## 数值分析实验报告

**摘 要**

本次数值分析实验报告包含三个板块：高斯系列的消元方法，LU分解法和迭代方法(Jacobi方法，Gauss-Sedier方法和SOR方法)以及非线性方程组求解的牛顿法和拟牛顿法。通过该次实验，掌握数值分析知识的同时，且应用MATLAB软件实现面向过程的编程，面向对象的编程。最后，把相关的结果汇总开发了一个线性方程组求解器app，供学习使用，相关成果源码存放Github仓库:git@github.com:curryqka/linearEquation.git。

## 实验3.1

### 问题重述：

考虑线性方程组，，,编制一个能自动选取主元,又能手动选取主元的求解线性代数方程组的Gauss消去过程。

（1）取矩阵，,则方程有解。取计算矩阵的条件数。让程序自动选取主元，分别用顺序Gauss消元、列主元Gauss消元和完全选主元Gauss消元方法求解，结果如何？

（2）现选择程序中手动选取主元的功能，每步消去过程都选取模最小或按模尽可能小的元素作为主元进行消元，观察并记录计算结果，若每步消去过程总选取按模最大的元素作为主元,结果又如何？分析实验的结果。

（3）取矩阵阶数n=20或者更大,重复上述实验过程,观察记录并分析不同的问题及消去过程中选择不同的主元时计算结果的差异，说明主元素的选取在消去过程中的作用.

(4）选取其他你感兴趣的问题或者随机生成的矩阵,计算其条件数，重复上述实验，观察记录并分析实验的结果。

### 1. 算法原理简述

首先,分析各种算法消去过程的计算公式，

**顺序高斯消去法:**

第k步消去中，设增广矩阵中的元素 (若等于零则可以判定系数矩阵为奇异矩阵，停止计算），则对k行以下各行计算消元因子

分别用乘以增广矩阵的第行并加到第行，则可将增广矩阵中第列中以下的元素消为零；重复此方法，从第1步进行到第n-1步,则可以得到最终的增广矩阵，即；

**列主元高斯消去法：**

第k步消去中，在增广矩阵中的子方阵中,选取使得

当时，对中第行与第行交换,然后按照和顺序消去法相同的步骤进行。重复此方法，从第1步进行第n-1步，就可以得到最终的增广矩阵，即；

**完全主元高斯消去法：**

第k步消去中，在增广矩阵中对应的子方阵中，选取使得

若或，则对中第行与第行、第列与第列交换，然后按照和顺序消去法相同的步骤进行即可。重复此方法，从第1步进行到第n-1步,就可以得到最终的增广矩阵,即；

接下来，分析回代过程求解的公式,容易看出，对上述任一种消元法,均有以下计算公式:

衡量方程组精度的指标设置为误差向量的范数，i.e,

### 2.实验程序的设计

* 输入实验要求及初始条件；
* 计算系数矩阵A的条件数及方程组的理论解；
* 对各不同方法编程计算,并输出最终计算结果.
* 本实验程序设计采用MATLAB面向过程的方法，主函数脚本为gauss31\_main.m，各个独立方法封装为函数文件。

|  |  |
| --- | --- |
| 文件名 | 功能 |
| set\_equation.m | 方程组系数矩阵和右端项赋值 |
| order\_gauss.m | 顺序高斯消去 |
| col\_gauss.m | 列主元高斯消去 |
| all\_gauss.m | 完全主元高斯消去 |
| free\_gauss.m | 手动选择主元（按模越大或越小） |
| gauss31\_main.m | 根据题干实现主函数选择逻辑 |

### 3. 程序实验结果及分析

(1)通过调用cal\_cond函数计算得到线性方程组的精确解:，以及各个条件数

|  |  |
| --- | --- |
| 条件数 | 值 |
| 矩阵A的1-条件数 | 2557.500000 |
| 矩阵A的2-条件数 | 1727.556025 |
| 矩阵A的无穷-条件数 | 2557.500000 |

可知系数矩阵的条件数较大,故此问题属于病态问题, b或A的扰动都可能引起解的较大误差；

采用顺序高斯消去法(order\_gauss.m)，计算结果如下，

|  |  |
| --- | --- |
| 未知数索引 | 值 |
| 1 | 1.00000000000000 |
| 2 | 1.00000000000000 |
| 3 | 1.00000000000000 |
| 4 | 1.00000000000000 |
| 5 | 0.999999999999998 |
| 6 | 1.00000000000000 |
| 7 | 0.999999999999993 |
| 8 | 1.00000000000001 |
| 9 | 0.999999999999979 |
| 10 | 1.00000000000003 |

使用无穷范数衡量误差,得到

可以发现，采用顺序高斯消元法求得的解与精确解之间误差较小。通过进一步观察，可以发现，按照顺序高斯消去法计算时，其选取的主元值和矩阵中其他元素大小相近，因此顺序高斯消去法方式并没有对结果造成特别大的影响。

若采用列主元高斯消元法(col\_gauss.m)，则结果为下表所示：

|  |  |
| --- | --- |
| 未知数索引 | 值 |
| 1 | 1.00000000000000 |
| 2 | 1.00000000000000 |
| 3 | 1.00000000000000 |
| 4 | 1.00000000000000 |
| 5 | 1.00000000000000 |
| 6 | 1.00000000000000 |
| 7 | 1.00000000000000 |
| 8 | 1.00000000000000 |
| 9 | 1.00000000000000 |
| 10 | 1.00000000000000 |

使用无穷范数衡量误差,得到

可以发现，采用列主元高斯消元法求得的解与精确解直接几乎无差。

若使用完全主元高斯消元法(all\_gauss.m)，则结果为下表所示：

|  |  |
| --- | --- |
| 未知数索引 | 值 |
| 1 | 1.00000000000000 |
| 2 | 1.00000000000000 |
| 3 | 1.00000000000000 |
| 4 | 1.00000000000000 |
| 5 | 1.00000000000000 |
| 6 | 1.00000000000000 |
| 7 | 1.00000000000000 |
| 8 | 1.00000000000000 |
| 9 | 1.00000000000000 |
| 10 | 1.00000000000000 |

使用无穷范数衡量误差,得到

可以发现，采用完全主元高斯消元法求得的解与精确解直接几乎无差。

(2)若每步都选取模最小或尽可能小的元素为主元(free\_gauss.m)，则结果为下表所示：

|  |  |
| --- | --- |
| 未知数索引 | 值 |
| 1 | 1 |
| 2 | 1.00000000000000 |
| 3 | 1.00000000000000 |
| 4 | 1.00000000000000 |
| 5 | 0.999999999999998 |
| 6 | 1.00000000000000 |
| 7 | 0.999999999999993 |
| 8 | 1.00000000000001 |
| 9 | 0.999999999999979 |
| 10 | 1.00000000000003 |

使用无穷范数衡量误差,得到

而按照每步都选取模最大，尽可能大的元素为主元，则与完全选主元方法等价

实验结果表明，列主元消去法和完全主元消去法均获得了精确解，而顺序高斯消去法和以尽可能小的元素为主元的消去法则未能达到精确解。在后两种消去法中，由于程序计算时存在舍入误差，对最终结果产生了一定的影响。然而，由于方程组的维度较低且元素之间差异不大，因此误差仍然相对较小。

为了进一步分析，我们计算了上述四种方法在每一步中选取的主元数值，并将结果列于下表进行比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 第n次消元 | 顺序 | 列主元 | 完全主元 | 模最小 |
| 1 | 6.000000000000000 | 8 | 8 | 6.000000000000000 |
| 2 | 4.666666666666667 | 8 | 8 | 4.666666666666667 |
| 3 | 4.285714285714286 | 8 | 8 | 4.285714285714286 |
| 4 | 4.133333333333333 | 8 | 8 | 4.133333333333333 |
| 5 | 4.064516129032258 | 8 | 8 | 4.064516129032258 |
| 6 | 4.031746031746032 | 8 | 8 | 4.031746031746032 |
| 7 | 4.015748031496063 | 8 | 8 | 4.015748031496063 |
| 8 | 4.007843137254902 | 8 | 8 | 4.007843137254902 |
| 9 | 4.003913894324853 | 8 | 8 | 4.003913894324853 |
| 10 | 4.001955034213099 | 0.015617370605469 | 0.015617370605469 | 4.001955034213099 |

从上表可以观察到，在这个方程组中，顺序高斯消去法选取的主元恰好是模最小的元素。另外，由于列主元消去法和完全主元消去法选取的元素为8，与4相比数量级差异较小，因此计算过程中的累积误差也较小，导致四种方法的输出结果都相对精确。

在这里，我们将解释顺序法和模最小法计算结果完全一致的原因。该矩阵在消元过程中，每次选取主元的一列只有两个非零元素，对角线上的元素约为4，而其正下方的元素为8，该列其余位置的元素均为0。在这种情况下，默认的主元就是该列中最小的元素，因此两种方法所得到的计算结果是一致的。

理论上，完全主元高斯消去法的误差最小，其次是列主元高斯消去法，而选取模最小的元素作为主元时的误差最大。然而，由于方程组的特殊性（元素之间差异不大且维度较低），在这里这个理论现象还没得到充分地体现。

(3)当，修改输入矩阵的维度，重复上述实验过程。各种方法计算的解和误差汇总在下表。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 顺序高斯消去法 | 列主元高斯消去 | 完全主元高斯消去 | 选取模最小或尽可能小元素作为主元消去 |
| X | 1.000000000000000 1.000000000000000 1.000000000000000 1.000000000000001 0.999999999999998 1.000000000000004 0.999999999999993 1.000000000000014 0.999999999999972 1.000000000000057 0.999999999999886  1.000000000000227 0.999999999999547 1.000000000000902 0.999999999998209 1.000000000003524 0.999999999993179 1.000000000012732 0.999999999978173 1.000000000029102 | 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 | 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 | 1.000000000000000 1.000000000000000 1.000000000000000 1.000000000000001 0.999999999999998 1.000000000000004 0.999999999999993 1.000000000000014 0.999999999999972 1.000000000000057 0.999999999999886  1.000000000000227 0.999999999999547 1.000000000000902 0.999999999998209 1.000000000003524 0.999999999993179 1.000000000012732 0.999999999978173 1.000000000029102 |
|  | 2.910205409989430e-11 | 0 | 0 | 2.910205409989430e-11 |

观察到在这种情况下，列主元和完全主元的计算结果仍为精确值，而顺序高斯消去和模尽可能小的元素方法仍然存在一定的误差，并且两者的误差是一致的。与时相比，时的误差增加了大约1000倍，这是由于计算过程中舍入误差的累积效应。因此，如果进一步增加矩阵的维度，误差累积现象将更严重。

(4)首先对于题干的矩阵，对其加大维度，考察其方程组解和误差的情况；其次输入其他的自定义矩阵，探究其方程组解和误差的情况。

设置维度为10，20，30，40，50，100，重复前述实验过程，得到下列的结果表。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 维度 | 2-条件数 | 顺序消去error | 列主元error | 完全主元error | 模尽量小error |
|  | 1.72e+3 | 2.84e-14 | 0 | 0 | 2.84e-14 |
|  | 1.78e+6 | 2.91e-11 | 0 | 0 | 2.91e-11 |
|  | 1.84e+9 | 2.98e-08 | 0 | 0 | 2.98e-08 |
|  | 1.88e+12 | 3.05e-05 | 0 | 0 | 3.05e-05 |
|  | 1.89e+15 | 3.12e-02 | 0 | 0 | 3.12e-02 |
|  | 2.49e+16 | 3.52e+13 | 1.83e-01 | 1.83e-01 | 3.52e+13 |

考虑线性方程组，，，其中矩阵为

，

设置维度为10，20，30，40，50，100，重复前述实验过程，得到下列关于新矩阵的结果表。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 维度 | 2-条件数 | 顺序消去error | 列主元error | 完全主元error | 模尽量小error |
|  | 17.99 | 4.44e-16 | 0 | 0 | 4.44e-16 |
|  | 60.07 | 6.66e-16 | 2.22e-16 | 2.22e-16 | 6.66e-16 |
|  | 193.30 | 5.77e-15 | 2.66e-15 | 2.66e-15 | 5.77e-15 |
|  | 951.02 | 1.93e-14 | 1.38e-14 | 1.38e-14 | 2.00e-14 |
|  | 3417.35 | 7.55e-14 | 4.75e-14 | 4.75e-14 | 7.43e-14 |
|  | 9080.53 | 2.05e-13 | 1.35e-13 | 1.35e-13 | 2.04e-13 |

## 二、实验3.3

### 问题重述：

考虑方程组的解,其中系数矩阵H为Hilbert矩阵：



这是一个著名的病态问题.通过首先给定解（例如取为各个分量均为1）再计算出右端的办法给出确定的问题。

(1)选择问题的维度为6,分别用Gauss消去法（即LU分解）、J迭代法、GS迭代法和SOR迭代法求解方程组，其各自的结果如何？将计算结果与问题的解比较,结论如何。

(2)逐步增大问题的维度，仍用上述的方法来解它们，计算的结果如何？计算的结果说明的什么?

(3)讨论病态问题求解的算法。

### 1. 算法原理简述

对任意线性方程组，分析各种方法的计算公式如下，

（1)Gauss消去法：

首先对系数矩阵进行LU分解，有,则原方程转化为，令，则原方程可以分为两步回代求解：

其中，为下三角部分，为上三角部分。LU矩阵可以经过反复行列计算得到。

1. J迭代法:

迭代式子

分解,其中 ，进行迭代计算，直到误差满足要求。

（3）GS迭代法：

迭代式子

分解,其中 ，进行迭代计算，直到误差满足要求。

（4）SOR迭代法：

迭代式子

分解,再构造迭代矩阵 ，其中，进行迭代计算，直到误差满足要求.

### 2.实验程序的设计

一、根据维度n确定矩阵H的各个元素和b的各个分量值；

二、选择计算方法（ Gauss消去法（LU分解），J迭代法，GS迭代法，SOR迭代法），对迭代法设定初值，对于SOR方法还需要设定松弛因子;

三、进行计算，直至满足误差要求(对迭代法，设定相邻两次迭代结果之差的无穷范数小于，取0.0001；对SOR方法，设定为输出迭代1000次之后的结果及误差值），输出实验结果。

四、为了使得程序清晰简短，使得上述功能顺利便捷地实现。此实验设计采用MATLAB面向对象的编程方法classdef，类文件是*itermodel.m*。

|  |  |
| --- | --- |
| 文件和类函数 | 功能 |
| Itermodel.m | itermodel类确定一个方程组 |
| set\_matrix | 如果没有给定右端项，则给定右端项 |
| get\_LU | 得到系数矩阵的LU分解 |
| get\_LU\_check | 这里还写了循环实现的版本，与上面的算法（矩阵加速）进行比较 |
| gauss\_method | 高斯消去的LU分解 |
| jacobi\_method | J法迭代 |
| gs\_method | GS法迭代 |

### 3.程序实验结果及分析

(1)n=6时，高斯消去法，即LU分解计算的结果如下表所示，

|  |  |
| --- | --- |
| 未知数索引 | 值 |
| 1 | 0.999999999999414 |
| 2 | 1.00000000001708 |
| 3 | 0.999999999882853 |
| 4 | 1.00000000030780 |
| 5 | 0.999999999657641 |
| 6 | 1.00000000013572 |

使用无穷范数衡量误差,得到

设定迭代初值为零,计算得到 J法的迭代矩阵B的谱半径为4.30853＞1，所以J法不收敛;

设定迭代初值为零，计算得到GS法的迭代矩阵G的谱半径为：0.999998＜1，故GS法收敛，经过761次迭代达到收敛，GS法的计算结果如下表所示，

|  |  |
| --- | --- |
| 未知数索引 | 值 |
| 1 | 0.999999999999407 |
| 2 | 1.00000000001732 |
| 3 | 0.999999999881006 |
| 4 | 1.00000000031288 |
| 5 | 0.999999999651809 |
| 6 | 1.00000000013808 |

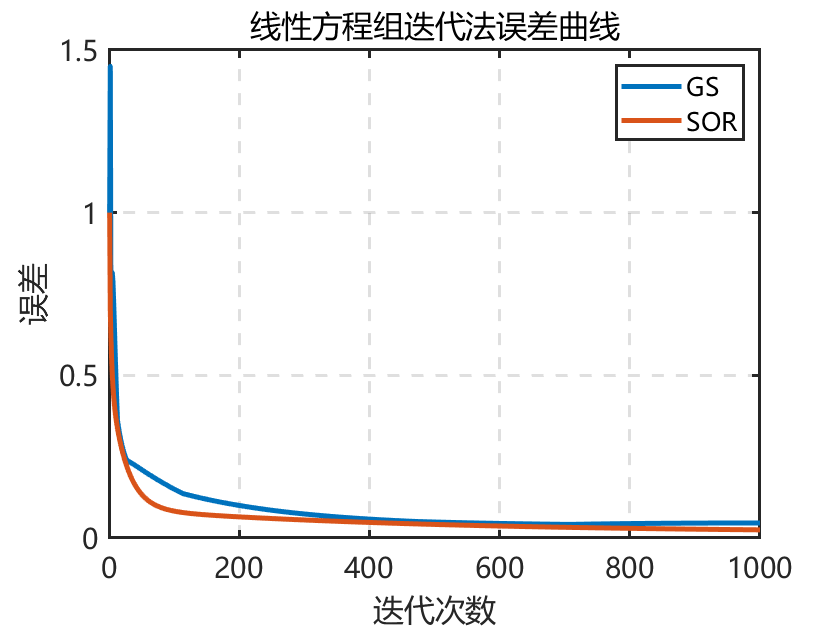
使用无穷范数衡量误差,得到

4.629045007723664e-02

设定迭代初值为零向量，并设定,计算得到SOR法迭代矩阵谱半径为0.9999994338152238，SOR法收敛，经过811次迭代达到收敛，SOR法的计算结果如下表所示，

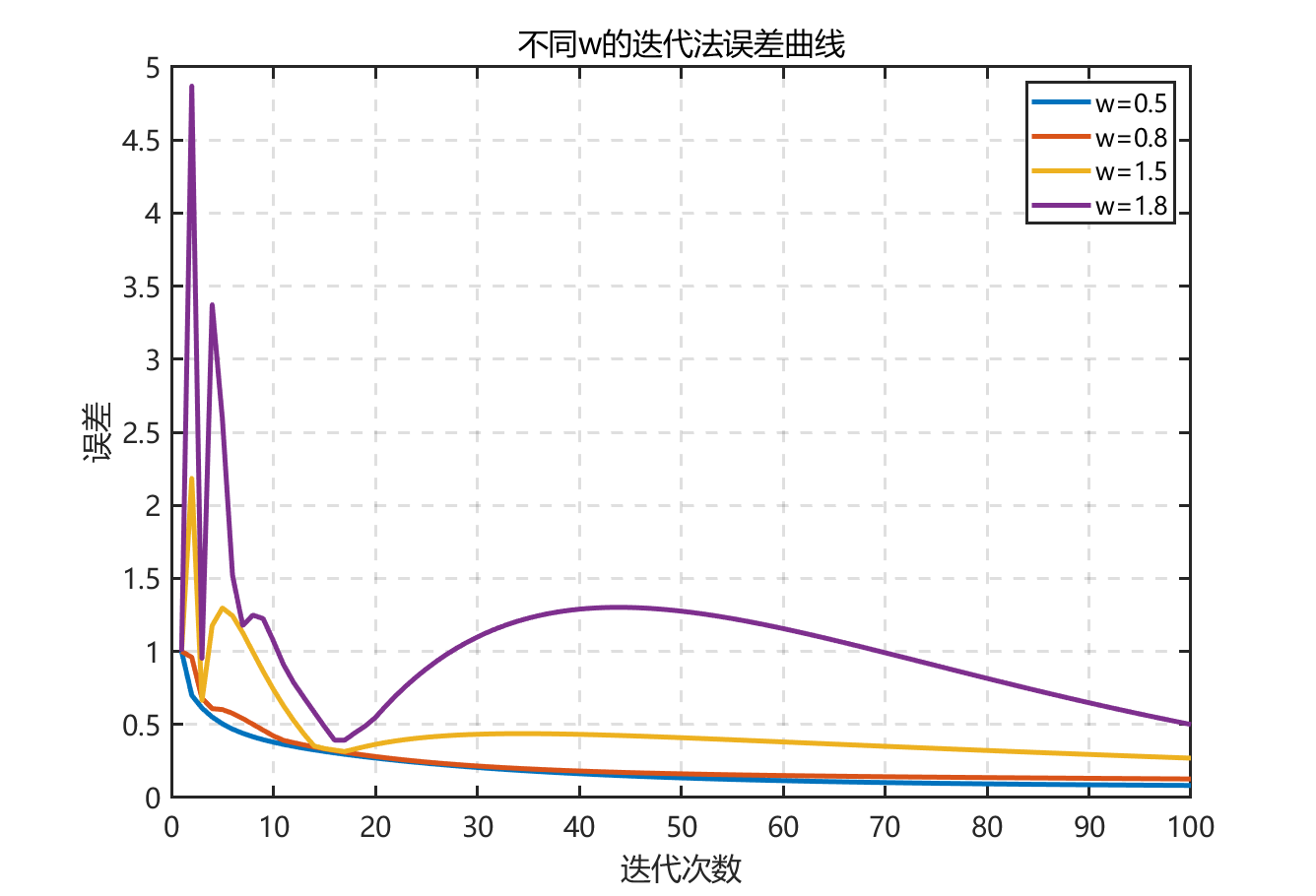
|  |  |
| --- | --- |
| 未知数索引 | 值 |
| 1 | 1.00000001227628 |
| 2 | 0.999999653770269 |
| 3 | 1.00000232294503 |
| 4 | 0.999993995104265 |
| 5 | 1.00000659650233 |
| 6 | 0.999997410682345 |

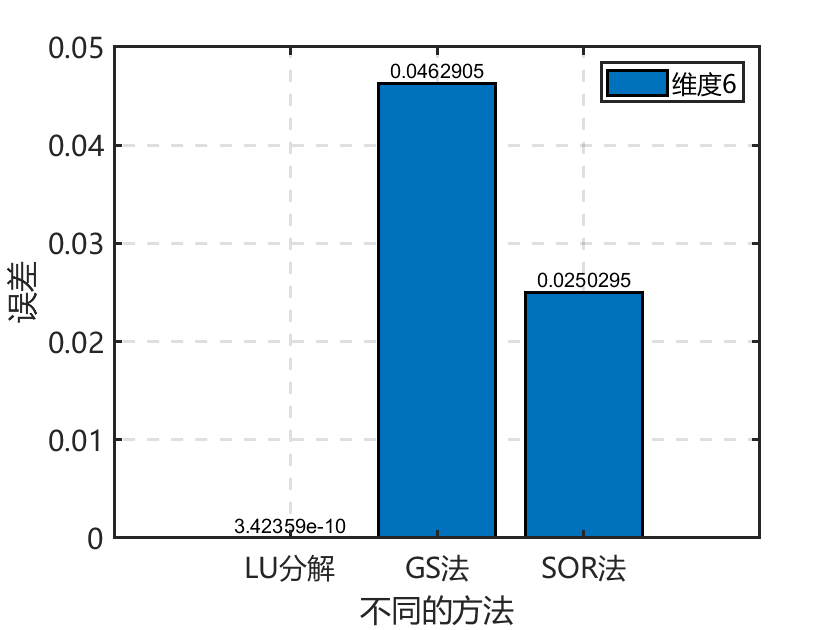
使用无穷范数衡量误差,得到



对SOR方法，可变，改变值，迭代100次，计算结果可以列表如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 迭代次数 | 100 | 100 | 100 | 100 |
| 迭代矩阵的谱半径 | 0.999999433815223 | 0.999998867083155 | 0.999996830135013 | 0.999982309342386 |
| X | 1.003653917714694 0.974666041209353 1.011814573842440 1.042837929171827 1.017190220902681 0.945462001336268 | 1.014676015634604  0.896636864424096  1.090444578936265  1.107070542628148  1.006315452225331  0.873244842279255 | 1.028022215505147  0.790604920509843  1.267167365524072  1.061689730857891  0.990084054872602  0.846005956774467 | 1.051857392323966  0.653408758549156  1.486449891152510  0.783650360698119  1.349665420488270  0.664202350634588 |
|  | 0.054537998663732 | 0.126755157720745 | 0.267167365524072 | 0.486449891152510 |





计算可得，矩阵H的条件数为>>1，因此为病态矩阵，产生病态问题。从上图可以看出，四种求解方法都存在误差。下面将对误差来源进行分析。

LU分解方法的误差主要源于Hilbert矩阵中的元素从分数形式转换为小数形式时，无法完全除尽的情况下会产生舍入误差。这个问题在进行LU分解时同样存在。因此，最终得到的结果不是方程的精确解。不过，结果表明该方法的误差非常小。

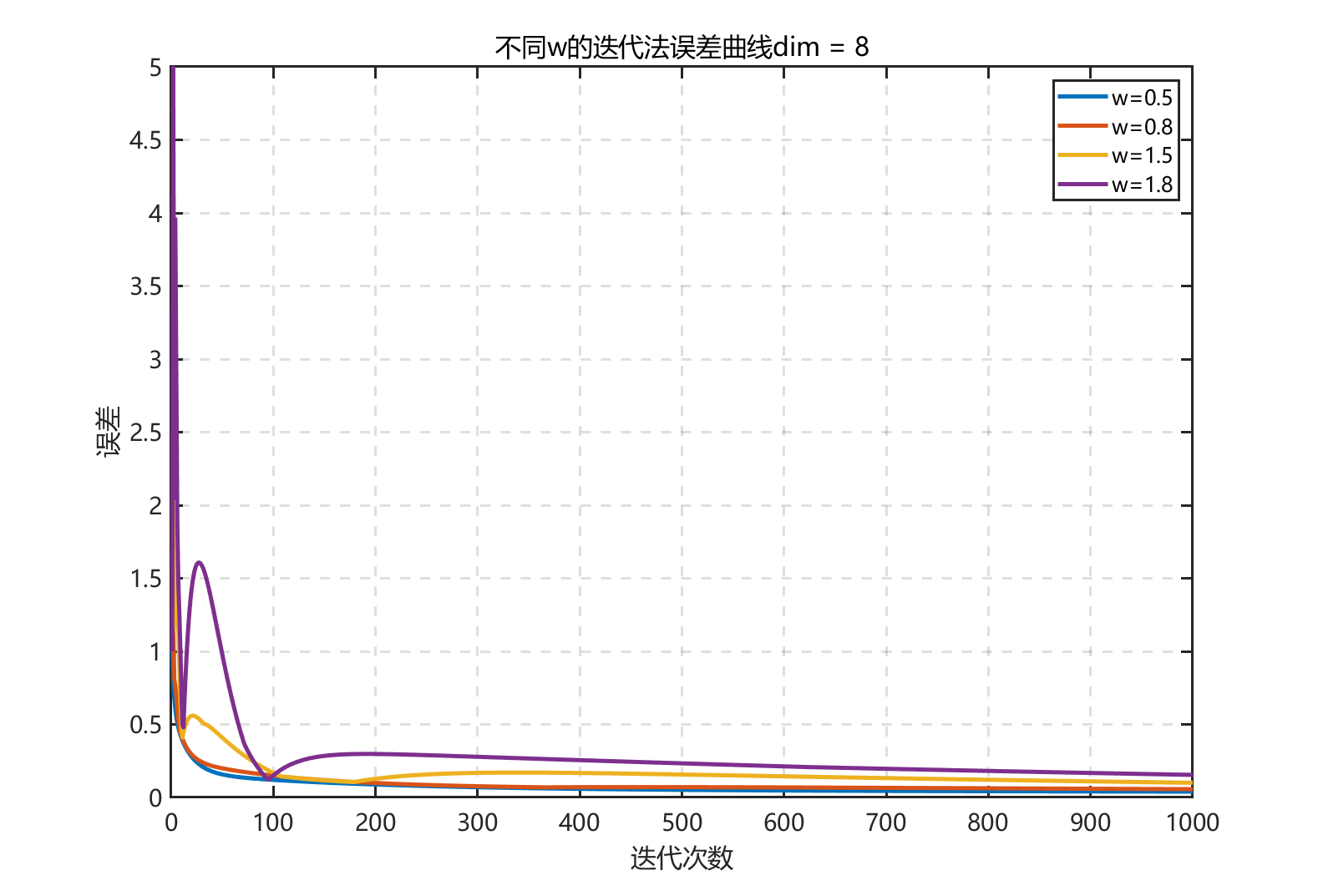
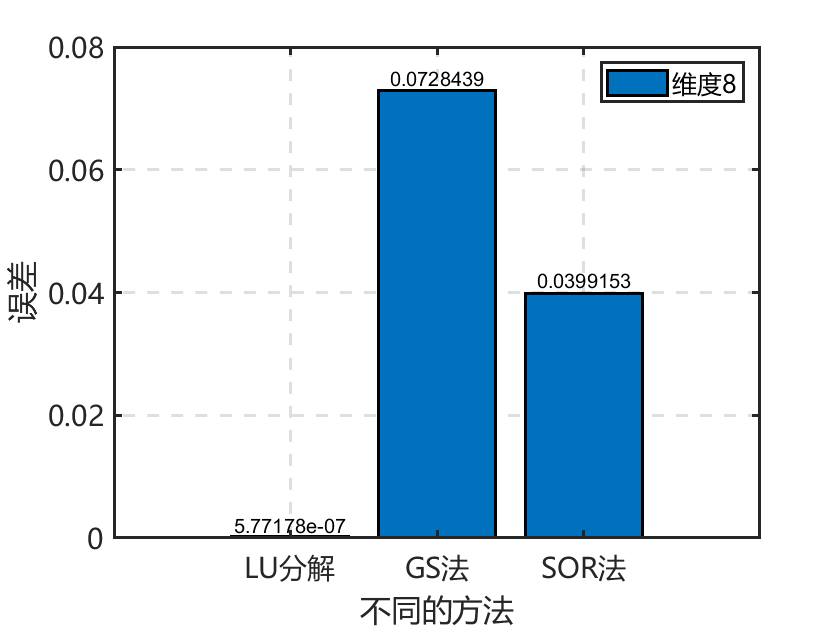
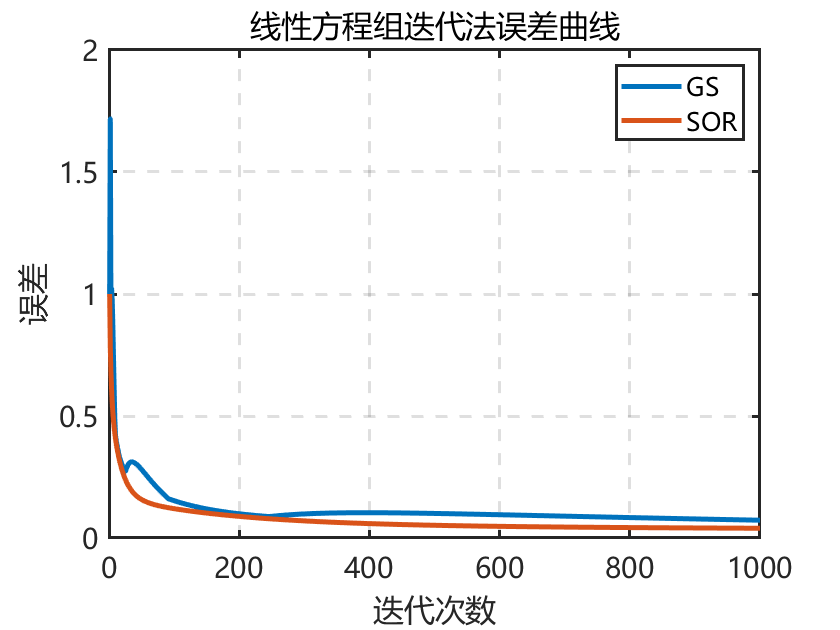
Jacobi迭代矩阵的谱半径为4.30853，故此迭代法不收敛；

GS迭代法在迭代次数为1000时得到方程的近似解，其误差约为0.046，相对较大。GS迭代矩阵的谱半径为0.999998，这也意味着它非常接近于1，所以GS迭代法收敛速度较慢；

SOR方法在进行1000次迭代后的误差约为0.025，相对较大。SOR迭代矩阵的谱半径为0.999999, 这也意味着它非常接近于1。因此，当松弛因子w=0.5时，SOR方法的收敛速度并不是非常快，但与GS方法相比，已经明显改善了迭代速度。此外，不同的松弛因子会影响SOR方法的迭代速度，通过选择最佳的松弛因子，可以实现更快的收敛。

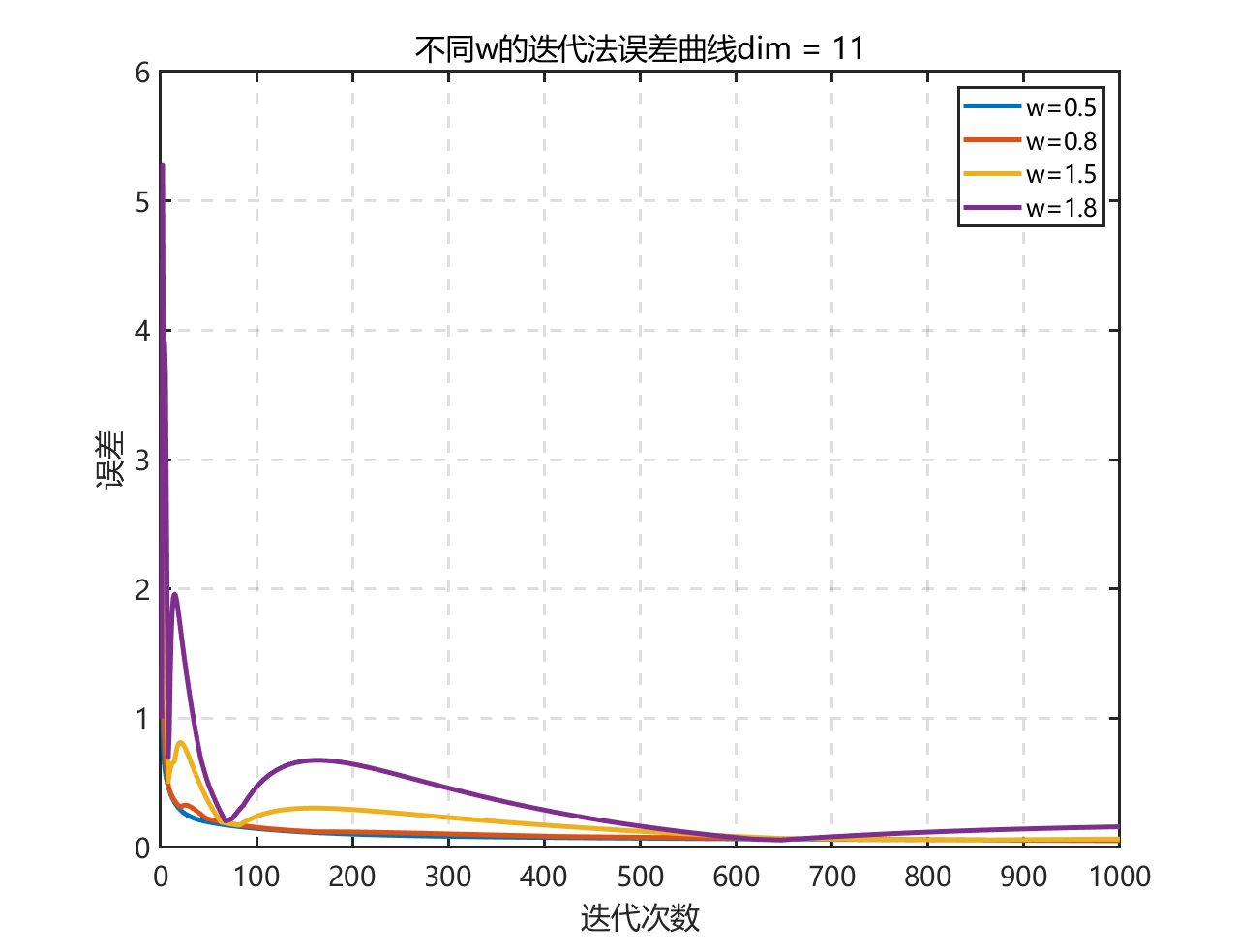
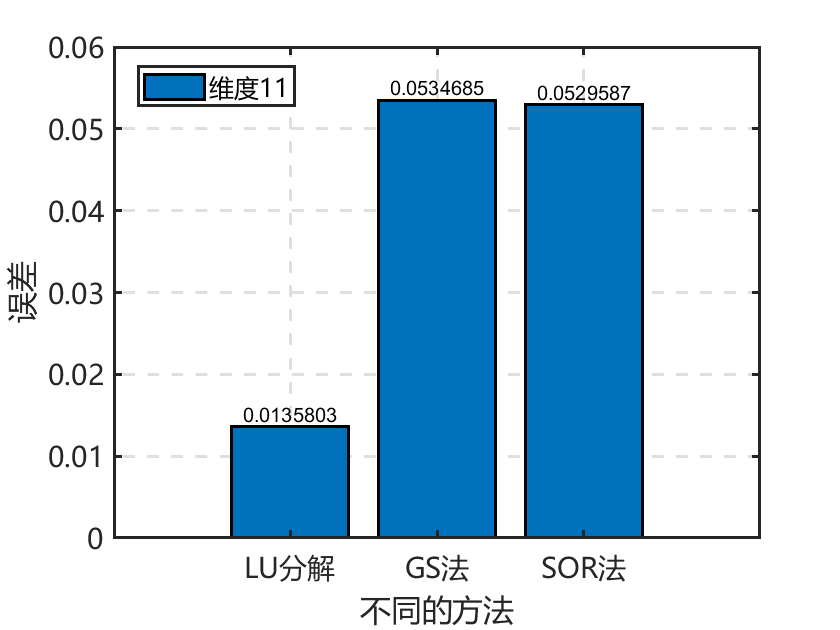
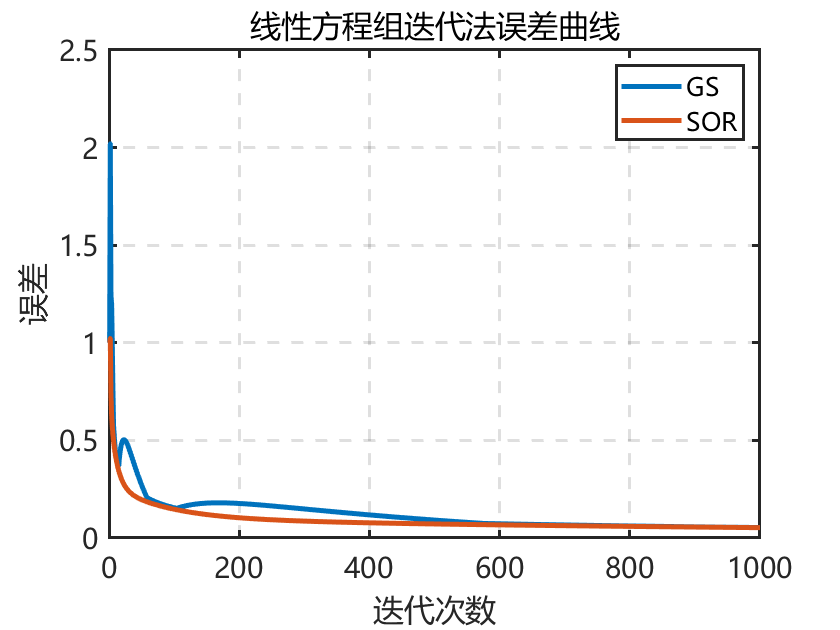
(2)逐步增大矩阵的维度，取矩阵维度为8，11，15，18。绘制各种迭代方法迭代1000次过程的误差变化情况和几种不同的方法最终的解误差比较柱状图。

当n=8时汇总实验(1)中的结果，J法仍不收敛，GS法经过1792次迭代达到预设的误差精度，SOR法经过679次迭代达到预设的误差精度。



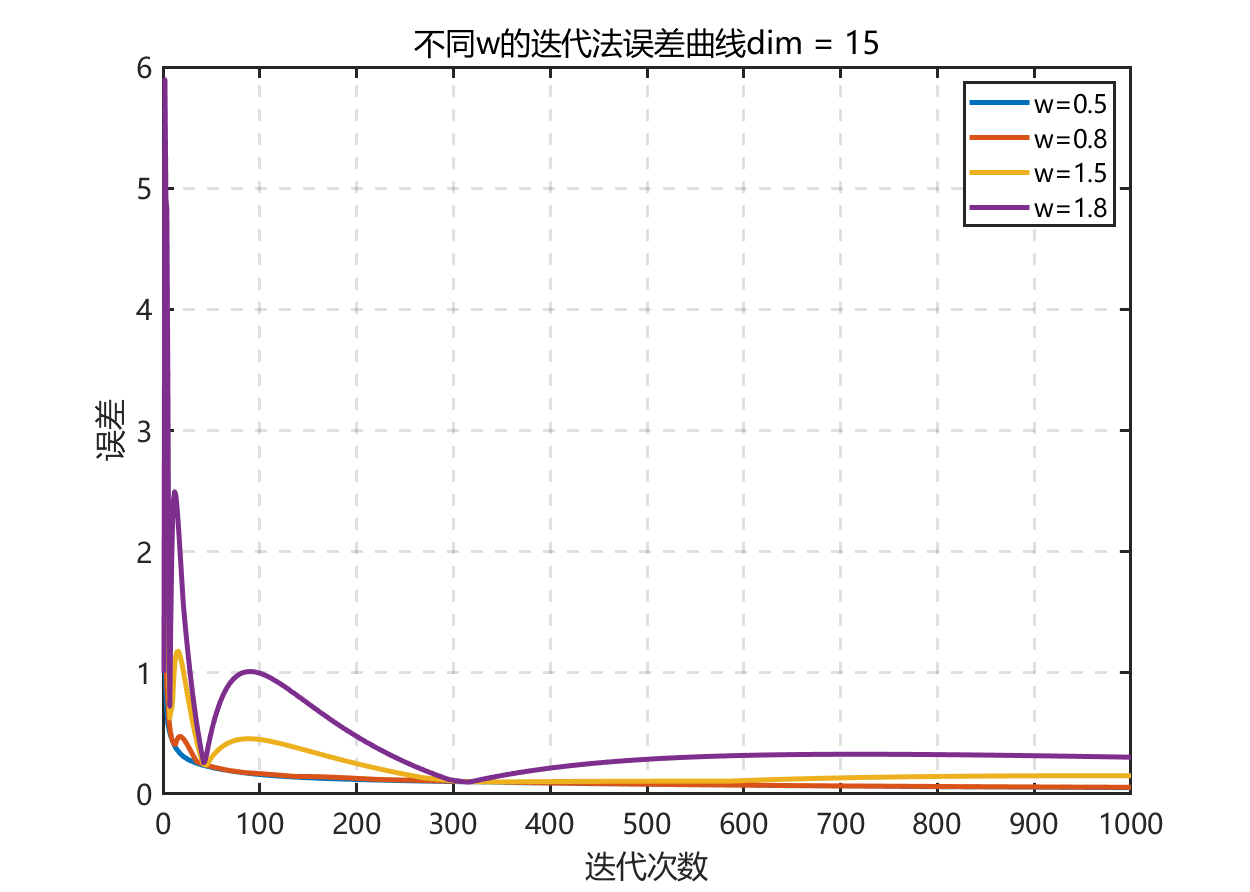
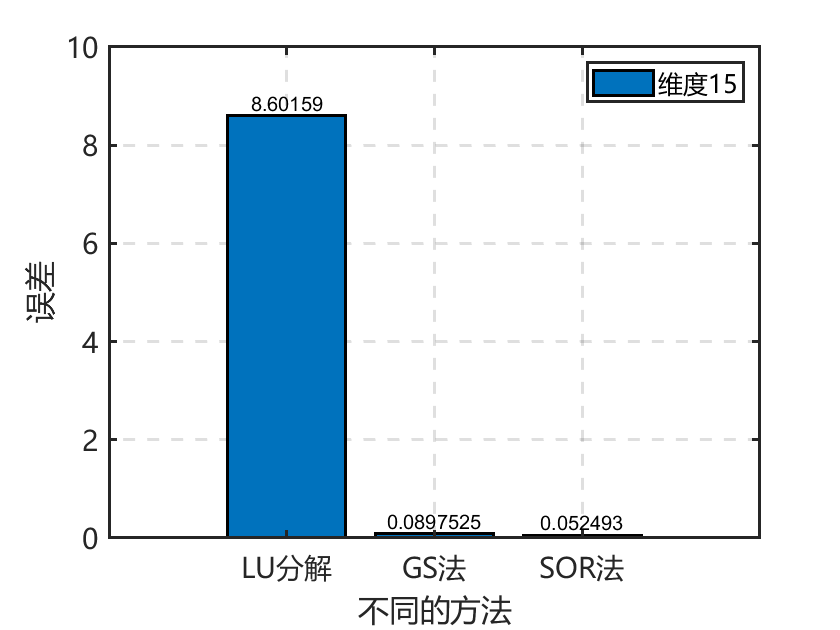
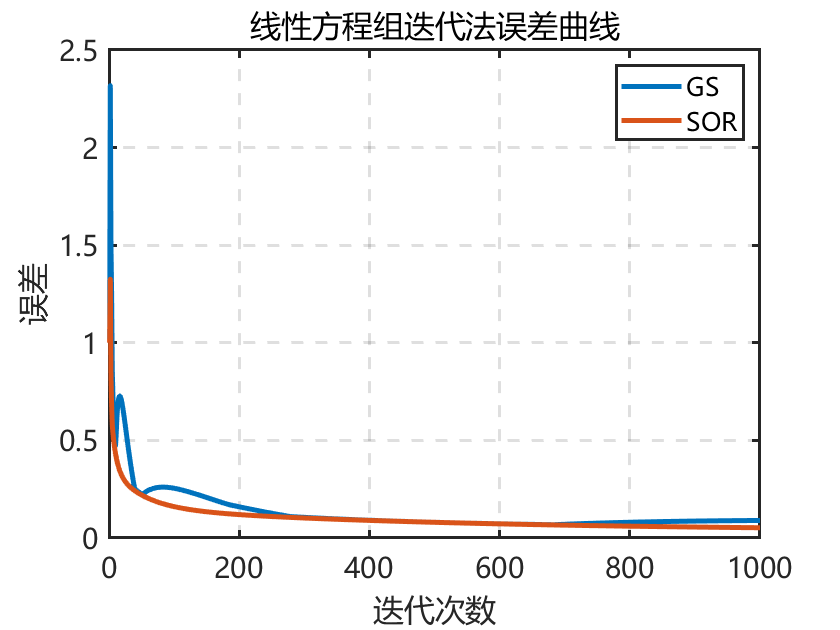
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | LU分解 | J法 | GS法 | SOR法（w=0.5） |
| 计算结果 | 0.999999999958815  1.00000000218803  0.999999971623834  1.00000015273327  0.999999590607157  1.00000057717834  0.999999590498117  1.00000011524019 | NaN | 0.997238186541099  1.03286442768534  0.927156095196408  1.00241712224855  1.04581542153932  1.04110726720578  1.00352672246015  0.947894765213143 | 0.998585114277986  1.01722329724673  0.964189491042261  0.991717141831002  1.02600479184758  1.03267511136432  1.00807682288710  0.960084698975707 |
| 迭代次数 |  |  | 1000 | 1000 |
| 谱半径 |  | 6.04213 | 0.999999997624837 | 0.999999999208775 |
|  |  |  |  | 0.0399 |

当n=11时，重复实验(1)中的结果，J法仍不收敛，GS法经过1780次迭代达到预设的误差精度，SOR法经过1843次迭代达到预设的误差精度。



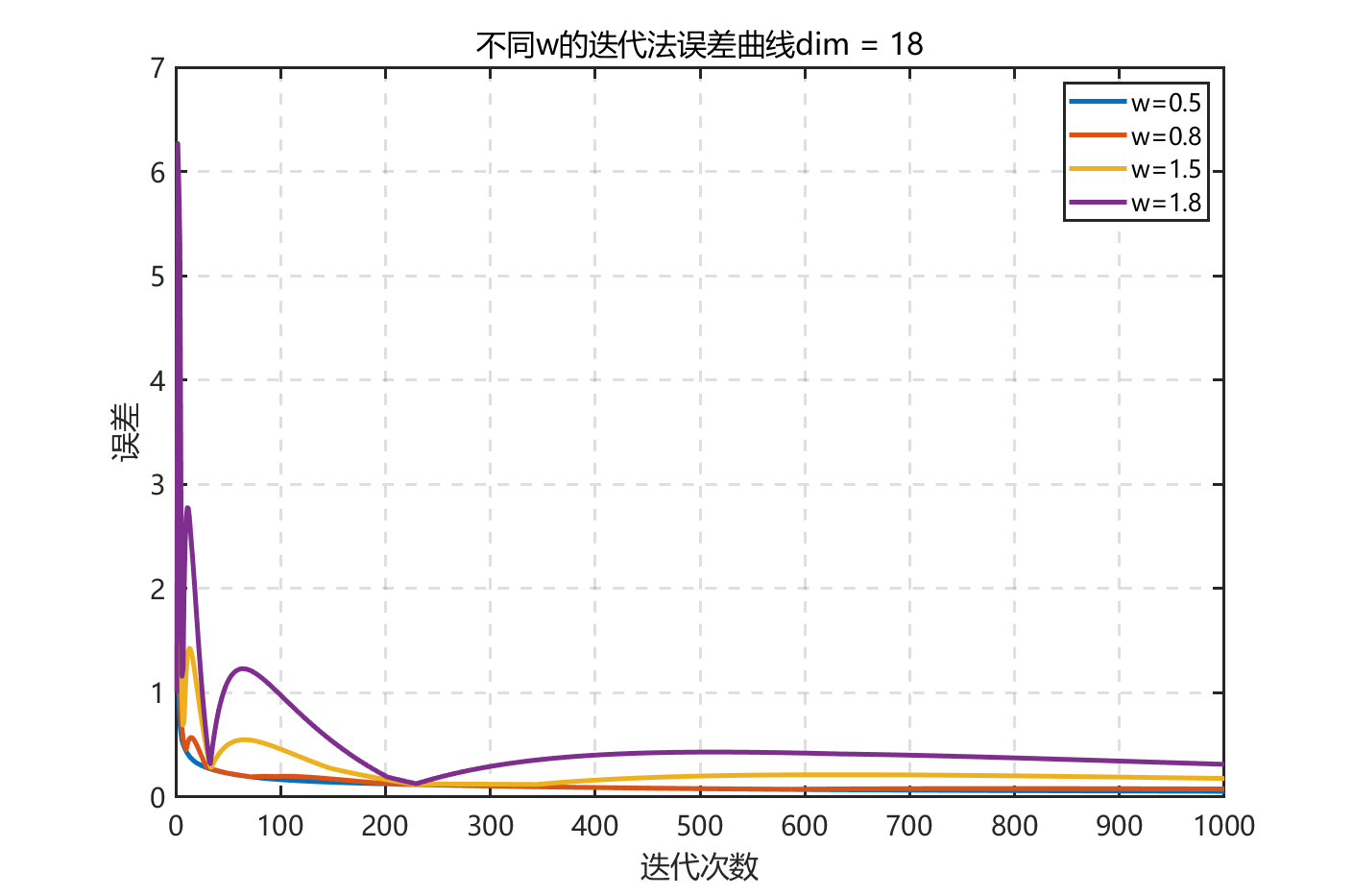
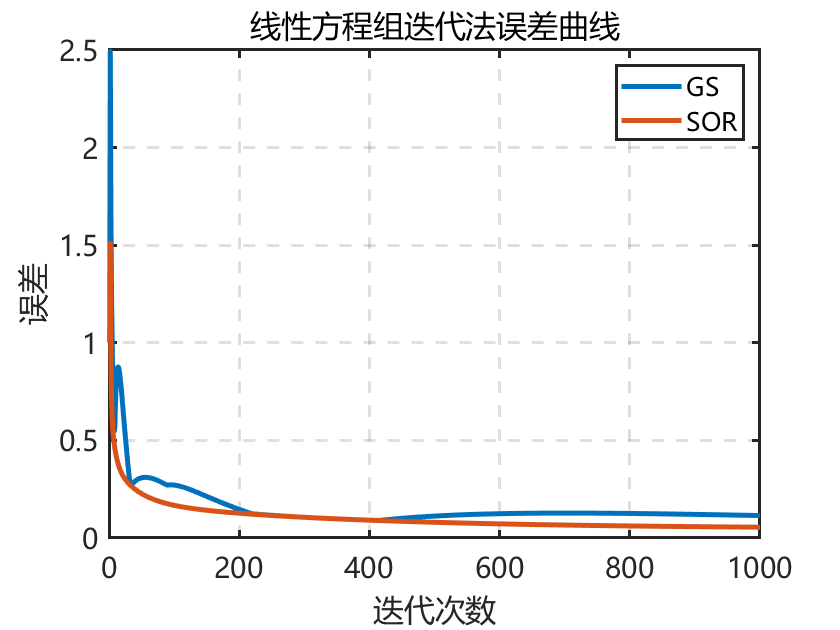
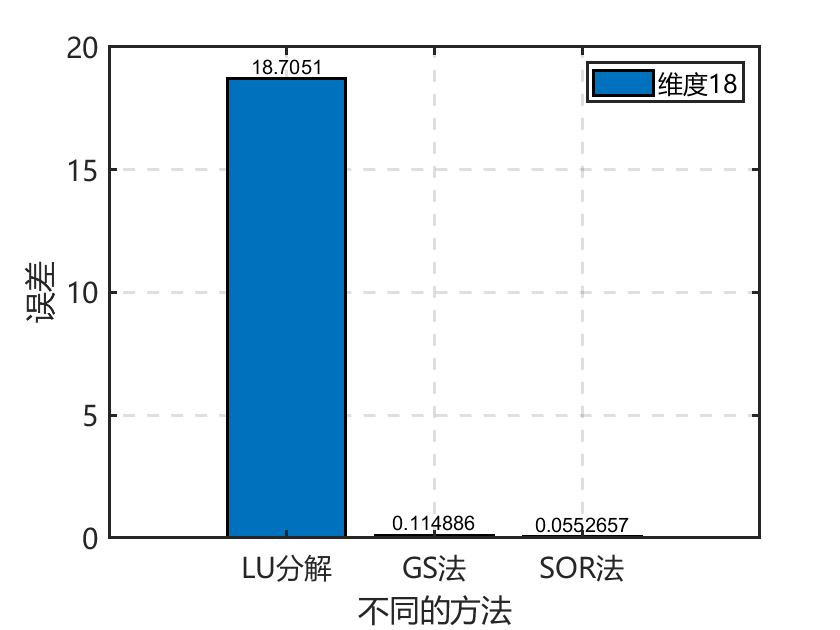
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | LU分解 | J法 | GS法 | SOR法（w=0.5） |
| 计算结果 | 0.999999991702147  1.00000084009811  0.999978782772638  1.00023203116137  0.998643166718095  1.00469510670745  0.989917306252405  1.01358027827325  0.988839656910635  1.00511453164685  0.998998304059348 | NaN | 0.997331585918091  1.02016331851805  0.990348390751406  0.956221720499595  0.987067091873407  1.02208362497637  1.03927228125814  1.03618486944844  1.01659736430454  0.985317845293761  0.946531539708760 | 0.997070829253337  1.02482215825969  0.974209291957971  0.969193115001991  0.992937514938847  1.02003788597375  1.03535445326100  1.03382507640431  1.01629507580315  0.986091590436317  0.947041306205549 |
| 迭代次数 |  |  | 1000 | 1000 |
| 谱半径 |  | 8.6496 | 0.999999999999898 | 0.999999999999966 |
|  |  |  |  | 0.0530 |

当n=15时，重复实验(1)中的结果，J法仍不收敛，GS法经过2576次迭代达到预设的误差精度，SOR法经过1580次迭代达到预设的误差精度。



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | LU分解 | J法 | GS法 | SOR法（w=0.5） |
| 计算结果 | 0.999999906030033  1.00001088035959  0.999713203328494  1.00266131602045  0.995319127256521  0.915072789354481  1.71785870279773  -1.76105317458505  7.22764342635122  -7.60159048692945  7.87388322769765  -1.27078899475217  0.191446938563066  1.94573514752011  0.764087870105062 | NaN | 1.00160072343525  0.970945200126149  1.08975247956403  0.970193229688543  0.939876133012553  0.960121743738896  0.993114413880972  1.02097450692593  1.03749773651291  1.04190526085476  1.03559520230657  1.02062454130321  0.999052297513606  0.972688774337110  0.943028392805565 | 0.998511700869848  1.00480284841386  1.01369333038742  0.990548174781840  0.973909991916994  0.976893857920405  0.993009721716317  1.01209383259011  1.02672645935781  1.03315011398394  1.03035573857501  1.01894983492861  1.00029500369177  0.975974985390462  0.947506993404381 |
| 迭代次数 |  |  | 1000 | 1000 |
| 谱半径 |  | 12.1330 | 1 | 1.000000000000000 |
|  | 8.61 |  | 0.0898 | 0.0525 |

当n=18时，重复实验(1)中的结果，重复实验(1)中的结果，J法仍不收敛，GS法经过3406次迭代达到预设的误差精度，SOR法经过1550次迭代达到预设的误差精度。



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | LU分解 | J法 | GS法 | SOR法（w=0.5） |
| 计算结果 | 0.999999993225408  0.999999012114641  1.00012891928710  0.996000282408603  1.05711020215382  0.540677486733538  3.28475422976803  -6.32639437030452  16.1290925365322  -17.7050950453816  10.2474633099838  9.80335349157807  -16.1943063432206  9.29586954885840  4.96870288554604  -5.65228489636280  4.06823676295329  0.486692015209126 | NaN | 1.00420165170304  0.948023998736634  1.11488621974144  1.00117826785613  0.943116310852264  0.939559893544536  0.961801713312608  0.990520469868197  1.01580574914131  1.03351128447179  1.04256338381597  1.04337812147572  1.03700262512382  1.02467244599443  1.00759643122738  0.986862394846592  0.963405077768748  0.938004847833065 | 1.00065492753373  0.986719561932899  1.03056949647015  1.01548844543484  0.983303515852728  0.965950487774958  0.968132279755893  0.982984410735307  1.00212921993303  1.01933892912093  1.03105057653174  1.03579450903222  1.03345176894076  1.02465041283936  1.01035523192959  0.991619085476704  0.969447399288330  0.944734294113372 |
| 迭代次数 |  |  | 1000 | 1000 |
| 谱半径 |  | 14.748 | 1.000000000000000 | 1.000000000000002 |
|  | 18.7 |  | 0.115 | 0.0553 |

(3)系数矩阵的条件数较大时为病态方程。由实验可知，LU分解方法在解上述高维度方程时，结果存在很大的误差。而对于收敛的迭代法，可以通过选取最优松弛因子的方法来求解，虽然迭代次数相对较多，但是结果较为精确。

总体来看，对于一般病态方程组的求解，可以采用以下方式:

①低维度下采用LU分解法直接求解是可行的；

②Jacobi迭代方法不适宜于求解病态问题；

③GS迭代方法可以解决维度较低的病态问题，但其谱半径非常趋近于1，导致迭代算法收敛速度很慢，维度较大的时候，GS法也不再收敛;

④SOR方法较适合于求解病态问题,特别是矩阵维度较高的时候，其优势更为明显；

⑤采用高精度的运算,如选用双倍或更多倍字长的运算，可以提高收敛速度；

⑥可以对原方程组作某些预处理，从而有效降低系数矩阵的条件数。

### 4. 实验结论

（1）对于Hilbert矩阵问题，其条件数随着维度的增加迅速增加，因此病态性会变得越来越明显。在维度较低时，可以使用LU分解法、Gauss-Seidel迭代法和逐次超松弛迭代法（SOR迭代法），并且优先考虑使用LU分解法。然而，如果需要解决高维Hilbert矩阵问题，则只有SOR迭代法可以求解。

（2）SOR方法特别适用于解决病态问题，尤其是在高维矩阵的情况下，其优势更加显著。实验结果表明，随着矩阵维数的增加，SOR方法误差基本保持稳定，说明它是解决病态问题的一个合适选择。

## 三、实验4.1

### 问题重述：

对牛顿法和拟牛顿法。进行非线性方程组的数值求解

Q1：用上述两种方法，分别计算下面的两个例子。在达到精度相同的前提下，比较其迭代次数、CPU时间等。

Q2：取其他初值，结果又如何?反复选取不同的初值，比较其结果.

Q3：总结归纳你的实验结果,试说明各种方法适用的问题.

**1. 算法原理简述**

对需要求解的非线性方程组而言，牛顿法和拟牛顿法的迭代公式如下，

(1)牛顿法：牛顿法为单步迭代法，需要取一个初值。

(2)拟牛顿法

其中，

拟牛顿法不需要求解的导数，因此节省了大量的运算时间，但需要给定矩阵的初值，取为。

### 2.实验程序的设计

一、输入初值；

二、根据误差要求，按公式进行迭代计算；

三、输出解数据和迭代次数，CPU运算时间；

四、为了使得程序清晰简短，使得上述功能顺利便捷地实现。此实验设计采用MATLAB面向对象的编程方法classdef，类文件是*NewtonMethod.m*。

|  |  |
| --- | --- |
| 文件和类函数 | 功能 |
| NewtonMethod.m | NewtonMethod类确定一个方程组 |
| get\_new\_F | 计算当前x自变量下的函数值，牛顿法和拟牛顿法都会用到 |
| get\_JM | 计算当前x自变量下的Jacobi矩阵，牛顿法和拟牛顿法都会用到，拟牛顿法仅用于初值计算 |
| get\_new\_X | 更新解向量的取值 |
| newton | 牛顿法的实现 |
| b\_newtons | 拟牛顿法的实现 |

### 3.程序实验结果及分析

（Q1）首先求解方程组1,在这里，设定精度要求，

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | | 牛顿法 | 拟牛顿法 |
| 初始值 | |  |  |
| 计算结果X | x1 | 0.905539609855913 | 0.905539609855937 |
| x2 | 1.085219168370031 | 1.085219168370044 |
| x3 | 0.672193668718306 | 0.672193668718633 |
| 迭代次数 | | 3 | 4 |
| CPU计算时间/s | | 0.025134 | 0.011626 |

在相同的初始值情况下，可以观察到牛顿法和拟牛顿法在达到相同的计算精度时得到的结果基本一致。在这两种方法中，牛顿法的迭代次数明显较少，但是由于每次迭代都需要求解矩阵的逆，导致牛顿法的每次迭代需要更长的CPU计算时间。

进一步，求解方程组2，同样设定精度要求为，

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | | 牛顿法 | 拟牛顿法 |
| 初始值 | |  |  |
| 计算结果X | x1 | 0.500000000000000 | 0.500000000000000 |
| x2 | 0.000000000000000 | -0.000000000000000 |
| x3 | -0.523598775598299 | -0.523598775598299 |
| 迭代次数 | | 5 | 5 |
| CPU计算时间/s | | 0.079201 | 0.059248 |

同样地，可以观察到，当初始值相同时，牛顿法和拟牛顿法在达到相同计算精度时得到基本相似的结果。不过，由于牛顿法每次迭代涉及求解矩阵的逆运算，导致其每次迭代的CPU计算时间较长。

(Q2)对方程组1，取其他的初值，设定精度要求，结果列表如下图所示，

|  |  |  |  |
| --- | --- | --- | --- |
| **初始值** | **方法** | **牛顿法** | **拟牛顿法** |
|  | 计算结果 | 0.905539609855914  1.08521916837003  0.672193668718305 | 0.905539610069747  1.08521916284571  0.672193662648011 |
| 迭代次数 | 5 | 8 |
| CPU计算时间/s | 0.072708 | 0.082578 |
|  | 计算结果 | 0.905539609855914  1.08521916837003  0.672193668718306 | 0.905539613167890  1.08521916965445  0.672193689554417 |
| 迭代次数 | 4 | 7 |
| CPU计算时间/s | 0.055251 | 0.079967 |
|  | 计算结果 | 0.905539609855914  1.08521916837003  0.672193668718305 | 0.905539609853265  1.08521916833190  0.672193668315815 |
| 迭代次数 | 4 | 6 |
| CPU计算时间/s | 0.010379 | 0.011837 |
|  | 计算结果 | 0.905539609855914  1.08521916837003  0.672193668718305 | 9.21185244848659  -5.57400515561907  18.1181739767053 |
| 迭代次数 | 11 | 82 |
| CPU计算时间/s | 0.155539 | 0.981788 |
|  | 计算结果 | 0.905539609855914  1.08521916837003  0.672193668718305 | 0.905539603252738  1.08521916755145  0.672193649277227 |
| 迭代次数 | 11 | 830 |
| CPU计算时间/s | 0.154876 | 9.810156 |
|  | 计算结果 | 0.905539609855914  1.08521916837003  0.672193668718305 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 |
| 迭代次数 | 14 | Max\_iter |
| CPU计算时间/s | 0.011373 |  |
|  | 计算结果 | 0.905539609855914  1.08521916837003  0.672193668718305 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 |
| 迭代次数 | 25 | Max\_iter |
| CPU计算时间/s | 0.351373 |  |
|  | 计算结果 | 9.21185244848637  -5.57400515558461  18.1181739768381 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 |
| 迭代次数 | 19 | Max\_iter |
| CPU计算时间/s | 0.267591 |  |

从上表可以看出，方程组（1）在附近存在另一个不动点。初值的选择会直接影响到牛顿法和拟牛顿法的收敛点。总的来说，初值离不动点越远，需要的迭代次数越多。因此，初值的选取非常重要，合适的初值可以更快地收敛。相反，如果初始值偏离精确解较远，会出现迭代次数增加直至无法收敛的情况。由于拟牛顿法是一种近似方法，明显需要更多的迭代次数，并且收敛情况不如牛顿法好，甚至会出现奇异矩阵的情况。值得注意的是牛顿法的求解方法和拟牛顿法计算的时间上总的时间几乎差不多。这是因为拟牛顿法需要的迭代步数比较多，在平均每步的迭代时间上，算法较为简单的拟牛顿法具有更短的平均迭代时长。

进一步，对方程组2，取其他的初值，设定精度要求，结果列表如下图所示，

|  |  |  |  |
| --- | --- | --- | --- |
| **初始值** | **方法** | **牛顿法** | **拟牛顿法** |
|  | 计算结果 | 0.498144684589491  -0.199605895543805  -0.528825977573388 | 0.498144675536865  -0.199604466356001  -0.528825910995774 |
| 迭代次数 | 7 | 10 |
| CPU计算时间/s | 0.109997 | 0.114888 |
|  | 计算结果 | 0.500000000000005  5.03025525044913e-13  -0.523598775598286 | 0.499999999691968  5.70777903801996e-08  -0.523598774756242 |
| 迭代次数 | 8 | 13 |
| CPU计算时间/s | 0.011329 | 0.016811 |
|  | 计算结果 | 0.500000000000916  1.00410004533121e-10  -0.523598775595673 | 0.500000000000013  3.83214283673815e-12  -0.523598775598340 |
| 迭代次数 | 6 | 9 |
| CPU计算时间/s | 0.101091 | 0.114424 |
|  | 计算结果 | 0.500000000000001  7.43330931426759e-14  -0.523598775598297 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 |
| 迭代次数 | 200 | Max\_iter |
| CPU计算时间/s | 4.563484 |  |
|  | 计算结果 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 |
| 迭代次数 | Max\_iter | Max\_iter |
| CPU计算时间/s |  |  |
|  | 计算结果 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 |
| 迭代次数 | Max\_iter | Max\_iter |
| CPU计算时间/s |  |  |
|  | 计算结果 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 |
| 迭代次数 | Max\_iter | Max\_iter |
| CPU计算时间/s |  |  |
|  | 计算结果 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 | Matlab警告矩阵接近奇异值，程序进入长期循环计算中 |
| 迭代次数 | Max\_iter | Max\_iter |
| CPU计算时间/s |  |  |

从这里可以发现，牛顿法可以在更广泛的区间内实现压缩映射原理，因此可以在更广泛的范围内选择初值，并最终收敛到精确解附近。相反，对于方程组2这样对于初值选取较为灵敏的特点，拟牛顿法的初值选取要求较高。此外，当初始值较接近不动点时，牛顿法和拟牛顿法得到的结果基本相同，尽管迭代次数略有差异，但所需的计算时间总体相近。这是因为拟牛顿法需要的迭代步数比较多，在平均每步的迭代时间上，算法较为简单的拟牛顿法具有更短的平均迭代时长。

(Q3)实验发现

在牛顿法的迭代过程中，需要进行矩阵的求逆运算。该方法的迭代收敛充分条件是在迭代过程中满足区间上的映内性。由于矩阵求逆的过程相对简单，因此适合用牛顿法在较大区间内进行计算。通常情况下，牛顿法适用于具有单调性或单值特性的函数，因为对初始值的敏感度较低，并且该算法具有良好的收敛性。拟牛顿法可以简化替代求逆这部分的运算，但是达到与牛顿法相同的精度，需要更多的迭代次数，对于初始值敏感度较高。

每次计算所需的CPU时间与计算机的当前运行状态相关，不同代码的运行时间也可能不一致，因此这些数据可能并不具有很高的参考价值。

### 4.实验总结

本次实验通过牛顿法和拟牛顿法求解两个非线性方程组，改变不同初值，对求解的结果进行比较分析。对于牛顿法和拟牛顿法，初始值越接近精确解，所需的迭代次数越小；

在应用上，牛顿法和拟牛顿法各有其优势。就迭代次数而言，由于更加精确，牛顿法所需的迭代次数更少；然而单次迭代的计算步骤更多、更复杂，因此每次迭代所需的时间更长。相比之下，拟牛顿法通过采用简化的近似公式，使得每次迭代更加迅速。当非线性方程组的求逆过程相对简单时（如方程组1的情况），拟牛顿法并没有明显的优势；但当求逆过程较为复杂时（如方程组2的情况），拟牛顿法则可以展现出优势，尽管循环次数增加，但CPU耗时却减少。

另外，在压缩映射区间方面，一般而言，具有单调性或单值特性的函数适合应用牛顿法，因其对初始值的敏感程度较低，确保了算法的良好收敛性。而拟牛顿法则无需在迭代过程中进行矩阵求逆操作，而是利用差商替代了矩阵的求导，因此当初始误差较大，会出现奇异矩阵的情况，从而对初始值的敏感程度较高。