实验报告

—— 数值计算 Homework1

姓名：王迎旭
学号：16340226
班级：16 级软件工程教务三班
邮箱：469410930@qq.com
日期：2018.5.23

目录

Part 1

1.1 问题描述

一、请实现下述算法，求解线性方程组 Ax=b，其中 A 为 n × n 维的已知矩阵，b 为 n 维的已知向量，x 为 n 维的未知向量。
（1）高斯消去法。
（2）列主元消去法。
A 与 b 中的元素服从独立同分布的正态分布。令 n=10、50、100、200，测试计算时间并绘制曲线。

1.2 算法设计

1.2.1 条件约束

1.2.1.1 矩阵 A 与向量 b

```
%生成n*n的随机矩阵
A = normrnd(10,1,n,n) ;

%生成n*1的随机矩矩阵
b = normrnd(10,1,n,1) ;
```

1.2.1.2 保证 Ax = b 解的唯一性

```
%测试模块
    %保证Ax = B有唯一解
    B = [A b] ;
    Rank_A = rank(A) ;
    Rank_B = rank(B) ;
    if( Rank_A ~= Rank_B )
        disp('Error , Rank_A != Rank_B') ;
        return ;
    end
%模块结束
```

1.2.1.3 实验重复次数

（注：进行 1000 次实验，保证程序运行的普遍性与非偶然性）

```
for count = 0 : 1 : 999
```

1.2.2 顺序高斯消去法

1.2.2.1 算法描述

将 Ax=b 按照从上至下、从左至右的顺序化为上三角方程组，中间过程不对矩阵进行交换，主要步骤如下：

Step 1

将第2行至第n行，每行分别与第一行做运算，消掉每行第一个参数。公式如：$a_{11}^{(1)} \neq 0$，$l_{i1} = \frac{a_{i1}}{a_{11}^{(1)}}(i = 2:n)$，第$i$行+$(-l_{i1})$×第1行($i$=2:n)

形成如下图所示新矩阵：

$$B^{(1)} \Rightarrow \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} & b_n^{(2)} \end{pmatrix} \triangleq B^{(2)}$$

Step 2

从新矩阵的a22开始（a22不能为0），以第二行为基准，将第三行至第n行分别与第二行做运算，消掉每行第二个参数。

公式如：$l_{i2}=\frac{a_{i2}^{(2)}}{a_{22}^{(2)}}(i=3{:}n)$，第$i$行$+(-l_{i2})\times$第2行$(i=3{:}n)$，形成如下图所示新矩阵：

$$B^{(2)}\Rightarrow\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} & b_3^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & a_{n3}^{(3)} & \cdots & a_{nn}^{(3)} & b_n^{(3)} \end{pmatrix}\triangleq B^{(3)}$$

Step k

设$a_{kk}^{(k)}\neq 0$，令$l_{ik}=\frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}(i=k+1\sim n)$

按照上述方法，当第k步运算时，公式为：第i行$+(-l_{ik})\times$第k行$(i=k+1\sim n)$

运算前后的矩阵为：

$$B^{(k)}=\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1k}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2k}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ \vdots & \vdots & & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} & b_k^{(k)} \\ \vdots & \vdots & \ddots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & a_{nk}^{(k)} & \cdots & a_{nn}^{(2)} & b_n^{(k)} \end{pmatrix}\Rightarrow$$

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1k}^{(1)} & a_{1,k+1}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2k}^{(2)} & a_{2,k+1}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & & \vdots & \vdots \\ 0 & 0 & \cdots & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{kn}^{(k)} & b_k^{(k)} \\ 0 & 0 & \cdots & 0 & a_{k+1,k+1}^{(k+1)} & \cdots & a_{k+1,n}^{(k+1)} & b_{k+1}^{(k+1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & a_{n,k+1}^{(k+1)} & \cdots & a_{nn}^{(k+1)} & b_n^{(k+1)} \end{pmatrix}\triangleq B^{(k+1)}$$

Step n-1

经过 n-1 步，方程组也就转化为了我们希望得到的上三角方程组，如下：

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots \\ & & & a_{nn}^{(n)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(n)} \end{bmatrix}$$

1.2.2.2 顺序高斯消去法 Matlab 程序

```
%顺序高斯消去
for k = 1 : n - 1
    for i = k + 1: n
        %判断Akk是否为零
        if(A(k,k) == 0)
            disp('Error , A(k,k) is 0 !') ;
            return ;
        end
        temp = A(i,k) / A(k,k) ;
        for j = k + 1 : n
            A(i,j) = A(i,j) - temp * A(k,j) ;
        end
        b(i) = b(i) - temp * b(k);
    end
end
%求出解集
```

1.2.3 列主元高斯消去法

1.2.3.1 算法描述

对比顺序高斯消去法，列主元高斯消去法在第 k 步消元前，找出 k 行下所有第 k 列元素最大的非零元素 a(pk)， 将第 p 行与第 k 行进行整行交换，这样既不影响原方程的解，也可以将绝对值最大的 a(pk)作为主元，放在处的位置上，尽可能减小引入误差。

全主元消除法与列主元消除法类似，只不过列主元消除法是从第 k 列中选取一个最大的元素，与第 k 行进行交换。 而全主元消除法释从第 k 行第 k 列开始的右下角矩阵中所有元素中选取一个最大的元素作为主元，同时交换 p 行与 q 列，从而保证稳定性。

1.2.3.2 列主元高斯消去法 Matlab 程序

```
for p=1:n-1
    %找出某一列主元最大的行
    t=find(abs(B(p:end,p))==max(abs(B(p:end,p))))+p-1;
    %如果该行不是最大主元所在的行
    if abs(B(t,p))~=abs(B(p,p))
        %完成当前行与最大的行交换
        l=B(t,:);
        if(t==5)
            B(t,:)=B(p,:);
            B(p,:)=l;
        end
    end
      %列主元判断
      for k=p+1:n
          if(B(p,p) == 0)
              disp('Error , A(k,k) is 0 !') ;
              return ;
          end
          temp = B(k,p) / B(p,p);
          %完成消元变化过程
          B(k,p:n+1) = B(k,p:n+1) - temp * B(p,p:n+1);
      end
  end
```

1.3 数值实验

1.3.1 顺序高斯消去法

1.3.1.1 顺序高斯消去法消元求值

注： ①为了方便展示结果使用 n = 10 进行运算
②x_1 是 Matlab 自带运算器 A\b 的运算结果
③x_2 是顺序高斯消去法求值结果

```
x_1 =

   -0.7738
    2.8762
   -1.2850
    0.1263
    0.3783
    0.4728
    1.7168
   -0.4116
   -2.1938
    0.0986
```

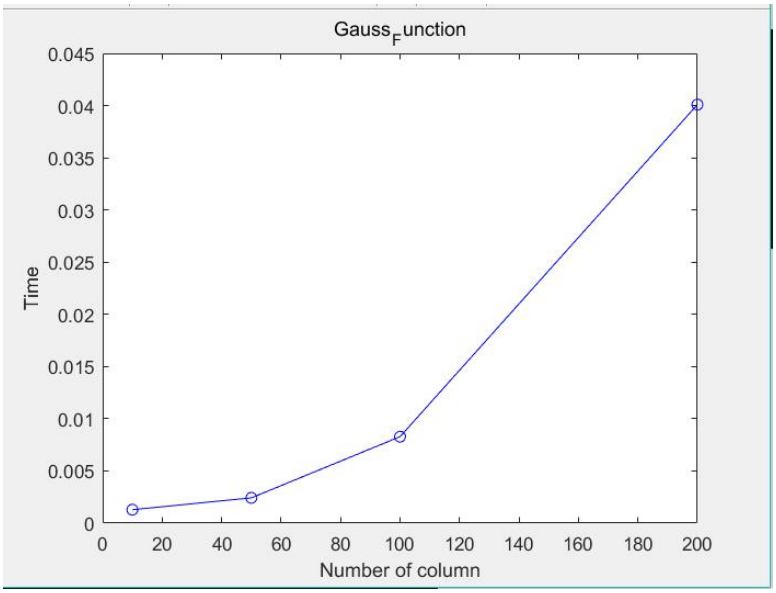
```
x_2 =

   -0.7738
    2.8762
   -1.2850
    0.1263
    0.3783
    0.4728
    1.7168
   -0.4116
   -2.1938
    0.0986
```

1.3.1.2 顺序高斯消去法 Matlab 时间测试程序

```
time1 = Gauss_function(10);
time2 = Gauss_function(50);
time3 = Gauss_function(100);
time4 = Gauss_function(200);
X = [10,50,100,200];
Y = [time1,time2,time3,time4];
plot(X,Y,'-bo') ;
title('Gauss_Function');
xlabel('Number of column');
ylabel('Time');
```

1.3.1.2 运行图例



1.3.2 列主元高斯消去法

1.3.2.1 列主元高斯消去法消元求值

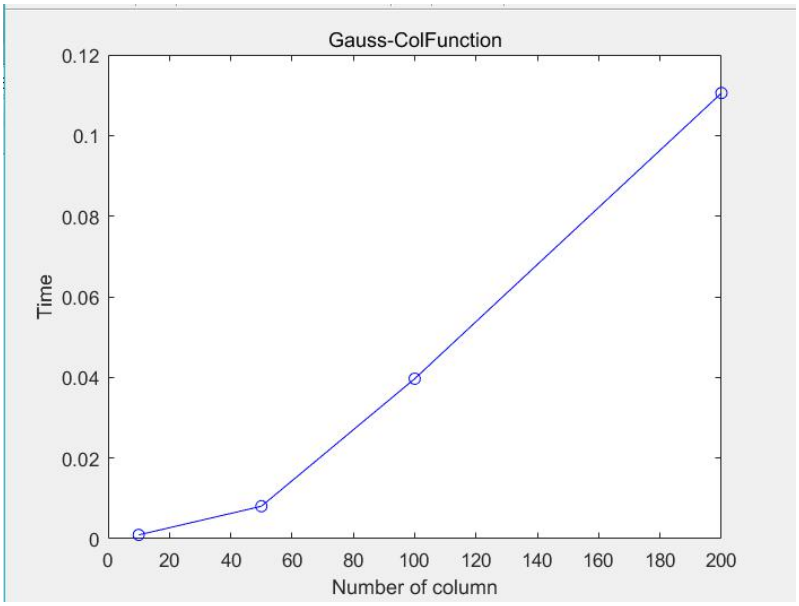
注： ①为了方便展示结果使用 n = 10 进行运算
②x_1 是 Matlab 自带运算器 A\b 的运算结果
③x_2 是列主元高斯消去法求值结果

x_1 =	x_2 =
-0.9020	-0.9020
7.4860	7.4860
-1.5552	-1.5552
-11.3054	-11.3054
3.4571	3.4571
0.4723	0.4723
3.6087	3.6087
3.8291	3.8291
-0.0591	-0.0591
-3.6045	-3.6045

1.3.2.1 列主元高斯消去法 Matlab 时间测试程序

```
time1 = Gauss_colfunction(10);
time2 = Gauss_colfunction(50);
time3 = Gauss_colfunction(100);
time4 = Gauss_colfunction(200);
X = [10,50,100,200];
Y = [time1,time2,time3,time4];
plot(X,Y,'-bo') ;
title('Gauss_ColFunction');
xlabel('Number of column');
ylabel('Time');
```

1.3.2.2 运行图例



1.3.3 顺序高斯消去法与列主元高斯消去法对比

1.3.2.1 顺序高斯消去法与列主元高斯消去法时间对比 Matlab 测试程序

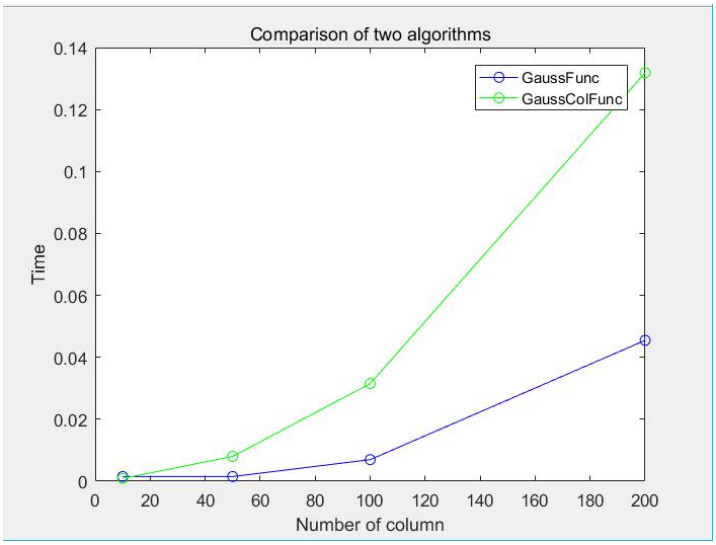
注：在对比高斯消去法与列主元消去法的时间时候，我对每一个维数进行了 100 次的实验，随后对时间取平均值，随后作出图例。

```
for k = 0 : 1 : 99
    %生成n*n的随机矩阵
    A = normrnd(10,1,10,10) ;
    %生成n*1的随机矩矩阵
    b = normrnd(10,1,10,1) ;
    time1 = time1 + Gauss_function(A,b,10);
    time5 = time5 + Gauss_colfunction(A,b,10);
end
```

```
%高斯消元法输出
X = [10,50,100,200];
Y = [time1/100,time2/100,time3/100,time4/100];
plot(X,Y,'-bo') ;
%模块结束
hold on ;

%列高斯消去法作图
X = [10,50,100,200];
Y = [time5/100,time6/100,time7/100,time8/100];
plot(X,Y,'-go') ;
%模块结束
title('Comparison of two algorithms')
xlabel('Number of column');
ylabel('Time');
legend('GaussFunc','GaussColFunc');
```

1.3.2.2 运行图例



1.4. 结果分析

从 1.3.2.2 的运行图例我们可以明显的发现，当方阵 A 的维数逐渐增加时候，列主元高斯消去法与顺序高斯消去法的时间差会越来越大。

我们从算法的本质出发进行分析：

顺序高斯消去算法：

```
%顺序高斯消去
for k = 1 : n - 1
    for i = k + 1: n
        %判断Akk是否为零
        if(A(k,k) == 0)
            disp('Error , A(k,k) is 0 !') ;
            return ;
        end
        temp = A(i,k) / A(k,k) ;
        for j = k + 1 : n
            A(i,j) = A(i,j) - temp * A(k,j) ;
        end
        b(i) = b(i) - temp * b(k);
    end
end
%求出解集
```

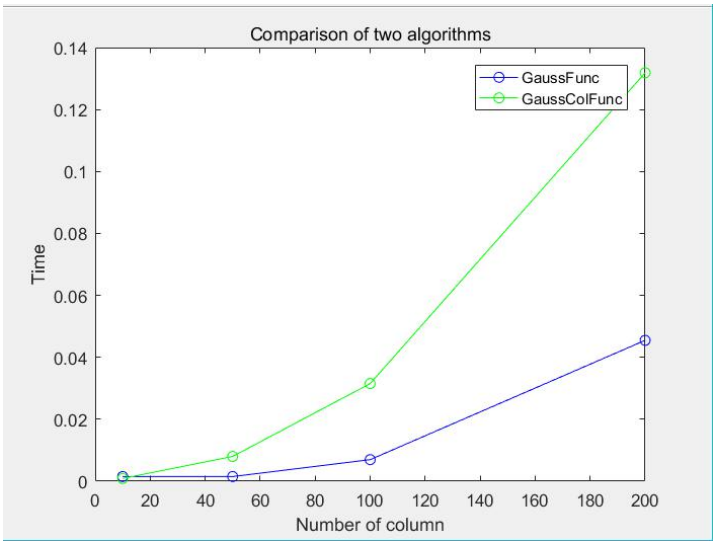
列主元高斯消去算法：

```
    for p=1:n-1
        %找出某一列主元最大的行
        t=find(abs(B(p:end,p))==max(abs(B(p:end,p))))+p-1;
        %如果该行不是最大主元所在的行
        if abs(B(t,p))~=abs(B(p,p))
            %完成当前行与最大的行交换
            l=B(t,:);
            if(t==5)
                B(t,:)=B(p,:);
                B(p,:)=l;
            end
        end
```

```
            %列主元判断
            for k=p+1:n
                if(B(p,p) == 0)
                    disp('Error , A(k,k) is 0 !') ;
                    return ;
                end
                temp = B(k,p) / B(p,p);
                %完成消元变化过程
                B(k,p:n+1) = B(k,p:n+1) - temp * B(p,p:n+1);
            end
        end
```

我们从算法的循环中可以明显的发现，列主元高斯消去比顺序高斯消去法多了一个选主元的过程与选完主元后的交换的过程，从时间复杂度上我们可以发现，这是个 n^2 级别的操作，如果再加上外围的循环，就相当于比顺序高斯消去法多了两个 n^3 级别的操作。

假设原先的顺序高斯消去法时间复杂度为 a * n^3 ，那么列主元高斯消去法的时间复杂度就是 a* n^3 + b * n^3，从数量级上看，这两个操作都是同样数量级（n^3）级别的操作，并没有太大的差异，但是从实际操作的步骤来看，列主元高斯消去法比顺序高斯消去法要多一个出常数级别的操作，这就导致了算法运行时间的扩大，而扩大的倍数也取决于(a+b)/a 这个数字的大小。



如果在进行列主元高斯消去时候，我们遇到了最坏的情况，即每一次需要操作的列主元都是需要交换的，这样我们第 k 次操作，就需要进行 n-k 次交换，交换的数据也是有 n-k 个，这样循环中就产生了一个 n^2 级别的操作。如果每一次的消元操作都是需要进行这样 n^2 级别列主元变换,那么我们的这个交换的过程与 A 矩阵的消元过程的时间复杂度也基本差不多了，甚至会大于消元过程，这也就说明了，为什么我们在 n = 200，n = 100 进行消元时候（图示），列主元高斯消去法消耗的时间比顺序高斯消去法消耗时间两倍还要多。

但是从误差分析上来看，书上 P149 给我们的例子已经很明确的体现出，列主元高斯消去法是更适合用于消元的，因为我们并不能保证在进行消元时候小主元精度的绝对准确，如果小主元精度有缺失，那么我们得到的答案差距就会特别大，会严重影响我们的结论。但是由于列主元高斯消去法所需要时间也是比较多的，我们在进行大矩阵运算的时候，会浪费较多的时间，所以我们在消元时候，都不会怎么去选用这两个方法，反而更倾向于选 part2 中的 CG 方法。

Part 2

2.1 问题描述

二、请实现下述算法，求解线性方程组 Ax=b，其中 A 为 n×n 维的已知矩阵，b 为 n 维的已知向量，x 为 n 维的未知向量。

（1）Jacobi 迭代法。

（2）Gauss-Seidel 迭代法。

（3）逐次超松弛迭代法。

（4）共轭梯度法。

A 为对称正定矩阵，其特征值服从独立同分布的[0,1]间的均匀分布；b 中的元素服从独立同分布的正态分布。令 n=10、50、100、200，分别绘制出算法的收敛曲线，横坐标为迭代步数，纵坐标为相对误差。比较 Jacobi 迭代法、Gauss-Seidel 迭代法、逐次超松弛迭代法、共轭梯度法与高斯消去法、列主元消去法的计算时间。改变逐次超松弛迭代法的松弛因子，分析其对收敛速度的影响。

2.2 算法设计

2.2.1 条件约束

2.2.1.1 方阵 A 与向量 b

```
    %替换矩阵
    V = diag(rand(n,1));
    M = orth(rand(n));
    A = M * V * M' ;

    %正态分布待测向量
    b = normrnd(0,1,n,1) ;
```

2.2.1.2 保证 Ax = b 解的唯一性

```
    %测试模块
        %保证Ax = b有唯一解
        test = [A b] ;
        Rank_A = rank(A) ;
        Rank_test = rank(test) ;
        if( Rank_A ~= Rank_test )
            disp('Error , Rank_A != Rank_test') ;
            return ;
        end
    %模块结束
```

2.2.1.3 保证雅可比迭代法的收敛性

```
    %判断雅可比迭代法是否收敛
    if(vrho((D\(L+U)))>1)
        k = k - 1 ;
        continue ;
    end
```

2.2.1.4 保证高斯赛德尔迭代法的收敛性

```
    %判断高斯赛德尔迭代是否收敛
    G = (D-L)\U ;
    if(vrho(G)>1)
        k = k - 1 ;
        continue ;
    end
```

2.2.1.5 设置实验重复次数

```
for count = 0 : 1 : 99
```

2.2.2 雅可比迭代法

2.2.2.1 算法描述

对方程组 Ax=b，其中 A 为非奇异矩阵。设 $a_{ii} \neq 0 (i = 1, 2, \cdots, n)$，并将 A 写为三部分：

$$A = \begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix} - \begin{bmatrix} 0 & & & \\ -a_{21} & 0 & & \\ \vdots & \vdots & \ddots & \\ -a_{n-1,1} & -a_{n-1,2} & \cdots & 0 \\ -a_{n1} & -a_{n2} & \cdots & -a_{n,n-1} & 0 \end{bmatrix} - \begin{bmatrix} 0 & -a_{12} & \cdots & -a_{1,n-1} & -a_{1n} \\ & 0 & \cdots & a_{2,n-1} & -a_{2n} \\ & & \ddots & \vdots & \vdots \\ & & & 0 & -a_{n-1,n} \\ & & & & 0 \end{bmatrix}$$

$$= D - L - U$$

于是 $Ax = b \Leftrightarrow (D - L - U)x = b$

即 $x = D^{-1}(L + U)x + D^{-1}b$

所以解 Ax=b 的基本迭代公式为

$$\begin{cases} x^{(0)} = (x_1^{(0)}, \cdots, x_n^{(0)}), \\ x_i^{(k+1)} = (b_i - \sum\limits_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)}) / a_{ii} (i = 1, 2, \cdots, n)(k = 0, 1, \cdots). \end{cases}$$

2.2.2.2 雅可比迭代法 Matlab 程序

```
%求出A矩阵的对角矩阵
D = diag(diag(A)) ;
%求出负下三角矩阵
L = -tril(A,-1) ;
%求出负上三角矩阵
U = -triu(A,1) ;
%求出B矩阵
B = D\(L+U) ;
%求出向量f
f = D\b ;
%设置误差量
Error_term = 1.0e-6;
%求迭代值
x_3 = B * x_2 + f ;
```

```
while norm(x_3 - x_2) >= Error_term
    x_2 = x_3 ;
    x_3 = B * x_2 + f ;
    Now_number = Now_number + 1 ;
    if( Now_number >= Max_number)
        disp('Error : The number of iterations reached the upper limit');
        break ;
    end
end
```

2.2.3 高斯赛德尔迭代法

2.2.3.1 算法描述

把矩阵 A 分解成

$$A = D - L - U \tag{6}$$

其中 $D = diag(a_{11}, a_{22}, \ldots, a_{nn})$，$-L, -U$ 分别为 A 的主对角元除外的下三角和上三角部分，于是，方程组(1)便可以写成

$$(D - L)x = Ux + b$$

即

$$x = B_2 x + f_2$$

其中

$$B_2 = (D - L)^{-1}U, \qquad f_2 = (D - L)^{-1}b \tag{7}$$

以 B_2 为迭代矩阵构成的迭代法(公式)

$$x^{(k+1)} = B_2 x^{(k)} + f_2 \tag{8}$$

称为高斯—塞德尔迭代法(公式)，用 量表示的形式为

$$\begin{cases} x_i^{(k+1)} = \dfrac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right] \\ i = 1,2,\cdots n, \qquad k = 0,1,2,\ldots \end{cases} \tag{9}$$

2.2.3.1 高斯赛德尔迭代法 Matlab 程序

```
38    %求出A矩阵的对角矩阵
39    D = diag(diag(A)) ;
40    %求出负下三角矩阵
41    L = -tril(A,-1) ;
42    %求出负上三角矩阵
43    U = -triu(A,1) ;
44    %求出G矩阵
45    G = inv(D-L)*U ;
46    %求出f向量
47    f = inv(D-L)*b ;
48    %设置误差量
49    Error_term = 1.0e-6;
50    %求迭代值
51    x_3 = G * x_2 + f ;
```

```
    while norm(x_3 - x_2) >= Error_term
        x_2 = x_3 ;
        x_3 = G * x_2 + f ;
        Now_number = Now_number + 1 ;
        if( Now_number >= Max_number)
            disp('Error : The number of iterations reached the upper limit');
            break ;
        end
    end
```

2.2.4 超松弛迭代法

（注：超松弛迭代法，只需要保证每次迭代之前使输入的松弛因子 0 < W < 2 与 矩阵的谱半径小于 1，就可以保证收敛性）

2.2.4.1 算法描述

解 $Ax=b$ 的 SOR 方法为

$$\begin{cases} x^{(0)}, & \text{初始向量,} \\ x^{(k+1)} = L_\omega x^{(k)} + f, & k = 0,1,\cdots, \end{cases} \tag{3.1}$$

其中 $L_\omega=(D-\omega L)^{-1}((1-\omega)D+\omega U)$，$f=\omega(D-\omega L)^{-1}b$.

下面给出解 $Ax=b$ 的 SOR 迭代法的分量计算公式. 记

$$x^{(k)} = (x_1^{(k)},\cdots,x_i^{(k)},\cdots,x_n^{(k)})^{\mathrm{T}},$$

由(3.1)式可得

$$(D-\omega L)x^{(k+1)} = ((1-\omega)D+\omega U)x^{(k)} + \omega b,$$

或

$$Dx^{(k+1)} = Dx^{(k)} + \omega(b + Lx^{(k+1)} + Ux^{(k)} - Dx^{(k)}).$$

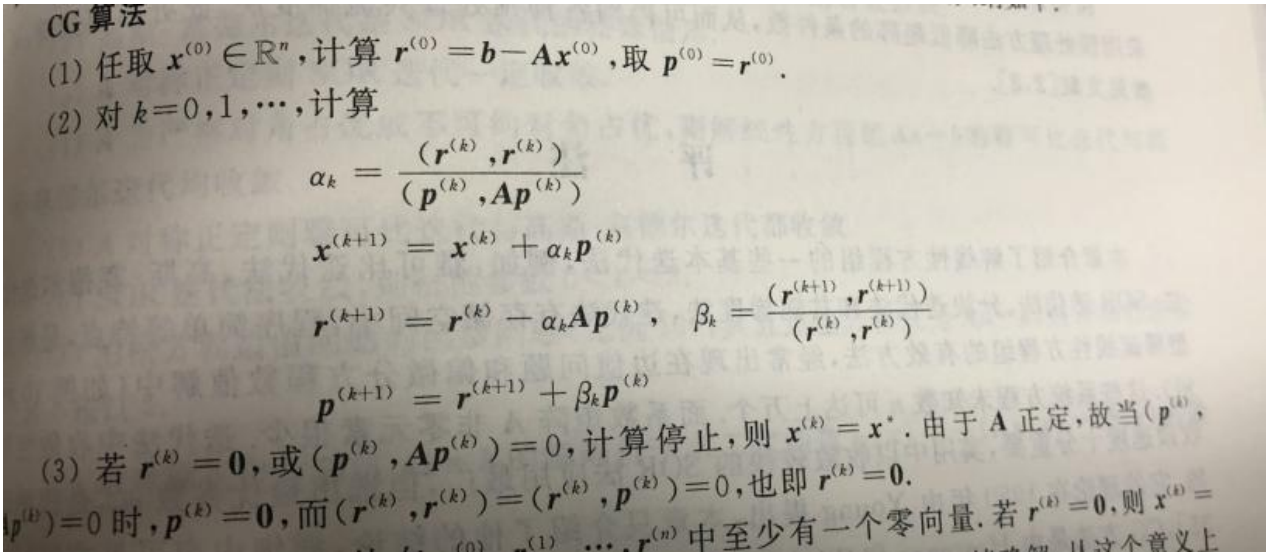
2.2.4.2 超松弛迭代法 Matlab 程序

```
    %求出A的对角矩阵
    D = diag(diag(A)) ;
    %求出负上三角矩阵
    U = -triu(A,1) ;
    %求出负下三角矩阵
    L = -tril(A,-1) ;
    %取出松弛因子
    w = W ;
    %求出B矩阵
    B = (D - w * L) \ ( (1-w) * D + w * U) ;
    %求出f
    f = w * ( (D- w * L) \ b );
    %设定误差项
    Error_term = 1.0e-6;
```

```
    for k = 1:1:10000000
        y = B * x_2 + f ;
        if(norm(y-x_2) < Error_term)
            break ;
        end
        if( k >= 10000000 )
            disp('Error : The number of iterations reached the upper limit');
            break ;
        end
        x_2 = y ;
    end
```

2.2.4 共轭梯度法

2.2.4.1 算法描述



CG 算法

(1) 任取 $x^{(0)} \in \mathbb{R}^n$，计算 $r^{(0)} = b - Ax^{(0)}$，取 $p^{(0)} = r^{(0)}$.

(2) 对 $k = 0, 1, \cdots$，计算

$$\alpha_k = \frac{(r^{(k)}, r^{(k)})}{(p^{(k)}, Ap^{(k)})}$$

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$$

$$r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)}, \quad \beta_k = \frac{(r^{(k+1)}, r^{(k+1)})}{(r^{(k)}, r^{(k)})}$$

$$p^{(k+1)} = r^{(k+1)} + \beta_k p^{(k)}$$

(3) 若 $r^{(k)} = 0$，或 $(p^{(k)}, Ap^{(k)}) = 0$，计算停止，则 $x^{(k)} = x^*$. 由于 A 正定，故当 $(p^{(k)}$,

$p^{(k)}) = 0$ 时，$p^{(k)} = 0$，而 $(r^{(k)}, r^{(k)}) = (r^{(k)}, p^{(k)}) = 0$，也即 $r^{(k)} = 0$.

$r^{(0)}, r^{(1)}, \cdots, r^{(n)}$ 中至少有一个零向量. 若 $r^{(k)} = 0$，则 $x^{(k)} =$

2.2.4.1 共轭梯度法 Matlab 程序

```
    %定义r(0)
    r = b - A * x_2 ;
    %设置测试变量
    test_random = zeros(1,100);
    %定义p(0)
    p = r ;
```

```
    for k = 1 :1:10000000
        %定义a(k)
        a_k = (norm(r)^2)/( p' * A * p) ;
        x_2 = x_2 + a_k * p ;
        %定义r(k+1)
        rr = r - a_k * A * p  ;
        %求出误差大小
        if( abs(rr) <= Error_term )
            break ;
        end
        %定义β(k)
        B = (norm(rr)^2) / (norm(r)^2) ;
        %定义p(k+1)
        p = rr + B * p ;
        r = rr ;
        if( k >= 10000000 )
            disp('Error : The number of iterations reached the upper limit');
            break ;
        end
    end
```

2.3 数值实验

2.3.1 算法求值实验

2.3.1.1 雅可比迭代法求值

注：　①为了方便展示结果使用 n = 10 进行运算
　　②x_1 是 Matlab 自带运算器 A\b 的运算结果
　　③x_3 是雅可比迭代法求值结果

x_1 =	x_3 =
-3.4066	-3.4066
-2.6458	-2.6458
-0.4282	-0.4282
0.4121	0.4121
6.1293	6.1293
-2.4108	-2.4108
-2.1442	-2.1442
-0.9915	-0.9915
1.0555	1.0555
4.6940	4.6940

2.3.1.2 高斯赛德尔迭代法求值

注： ①为了方便展示结果使用 n = 10 进行运算
②x_1 是 Matlab 自带运算器 A\b 的运算结果
③x_3 是高斯赛德尔迭代法求值结果

x_1 =	x_3 =
0.3918	0.3918
-8.7076	-8.7076
-2.0353	-2.0353
-0.8532	-0.8532
4.5397	4.5397
1.8017	1.8017
-0.0332	-0.0332
-3.4651	-3.4651
3.4039	3.4039
4.5522	4.5522

2.3.1.3 超松弛迭代法求值

注： ① 为了方便展示结果使用 n = 10 进行运算
② 选用松弛因子 W = 1.5
③ x_1 是 Matlab 自带运算器 A\b 的运算结果
④ x_3 是高斯赛德尔迭代法求值结果

x_1 =	x_2 =
-2.7579	-2.7579
2.6007	2.6007
0.1934	0.1934
3.1160	3.1160
0.9738	0.9738
-1.5926	-1.5926
-2.8579	-2.8579
0.3801	0.3801
-2.9047	-2.9047
0.4555	0.4555

2.3.1.4 共轭梯度法求值

注： ①为了方便展示结果使用 n = 10 进行运算
②x_1 是 Matlab 自带运算器 A\b 的运算结果
③x_3 是共轭梯度法求值结果

```
x_1 =       x_2 =

    8.8738      8.8738
  -14.8322    -14.8322
    9.4291      9.4291
   15.0536     15.0536
  -10.6778    -10.6778
    1.3369      1.3369
   -5.7877     -5.7877
    1.1220      1.1220
    4.6383      4.6383
  -10.1000    -10.1000
```

2.3.2 Gauss_function、Gauss_Colfuction、Jacobi、Gauss_Seidel、SOR、CG 求解方程**运行时间对比**

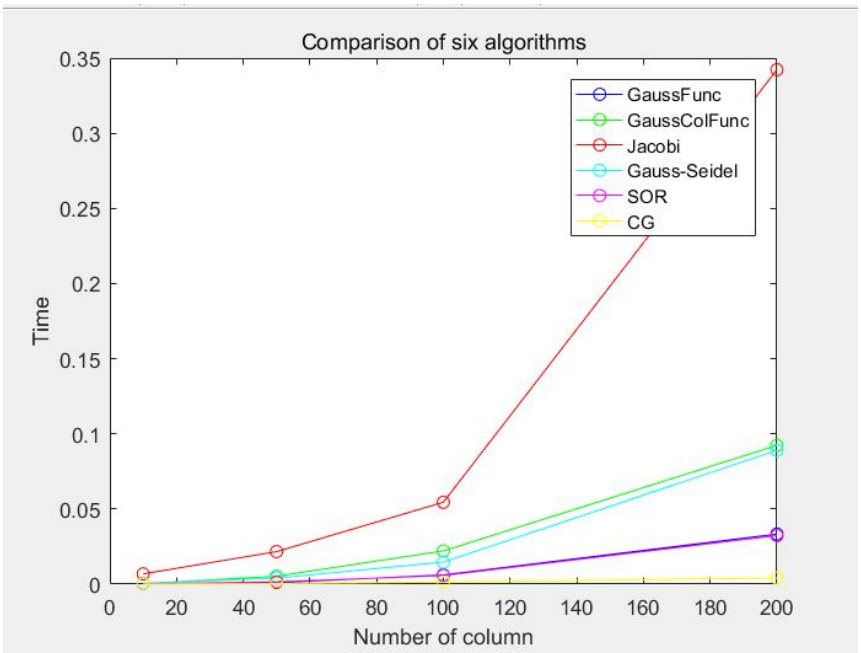
2.3.2.1 测试程序

```
for k = 0 : 1 : 99
    %矩阵A
    V = diag(rand(10,1));
    M = orth(rand(10));
    A = M * V * M' ;
    %待测值向量
    b = normrnd(0,1,10,1) ;
    time1 = time1 + Gauss_function(A,b,10);
    time5 = time5 + Gauss_colfunction(A,b,10);
    time9 = time9 + Jacobi_function(A,b,10) ;
    time13 = time13 + GaussSeidel_function(A,b,10) ;
    time17 = time17 +  SOR_function(A,b,10,1.5) ;
    time21 = time21 +  CG(A,b,10);
end
```

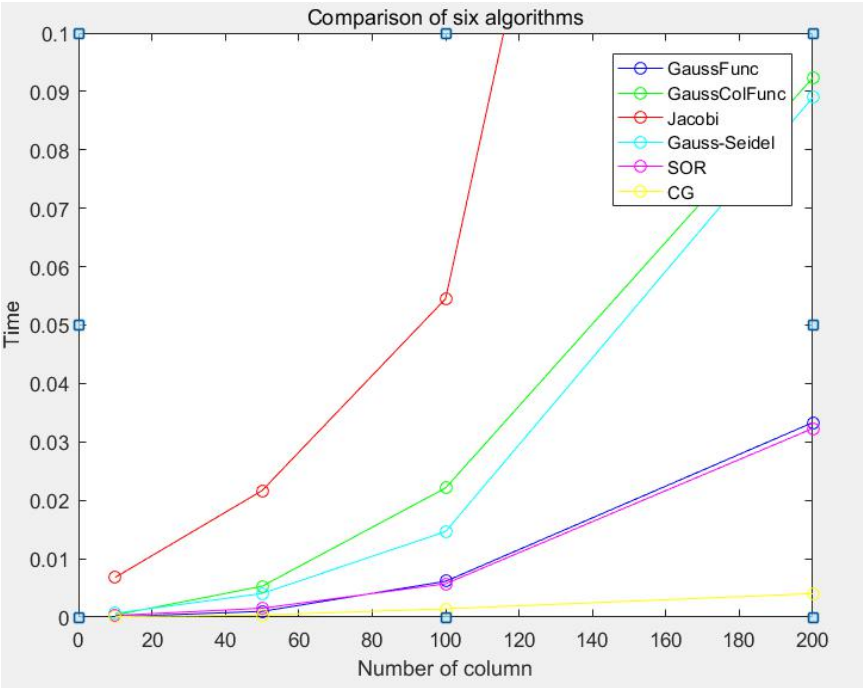
```
%高斯消元法输出
X = [10,50,100,200];
Y = [time1/100,time2/100,time3/100,time4/100];
plot(X,Y,'-bo') ;
%模块结束
hold on ;
```

（注：time1，time5，time9，time13，time17，time21 依次对应顺序高斯消去算法，列主元高斯消去算法，雅可比迭代算法，高斯赛德尔迭代算法，超松弛迭代算法，共轭梯度算法在 n = 10 条件下的算法运行时间[n=50，100，200 情况类似]，运行 100 次之后，求平均运行时间）

2.3.2.2 运行时间对比图例

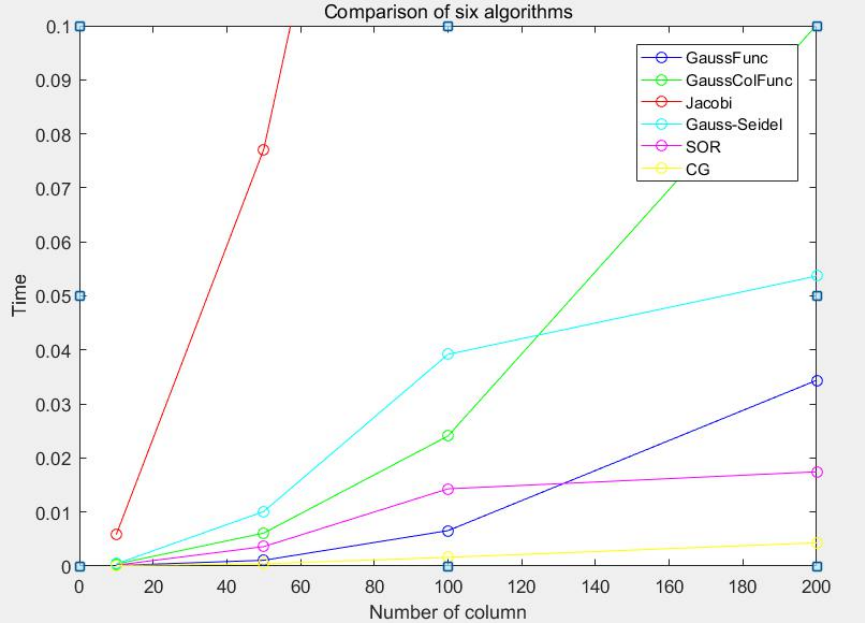


为了更方便观察，我截取了 0 < y < 0.1 这一段的数据再次进行绘图



从图上，我们明显能够发现，时间从高到低依次是：雅可比迭代算法 > 列主元高斯消去算法 > 高斯赛德尔迭代法 > 高斯消去算法 > 超松弛算法 > 共轭梯度法

但是我们同时也发现，超松弛迭代算法与高斯消去算法所花费的时间近乎相同，为了避免偶然性，于是我又再次进行了一次实验，再取了一次 100 次实验的结果进行分析。



结果显示，超松弛迭代算法与顺序高斯消去算法的时间是近似的，并伴随着实验可能会有偏差。

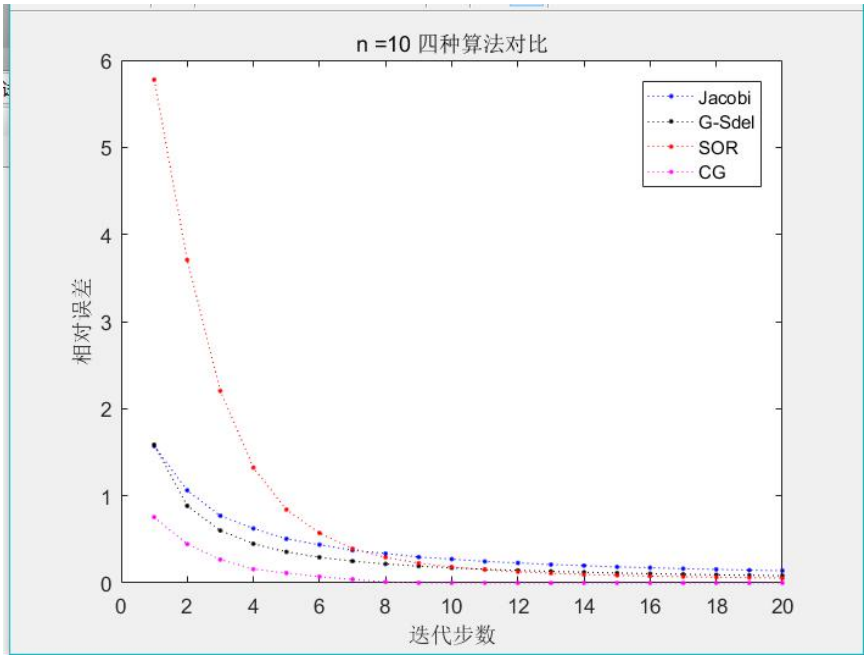
2.3.3 Jacobi、Gauss-Seidel、SOR、CG 收敛速度对比

2.3.3.1 n = 10

注： ① 这里迭代步数均设为 20

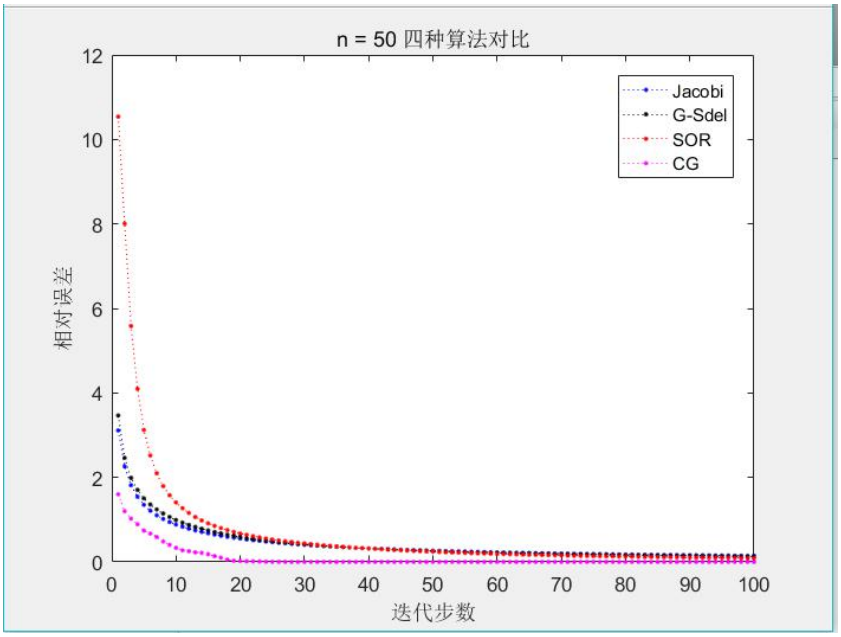
② 相对误差均为迭代值与原先值的一范数

③ 每次绘图前，重复 100 次实验，将 100 次实验中，每一步的误差加起来，随后再取平均值



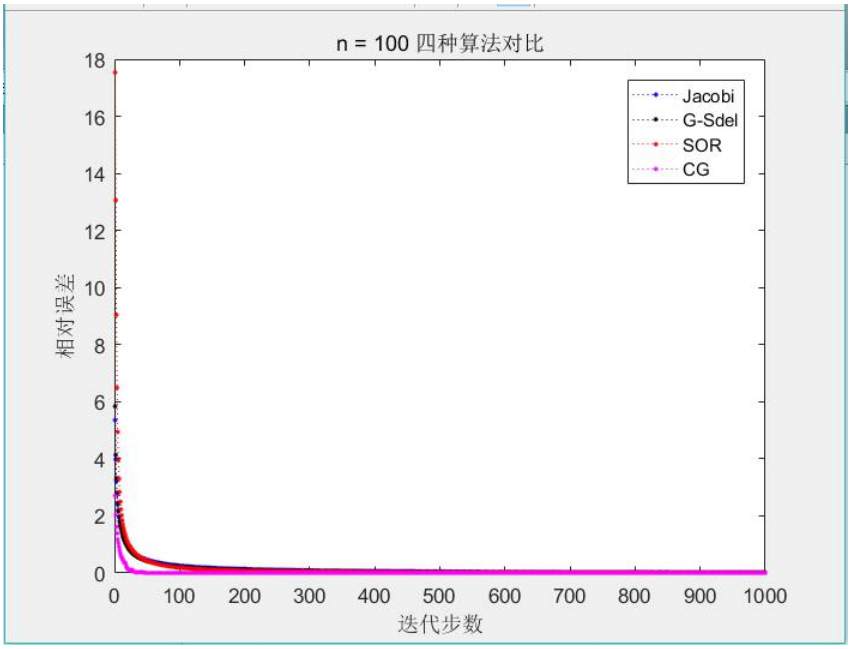
2.3.3.2 n = 50

注： ① 这里迭代步数均设为 100
② 相对误差均为迭代值与原先值的一范数
③ 每次绘图前，重复 100 次实验，将 100 次实验中，每一步的误差加起来，随后再取平均值

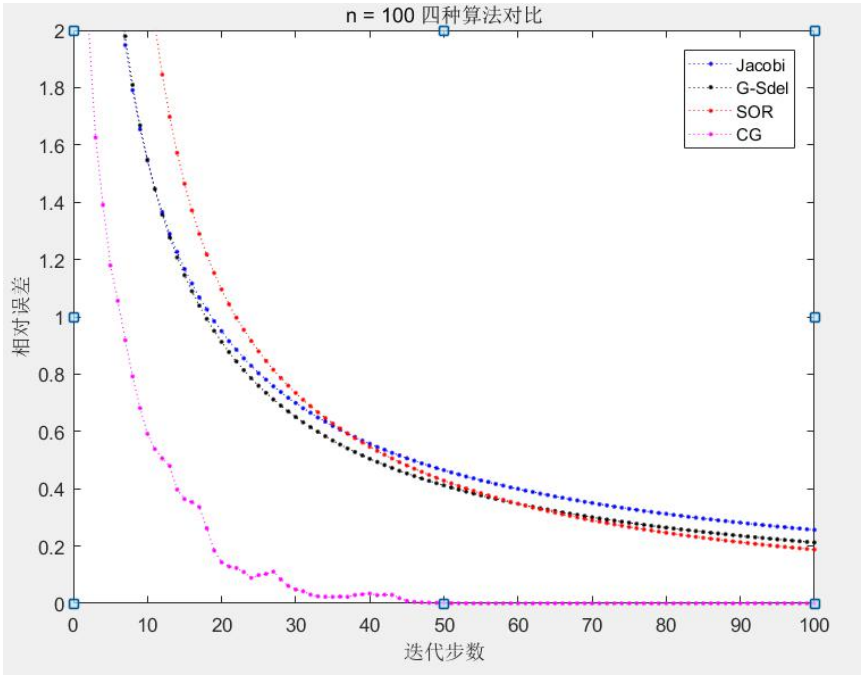


2.3.3.3 n = 100

注： ① 这里迭代步数均设为 1000
② 相对误差均为迭代值与原先值的一范数
③ 每次绘图前，重复 100 次实验，将 100 次实验中，每一步的误差加起来，随后再取平均值



为了进一步反映出迭代速度的变化，我取 0 < x < 100 且 0 < y < 2 这个区间进行放大观察

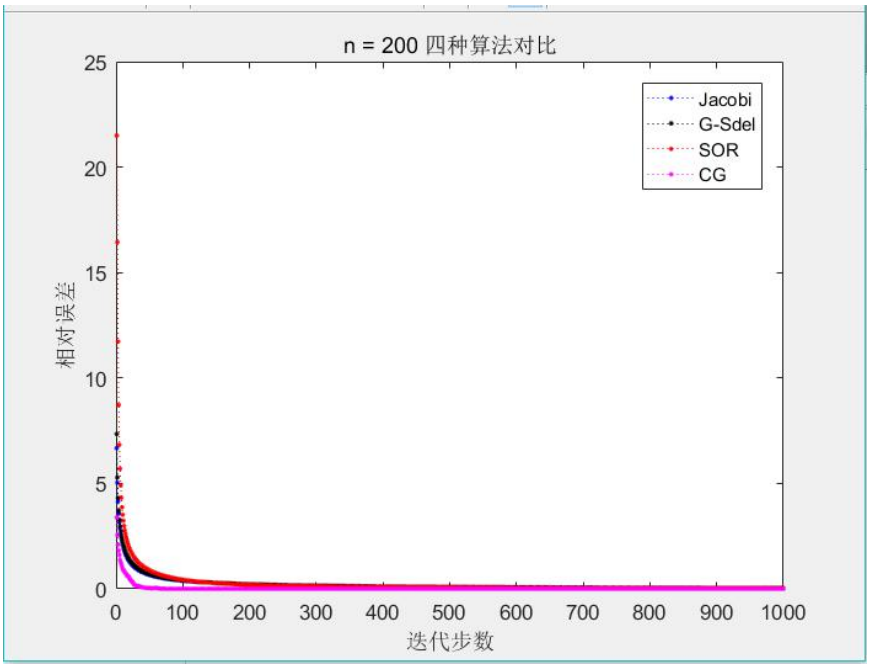


2.3.3.4 n = 200

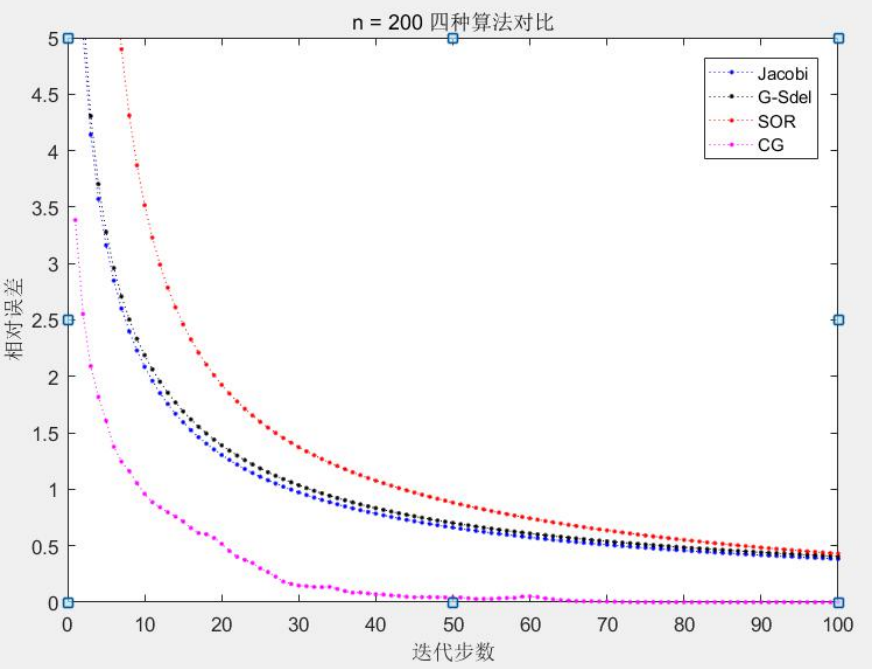
注： ① 这里迭代步数均设为 1000

② 相对误差均为迭代值与原先值的一范数

③ 每次绘图前，重复 100 次实验，将 100 次实验中，每一步的误差加起来，随后再取平均值



为了进一步反映出迭代速度的变化，我取 0 < x < 100 且 0 < y < 5 这个区间进行放大观察



2.3.3.5 总结

从以上的图像中我们发现，CG 算法的迭代是最快完成的，同时迭代速度也是最快的，而 SOR, Jacobi, Gauss-Seidel 算法三者之间的收敛速度相对来说是相同的，不过 Jacobi 算法起始的误差要大于 SOR 与 Gauss-Seidel 算法，这就导致了 Jacobi 算法的收敛速度相对缓慢一点，但但最终三者会趋于相同。

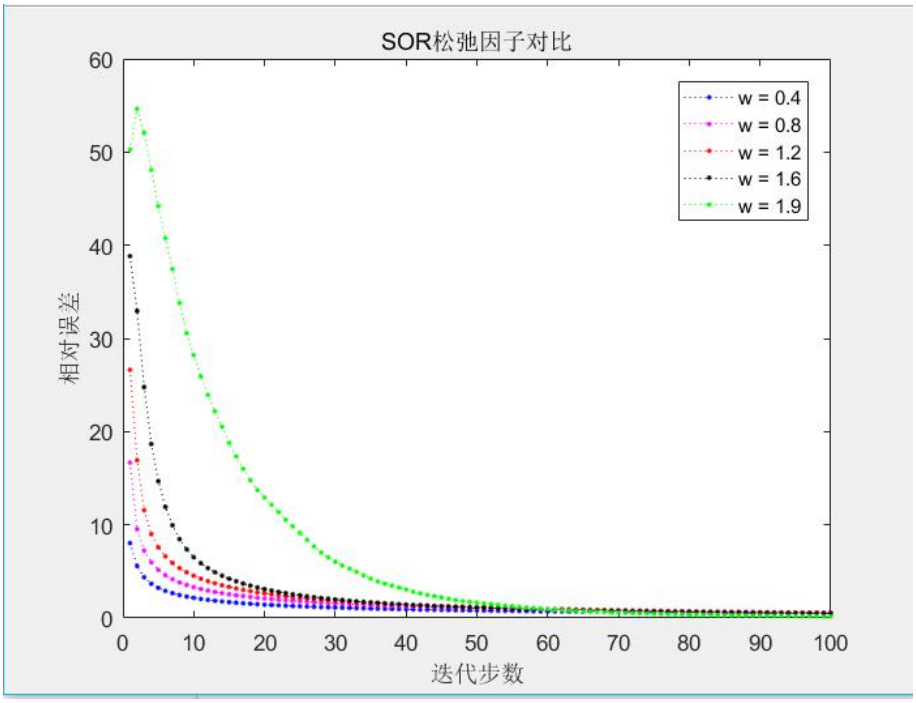
2.3.4 探究 W 对 SOR 算法收敛速度的影响

2.3.4.1 以迭代步数为自变量，相对误差为因变量，对比不同 W 的收敛速度

注： ① 实验共进行 100 次，求出 100 次实验中每一步迭代的误差的平均值
② 分别测试 W = 0.4 、 0.8 、 1.2 、 1.6 、 1.9 时的收敛曲线
③ 迭代步数 n = 100

```
for k = 1 : 1 : 100
    %矩阵A
    V = diag(rand(100,1));
    M = orth(rand(100));
    A = M * V * M' ;
    %待测值向量
    b = normrnd(0,1,100,1) ;
    error1 = error1 + SOR_function(A,b,100,0.4,100);
    error2 = error2 + SOR_function(A,b,100,0.8,100);
    error3 = error3 + SOR_function(A,b,100,1.2,100);
    error4 = error4 + SOR_function(A,b,100,1.6,100);
    error5 = error5 + SOR_function(A,b,100,1.9,100);
end
```

2.3.4.2 对比图例



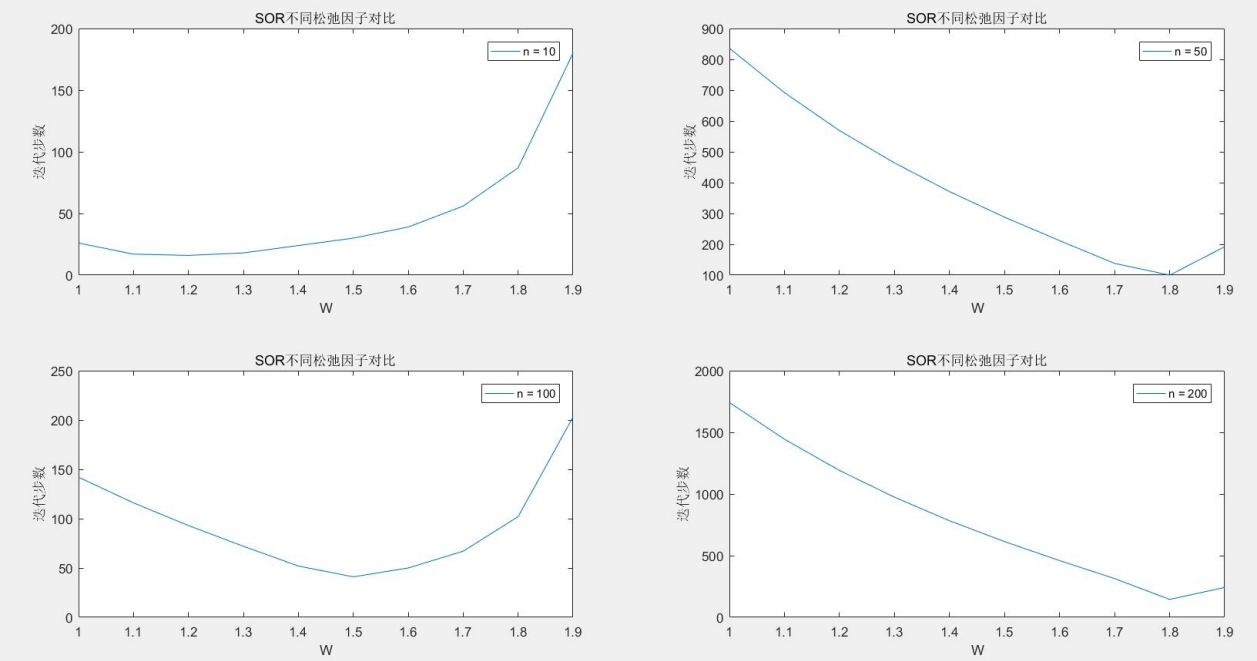
从图上我们可以直观的看出，当迭代步数等于 100 时候，松弛因子越大反而算法的收敛速度越低，同时当 W 趋近于 2 的时候，收敛速度会变得特别慢，但是这样并不能直观反映出 W 对 SOR 算法收敛的影响，于是我进行了 2.3.4.3 中进一步的实验。

2.3.4.3 以 W 为自变量，迭代步数为因变量，限定迭代过程中相对误差进行对比

注： ① 实验共进行 100 次，求出 100 次实验中每一步迭代的误差的平均值
② 限定迭代误差为 10^-6 次方

```
V = diag(rand(10,1));
M = orth(rand(10));
A = M * V * M' ;
%待测值向量
b = normrnd(0,1,10,1) ;
for  i = 10 : 1 : 19
    step1(i - 9 , :) = SOR_function(A,b,10,i/10,1.0e-6)
end
subplot(2,2,1);
plot(1:0.1:1.9,step1) ;
title('SOR不同松弛因子对比')
xlabel('W');
ylabel('迭代步数');
legend('n = 10');
```

2.3.4.4 对比图例

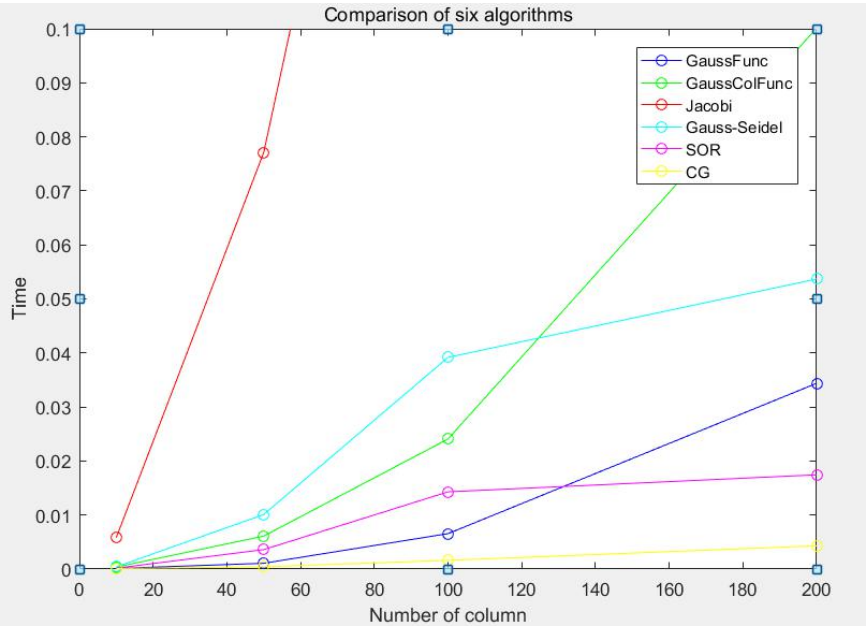


从这四个图的对比中我们可以发现，当方阵的 n 不同的时候，W 对矩阵迭代步数的影响也是不尽相同的，但是总会出现一个可以使步数达到最优的 W 值。

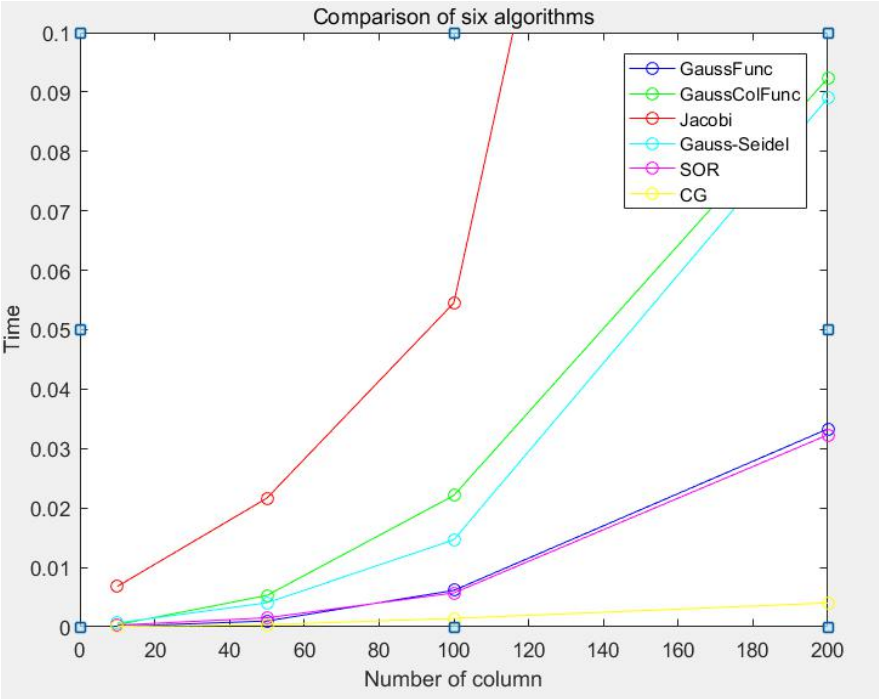
所以这个图，也就可以直观的显示出 W 值对迭代步数的影响。

2.4 结果分析

2.4.1 时间对比分析



从图上我们可以直观的看出，同等维数下，Jacobi 算法的运行时间是最长的并且随着 n 的扩大，算法的运行时间呈阶梯化上升的趋势，这就说明了 Jacobi 算法的不稳定性，十分容易收到矩阵维数的影响。但相对来说，CG 算法就比较稳定了，当矩阵的维数发生很大的变化时候，CG 算法的时间曲线也只是有小范围的波动，这也客观体现了 CG 算法的优越性。



同时我们也发现，不同的实验中，Gauss_function、Gauss_Colfunction、Gauss-Seidel、SOR 四个算法的不稳定性是比较大的，Gauss_Colfunction 与 Gauss-Seidel 算法运行时间总是要大于 SOR 与 Gauss_function 算法。而两个算法（SOR 与 Gauss_function、Gauss_Colfunction 与 Gauss-Seidel）彼此之间，可能会有运行时间的交替上升，这可能是因为算法在运行时候可能会受到某个特殊矩阵的影响而发生小范围内的变动，而与算法本身的设计是没有太大关系的。

Gauss_function、Gauss_Colfunction 的时间差异前面我们已经分析过了，下面大概讲一下剩下四种的算法差异。

① Jacobi 算法是在初始矩阵 A，与待求向量 b 上完成迭代，所以待求的值更新的比较慢，迭代的次数也比较多，进而算法的时间也就多。

② Gauss_Seidel 算法是在 Jacobi 的基础上做了优化，在进行迭代时候，如果有值进行了更新，那么就使用新值进行迭代，这样的做法就会使求值的速度有较大的提高。

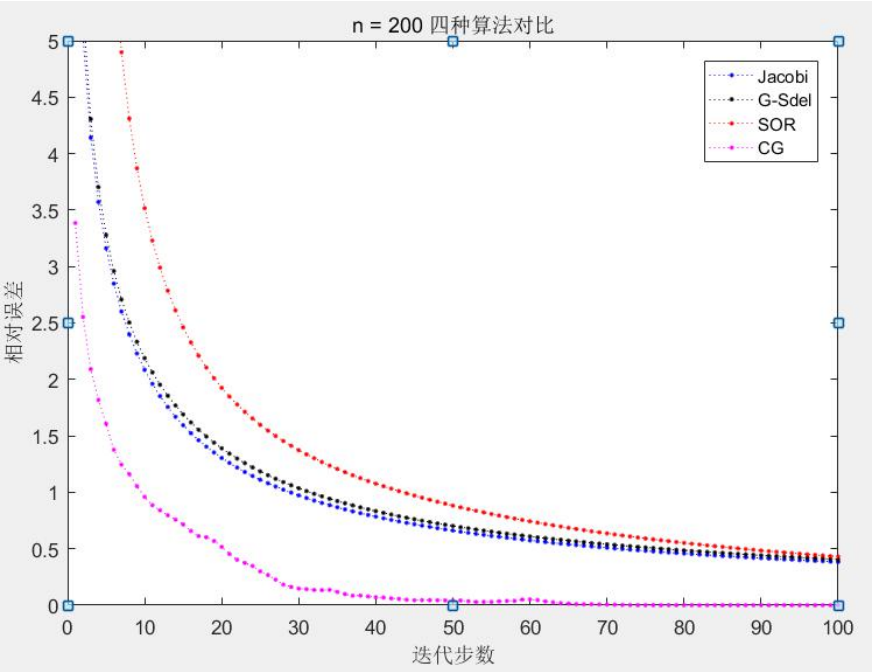
③ XOR 算法与前两者相比，是进一步进行优化，当迭代求得新值时候，取新值和旧值加权平均数，当权重合适的时候，就会有较快速度的收敛，同时算法的时间也会降低，而这个权重也就是我们自主设定的 W。

④ 至于 CG，则是使用了与前三者完全不同的算法，CG 使用迭代值与当前值的二模进行迭代，这样可以最大限度的减小误差，并且极快的逼近真实值。

⑤ Gauss_function 与 Gauss_Colfunction 算法，则是完全走的按行消元的路线，并不是根据当前值与计算值的迭代进行计算。

2.4.2 收敛速度对比分析

基于上一问的运算时间分析，我们也可以得到一定的结论，算法的时间受算法的收敛速度的影响。



同时算法的收敛的速度还受到起始相对误差的影响，以 CG 算法为例，起始点的相对误差较小，那么 CG 在同样的收敛趋势下面，很快的就可以完成收敛；反观 Jacobi 算法，一开始的相对误差就特别大，那么在收敛趋势相同的情况下，收敛的速度就会慢了很多，从而迭代的步数就会多出很多，但是最终 Jacobi 会与 Gauss_Seidel、SOR 以相同的收敛速度和相同的计算误差，完成算法的迭代。

算法的迭代速度，同时与算法完成迭代的计算方式有关，Jacobi、Gauss_Seidel、SOR 都是使用初始矩阵与初始给定向量进行迭代，同时在迭代过程中伴有对当前值与迭代值的误差求取，所以这三者的收敛速度与方式近乎相同，而 CG 则是进行对迭代值与当前值进行二模相除操作，随后再使用该值进行迭代，以较快的速度完成对真实值的逼近。

2.4.3 SOR 对收敛速度的影响分析

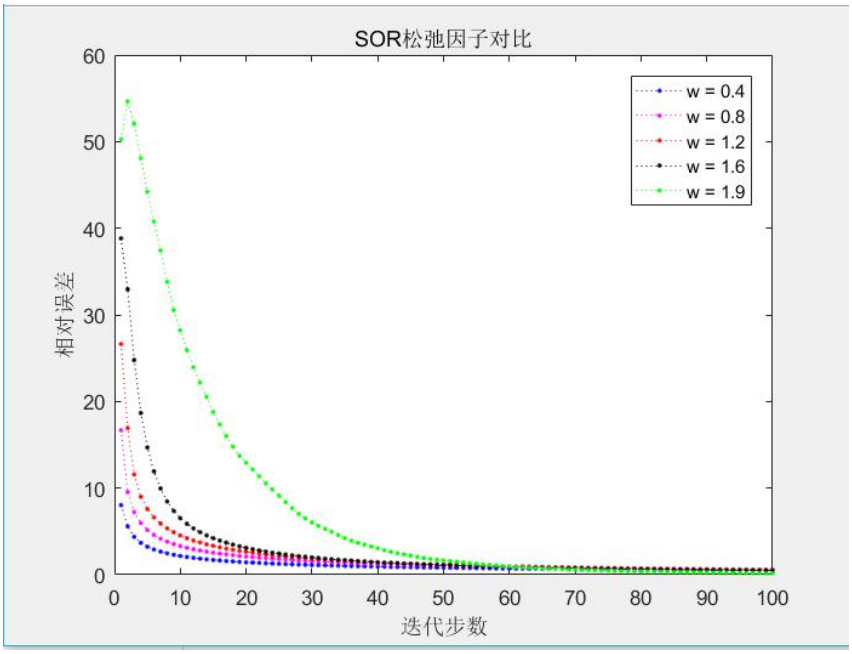
当方阵 A 的 n = 100 时，从图上我们直观的可以看出，W 由小变大，算法的收敛速度依次递减，并且算法的初始误差值变得越来越大。

我们在使用 SOR 算法进行迭代求值的时候，有这么一步

```
%求出 B 矩阵
B = (D - w * L) \ ( (1-w) * D + w * U) ;
%求出 f
f = w * ( (D - w * L) \ b );
y = B * x_2 + f ;
```

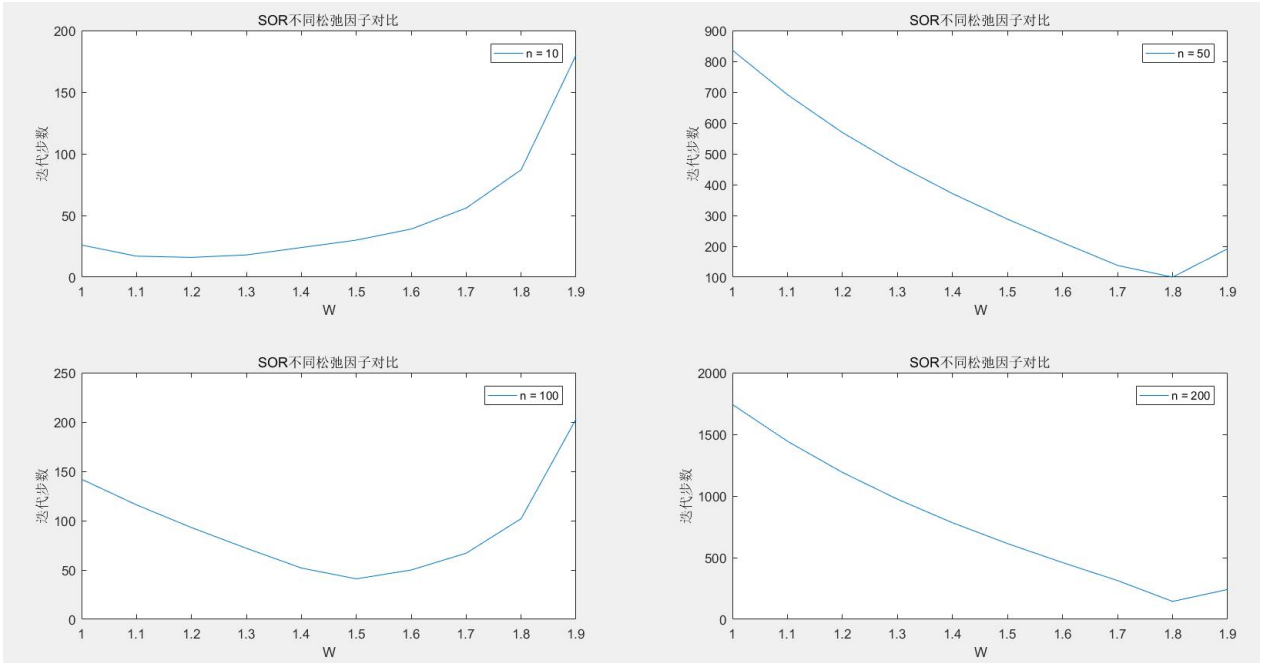
从这个求解形式上，我们可以直观的看出，SOR 迭代法是基于初始矩阵 A 的上三角矩阵 U，下三角矩阵 L，以及对角矩阵 D 进行比例调配之后进行迭代。SOR 算法比 Jacobi 和 Gauss_Seidel 算法先进的地方也在于此步。当我们选择正确的 W 时，就可以合理的调配当前值和迭代值之间的比例，算法的收敛性会大大提升。

（图 A）



但是仅从此图 A 上，我们还并不能得出 W 值越大，收敛性越差的结论；参考课本 P195 下方的表格中给出的数据集合就可以知道，单单从实验的图上直观得到的结论并不准确。所以如果要想真正的获取收敛速度与 W 的关系，还需要作进一步的实验。

（图 B）



还可以从图 A 观察到的一点就是，当 W->2 的时候，SOR 的收敛性会变得与之前的差距很大，类比 W=1.6 与 W=1.2、W=1.6 和 W=1.9 可看出，近乎相同的 W 变化量，但是在图上的收敛程度差异却特别大。

为了能探究 W 对 SOR 算法收敛性的影响，我重新调整了思路，对算法的每一步迭代的误差与方阵 A 的维数 n 进行了限定，令 W 为自变量，迭代步数为因变量进行运算比较，结果得到图 B，图 B 就可以非常直观的显示出 W 对 SOR 算法中迭代步数的影响。当 1 < W < 2 时，总会存在一个 W 使得 SOR 算法的迭代步数会变得尽可能的小，而不是并不是一开始得到的结论，W 越大，XOR 算法的收敛性越差。但是矩阵的维数与矩阵中的数据也是对 SOR 算法的收敛性有一定影响的，从图 B 的迭代步数曲线中，我们就可以清楚的看出这一点。

Part 3

3.1 问题描述

三、在 Epinions 社交数据集(https://snap.stanford.edu/data/soc-Epinions1.html)中，每个网络节点可以选择信任其它节点。借鉴 Pagerank 的思想编写程序，对网络节点的受信任程度进行评分。在实验报告中，请给出伪代码。

3.2 算法设计

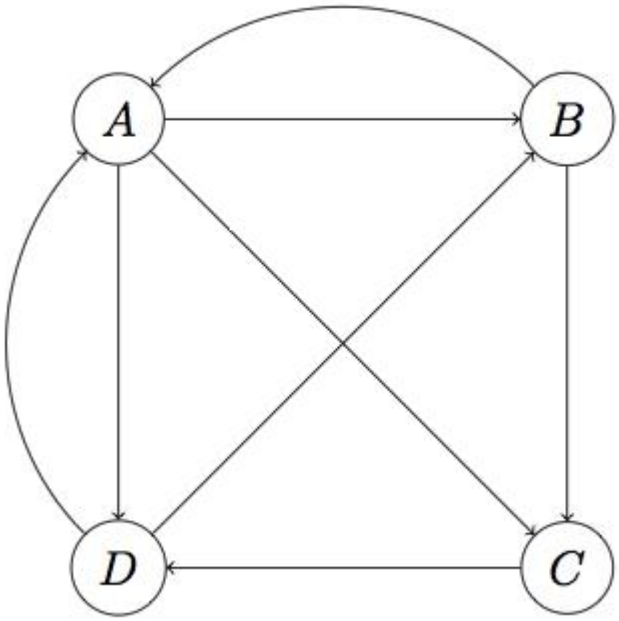
3.2.1 算法描述

PageRank 的作用是评价网页的重要性，以此作为搜索结果的排序重要依据之一。实际中，为了抵御 spam，各个搜索引擎的具体排名算法是保密的，PageRank 的具体计算方法也不尽相同，本节介绍一种最简单的基于页面链接属性的 PageRank 算法。这个算法虽然简单，却能揭示 PageRank 的本质，实际上目前各大搜索引擎在计算 PageRank 时链接属性确实是重要度量指标之一。

3.2.1.1 简单 PageRank 计算

首先，我们将 Web 做如下抽象：1、将每个网页抽象成一个节点；2、如果一个页面 A 有链接直接链向 B，则存在一条有向边从 A 到 B（多个相同链接不重复计算边）。因此，整个 Web 被抽象为一张有向图。

现在假设世界上只有四张网页：A、B、C、D，其抽象结构如下图：



显然这个图是强连通的（从任一节点出发都可以到达另外任何一个节点）。

然后需要用一种合适的数据结构表示页面间的连接关系。其实，PageRank 算法是基于这样一种背景思想：被用户访问越多的网页更可能质量越高，而用户在浏览网页时主要通过超链接进行页面跳转，因此我们需要通过分析超链接组成的拓扑结构来推算每个网页被访问频率的高低。最简单的，我们可以假设当一个用户停留在某页面时，跳转到页面上每个被链页面的概率是相同的。例如，上图中 A 页面链向 B、C、D，所以一个用户从 A 跳转到 B、C、D 的概率各为 1/3。设一共有 N 个网页，则可以组织这样一个 N 维矩阵：其中 i 行 j 列的值表示用户从页面 j 转到页面 i 的概率。这样一个矩阵叫做转移矩阵（Transition Matrix）。下面的转移矩阵 M 对应上图：

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \\ 1/3 & 0 & 1 & 0 \end{bmatrix}$$

然后，设初始时每个页面的 rank 值为 1/N，这里就是 1/4。按 A-D 顺序将页面 rank 为向量 v：

$$v = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

注意，M 第一行分别是 A、B、C 和 D 转移到页面 A 的概率，而 v 的第一列分别是 A、B、C 和 D 当前的 rank，因此用 M 的第一行乘以 v 的第一列，所得结果就是页面 A 最新 rank 的合理估计，同理，Mv 的结果就分别代表 A、B、C、D 新 rank：

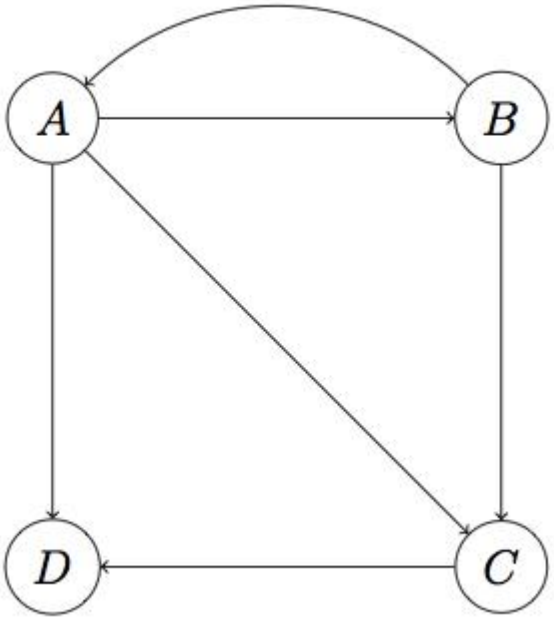
$$Mv = \begin{bmatrix} 1/4 \\ 5/24 \\ 5/24 \\ 1/3 \end{bmatrix}$$

然后用 M 再乘以这个新的 rank 向量，又会产生一个更新的 rank 向量。迭代这个过程，可以证明 v 最终会收敛，即 v 约等于 Mv，此时计算停止。最终的 v 就是各个页面的 pagerank 值。例如上面的向量经过几步迭代后，大约收敛在（1/4, 1/4, 1/5, 1/4），这就是 A、B、C、D 最后的 pagerank。

3.2.1.2 处理 Dead Ends

上面的 PageRank 计算方法假设 Web 是强连通的，但实际上，Web 并不是强连通（甚至不是联通的）。下面看看 PageRank 算法如何处理一种叫做 Dead Ends 的情况。

所谓 Dead Ends，就是这样一类节点：它们不存在外链。看下面的图：



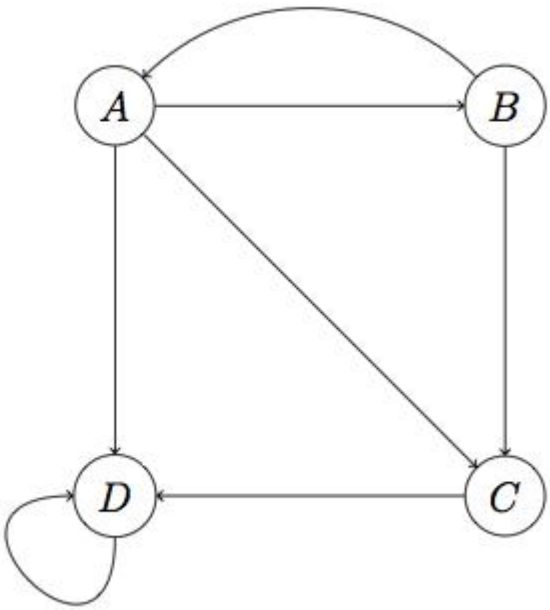
注意这里 D 页面不存在外链，是一个 Dead End。上面的算法之所以能成功收敛到非零值，很大程度依赖转移矩阵这样一个性质：每列的加和为 1。而在这个图中，M 第四列将全为 0。在没有 Dead Ends 的情况下，每次迭代后向量 v 各项的和始终保持为 1，而有了 Dead Ends，迭代结果将最终归零（要解释为什么会这样，需要一些矩阵论的知识，比较枯燥，此处略）。

处理 Dead Ends 的方法如下：迭代拿掉图中的 Dead Ends 节点及 Dead Ends 节点相关的边（之所以迭代拿掉是因为当目前的 Dead Ends 被拿掉后，可能会出现一批新的 Dead Ends），直到图中没有 Dead Ends。对剩下部分计算 rank，然后以拿掉 Dead Ends 逆向顺序反推 Dead Ends 的 rank。

以上图为例，首先拿到 D 和 D 相关的边，D 被拿到后，C 就变成了一个新的 Dead Ends，于是拿掉 C，最终只剩 A、B。此时可很容易算出 A、B 的 PageRank 均为 1/2。然后我们需要反推 Dead Ends 的 rank，最后被拿掉的是 C，可以看到 C 前置节点有 A 和 B，而 A 和 B 的出度分别为 3 和 2，因此 C 的 rank 为：1/2 * 1/3 + 1/2 * 1/2 = 5/12；最后，D 的 rank 为：1/2 * 1/3 + 5/12 * 1 = 7/12。所以最终的 PageRank 为（1/2，1/2，5/12，7/12）。

3.2.1.3 Spider Traps 及平滑处理

可以预见，如果把真实的 Web 组织成转移矩阵，那么这将是一个极为稀疏的矩阵，从矩阵论知识可以推断，极度稀疏的转移矩阵迭代相乘可能会使得向量 v 变得非常不平滑，即一些节点拥有很大的 rank，而大多数节点 rank 值接近 0。而一种叫做 Spider Traps 节点的存在加剧了这种不平滑。例如下图：



D 有外链所以不是 Dead Ends，但是它只链向自己（注意链向自己也算外链，当然同时也是个内链）。这种节点叫做 Spider Trap，如果对这个图进行计算，会发现 D 的 rank 越来越大趋近于 1，而其它节点 rank 值几乎归零。

为了克服这种由于矩阵稀疏性和 Spider Traps 带来的问题，需要对 PageRank 计算方法进行一个平滑处理，具体做法是加入“心灵转移（teleporting）”。所谓心灵转移，就是我们认为在任何一个页面浏览的用户都有可能以一个极小的概率瞬间转移到另外一个随机页面。当然，这两个页面可能不存在超链接，因此不可能真的直接转移过去，心灵转移只是为了算法需要而强加的一种纯数学意义的概率数字。

加入心灵转移后，向量迭代公式变为：

$$v' = (1-\beta)Mv + e\frac{\beta}{N}$$

其中β往往被设置为一个比较小的参数（0.2或更小），e为N维单位向量，加入e的原因是这个公式的前半部分是向量，因此必须将β/N转为向量才能相加。这样，整个计算就变得平滑，因为每次迭代的结果除了依赖转移矩阵外，还依赖一个小概率的心灵转移。

以上图为例，转移矩阵M为：

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 1/2 & 0 & 0 \\ 1/3 & 0 & 1 & 1 \end{bmatrix}$$

设β为0.2，则加权后的M为：

$$M = \begin{bmatrix} 0 & 2/5 & 0 & 0 \\ 4/15 & 0 & 0 & 0 \\ 4/15 & 2/5 & 0 & 0 \\ 4/15 & 0 & 4/5 & 4/5 \end{bmatrix}$$

因此：

$$v' = \begin{bmatrix} 0 & 2/5 & 0 & 0 \\ 4/15 & 0 & 0 & 0 \\ 4/15 & 2/5 & 0 & 0 \\ 4/15 & 0 & 4/5 & 4/5 \end{bmatrix} v + \begin{bmatrix} 1/20 \\ 1/20 \\ 1/20 \\ 1/20 \end{bmatrix}$$

如果按这个公式迭代算下去，会发现Spider Traps的效应被抑制了，从而每个页面都拥有一个合理的pagerank。

3.2.2 PageRank算法Matlab程序

① 从网站的源文件中读取数据

（i为起始点，j为目标点，由于矩阵下标从1开始，所以所有坐标均+1）

```
[i,j] = textread('C:\Users\dell\Desktop\Data.txt','%f%f','headerlines',2)
i = i + 1 ;
j = j + 1 ;
```

② 创建稀疏矩阵P

```
P = sparse(transpose(i),transpose(j),1,75890,75890);
```

③ 约定阻尼系数d = 0.85，处理算法描述中的“心灵转移”问题

```
d = 0.85;
```

④ 判断是否存在死点，并进行处理

```
for k = 1:1:75890
    if(sum(P(:,i)) == 0)
        P(:,i) = 1 ;
    end
end
```

⑤ 对矩阵 P 进行归一化处理

```
for i=1:n
    if(sum(P(i,:)))
        P(i,:)=P(i,:)/sum(P(i,:));
    end
end
disp('归一化计算完成');
```

⑥ 创建稀疏 n*1 向量 e

```
e = sparse(n,1);
```

⑦ 完成对 r 的迭代

```
error = 1.0e-6 ;
P = P';
x = ones(n,1);
r = d*P*x+(1-d)*e*(e'*x)/n;
while (norm(r-x)>error)
    x = r;
    r = d*P*x+(1-d)*e*(e'*x)/n;
end
```

⑧ 创建新矩阵 B，并将 r 赋给 B 的第一列，将 B 的第二列按 1-75890 的顺序编上序号

```
B = zeros(75890,2) ;
B(:,1) = r ;
for k = 1 : 1 : 75890
    B(k,2) = k ;
end
```

⑨ 以 B 的第一列为基准进行逆排序并输出访问概率最大前十结点的序号与概率

```
%排序
C = -sortrows(-B,1);
%输出第一列数据大小
C(1:10,1)
%输出第二列序号
C(1:10,2)
```

3.2.3 PageRank 程序伪代码

```
PROGARM BEGIN:

    Read data from target.txt ;

    Create 稀疏矩阵 P, 创建 n*1 稀疏向量 e, 迭代 n*1 向量 r, n*1 向量 x;

    Write data[行, 列] to P ;

    JUDGE
        FOR A 所有列
            IF SUM(P 的某一列) == 0
                P 的该列所有元素赋为 1
            ELSE
                CONTINUE
            END IF
        END FOR
     END JUDGE

    归一化处理矩阵 P;

    WHILE( norm(r-x) > 误差值 )

        迭代 r (r = d * P * x + (1-d) * e * (e' * x )/n) ;

    END WHILE

    输出最大几率的前十网页的序号

PROGARM END ;
```

3.3 数值实验

3.3.1 PageRank 算法 Matlab 程序运行

访问概率最大前十结点的序号及其概率：

```
ans =
```

'19' 0.0045689
'738' 0.00305
'1720' 0.0021246
'119' 0.0021067
'791' 0.0020381
'137' 0.0020131
'144' 0.0019684
'41' 0.0017406
'1620' 0.0015473
'1180' 0.0014022

3.4 结果分析

从实验结果我们能够得到访问概率最大的十个点的序号，同时也证明了 PageRank 算法的正确性。

最后，实验结束，可以总结一下 PageRank 算法的优缺点。

优点：

PageRank 算法是一个与查询无关的静态算法，所有网页的 PageRank 值通过离线计算获得；有效减少在线查询时的计算量，极大降低了查询响应时间。

缺点：

① 没有区分站内导航链接。很多网站的首页都有很多对站内其他页面的链接，称为站内导航链接。这些链接与不同网站之间的链接相比，肯定是后者更能体现 PageRank 值的传递关系。

② 没有过滤广告链接和功能链接（例如常见的“分享到微博”）。这些链接通常没有什么实际价值，前者链接到广告页面，后者常常链接到某个社交网站首页。

③ 对新网页不友好。一个新网页的一般入链相对较少，即使它的内容的质量很高，要成为一个高 PR 值的页面仍需要很长时间的推广。