, 3.0000002743, 4.0000004519, 4.9999966478]^T$$

迭代次数为：40次，误差值为1.4434×10^{-5}

```
function [x, n, err] = Gauss(A, b, p, k, N, wp, r)
%n 为迭代次数， x 为最终的迭代值 err为误差值
%A 为线性方程组，b为方程式右端值， p为收敛条件
%k为迭代方法的选择，如果k=1则使用Gauss方法，若k = 2则使用SOR迭代法
%N为矩阵范数的选择，输入1，2，Inf分别代表1范数，2范数与无穷范数
%r为精确解，用来计算误差,如果没有精确值则不输入
%wp为如果没有最佳松弛因子时自己选择输入的松弛因子
%初始解始终设为  x1 = 0
  U = triu(A, 1);      %提出上三角
  L = tril(A, -1);      %提出下三角
  d = diag(A);      %提取对角线
  D = diag(d);      %组成矩阵
  n = length(A);
  Bjacobi = -D\(L + U);     %求出Jacobi迭代矩阵B
  TB = max(abs(eig(Bjacobi)));    %求出Jacobi谱半径
  Bgauss = -(D + L)\U;        %求出gauss迭代矩阵B
  GB = max(abs(eig(Bgauss)));    %求出gauss谱半径
  if k == 1
      disp('使用Gauss方法')
      if GB < 1
          disp('Gauss方法收敛')
          x1 = zeros(n,1);      %构造初始值
          x2 = zeros(n, 1) + 0.5;
          n = 0;                %记录迭代次数
          if exist('r', 'var')                %根据是否有精确解来计算误差，如果没有则根据后验误差计算
              while norm((r - x1), N) ≥ p    %判断收敛条件
                  x2 = x1;
```

```
                x1 = Bgauss*x2 + (D + L)\b;
                n = n + 1;
            end
            err = norm(x1 - r);      %计算误差值，统一采用矩阵二范数计算
        else
            while norm((x2 - x1), N) ≥ p    %判断收敛条件
                x2 = x1;
                x1 = Bgauss*x2 + (D + L)\b;
                n = n + 1;
            end
            err = norm(x1 - x2);      %计算误差值，统一采用矩阵二范数计算
        end
        x = x1;
    else
        disp('Gauss方法不收敛')
    end
elseif k == 2
    disp('使用SOR方法')
    if isreal(eig(Bjacobi)) && TB < 1
        disp('SOR方法存在最佳松弛因子')
        w = 2/(1 + sqrt(1 - max(abs(eig(-inv(D)*(L+U))))^2));   %计算松弛因子
        BS = (D + w*L)\((1 - w)*D - w*U);   %计算SOR迭代中的B矩阵
        if max(abs(eig(BS))) < 1    %判断SOR方法是否收敛
            disp('SOR迭代法收敛')
            x1 = zeros(n,1);      %构造初始值
            x2 = zeros(n, 1) + 1;
            n = 0;                %记录迭代次数
            if exist('r', 'var')                %根据是否有精确解来计算误差，如果没有则根据后验误差计算
                while norm((r - x1), N) ≥ p    %判断收敛条件
                    x2 = x1;
                    x1 = BS*x2 + (D + w*L)\(w*b);
                    n = n + 1;
                end
                err = norm(x1 - r);      %计算误差值，统一采用矩阵二范数计算
            else
                while norm((x2 - x1), N) ≥ p
                    x2 = x1;
                    x1 = BS*x2 + (D + w*L)\(w*b);
                    n = n + 1;
                end
                err = norm(x1 - x2);
            end
            x = x1;
        end
    elseif isreal(eig(Bjacobi)) == 0
        disp('请使用正交化对矩阵进行处理')
    elseif isreal(eig(Bjacobi)) &&  TB > 1                          %需要注意的是对于使用A进行迭代还是A'*A进行迭代迭代次数会有显著的变化
        disp('不存在最佳松弛因子，请输入松弛因子使SOR方法收敛')
```

```
            if wp > 0 && wp < 2
                disp('松弛因子选择合理，SOR方法收敛')
                BS = (D + wp*L)\((1 - wp)*D - wp*U);   %计算无最佳松弛因子的SOR迭代中的迭代矩阵
                x1 = zeros(n,1);      %构造初始值
                x2 = zeros(n, 1) + 1;
                n = 0;                %记录迭代次数
                if exist('r', 'var')              %根据是否有精确解来计算误差，如果没有则根据后验误差计算
                    while norm((r - x1), N) ≥ p    %判断收敛条件
                        x2 = x1;
                        x1 = BS*x2 + (D + wp*L)\(wp*b);
                        n = n + 1;
                    end
                    err = norm(x1 - r);      %计算误差值，统一采用矩阵二范数计算
                else
                    while norm((x2 - x1), N) ≥ p
                        x2 = x1;
                        x1 = BS*x2 + (D + wp*L)\(wp*b);
                        n = n + 1;
                    end
                    err = norm(x1 - x2);
                end
                x = x1;
            else
                disp('松弛因子选择不合理，SOR方法不收敛,请从新选择')
                return
            end
        end
    end
end
```

STEP THREE:计算时间

计算时间皆取五次计算时间的平均值

$\omega = 0.6864$ 的 JOR方法：0.002327秒

Gauss迭代方法：0.001625秒

$\omega = 1.3056$ 的 SOR方法：0.001280秒

综上所述，JOR迭代法误差略小于SOR迭代法，但所需时间与迭代次数均大于SOR迭代法，因此我们可以认为SOR迭代法效果更好。并且Gauss迭代法也优于JOR方法，其原因是JOR的松弛因子选择不恰当。

三、实验三:使用Krylov子空间方法求解线性方程组

1.题目：

试分别用最速下降法、共轭梯度法及预优共轭梯度法求解线性方程组 AX = b, 其中系数阵 $A = (a_{ij}) \in R^{50\times 50}$ 及向量 $b = [b_1, b_2, \ldots, b_{50}]^{\mathrm{T}}$ 的元素由下式确定:

$$a_{ij} = \begin{cases} \max\{i, j\}, & i \neq j, \\ 50i, & i = j; \end{cases} \qquad b_i = \sum_{j=1}^{50} a_{ij}(51 - j)$$

要求迭代解 X_k 的精度满足 $\|b - AX_k\|_2 < 10^{-8}$

其中预优矩阵给定为 $M = \mathrm{diag}\,(a_{11}, \ldots, a_{50})$

解:

STEP ONE：计算A, b的显式值

其计算代码如下所示：

```
%生成矩阵A
A = zeros(50,50);
for i = 1:50
    for j = 1:50
        if i==j
            A(i,j) = 50*i;
        else
            A(i,j) = max(i, j);
        end
    end
end
%生成向量b
b = zeros(50, 1);
for i = 1:50
    for j = 1:50
        b(i) = b(i) + A(i,j)*(51 - j);
    end
end
```

STEP TWO:求解线性方程组

首先我们需要判定矩阵A是否为对称正定的，由于从A的生成过程中我们可以发现其对称显然，而通过matlab 中的chol函数我们可以判定A为正定的。因此这三种方法均可使用。

(1)最速下降法

最速下降法对应的代码如下所示：

```
 function [X, n] = sdescent(A, b, p, N)
 %n 为迭代次数， X 为最终的迭代值
 %A 为线性方程组，b为方程式右端值， p为收敛条件
 %N为矩阵范数的选择，输入1，2，Inf分别代表1范数，2范数与无穷范数
     n = length(b);
     X0 = zeros(n, 1);
     r0 = b - A*X0;
     n = 0;
```

```
    while norm(r0, N) ≥ p
        n = n + 1;
        alpha0 = r0'*r0/(r0'*A*r0);
        X1 = X0 + alpha0*r0;
        r0 = b - A*X1;
        X0 = X1;
    end
    X = X0;
end
```

方程的解如下表所示：

表 3: 最速下降法的计算结果

1	2	3	4	5
49.9999999997595	48.9999999999374	47.9999999999663	46.9999999999784	45.9999999999850
44.9999999999892	43.9999999999922	42.9999999999943	41.9999999999960	40.9999999999973
39.9999999999984	38.9999999999993	38.0000000000000	37.0000000000006	36.0000000000012
35.0000000000017	34.0000000000021	33.0000000000024	32.0000000000028	31.0000000000031
30.0000000000033	29.0000000000035	28.0000000000038	27.0000000000039	26.0000000000041
25.0000000000043	24.0000000000044	23.0000000000045	22.0000000000046	21.0000000000048
20.0000000000049	19.0000000000049	18.0000000000050	17.0000000000051	16.0000000000052
15.0000000000052	14.0000000000053	13.0000000000053	12.0000000000054	11.0000000000054
10.0000000000055	9.00000000000552	8.00000000000556	7.00000000000562	6.00000000000562
5.00000000000566	4.00000000000570	3.00000000000573	2.00000000000576	1.00000000000581

注：结果保留15位有效数字

其迭代次数为1439次

(2)共轭梯度法

共轭梯度法对应的代码如下所示：

```
function [X, n] = congrad(A, b, p, N)
%n 为迭代次数， X 为最终的迭代值
%A 为线性方程组，b为方程式右端值， p为收敛条件
%N为矩阵范数的选择，输入1，2，Inf分别代表1范数，2范数与无穷范数
    n = length(b);
    X0 = zeros(n, 1);
    r0 = b - A*X0;
    P0 = r0;
    n = 0;
    while norm(r0, N) ≥ p
        n = n + 1;
        alpha = r0'*r0/(P0'*A*P0);
        X1 = X0 + alpha*P0;
        r1 = r0 - alpha*A*P0;
        beta = r1'*r1/(r0'*r0);
        P1 = r1 + beta*P0;
        r0 = r1;
```

```
18            X0 = X1;
19            P0 = P1;
20        end
21        X = X0;
22    end
```

方程的解如下表所示：

表 4: 共轭梯度法的计算结果

1	2	3	4	5
50.0000000000000	49.0000000000000	48.0000000000000	47.0000000000000	46.0000000000000
45.0000000000000	44.0000000000000	43.0000000000000	42.0000000000000	41.0000000000000
40.0000000000000	39.0000000000001	38.0000000000000	37.0000000000001	35.9999999999999
35.0000000000002	33.9999999999998	33.0000000000004	31.9999999999995	31.0000000000007
29.9999999999992	29.0000000000008	27.9999999999994	27.0000000000004	25.9999999999999
24.9999999999998	24.0000000000004	22.9999999999994	22.0000000000006	20.9999999999995
20.0000000000004	18.9999999999997	18.0000000000002	16.9999999999999	16.0000000000001
15.0000000000000	14.0000000000000	13.0000000000000	12.0000000000000	11.0000000000000
10.0000000000000	9.00000000000001	8.00000000000002	7.00000000000002	6.00000000000002
5.00000000000002	4.00000000000003	3.00000000000003	2.00000000000002	1.00000000000001

注：结果保留15位有效数字
其迭代次数为49次，迭代次数大幅下降

(3)预优共轭梯度法
此处预优矩阵$M = \mathrm{diag}\,(a_{11}, \ldots, a_{50})$已给定，因此直接带入计算得解如下：
其计算代码如下所示：

```
1   function [X, n] = precongrad(A, b, M, p, N)
2   %n 为迭代次数， X 为最终的迭代值
3   %A 为线性方程组，b为方程式右端值， p为收敛条件 M为预优矩阵
4   %N为矩阵范数的选择，输入1，2，Inf分别代表1范数，2范数与无穷范数
5       n = length(b);
6       X0 = zeros(n, 1);
7       r0 = b - A*X0;
8       zeta0 = M\r0;
9       rho0 = r0'*zeta0;
10      P0 = zeta0;
11      while norm(r0, N) ≥ p
12          n = n + 1;
13          omega = A*P0;
14          alpha = rho0/(P0'*omega);
15          X0 = X0 + alpha*P0;
16          r0 = r0 - alpha*omega;
17          zeta1 = M\r0;
18          rho1 = r0'*zeta1;
19          beta = rho1/rho0;
```

```
            P0 = zeta1 + beta*P0;
            rho0 = rho1;
        end
        X = X0;
    end
```

方程的解如下表所示：

表 5: 共轭梯度法的计算结果

1	2	3	4	5
49.9999999999997	48.9999999999998	48.0000000000000	47.0000000000000	46.0000000000000
45.0000000000000	43.9999999999999	43.0000000000000	41.9999999999999	41.0000000000000
40.0000000000000	39.0000000000000	38.0000000000000	37.0000000000000	36.0000000000000
35.0000000000000	34.0000000000000	33.0000000000000	32.0000000000000	31.0000000000000
30.0000000000000	29.0000000000000	28.0000000000000	27.0000000000000	26.0000000000000
25.0000000000000	24.0000000000000	23.000000000000	22.0000000000000	21.0000000000000
20.0000000000000	19.0000000000000	18.0000000000000	17.0000000000000	16.0000000000000
15.0000000000000	14.0000000000000	13.0000000000000	12.0000000000000	11.0000000000000
10.0000000000000	9.00000000000001	8.00000000000003	7.00000000000002	6.00000000000003
5.00000000000002	4.00000000000000	3.00000000000000	2.00000000000001	1.00000000000001

注：结果保留15位有效数字

其迭代次数为60次

四、实验四:非线性方程组问题

1.题目：

设有非线性方程组

$$\begin{cases} 6x - 2\cos(yz) - 1 = 0, \\ \sqrt{x^2 + \sin z + 1.06} - 9(y + 0.1) = 0, \\ 3\exp(-xy) + 60z + 10\pi - 3 = 0. \end{cases} \tag{8}$$

试分别应用Picard迭代法、Picard加速迭代法及Newton迭代法求解该方程,要求精确到$\|X_k - X_{k-1}\|_\infty < 10^{-8}$，并比较三者的计算效率。

解：

STEP ONE：证明该方程组有唯一解

首先，原方程组等价于如下形式：

$$\begin{cases} \frac{1}{3}\cos(yz) + \frac{1}{6} = x, \\ \frac{1}{9}\sqrt{x^2 + \sin z + 1.06} - 0.1 = y, \\ -\frac{1}{20}\exp(-xy) - \frac{1}{6}\pi + \frac{1}{20} = z. \end{cases} \tag{9}$$

记$X = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$, $\phi(X) = \begin{pmatrix} \frac{1}{3}\cos(yz) + \frac{1}{6} \\ \frac{1}{9}\sqrt{x^2 + \sin z + 1.06} - 0.1 \\ -\frac{1}{20}\exp(-xy) - \frac{1}{6}\pi + \frac{1}{20} \end{pmatrix}$ 则(9)等价于$X = \phi(X)$ 显然, $\phi(X)$在

闭凸集 $D_0 = \{X \in R^3 \| X\|_\infty \leq 1\}$上是连续可微的, 且有 $\|X\|_\infty$的值为0.5236 其小于1，此即表明有$\phi(D_0) \in D_0$

又有$\phi'(X) = \begin{pmatrix} 0 & -\frac{z}{3}\sin(yz) & -\frac{y}{3}\sin(yz) \\ \frac{x}{9\sqrt{x^2+\sin z+1.06}} & 0 & \frac{\cos z}{18\sqrt{x^2+\sin z+1.06}} \\ \frac{y}{20}\exp(-xy) & \frac{x}{20}\exp(-xy) & 0 \end{pmatrix}$, 则$\forall X \in D_0$ 我们有$\|\phi'(X)\|_\infty$的值为0.1293其小于1

故据定理6.2，原方程组有唯一不动点$X^* \in D_0$, 迭代格式$X_{k+1} = \phi(X_k), k = 0, 1, 2, \ldots$ 自任意初始值$X_0 \in D_0$出发收敛于该不动点。

STEP TWO:应用Picard迭代法求解

由STEP ONE 我们知Picard迭代法收敛，因此取初值为$X_0 = (0, 0, 0)^T$ 求得解为

$X = (0.500000000000000, 0.000000000033196, -0.523598775598299)^T$,迭代次数k为7

具体迭代代码如下所示：

```
function [X, k] = Picard(A, b, N, L, p, xnew, ynew, znew)
%X 表示最终输出的解 k表示迭代次数
%A表示非线性方程组左端，b表示非线性方程组左端， N表示使用的矩阵范数（在最开始检验时均使用矩阵无穷范数）p为误差
%L表示定义域的范围
%在输入参数时，由于矩阵A中带有x，因此需要先输入"syms xnew real" 从而对x进行定义，否则报错，其余参数同理(real 防止转置后出现复数)
%在输入变量时，格外建议参数使用形如"xnew"的形式，避免后续程序因为变量重复而出错
%由于本人能力有限，在变量数目增多时需要手动更改程序的几处与变量数目有关的地方（已经在程序中注明）
%如果后面的同学有更好的思路欢迎分享
n = length(b);      %确定方程有几个参数
%求Jacobi矩阵
J = sym(zeros(1, n));
F1 = sym(zeros(1, n));
F2 = sym(zeros(1, n));
S1 = zeros(1, n);          %构造第一个储存矩阵，用于储存A的每一行元素绝对值之和的最大值
S2 = zeros(1, n);          %构造第二个储存矩阵，用于储存A的Jacobi矩阵的每一行元素绝对值之和的最大值
for i = 1:n
    J(1, i) = sum(A(i,:));
    F1(1, i) = sum(abs(A(i,:)));          %构造对应的无穷范数矩阵，便于求无穷范数
end
AJocobi = jacobian(J, [xnew;ynew;znew]);          %注意xyz的数量要随着方程更改!!!
for i = 1:n
    F2(1, i) = sum(abs(AJocobi(i,:)));
end
%验证是否收敛
%A矩阵范数
for i = 1:n
    f = -F1(i);                          %取负值便于求最大值
    xm = sym(ones(1,n));                 %给符号变量赋初值
```

```
        for ii=1:n
            xm(ii)=['x' num2str(ii)];        %定义符号变量的形式为x1,...,xn
        end
        f = char(f);                         %转化为字符变量
        f = strrep(f,'xnew','x(1)');          %将x，y转化为向量的形式，便于后面求极值
        f = strrep(f,'ynew','x(2)');          %若有更多的变量使用相同的方法操作!!!
        f = strrep(f,'znew','x(3)');
        for iii = n:-1:1
            f = replace(f,['x' num2str(iii)],['x(' num2str(iii) ')']);
        end
        f = eval(['@(x)' char(f) ';']);
        f(1:n);                                                          %注意此处也要修改!!!
        lb = [-L,-L,-L]; ub = [L,L,L]; T = [];b = [];Aeq = [];beq = [];   %求极值函数的对应参数设置
        x0 = (lb + ub)/2;
        [xd, fval] = fmincon(f,x0,T,b,Aeq,beq,lb,ub);
        S1(i) = fval;                                                    %记录每一列绝对值之和的值
        f = 0;
    end
    %Ajacobi矩阵范数
    for i = 1:n
        f = -F2(i);                          %取负值便于求最大值
        xm = sym(ones(1,n));                  %给符号变量赋初值
        for ii=1:n
            xm(ii)=['x' num2str(ii)];        %定义符号变量的形式为x1,...,xn
        end
        f = char(f);                         %转化为字符变量
        f = strrep(f,'xnew','x(1)');          %将x，y转化为向量的形式，便于后面求极值
        f = strrep(f,'ynew','x(2)');          %若有更多的变量使用相同的方法操作!!!
        f = strrep(f,'znew','x(3)');
        for iii = n:-1:1
            f = replace(f,['x' num2str(iii)],['x(' num2str(iii) ')']);
        end
        f = eval(['@(x)' char(f) ';']);
        f(1:n);                                                          %注意此处也要修改!!!
        lb = [-L,-L,-L]; ub = [L,L,L]; T = [];b = [];Aeq = [];beq = [];   %求极值函数的对应参数设置
        x0 = (lb + ub)/2;
        [xd, fval] = fmincon(f,x0,T,b,Aeq,beq,lb,ub);
        S2(i) = fval;                                                    %记录每一列绝对值之和的值
        f = 0;
    end
    %开始计算
    if  max(abs(S1)) ≤ L && max(abs(S2)) ≤ 1
        disp('Picard迭代法收敛')
        k = 0;                                %记录迭代次数
        x1 = zeros(n, 1);
        x2 = ones(n, 1);
```

```
        X = zeros(n, 1);
        while norm(x2 - x1, N) ≥ p
            x2 = x1;
            xx = x2(1); yy = x2(2); zz = x2(3);    %注意修改变量个数!!!
            B = A;
            A = subs(A,xnew,xx);
            A = subs(A,ynew,yy);
            A = subs(A,znew,zz);
            for i = 1:n
               x1(i) = sum(A(i,:));
            end
            A = B;
            k = k + 1;
        end
        X = x1;
    else
        disp('Picard迭代法不收敛')
        return
    end
    end
```

STEP THREE:应用Picard加速迭代法求解首先需要计算相似矩阵P，由我们取得的初始向量$X_0 = (0,0,0)'$，得到$P = \phi'(X_0) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & \frac{1}{18\sqrt{1.06}} \\ 0 & 0 & 0 \end{pmatrix}$ 从而由加速迭代格式 $X_{k+1} = (I-P)^{-1}[\phi(X_k) - PX_k], k = 0,1,\ldots$ 得方程组的解:

$X = (0.500000000000000, 0.000000000004526, -0.523598775605672)^T$,迭代次数k为5

具体迭代代码如下所示：

```
%开始计算
if  max(abs(S1)) ≤ L && max(abs(S2)) ≤ 1
    disp('Picard加速方法收敛')
    k = 0;                                    %记录迭代次数
    x1 = zeros(n, 1);
    x2 = ones(n, 1);
    X = zeros(n, 1);
    PP = (eye(n) - P)\eye(n);
    while norm(x2 - x1, N) ≥ p
        x2 = x1;
        xx = x2(1); yy = x2(2); zz = x2(3);    %注意修改变量个数!!!
        B = A;
        A = subs(A,xnew,xx);
        A = subs(A,ynew,yy);
        A = subs(A,znew,zz);
        for i = 1:n
```

```
            x1(i) = sum(A(i,:));
        end
        x1 = PP * (x1 - P*x2);
        A = B;
        k = k + 1;
    end
    X = x1;
else
    disp('Picard加速方法不收敛')
    return
end
```

其中判断迭代法是否收敛的步骤与Picard迭代法相同，故没有呈现相关代码

STEP FOUR:应用Newton迭代法求解

我们首先首先记$F(X)=\begin{pmatrix}6x-2\cos(yz)-1\\ \sqrt{x^2+\sin z+1.06}-9(y+0.1)\\ 3\exp(-xy)+60z+10\pi-3\end{pmatrix}$，则原方程组等价于$F(X)=0$,而F(X)的雅可比矩阵为$F'(X)=\begin{pmatrix}6 & 2z\sin(yz) & 2y\sin(yz)\\ \frac{x}{\sqrt{x^2+\sin z+1.06}} & -9 & \frac{\cos z}{2\sqrt{x^2+\sin z+1.06}}\\ -3y\exp(-xy) & -3x\exp(-xy) & 60\end{pmatrix}$

选取初始迭代向量为$X=(0,0,0)^T$ 并且$F'(X_0)$为可逆矩阵，因此初始向量选择合理,得方程组的解:

$X=(0.499999998567772,0.000000000007295,-0.523598775598299)^T$,迭代次数k为4

具体迭代代码如下所示：

```
function [X, k] = Newton(F, N, n, p, xnew, ynew, znew)
%表示最终输出的解 k表示迭代次数
%F表示给定的非线性方程组,N表示使用的矩阵范数（在最开始检验时均使用矩阵无穷范数）p为误差 n为有几个未知变量
%在输入变量时，格外建议参数使用形如"xnew"的形式，避免后续程序因为变量重复而出错
%求Jacobi矩阵
J = sym(zeros(1, n));
for i = 1:n
    J(1, i) = sum(F(i,:));
end
FJocobi = jacobian(J, [xnew;ynew;znew]);            %注意xyz的数量要随着方程更改!!!
k = 0;                                    %记录迭代次数
x1 = zeros(n, 1);
x2 = ones(n, 1);
X = zeros(n, 1);
while norm(x2 - x1, N) ≥ p
     x2 = x1;
     xx = x2(1); yy = x2(2); zz = x2(3);    %注意修改变量个数!!!
     B = F;
```

```
19      F = subs(F,xnew,xx); F = subs(F,ynew,yy);F = subs(F,znew,zz);
20      BJocobi = FJocobi;
21      FJocobi = subs(FJocobi,xnew,xx); FJocobi = ...
            subs(FJocobi,ynew,yy);FJocobi = subs(FJocobi,znew,zz);
22      x1 = x2 - FJocobi\F;
23      x1 = double(x1);
24      F = B;
25      FJocobi= BJocobi;
26      k = k + 1;
27  end
28  X = x1;
29  end
```

再对Newton法进行优化改进后的代码如下

```
1   function [X, k] = Newton(F, N, n, p, xnew, ynew, znew)
2   %求Jacobi矩阵
3   %表示最终输出的解 k表示迭代次数
4   %F表示给定的非线性方程组,N表示使用的矩阵范数（在最开始检验时均使用矩阵无穷范数）p为误差
        n为有几个未知变量
5   %在输入变量时，格外建议参数使用形如"xnew"的形式，避免后续程序因为变量重复而出错
6   %求Jacobi矩阵
7   J = sym(zeros(1, n));
8   for i = 1:n
9       J(1, i) = sum(F(i,:));
10  end
11  FJocobi = jacobian(J, [xnew;ynew;znew]);     %注意xyz的数量要随着方程更改!!!
12  k = 0;                                       %记录迭代次数
13  x1 = zeros(n, 1);
14  x2 = ones(n, 1);
15  X = zeros(n, 1);
16  while norm(x2 - x1, N) ≥ p
17       x2 = x1;
18       xx = x2(1); yy = x2(2); zz = x2(3);    %注意修改变量个数!!!
19       B = F;
20       F = subs(F,xnew,xx); F = subs(F,ynew,yy);F = subs(F,znew,zz);
21       BJocobi = FJocobi;
22       FJocobi = subs(FJocobi,xnew,xx); FJocobi = ...
             subs(FJocobi,ynew,yy);FJocobi = subs(FJocobi,znew,zz);
23       FJocobi = double(FJocobi); F = double(-F);
24      [∆, c] = congrad(FJocobi, F, p, N);
25      x1 = x2 + ∆;
26      x1 = double(x1);
27      F = B;
28      FJocobi= BJocobi;
29      k = k + 1;
30  end
31  X = x1;
32  end
```

其中我们使用了预优共轭梯度法来提高计算效率

STEP FIVE:比较计算效率

三种迭代方法的迭代次数与迭代时间如下表所示(计算时间取5次平均)：

表 6: 迭代效率比较表

	Picard迭代	Picard加速迭代	Newton迭代
迭代次数	7	5	4
迭代时间	0.207656	0.158847	0.161437

从表中我们可以发现Newton迭代法的迭代次数最少，迭代速度比Picard加速迭代稍慢，因此我们发现在对Newton法进行改进后，其计算时间大幅度缩短，并且计算迭代次数少，因此我们可以认为Newton迭代法是比较优秀的迭代方法，计算效率较高。而Picard加速迭代法迭代次数只比Newton法多一步，计算时间稍微比Newton法快一些，因此我们也可以认为Picard加速迭代法也是很有计算效率的方法。