HW 8

Math 501
Numerical analysis

Spring, 2007
California State University, Fullerton

Nasser M. Abbasi

August 13, 2021 Compiled on August 13, 2021 at 9:59pm

Contents

1 Analytic problems

1.1 section 4.6, problem 2

question: Prove that if A has this property (unit row diagonal dominant)

$$a_{ii} = 1 > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \qquad (1 \leq i \leq n)$$

then Richardson iteration is successful.

Solution:

Since the iterative formula is

$$\begin{aligned} x_{k+1} &= x_k + Q^{-1}(b - Ax_k) \\ &= x_k + Q^{-1}b - Q^{-1}Ax_k \\ &= (I - Q^{-1}A)x_k + Q^{-1}b \end{aligned}$$

This converges, by theorem (1) on page 210 when $\|I - Q^{-1}A\| < 1$

In Richardson method, $Q = I$, hence Richardson converges if $\|I - A\| < 1$

But $\|I - A\| = \| \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ 0 & \cdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 1 \end{bmatrix} \begin{bmatrix} 1 & a_{12} & \cdots & a_{1n} \\ a_{21} & 1 & \cdots & a_{2n} \\ \vdots & \cdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 1 \end{bmatrix} \| = \| \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \cdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix} \|$

But since row unit diagonal dominant, then the sum of each row elements remaining (after a_{ii} was annihilated) is a sum which is less than 1. Hence each row about will sum to some value which is less than 1. Hence the infinity norm of the above matrix, which is the maximum row sum, is less than 1. Hence

$$\|I - A\| < 1$$

Hence Richardson will converge. Each iteration will move closer to the solution x^*

1.2 problem 14

Problem: Prove that the eigenvalues of a Hermitian matrix are real.

Answer: A is Hermitian if $\overline{\left(A^T\right)} = A$, where the bar above indicates taking the complex conjugate. Hence the matrix is transposed and then each element will be complex conjugated.

Now, an eigenvalue of a matrix is defined such as

$$Ax = \lambda x$$

pre mutliply both sides by $\overline{\left(x^T\right)}$

$$\begin{aligned}\overline{\left(x^T\right)}Ax &= \overline{\left(x^T\right)}\lambda x \\ \overline{\left(x^T\right)}Ax &= \lambda\overline{\left(x^T\right)}x\end{aligned}$$

But since A is Herminitian, then $\overline{\left(A^T\right)} = A$, hence the above becomes

$$\begin{aligned}\overline{\left(x^T\right)\left(A^T\right)}x &= \lambda\overline{\left(x^T\right)}x \\ \overline{\left(x^TA^T\right)}x &= \lambda\overline{\left(x^T\right)}x \\ \overline{(Ax)^T}x &= \lambda\overline{\left(x^T\right)}x\end{aligned}$$

But $Ax = \lambda x$, hence the above becomes

$$\begin{aligned}\overline{(\lambda x)^T}x &= \lambda\overline{\left(x^T\right)}x \\ \overline{\left(x^T\lambda\right)}x &= \lambda\overline{\left(x^T\right)}x \\ \overline{\left(x^T\right)}\overline{\lambda}x &= \lambda\overline{\left(x^T\right)}x \\ \overline{\lambda}\overline{\left(x^T\right)}x &= \lambda\overline{\left(x^T\right)}x \\ \overline{\lambda} &= \lambda\end{aligned}$$

Hence since complex conjugate of eigenvalue is the same as the eigenvalue, therefor λ is real.

1.3 problem 16

Problem: Prove that if A is nonsingular, then $A\overline{A^T}$ is positive definite.

Answer: A is nonsingular, meaning its left and right inverses exist and are the same. To show that a matrix is positive definite, need to show that $x^T Ax > 0$ for all $x \neq 0$.

Let $\overline{A^T} = B$, then let $n = x^T A\overline{A^T}x$, we need to show that $n > 0$

$$n = x^T (AB) x$$

Since A^{-1} exist, then multiply both sides by A^{-1} and B^{-1}

$$\begin{aligned} A^{-1}B^{-1}n &= A^{-1}B^{-1}x^T (AB) x \\ &= x^T x \end{aligned}$$

But $x^T x = \|x\|^2 > 0$ unless $x = 0$, hence above becomes

$$\begin{aligned} A^{-1}B^{-1}n &> 0 \\ A^{-1}\left(\overline{A^T}\right)^{-1} n &> 0 \\ \left(\overline{A^T}A\right)^{-1} n &> 0 \end{aligned}$$

Multiply both sides by $\overline{A^T}A$ leads to $n > 0$, hence $A\overline{A^T}$ is positive definite.

1.4 problem 17

Problem: Prove that if A is positive definite, then its eigenvalues are positive.

Answer: A is positive definite implies $x^T Ax > 0$ for all $x \neq 0$. i.e. $\langle x, Ax \rangle > 0$.

But $Ax = \lambda x$, hence

$$\langle x, \lambda x \rangle > 0$$

or

$$\lambda \langle x, x \rangle > 0$$

But $\langle x, x \rangle = \|x\|^2 > 0$ unless $x = 0$, therefor

$$\lambda > 0$$

1.5 section 4.7, problem 1

question: Prove that if A is symmetric, then the gradient of the function $q(x) = \langle x, Ax\rangle - 2\langle x, b\rangle$ at x is $2(Ax - b)$. Recall that the gradient of a function $g : \mathbb{R}^n \to \mathbb{R}$ is the vector whose components are $\frac{\partial g}{\partial x_i}$ for $i = 1, 2, \cdots n$

answer:

$\langle a, b\rangle$ is $a^T b$, hence using this definition, we can expend the RHS above and see that it will give the result required.

$$\begin{aligned}
\langle x, Ax\rangle &= x^T(Ax)\\
&= [x_1\ x_2\ \cdots\ x_n]\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n}\\ a_{21} & a_{22} & \cdots & a_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ a_{n1} & a_{n2} & \cdots & a_{nn}\end{bmatrix}\begin{bmatrix} x_1\\ x_2\\ \vdots\\ x_n\end{bmatrix}\\
&= [x_1\ x_2\ \cdots\ x_n]\begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n\\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n\\ \vdots\\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n\end{bmatrix}\\
&= x_1(a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n) + x_2(a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n)\\
&+ x_3(a_{31}x_1 + a_{32}x_2 + \cdots + a_{3n}x_n) + \cdots + x_n(a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n)\\
&= \left(a_{11}x_1^2 + a_{12}x_1x_2 + \cdots + a_{1n}x_1x_n\right) + \left(a_{21}x_2x_1 + a_{22}x_2^2 + \cdots + a_{2n}x_2x_n\right)\\
&+ \left(a_{31}x_1x_3 + a_{32}x_2x_3 + a_{33}x_3^2 \cdots + a_{3n}x_nx_3\right) + \cdots + \left(a_{n1}x_nx_1 + a_{n2}x_nx_2 + \cdots + a_{nn}x_n^2\right)
\end{aligned}$$

And

$$\begin{aligned}
\langle x, b\rangle &= x^T b\\
&= [x_1\ x_2\ \cdots\ x_n]\begin{bmatrix} b_1\\ b_2\\ \vdots\\ b_n\end{bmatrix}\\
&= x_1b_1 + x_2b_2 + \cdots + x_nb_n
\end{aligned}$$

Hence Putting the above together, we obtain

$$\begin{aligned}
q(x) &= \langle x, Ax\rangle - 2\langle x, b\rangle\\
&= \left(a_{11}x_1^2 + a_{12}x_1x_2 + \cdots + a_{1n}x_1x_n\right) + \left(a_{21}x_2x_1 + a_{22}x_2^2 + \cdots + a_{2n}x_2x_n\right)\\
&+ \left(a_{31}x_1x_3 + a_{32}x_2x_3 + a_{33}x_3^2 + \cdots + a_{3n}x_nx_3\right) + \cdots + \left(a_{n1}x_nx_1 + a_{n2}x_nx_2 + \cdots + a_{nn}x_n^2\right)\\
&- 2(x_1b_1 + x_2b_2 + \cdots + x_nb_n)
\end{aligned}$$

Now taking the derivative of the above w.r.t $x_1, x_2, \cdots, x_n$ to generate the gradient vector, we obtain

$$\frac{\partial q(x)}{\partial x_1} = (2a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n) + (a_{21}x_2) + (a_{31}x_3) + \cdots + (a_{n1}x_n) - 2\,(b_1)$$
$$\frac{\partial q(x)}{\partial x_2} = (a_{12}x_1) + (a_{21}x_1 + 2a_{22}x_2 + \cdots + a_{2n}x_n) + (a_{32}x_3) + \cdots + (a_{n2}x_n) - 2\,(b_2)$$
$$\vdots$$
$$\frac{\partial q(x)}{\partial x_n} = (a_{1n}x_1) + (a_{2n}x_2) + (a_{3n}x_3) + \cdots + (a_{n1}x_1 + a_{n2}x_2 + \cdots + 2a_{nn}x_n) - 2\,(b_n)$$

Hence

$$\frac{\partial q(x)}{\partial x_1} = \begin{bmatrix} 2a_{11}, & a_{12}, & \cdots, a_{1n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + [0\,, a_{21}, a_{31}, & \cdots, a_{n1}] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} - 2\,(b_1)$$
$$\frac{\partial q(x)}{\partial x_2} = [a_{21}\,, 2a_{22}\,, \cdots, a_{2n}] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + [a_{12}\,, 0, & \cdots, a_{n2}] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} - 2\,(b_2)$$
$$\vdots$$
$$\frac{\partial q(x)}{\partial x_n} = [a_{n1}\,, a_{n2}, & \cdots, 2a_{nn}] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} a_{n1}, & a_{n2}, & \cdots, 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} - 2\,(b_n)$$

Combine, we get

$$\frac{\partial q(x)}{\partial x_1} = \begin{bmatrix} 2a_{11}, & 2a_{12}, & \cdots, 2a_{1n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} - 2\,(b_1)$$
$$\frac{\partial q(x)}{\partial x_2} = [2a_{21}\,, 2a_{22}\,, \cdots, 2a_{2n}] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} - 2\,(b_2)$$
$$\vdots$$
$$\frac{\partial q(x)}{\partial x_n} = [2a_{n1}\,, 2a_{n2}, & \cdots, 2a_{nn}] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} - 2\,(b_n)$$

Hence, in Vector/Matrix notation we obtain

$$\begin{aligned}\frac{\partial q(x)}{\partial \vec{x}} &= 2A\vec{x} - 2\vec{b} \\ &= 2\left(A\vec{x} - \vec{b}\right)\end{aligned}$$

Which is what we are asked to show.

(it would also have been possible to solve the above by expressing everything in the summation notations, i.e. writing $(Ax)_i = \sum_j^n A\left(i,j\right) x_j$, and apply differentiations directly on these summation expression as is without expanding them as I did above, it would have been probably shorter solution)

1.6 problem 2

Question: Prove that the minimum value of $q(x)$ is $-\left\langle b, A^{-1}b\right\rangle$

Solution:

$$q(x) = \langle x, Ax\rangle - 2\langle x, b\rangle$$

From problem (1) we found that $\frac{\partial q(x)}{\partial \vec{x}} = 2\left(A\vec{x} - \vec{b}\right)$, hence setting this to zero

$$\begin{aligned} A\vec{x} - \vec{b} &= 0 \\ \vec{x} &= A^{-1}b \end{aligned}$$

To check if this is a min,max, or saddle, we differentiate $\frac{\partial q(x)}{\partial \vec{x}}$ once more, and plug in this solution and check

$$\begin{aligned} \frac{\partial}{\partial \vec{x}}\left(\frac{\partial q(x)}{\partial \vec{x}}\right) &= 2\frac{\partial}{\partial \vec{x}}\left(A\vec{x} - \vec{b}\right) \\ &= 2A \end{aligned}$$

(A needs to be positive definite here, problem did not say this?) Hence this is a minimum.

Hence minimum value $q(x)$ is

$$q_{\min}(x) = \left\langle A^{-1}b, Ax\right\rangle - 2\left\langle A^{-1}b, b\right\rangle$$

But $Ax = b$, hence

$$\begin{aligned} q_{\min}(x) &= \left\langle A^{-1}b, b\right\rangle - 2\left\langle A^{-1}b, b\right\rangle \\ &= -\left\langle A^{-1}b, b\right\rangle \\ &= -\left\langle b, A^{-1}b\right\rangle \end{aligned}$$

1.7 problem 6

question: Prove that if $\hat{t} = \frac{\langle v, b-Ax \rangle}{\langle v, Av \rangle}$,and if $y = x + \hat{t}v$ then $\langle v, b - Ay \rangle = 0$

answer:

$$y = x + \frac{\langle v, b - Ax \rangle}{\langle v, Av \rangle} v$$

Then

$$\begin{aligned}
\langle v, b - Ay \rangle &= \left\langle v, b - A\left(x + \frac{\langle v, b - Ax \rangle}{\langle v, Av \rangle} v \right) \right\rangle \\
&= \left\langle v, b - Ax - A\frac{\langle v, b - Ax \rangle}{\langle v, Av \rangle} v \right\rangle \\
&= \left\langle v, b - Ax - Av\frac{\langle v, b - Ax \rangle}{\langle v, Av \rangle} \right\rangle \\
&= v^T \left(b - Ax - Av\frac{\langle v, b - Ax \rangle}{\langle v, Av \rangle} \right) \\
&= v^T b - v^T Ax - v^T Av\frac{\langle v, b - Ax \rangle}{\langle v, Av \rangle} \\
&= v^T b - v^T Ax - \langle v, Av \rangle \frac{\langle v, b - Ax \rangle}{\langle v, Av \rangle} \\
&= v^T b - v^T Ax - \langle v, b - Ax \rangle \\
&= v^T (b - Ax) - \langle v, b - Ax \rangle \\
&= \langle v, b - Ax \rangle - \langle v, b - Ax \rangle \\
&= 0
\end{aligned}$$

2 Computer problems

2.1 Jacobi, Gauss-Seidel, SOR

Jacobi, Gauss-Seidel, and SOR iterative solvers were implemented (in Matlab) and compared for rate of convergence. The implementation was tested on the following system

$$A = \begin{bmatrix} -4 & 2 & 1 & 0 & 0 \\ 1 & -4 & 1 & 1 & 0 \\ 2 & 1 & -4 & 1 & 2 \\ 0 & 1 & 1 & -4 & 1 \\ 0 & 0 & 1 & 2 & 04 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} -4 \\ 11 \\ -16 \\ 11 \\ -4 \end{bmatrix}$$

and the result was compared to Matlab $A\backslash b'$ result, which is

```
A\b'
ans =
   1.00000000000000
  -2.00000000000000
   4.00000000000000
  -2.00000000000000
   1.00000000000000
>>
```

These iterative solvers solve the linear system $Ax = b$ by iteratively approaching a solution. In all of these methods, the general iteration formula is

$$x_{k+1} = x_k + Q^{-1}(I - Ax_k)$$

Each method uses a different Q matrix. For Jacobi, $Q = diag(A)$, and for Gauss-Seidel, $Q = L(A)$, which is the lower triangular part of A. For SOR, $Q = \frac{1}{\omega}\left(diag(A) + \omega L^0(A)\right)$ where $L^0(A)$ is the strictly lower triangular part of A, and ω is an input parameter generally $0 < \omega < 2$.

For the Jacobi and Gauss-Seidel methods, we are guaranteed to converge to a solution if A is diagonally dominant. For SOR the condition of convergence is $\rho(G) < 1$, where $G = \left(I - Q^{-1}A\right)$, and $\rho(G)$ is the spectral radius of G.

2.2 Results

The following table shows the result of running Jacobian and Gauss-Seidel on the same Matrix. The columns are the relative error between each successive iteration. Defined as $\frac{|x_{k+1}-x_k|}{|x_{k+1}|}$. The first column is the result from running the Jacobian method, and the second is the result from the Gauss-Seidel method.

Iteration	Jacobi	Gauss-Seidel
1	0.94197873843414	0.93435567469174
2	0.22011159808466	0.13203098317070
3	0.04613055928361	0.03056375635161
4	0.02582530967034	0.02878251571226
5	0.02166103846735	0.02294426703511
6	0.01508738881859	0.01935275191913
7	0.01447950687767	0.01618762392092
8	0.01246529285252	0.01353611057259
9	0.01166876946105	0.01130586303057
10	0.01055706870353	0.00943545181644
11	0.00971262454554	0.00786920716886
12	0.00886458421973	0.00655940732143
13	0.00811604581680	0.00546521547469
14	0.00741643318343	0.00455191661070
15	0.00677994422374	0.00379012917894
16	0.00619437598238	0.00315507291046
17	0.00565882948442	0.00262590555197
18	0.00516810821828	0.00218513512307
19	0.00471915312779	0.00181810678260
20	0.00430837834153	0.00151255968550

We see from the above table that Gauss-Seidel method has faster convergence than Jacobi method. The following is a plot of the above table.

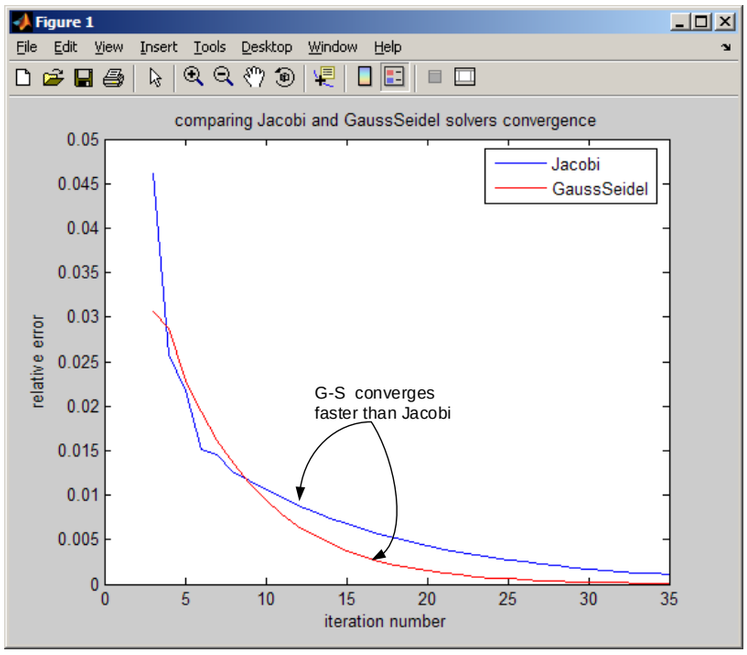

Figure 1: Plot

For the SOR method, it depends on the value of ω used. First we look at SOR on its own, comparing its performance as ω changes. Next, we pick 2 values for ω and compare SOR using these values with the Jacobian and the Gauss-Seidel method.

This table below shows the values of the relative error for difference ω. Only the first 20 iterations are shown. The following ω values are used: $.25, .5, .75, 1, 1.25, 1.5, 1.75$. This table also shows the number of iterations needed to achieve the same error tolerance specified by the user. Smaller number of iterations needed to reach this error limit indicates that ω selected was better than otherwise.

Table showing relative error as function of omega for SOR method

ns	1	2	3	4	5	6	7	8
	0.56636149	0.31679444	0.19062856	0.12117129	0.07975602	0.05368407	0.03668826	0.025375
	0.80386383	0.26815876	0.10423762	0.04325096	0.02004347	0.01252633	0.01064611	0.010023
	0.89416573	0.13002078	0.02796861	0.01947413	0.01787411	0.01629882	0.01477604	0.013360
	0.93435567	0.13203098	0.03056376	0.02878252	0.02294427	0.01935275	0.01618762	0.013536
	0.95492757	0.55277501	0.16654878	0.09785504	0.02868588	0.02521435	0.01102950	0.009760
	0.96647513	1.49382165	0.48287210	0.49652491	0.23641796	0.17683813	0.09584100	0.058404
	0.97327911	4.80887503	0.83556641	2.53932500	0.76442818	1.42972044	0.58706530	0.776483

Figure 2: Table

From the above table, and for the specific data used in this exercise, we observe that $\omega = 1.25$ achieved the best convergence

This is a plot of the above table. (using zoom to make the important part more visible).

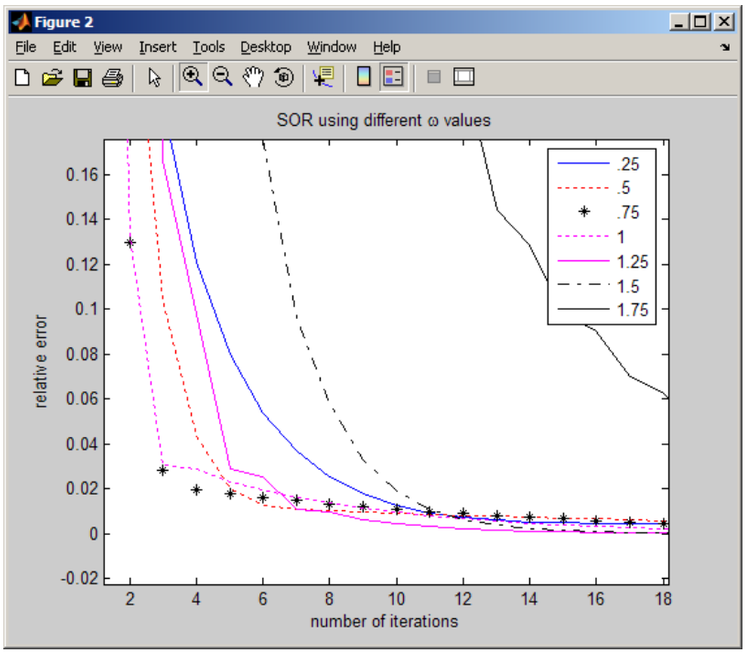

Figure 3: Zoom plot

Now we show the difference between Jacobi, Gauss-Seidel and SOR (using $\omega = 1.25$) as this ω gave the best convergence using SOR. The following is a plot showing that SOR with $\omega = 1.25$ achieved best convergence.

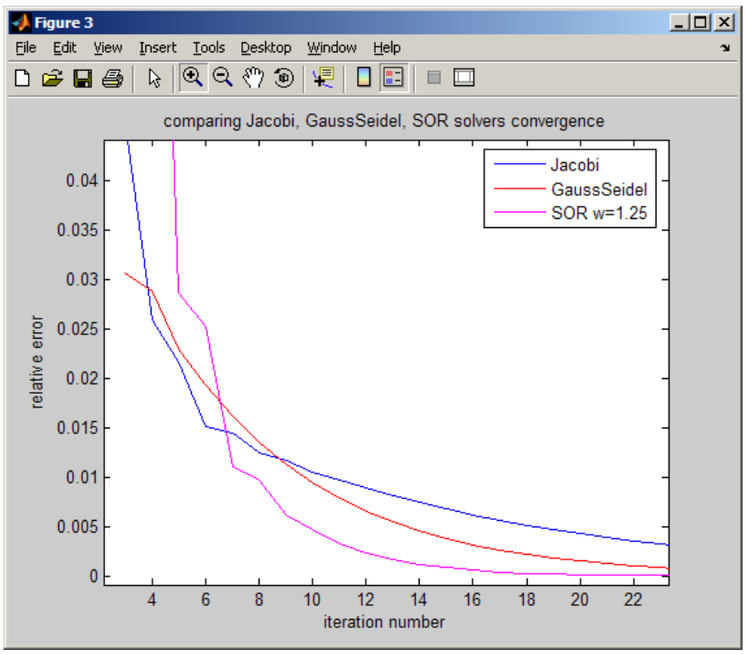

Figure 4: SOR plot

2.3 Example Matlab output

The following is an output from one run generated by a Matlab script written to test these implementations. 2 test cases used. A,b input as given in the HW assignment sheet given in the class, and a second A,b input shown in the textbook (SPD matrix, but not diagonally dominant) shown on page 245. This is the output.

```
***** TEST 1 *********
A =
    -4     2     1     0     0
     1    -4     1     1     0
     2     1    -4     1     2
     0     1     1    -4     1
     0     0     1     2    -4

b =
    -4    11   -16    11    -4
======>Matlab linear solver solution, using A\b
   1.00000000000000
  -2.00000000000000
   4.00000000000000
  -2.00000000000000
   1.00000000000000

Solution from Jacobi
   1.00037323222607
  -1.99962676777393
   4.00061424742539
  -1.99962676777393
   1.00037323222607

Solution from Gauss-Seidel
   1.00019802638995
  -1.99981450377552
   4.00028764196082
  -1.99983534139755
   1.00015423979143

Solution from SOR, w=0.250000
   1.00355389692825
  -1.99647784171454
   4.00574940211158
  -1.99653496988616
   1.00343408511030

Solution from SOR, w=0.500000
   1.00064715564763
  -1.99936629645376
   4.00102314817150
  -1.99939010276819
   1.00059721960251

Solution from SOR, w=0.750000
   1.00036324343971
  -1.99965036525822
   4.00055574249388
  -1.99967387439430
   1.00031390750518

Solution from SOR, w=1.000000
   1.00019802638995
  -1.99981450377552
   4.00028764196082
  -1.99983534139755
```

```
   1.00015423979143

Solution from SOR, w=1.250000
   1.00009213101077
  -1.99991820839097
   4.00012079232435
  -1.99993416539569
   1.00005844632680

Solution from SOR, w=1.500000
   0.99998926445485
  -1.99999117155337
   3.99998500044858
  -1.99999354920373
   0.99999484763949

Solution from SOR, w=1.750000
   1.00001519677595
  -2.00001369840580
   4.00002560215918
  -2.00001192126720
   1.00001112407411

***** TEST 2 *********
A =
    10     1     2     3     4
     1     9    -1     2    -3
     2    -1     7     3    -5
     3     2     3    12    -1
     4    -3    -5    -1    15

b =
    12   -27    14   -17    12

======>Matlab linear solver solution, using A\b
   1.00000000000000
  -2.00000000000000
   3.00000000000000
  -2.00000000000000
   1.00000000000000

Jacobi solution
   1.00018168868849
  -2.00016761404521
   2.99969203212914
  -1.99995200794879
   0.99978228017035

Gauss Seidel solution
   1.00009837024472
  -2.00008075631740
   2.99986024703377
  -1.99998691907114
   0.99991190441111

SOR solution, w=1.25
   1.00002736848705
  -2.00001960859025
   2.99996840603470
  -1.99999920522245
   0.99998285315310
```

Conclusion SOR with $\omega = 1.25$ achieved the best convergence. However, one needs to find determine which ω can do the best job, and this also will depend on the input. For different input different ω might give better result. Hence to use SOR one must first do a number of trials to determine the best value to use. Between the Jacobian and the Gauss-Seidel methods, the Gauss-Seidel method showed better convergence.

2.4 Long operation count

In these operations long count, since these algorithms are iterative, hence the operation count will be based on one iteration. One would then need to multiply this count by the number of iterations need to converge as per the specification that the user supplies. Gauss-Seidel will require less iterations to converge to the same solution as compared to Jacobi method. With the SOR method, it will depend on the ω selected.

Jacobi Long operations count for one iteration of the Jacobi method.

```
while keepLookingForSolution
   k=k+1;
   xnew=xold+Qinv*(b-A*xold);- - - - - (1)

   currentError=norm(xnew-xold);- - - - - (2)
   relError(k)=currentError/norm(xnew);- - - - - (3)

   if norm(b-A*xnew)<=resLimit || currentError<=errorLimit || k>maxIter
       keepLookingForSolution=FALSE;
   else
     xold=xnew;
   end

end
```

where

$$Qinv = eye(nRow)/diag(diag(A));$$

For line(1): For multiplying an $n \times n$ matrix by $n \times 1$ vector, n^2 ops are required. Hence for $A * xold$ we require n^2 ops. The result of $(b - A * xold)$ is an n long vector. Next we have $Qinv$ multiplied by this n long vector. This will require only n ops since $Qinv$ is all zeros except at the diagonal. Hence to calculate xnew we require $n + n^2$ ops.

For line(2,3): It involves 2 norm operations and one division. Hence we need to find the count for the norm operation. Assuming norm 2 is being used which is defined as $\sqrt{\sum_{i=1}^{n} x_i^2}$ hence this requires n multiplications and one long operation added for taking square root (ok to do?). Hence the total ops for finding relative error is $2n+2+1 = 2n+3$

The next operation is the check for the result limit:

$$norm(b - A * xnew) <= resLimit$$

This involves n^2 operations for the matrix by vector multiplication, and $n+1$ operations for the norm. Hence $n^2 + n + 1$

Adding all the above we obtain: $n + n^2 + 2n + 3 + n^2 + n + 1 = 2n^2 + 4n + 4$ hence this is $O\left(2n^2\right)$

The above is the cost per one iteration. Hence if M is close to n, then it is worst than G.E. But if the number of iteration is much less than N, then this method will be faster than non-iterative methods based on Gaussian elimination which required $O\left(n^3\right)$.

Gauss-Seidel Long operations count for one iteration of the Gauss-Seidel. For the statement

```
 while keepLookingForSolution
    k=k+1;

    xnew=xold;
    for i=1:nRow
        xnew(i)=xnew(i)+ (b(i)-A(i,:)*xnew)/A(i,i);- - - - -(1)
    end

    currentError=norm(xnew-xold); - - - - -(2)
    relError(k)=currentError/norm(xnew); - -- - - -(3)

    if norm(b-A*xnew)<=resLimit || currentError<=errorLimit || k>maxIter
        keepLookingForSolution=FALSE;
    else
        xold=xnew;
    end
end
```

For line (1): For each i, there is n for the operation A(i,:)*xnew, and one operation for the division. Hence the total cost is n^2 since there are n rows.

For line(2+3): The cost is the same as with the Jacobi method, which was found to be $2n+3$

The next operation is the check for the result limit:

$$norm(b - A * xnew) <= resLimit$$

This involves n^2 operations for the matrix by vector multiplication, and $n+1$ operations for the norm.

Hence the total operation is $n^2 + 2n + 3 + n + 1 = \boxed{n^2 + 3n + 4}$ hence this is $O\left(n^2\right)$ per one iteration.

SOR This is the same as Gauss-Seidel, except there is an extra multiplication by ω per one row per one iteration. Hence we need to add n to the count found in Gauss-Seidel.

Hence the count for SOR is $\boxed{n^2 + 4n + 4}$ or $O\left(n^2\right)$

2.5 Steepest descent iterative solver algorithm

The following is the output from testing the Steepest descent iterative algorithm.

```
======>Matlab linear solver solution, using A\b
   1.00000000000000
  -2.00000000000000
   3.00000000000000
  -2.00000000000000
   1.00000000000000

Steepest descent solution
   1.00013109477547
  -2.00012625234312
   2.99976034511258
  -1.99996859964602
   0.99987563548937

>>
```

2.6 Steepest Descent operation count

```
while keepLookingForSolution
    k=k+1;

    v=b-A*xold;    - - - - - - (1)
    t=dot(v,v)/dot(v,A*v); - - - - -(2)
    xnew=xold+t*v; - - - - -(3)

    currentError=norm(xnew-xold);   - - - - -(4)
    relError(k)=currentError/norm(xnew); - - - -(5)

    if norm(b-A*xnew)<=resLimit || currentError<=errorLimit || k>maxIter
        keepLookingForSolution=FALSE;
    else
        xold=xnew;
    end
end
```

For line (1): n^2

For line (2): For numerator: dot operation is n. For denominator: for A*v need n^2, and then need n added to the dot operation, hence need $n + n^2 + n$ operations. Add 1 to the division, hence need $n^2 + 2n + 1$ for line (2).

For line(3): n multiplications.

For line(4): $n + 1$ for the norm.

For line(5): $n + 2$ operations.

And finally for norm(b-A*xnew) in the final check, requires $n^2 + n + 1$

Hence total is $n^2 + n^2 + 2n + 1 + n + n + 1 + n + 2 + n^2 + n + 1 = 3n^2 + 6n + 5$

3 Source code

3.1 nma_driverTestIterativeSolvers.m

```
%
%This script is the driver to test and gather data for plotting
%for computer assignment 3/19/07 for Math 501, CSUF
%
%Nasser Abbasi 032607
%
%file name: nma_driverTestIterativeSolvers.m

close all;
clear all;

DISP_FOR_TABLE=0;  %turn to 1 to get output for table display

A=[-4 2 1 0 0;1 -4 1 1 0;2 1 -4 1 2;0 1 1 -4 1;0 0 1 2 -4];
b=[-4 11 -16 11 -4];
maxIter=200;
errorLimit=0.0001;
resLimit=0.00001;
oTable=zeros(maxIter,2);

%nma_getSpectraRadiusOfMatrix(
%find spectral radius for (I-Q^-1 A)

Q=diag(diag(A));
r=max(abs(eig(  eye(size(A,1)) - inv(Q)*A  ) ));
fprintf('Jacboi: spectral radius is %f\n',r);
if r>1
   fprintf('WARNING, spectral radius of (I-Q^-1 A)  should be less than 1 for convergence\n'
end
[x,k,relError]=nma_JacobiIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit);
figure;
plot(3:35,relError(3:35)); % plot(relError(1:k));
oTable(1:k,1)=relError(1:k);
fprintf('Solution from Jacobi\n');
format long;
disp(x)


Q=tril(A);
r=max(abs(eig(  eye(size(A,1)) - inv(Q)*A  ) ));

fprintf('Jacboi: spectral radius is %f\n',r);
if r>1
   fprintf('WARNING, spectral radius of (I-Q^-1 A)  should be less than 1 for convergence\n'
end

[x,k,relError]=nma_GaussSeidelIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit);
hold on;
plot(3:35,relError(3:35),'r'); % plot(relError(1:k));
legend('Jacobi','GaussSeidel');
title('comparing Jacobi and GaussSeidel solvers convergence');
xlabel('iteration number');
ylabel('relative error');

fprintf('Solution from Gauss-Seidel\n');
format long;
disp(x)
```

```
oTable(1:k,2)=relError(1:k);
fprintf('J\tG-S\n');
for i=1:35
    fprintf('%d\t%16.15f\t%16.15f\n',i,oTable(i,1),oTable(i,2));
end

%do it again for inclusion into Latex
if DISP_FOR_TABLE
    fprintf('Jacob\n');
    format long;
    for i=1:20
        disp(oTable(i,1))
    end

    %do it again for inclusion into Latex
    fprintf('G-S\n');
    for i=1:20
        disp(oTable(i,2))
    end
end

figure;
omegaValues=0.25*[1 2 3 4 5 6 7];
oTable=zeros(length(omegaValues),maxIter+1);
mycolor={'b','r:','k*','m:','m','k-.','k'};
for i=1:length(omegaValues)
    [x,k,relError]=nma_SORIterativeSolver(A,[1,1,1,1,1]',b',...
        maxIter,errorLimit,resLimit,omegaValues(i));
    plot(relError(1:k),mycolor{i});
    oTable(i,1)=omegaValues(i);
    oTable(i,2)=k;
    oTable(i,3:3+k-1)=relError(1:k);
    hold on;
    fprintf('Solution from SOR, w=%f\n',omegaValues(i));
    disp(x)

end
title('SOR using different \omega values');
xlabel('number of iterations');
ylabel('relative error');
legend('.25','.5','.75','1','1.25','1.5','1.75');

fprintf('omega\titeration\trelative errors\n');
for i=1:length(omegaValues)
    fprintf('%3.2f\t%d',oTable(i,1),oTable(i,2));
    if(oTable(i,2)>35)
        cutOff=35;
    else
        cutOff=oTable(i,2);
    end
    fprintf('\t\t');
    for j=1:cutOff
        fprintf('\t%16.15f',oTable(i,j+2));
        %fprintf('   %9.8f',oTable(i,j+2));
    end
    fprintf('\n');
end

%do it again for inclusion into Latex
if DISP_FOR_TABLE
    fprintf('SOR\n');
```

```
    for j=1:length(omegaValues)
        fprintf('SOR======>\n');
        for i=1:20
            disp(oTable(j,i+2))
        end
    end
end



%now compare the 3 methods using omega 1.25 only
[x,k,relError]=nma_JacobiIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit);
figure;
plot(3:35,relError(3:35)); % plot(relError(1:k));
[x,k,relError]=nma_GaussSeidelIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit);
hold on;
plot(3:35,relError(3:35),'r'); % plot(relError(1:k));

[x,k,relError]=nma_SORIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit,1.25);
plot(3:35,relError(3:35),'m'); % plot(relError(1:k));

legend('Jacobi','GaussSeidel','SOR w=1.25');
title('comparing Jacobi, GaussSeidel, SOR solvers convergence');
xlabel('iteration number');
ylabel('relative error');


%%%% Now run the test again. short version to paste into document.


A=[-4 2 1 0 0;1 -4 1 1 0;2 1 -4 1 2;0 1 1 -4 1;0 0 1 2 -4];
b=[-4 11 -16 11 -4];

fprintf('***** TEST 1 *********\n');
A
b
fprintf('======>Matlab linear solver solution, using A\\b  \n');
disp(A\b');

[x,k,relError]=nma_JacobiIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit);
fprintf('Jacobi solution\n');
disp(x);

[x,k,relError]=nma_GaussSeidelIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit);
fprintf('Gauss Seidel solution\n');
disp(x);

[x,k,relError]=nma_SORIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit,1.25);
fprintf('SOR solution, w=1.25\n');
disp(x);



%%%% Now run another test. Use an SPD matrix, which is shown on
%page 245 of textbook (Numerical Analysis, Kincaid.Cheney)
fprintf('***** TEST 2 *********\n');
A=[10 1 2 3 4;1 9 -1 2 -3;2 -1 7 3 -5;3 2 3 12 -1;4 -3 -5 -1 15];
```

```
b=[12 -27 14 -17 12];
A
b
fprintf('======>Matlab linear solver solution, using A\\b  \n');
disp(A\b');

[x,k,relError]=nma_JacobiIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit);
fprintf('Jacobi solution\n');
disp(x);

[x,k,relError]=nma_GaussSeidelIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit);
fprintf('Gauss Seidel solution\n');
disp(x);

[x,k,relError]=nma_SORIterativeSolver(A,[1,1,1,1,1]',b',...
    maxIter,errorLimit,resLimit,1.25);
fprintf('SOR solution, w=1.25\n');
disp(x);
```

3.2 nma_SORIterativeSolver.m

```
function [xnew,k,relError]=nma_SORIterativeSolver(A,x,b, ...
    maxIter,errorLimit,resLimit,omega)
%function [xnew,k,relError]=nma_GaussSeidelIterativeSolver(A,x,b,...
%                                    maxIter,errorLimit,resLimit,omega)
%
% Solve Ax=b using the SOR Iterative method
%
%INPUT:
% A: the A matrix
% x: Initial guess for solution
% b: right hand side
% maxIter:  max number of iterations allowed
% errorLimit: error tolerance. difference between successive x iteration
%             values. if such a difference is less than this error, stop.
% resLimit: if |b-A*x| is less than this limit, stop the iterative process.
% omega: SOR factor
%
%OUTPUT
% xnew: the solution found by iterative method.
% k: actual number of iterations used to obtain the above solution.
% relError: array that contains the relative error found at each iteration

%example call
% A=[-4 2 1 0 0;1 -4 1 1 0;2 1 -4 1 2;0 1 1 -4 1;0 0 1 2 -4];
% b=[-4 11 -16 11 -4]; maxIter=200; errorLimit=0.0001; resLimit=0.00001;
%[x,k,relError]=nma_JacobiIterativeSolver(A,[1,1,1,1,1]',b',...
%                                        maxIter,errorLimit,resLimit)

%by Nasser Abbasi  3/26/07
%

% do some error checking on input....

if nargin ~=7
    error 'wrong number of arguments. 7 inputs are required';
end

if ~isnumeric(omega)
    error 'omega must be numeric';
end
```

```
TRUE=1; FALSE=0;

[res,msg]=nma_IterativeSolversIsValidInput(A,x,b,...
    maxIter,errorLimit,resLimit);
if ~res
    error(msg);
end

[nRow,nCol]=size(A);
xold=x(:);
b=b(:);
k=0;
relError=zeros(maxIter,1);

keepLookingForSolution=TRUE;

while keepLookingForSolution
    k=k+1;

    xnew=xold;
    for i=1:nRow
        xnew(i)=xnew(i)+ omega*(b(i)-A(i,:)*xnew)/A(i,i);
    end

    currentError=norm(xnew-xold);
    relError(k)=currentError/norm(xnew);

    if norm(b-A*xnew)<=resLimit || currentError<=errorLimit || k>maxIter
        keepLookingForSolution=FALSE;
    else
        xold=xnew;
    end
end

end
```

3.3 nma_JacobiIterativeSolver.m

```
function [xnew,k,relError]=nma_JacobiIterativeSolver(A,x,b, ...
                                            maxIter,errorLimit,resLimit)
%function [xnew,k]=nma_JacobiIterativeSolver(A,x,b,...
%                                            maxIter,errorLimit,resLimit)
%
% Solve Ax=b using the Jacobi Iterative method
%
%INPUT:
% A: the A matrix
% x: Initial guess for solution
% b: right hand side
% maxIter:  max number of iterations allowed
% errorLimit: error tolerance. difference between successive x iteration
%             values. if such a difference is less than this error, stop.
% resLimit: if |b-A*x| is less than this limit, stop the iterative process.
%
%OUTPUT
% xnew: the solution found by iterative method.
% k: actual number of iterations used to obtain the above solution.
% relError: array that contains the relative error found at each iteration
%
%example call
% A=[-4 2 1 0 0;1 -4 1 1 0;2 1 -4 1 2;0 1 1 -4 1;0 0 1 2 -4];
% b=[-4 11 -16 11 -4]; maxIter=200; errorLimit=0.0001; resLimit=0.00001;
```

```
%[x,k,relError]=nma_JacobiIterativeSolver(A,[1,1,1,1,1]',b',...
%                                         maxIter,errorLimit,resLimit)

%by Nasser Abbasi  3/26/07
%

if nargin ~=6
    error 'wrong number of arguments. 6 inputs are required';
end

TRUE=1; FALSE=0;

[res,msg]=nma_IterativeSolversIsValidInput(A,x,b,maxIter,errorLimit,resLimit);
if ~res
    error(msg);
end

[nRow,nCol]=size(A);
xold=x(:);
b=b(:);
k=0;
relError=zeros(maxIter,1);
Qinv=eye(nRow)/diag(diag(A));

keepLookingForSolution=TRUE;

while keepLookingForSolution
   k=k+1;
   xnew=xold+Qinv*(b-A*xold);

   currentError=norm(xnew-xold);
   relError(k)=currentError/norm(xnew);

   if norm(b-A*xnew)<=resLimit || currentError<=errorLimit || k>maxIter
       keepLookingForSolution=FALSE;
   else
     xold=xnew;
   end

end

end
```

3.4 nma_GaussSeidelIterativeSolver.m

```
function [xnew,k,relError]=nma_GaussSeidelIterativeSolver(A,x,b, ...
    maxIter,errorLimit,resLimit)
%function [xnew,k]=nma_GaussSeidelIterativeSolver(A,x,b,...
%                                         maxIter,errorLimit,resLimit)
%
% Solve Ax=b using the Gauss-Seidel Iterative method
%
%INPUT:
% A: the A matrix
% x: Initial guess for solution
% b: right hand side
% maxIter:  max number of iterations allowed
% errorLimit: error tolerance. difference between successive x iteration
%             values. if such a difference is less than this error, stop.
% resLimit: if |b-A*x| is less than this limit, stop the iterative process.
%
%OUTPUT
% xnew: the solution found by iterative method.
```

```
% k: actual number of iterations used to obtain the above solution.
% relError: array that contains the relative error found at each iteration
%
%example call
% A=[-4 2 1 0 0;1 -4 1 1 0;2 1 -4 1 2;0 1 1 -4 1;0 0 1 2 -4];
% b=[-4 11 -16 11 -4]; maxIter=200; errorLimit=0.0001; resLimit=0.00001;
%[x,k,relError]=nma_GaussSeidelIterativeSolver(A,...
%                            [1,1,1,1,1]',b',maxIter,errorLimit,resLimit)
%

%by Nasser Abbasi  3/26/07

% do some error checking on input....

if nargin ~=6
    error 'wrong number of arguments. 6 inputs are required';
end

TRUE=1; FALSE=0;

[res,msg]=nma_IterativeSolversIsValidInput(A,x,b,...
    maxIter,errorLimit,resLimit);
if ~res
    error(msg);
end

[nRow,nCol]=size(A);
xold=x(:);
b=b(:);
k=0;
relError=zeros(maxIter,1);

keepLookingForSolution=TRUE;

while keepLookingForSolution
    k=k+1;

    xnew=xold;
    for i=1:nRow
        xnew(i)=xnew(i)+ (b(i)-A(i,:)*xnew)/A(i,i);
    end

    currentError=norm(xnew-xold);
    relError(k)=currentError/norm(xnew);

    if norm(b-A*xnew)<=resLimit || currentError<=errorLimit || k>maxIter
        keepLookingForSolution=FALSE;
    else
        xold=xnew;
    end
end

end
```

3.5 nma_IterativeSolversIsValidInput.m

```
function [res,msg]=nma_IterativeSolversIsValidInput(A,x,b,maxIter,errorLimit,resLimit)
%function[res,msg]=nma_IterativeSolversIsValidInput(A,x,b,maxIter,errorLimit,resLimit)
%
%helper function. Called by iterative liner solvers to validate input
%
%Nasser Abbasi 03/26/07

res=0;
msg='';
if ~isnumeric(A)|~isnumeric(b)|~isnumeric(x)|~isnumeric(maxIter) ...
        |~isnumeric(errorLimit)|~isnumeric(resLimit)
    msg='non numeric input detected';
    return;
end

[nRow,nCol]=size(A);
if nRow~=nCol
    msg='square A matrix expected';
    return;
end

[m,n]=size(b);
if n>1
    msg='b must be a vector';
    return;
end

if m~=nRow
    msg='length of b does not match A matrix size';
    return;
end

[m,n]=size(x);
if n>1
    msg='x must be a vector';
    return;
end

if m~=nRow
    msg='length of x does not match A matrix size';
    return;
end

res=1;
return;

end
```

3.6 nma_SteepestIterativeSolver.m

```
function [xnew,k,relError]=nma_SteepestIterativeSolver(A,x,b, ...
    maxIter,errorLimit,resLimit)
%function [xnew,k]=nma_SteepestIterativeSolver(A,x,b,...
%                                              maxIter,errorLimit,resLimit)
%
% Solve Ax=b using the Steepest descent Iterative method
%
%INPUT:
% A: the A matrix
% x: Initial guess for solution
% b: right hand side
```

```
% maxIter:  max number of iterations allowed
% errorLimit: error tolerance. difference between successive x iteration
%             values. if such a difference is less than this error, stop.
% resLimit: if |b-A*x| is less than this limit, stop the iterative process.
%
%OUTPUT
% xnew: the solution found by iterative method.
% k: actual number of iterations used to obtain the above solution.
% relError: array that contains the relative error found at each iteration
%
%example call
% A=[-4 2 1 0 0;1 -4 1 1 0;2 1 -4 1 2;0 1 1 -4 1;0 0 1 2 -4];
% b=[-4 11 -16 11 -4]; maxIter=200; errorLimit=0.0001; resLimit=0.00001;
%[x,k,relError]=nma_SteepestIterativeSolver(A,...
%                            [1,1,1,1,1]',b',maxIter,errorLimit,resLimit)
%

%by Nasser Abbasi  3/26/07

% do some error checking on input....

if nargin ~=6
    error 'wrong number of arguments. 6 inputs are required';
end

TRUE=1; FALSE=0;

[res,msg]=nma_IterativeSolversIsValidInput(A,x,b,...
    maxIter,errorLimit,resLimit);
if ~res
    error(msg);
end

[nRow,nCol]=size(A);
xold=x(:);
b=b(:);
k=0;
relError=zeros(maxIter,1);

keepLookingForSolution=TRUE;

while keepLookingForSolution
    k=k+1;

    v=b-A*xold;
    t=dot(v,v)/dot(v,A*v);
    xnew=xold+t*v;

    currentError=norm(xnew-xold);
    relError(k)=currentError/norm(xnew);

    if norm(b-A*xnew)<=resLimit || currentError<=errorLimit || k>maxIter
        keepLookingForSolution=FALSE;
    else
        xold=xnew;
    end
end

end
```

3.7 nma_driverTestSteepest.,

```
%
%This script is the driver to test steepest descent solver
%for computer assignment 3/19/07 for Math 501, CSUF
%
%Nasser Abbasi 033007
%
%file name: nma_driverTestSteepest.m

close all;
clear all;


maxIter=200;
errorLimit=0.0001;
resLimit=0.00001;

%%%% Now run another test. Use an SPD matrix, which is shown on
%page 245 of textbook (Numerical Analysis, Kincaid.Cheney)
fprintf('***** TEST steepest descent *********\n');
A=[10 1 2 3 4;1 9 -1 2 -3;2 -1 7 3 -5;3 2 3 12 -1;4 -3 -5 -1 15];
b=[12 -27 14 -17 12];
A
b
fprintf('======>Matlab linear solver solution, using A\\b  \n');
disp(A\b');

[x,k,relError]=nma_SteepestIterativeSolver(A,[1,1,1,1,1]',b', ...
    maxIter,errorLimit,resLimit);
fprintf('Steepest descent solution\n');
disp(x);
```