Enhancing Basic Iterative Solutions for Linear Equations

By

Namir C. Shammas

This article presents new algorithms that enhance classical methods for basic iterative solutions for linear equations. They are also classified as *stationary methods*. Such classical algorithms are simpler than their optimization-based counterparts that are based on conjugate gradient methods. The latter methods are classified as *non-stationary methods*. They use more elaborate optimization-based method and generally perform better. This article focuses mainly on the stationary methods, and how to enhance them. I also include a comparison with an implementation of the Conjugate Gradient method—the simplest non-stationary method. The article contains MATLAB code used to obtain the results.

Basics of Iterative Solutions of Linear Equations

There are two popular basic algorithms for basic iterative solutions for linear equations, and a popular enhancement for one of these algorithms. They are:

- The Jacobi method.
- The Gauss-Seidel method.
- The Successive Over Relaxation (SOR) enhancement for Gauss-Seidel.

Jacobi Method

The Jacobi method solves a set of n linear equations by systematically going through each equation, one at a time, to update the variable x_i using the current values for all the other variables. The algorithm then finalizes updating all the variables after it has done updating each variable. The method stores the updates in temporary variables until it reaches the end of one pass that updates all n variables. The Jacobi method uses the following equation:

$$x_i^{k+1} = \frac{1}{a_{i,i}} \left[b_i - \sum_{j=1, j \neq i}^{n} a_{i,j} x_j^k \right] \text{ for i=1, 2, 3, …, n} \qquad (1a)$$

Equation (1a) shows that the right-hand side uses only the old values of the variables (at iteration k) to calculate the values at iteration k+1. The matrix form of equation 1a is:

$$x^{k+1} = \boldsymbol{D}^{-1}\,(\boldsymbol{b} - \boldsymbol{R}\,\boldsymbol{x}^{k}) \qquad (1b)$$

Where **D** is the diagonal elements matrix and **R** = **A** – **D**.

The Jacobi method is not very efficient, since it delays using updated variables x_i. The good news with the Jacobi method is that it is very suitable for parallel computing. Using such approach speeds up the implementation of the Jacobi method.

The pseudo code for equation (1a) is:

```
Given an n by n matrix A and n-sized vector b, to solve Ax=b.
For k=1 to maxIters
  Vector xx = vector x
  For j=1 to n
    x(j)=b(j)
    For j=1 to n
      If i<>j Then
        x(j) = x(j) + a(j,i)*xx(i)
      End If
    End For
    x(j) = x(j)/a(j,j)
  End For
  r=norm(A*x-b)/n
  If r<=tolerance Then Exit For
End For
```

Gauss-Seidel Method

The Gauss-Seidel method is like the Jacobi method, with one exception. The algorithm uses x_i when calculating subsequent variables x_{i+j}. The algorithm uses the following equation:

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left[b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{k+1} - \sum_{j=i+1}^{n} a_{i,j}x_j^{k}\right] \text{ for i=1, 2, 3, …, n} \qquad (2a)$$

Notice that equation 2a has variable x^{k+1} appear on both sides of the equation. This confirms that the method uses the updated values as soon as they become available. The matrix form for equation 2a is:

$$x^{k+1} = (\boldsymbol{L} + \boldsymbol{D})^{-1}\,(\boldsymbol{b} - \boldsymbol{U}\,\boldsymbol{x}^{k}) \qquad (2b)$$

Where **D**, **U**, **L** are the diagonal elements matrix, strictly upper matrix, and strictly lower matrix, respectively.

The pseudo code for equation (2a) is:

```
Given an n by n matrix A and n-sized vector b, to solve Ax=b.
For k=1 to maxIters
  For j=1 to n
    x(j)=b(j)
    For j=1 to n
      If i<>j Then
        x(j) = x(j) + a(j,i)*x(i)
      End If
    End For
    x(j) = x(j)/a(j,j)
  End For
  r=norm(A*x-b)/n
  If r<=tolerance Then Exit For
End For
```

Successive Over Relation Method

The Successive Over Relaxation (SOR) method accelerates the Gauss-Seidel method by using a factor ω (with values between 0 and 2). The algorithm uses the following equation:

$$x_i^{k+1} = (1-\omega)x_j^k + \frac{\omega}{a_{i,i}}\left[b_i - \sum_{j<1}^{i-1} a_{i,j}x_j^{k+1} - \sum_{j>i}^{n} a_{i,j}x_j^k\right]$$

for i=1, 2, 3, …, n (3a)

The matrix form of equation 3a is:

$$\boldsymbol{x}^{k+1} = (\boldsymbol{D} + \omega\boldsymbol{L})^{-1}(\omega\boldsymbol{b} - [\omega\boldsymbol{U} + (\omega-1)\boldsymbol{D}]\boldsymbol{x}^k) \quad (3b)$$

The pseudo code for equation (3a) is:

```
Given an n by n matrix A and n-sized vector b, and over-relaxation factor w,
to solve Ax=b.
For k=1 to maxIters
  For j=1 to n
    s=b(j)
    For j=1 to n
      If i<>j Then
        s = s + a(j,i)*x(i)
      End If
    End For
    x(j) = (1-w)*x(j) + w*s/a(j,j)
  End For
  r=norm(A*x-b)/n
  If r<=tolerance Then Exit For
End For
```


Enhancing the Classical Algorithms

The basic approach for enhancing the Jacobi, Gauss-Seidel, and SOR methods is rather simple. Instead of updating one variable x_i at a time, we update m variables at a time. This multi-variable update requires the calculations of a submatrix A' and solution vector b' based on the matrix A, solution vector b, and subset vector of variables x. The algorithm then solves **A'x'=b'** for vector **x'** and replaces x_i with x_i' (either later or immediately). Increasing the size of subset of m linear equations tends to reduce the number of iterations needed to reach a specified tolerance for the average norm of residuals calculated as $r=\|\mathbf{Ax}-\mathbf{b}\|/n$. The relationship between the value of m and the number of iterations is not linear.

I will call the method for enhancing the class algorithms as *multi-variable subset updates* or MVSE for short.

Here is a simple example for applying the MVSE method to the Jacobi method using sets of two equations. This scheme yields the following set of sub-systems of linear equations:

$$a_{i,i}\, x_i + a_{i,i+1}\, x_{i+1} = b_i' \quad (4a)$$

$$a_{i+1,i}\, x_i + a_{i+1,i+1}\, x_{i+1} = b_{i+1}' \quad (4b)$$

Where b_i' and b_{i+1}' are:

$$b_i' = \left[b_i - \sum_{j=1,j\neq i,j\neq i+1}^{n} a_{i,j} x_j^k\right] \text{ for i=1, 3, 5, …, n-1} \quad (4c)$$

$$b_{i+1}' = \left[b_{i+1} - \sum_{j=1,j\neq i,j\neq i+1}^{n} a_{i+1,j} x_j^k\right] \text{ for i=1, 3, 5, …, n-1} \quad (4c)$$

The MSVE method calculates the coefficients b_i' and b_{i+1}' and then solves the sub-system of linear equations in 4a and 4b. The algorithm repeat this step for every set of two equations. The number of linear equations must be a multiple of 2. If they are not, then you apply the Jacobi method to the last equation.

In the case of using sets of three equations, we have:

$$a_{i,i}\, x_i + a_{i,i+1}\, x_{i+1} + a_{i,i+2}\, x_{i+2} = b_i' \quad (5a)$$

$$a_{i+1,i}\, x_i + a_{i+1,i+1}\, x_{i+1} + a_{i+1,i+2}\, x_{i+2} = b_{i+1}' \quad (5b)$$

$$a_{i+2,i}\, x_i + a_{i+2,i+1}\, x_{i+1} + a_{i+2,i+2}\, x_{i+2} = b_{i+2}' \quad (5c)$$

Where b_i', b_{i+1}', and b_{i+2}' are:

$$b'_i = \left[b_i - \sum_{j=1, j\neq i, j\neq i+1, j\neq i+2}^{n} a_{i,j} x_j^k\right] \text{ for i=1, 4, 7, …, n -2} \quad (5d)$$

$$b'_{i+1} = \left[b_{i+1} - \sum_{j=1, j\neq i, j\neq i+1, j\neq i+2}^{n} a_{i+1,j} x_j^k\right] \text{ for i=1, 4, 7, …, n-2} \quad (5e)$$

$$b'_{i+2} = \left[b_{i+2} - \sum_{j=1, j\neq i, j\neq i+1, j\neq i+2}^{n} a_{i+2,j} x_j^k\right] \text{ for i=1, 4, 7, …, n-2} \quad (5f)$$

Again, the MSVE method calculates the coefficients b'_i, b'_{i+1}, and b'_{i+2} and then solves the sub-system of linear equations in 5a, 5b, and 5c. The algorithm repeats this step for every set of three equations. The number of linear equations must be a multiple of 3. If they are not, then you apply the Jacobi method to the last two equations.

As you increase the number of equations in MVSE you increase the number of sub-linear equations shown in equations 4a and 4b, and in equations 5a, 5b, and 5c. The equations for applying MVSE to the Gauss-Seidel and SOR method are similar, albeit a bit more elaborate, since these methods use the updated values of x_i as soon as they become available. Like the basic Jacobi method, its MVSE-enhanced version is suitable for parallel computations.

I will apply the MVSE to the following categories of matrix coefficients:

1. All the matrix coefficients have random positive values that are uniformly distributed. The diagonal coefficients have positive and dominant values.
2. The non-diagonal matrix coefficients have normally distributed values with random positive and negative values. The diagonal coefficients have positive and dominant values.
3. All the matrix coefficients have random negative values that are uniformly distributed. The diagonal coefficients have negative and dominant values.
4. All the matrix coefficients have random negative values that are uniformly distributed. The diagonal coefficients have positive and dominant values.

The solution seeks vectors with value from 1 to n, the number of linear equations. The constant coefficients in vector **b** are calculated as **Ax**.

Enhanced Jacobi Method

The Jacobi method benefits greatly from the MVSE method. Listing 1 has MATLAB code for the basic Jacobi method. Listing 2 has MATLAB code for the MVSE using 2 variables. Listing 3 has MATLAB code for the MVSE using 5 variables. Listing 4 has MATLAB code for the MVSE using N variables. I did use

additional versions of the MATLAB code for using subsets of three variables. I am not including them because they are like the other versions.

The MATLAB listings in this article assign the number of equations to variable n. You can change the assigned value directly or by adjusting the assigned value to variable nvars when that variable is used in some listings. The code generates random matrix **A** with positive values and with diagonally-dominant matrix elements. The solution vectors **x** are sequentially set to be in the range of (1, n). The coefficient vector **b** is calculated as **Ax**. Thus, the vector **b** and matrix **A** are both random.

```
% Jacobi method
n=100;
A0=1+n*rand(n,n);
A=A0;
for i=1:n
    A(i,i)=sum(A(i,:));
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(2,1);
D=diag(diag(A));
R=A-D;
Dinv=inv(D);
for k=1:100*n
    x=Dinv*(b-R*x);
    r=norm(A*x-b)/n;
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iterations=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 1. The MATLAB code for the basic Jacobi method.

```
% MVSE with 2 variables for Jacobi method
n=100;
A0=1+n*rand(n,n);
A=A0;
for i=1:n
    A(i,i)=sum(A(i,:));
```



```
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(2,1);
lastr=norm(A*x-b)/n;
for k=1:100*n
    x2=x;
    for i=1:2:n-1
        xx(1)=x(i);
        xx(2)=x(i+1);
        dotx=dot(A(i,:),x);
        AA=[A(i,i) A(i,i+1);A(i+1,i) A(i+1,i+1)];
        sumx=dotx-A(i,i)*xx(1)-A(i,i+1)*xx(2);
        bb(1)=b(i)-sumx;
        dotx=dot(A(i+1,:),x);
        sumx=dotx-A(i+1,i)*xx(1)-A(i+1,i+1)*xx(2);
        bb(2)=b(i+1)-sumx;
        xx=AA\bb;
        x2(i)=xx(1);
        x2(i+1)=xx(2);
    end
    x=x2;
    r=norm(A*x-b)/n;
    if abs(lastr-r)/lastr>=0.1
      lastr=r;
      fprintf('k= %i, r = %e\n', k,r)
    end
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iterations=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 2. MATLAB code for the MVSE for Jacobi using 2 variables.

Listing 2 shows a typical and simple implementation of the MVSE method. Notice that the *for i* loop calculates values for a 2 by 2 matrix AA, and 2-element vectors xx and bb. These variables make up the data for solving sets of two linear equations using the statement xx=AA\bb. The solutions are temporarily copied into vector x2. When the *for i* loop terminates, the code copies the values of vector x2 back into vector x. This assignment implements the differed-assignment scheme which is a feature of the Jacobi method.


```
% MVSE with 5 variables for Jacobi method
n=100; % must be a multiple of 5
A=1+n*rand(n,n);
for i=1:n
    A(i,i)=sum(A(i,:));
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(1,5);
AA=zeros(5,5);

for k=1:100*n
    x2=x;
    for i=1:5:n
        for j=1:5
            xx(j)=x(i+j-1);
        end
        AA=[A(i,i) A(i,i+1) A(i,i+2) A(i,i+3) A(i,i+4); ...
            A(i+1,i) A(i+1,i+1) A(i+1,i+2) A(i+1,i+3) A(i+1,i+4); ...
            A(i+2,i) A(i+2,i+1) A(i+2,i+2) A(i+2,i+3) A(i+2,i+4); ...
            A(i+3,i) A(i+3,i+1) A(i+3,i+2) A(i+3,i+3) A(i+3,i+4); ...
            A(i+4,i) A(i+4,i+1) A(i+4,i+2) A(i+4,i+3) A(i+4,i+4)];

        dotx=dot(A(i,:),x);
        sumx=dotx-A(i,i)*xx(1)-A(i,i+1)*xx(2)-A(i,i+2)*xx(3) - ...
                  A(i,i+3)*xx(4) -A(i,i+4)*xx(5);
        bb(1)=b(i)-sumx;
        dotx=dot(A(i+1,:),x);
        sumx=dotx-A(i+1,i)*xx(1)-A(i+1,i+1)*xx(2)-A(i+1,i+2)*xx(3) - ...
                  A(i+1,i+3)*xx(4) -A(i+1,i+4)*xx(5);
        bb(2)=b(i+1)-sumx;
        dotx=dot(A(i+2,:),x);
        sumx=dotx-A(i+2,i)*xx(1)-A(i+2,i+1)*xx(2)-A(i+2,i+2)*xx(3) - ...
                  A(i+2,i+3)*xx(4) -A(i+2,i+4)*xx(5);
        bb(3)=b(i+2)-sumx;
        dotx=dot(A(i+3,:),x);
        sumx=dotx-A(i+3,i)*xx(1)-A(i+3,i+1)*xx(2)-A(i+3,i+2)*xx(3) - ...
                  A(i+3,i+3)*xx(4) -A(i+3,i+4)*xx(5);
        bb(4)=b(i+3)-sumx;
        dotx=dot(A(i+4,:),x);
        sumx=dotx-A(i+4,i)*xx(1)-A(i+4,i+1)*xx(2)-A(i+4,i+2)*xx(3) - ...
                  A(i+4,i+3)*xx(4) -A(i+4,i+4)*xx(5);
        bb(5)=b(i+4)-sumx;
        xx=AA\bb';
        for j=1:5
            x2(i+j-1)=xx(j);
        end
    end
    x=x2;
    r=norm(A*x-b)/n;
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
```



```
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iteratins=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 3. MATLAB code for the MVSE for Jacobi using 5 variables.

Listing 3 resembles Listing 2, except it handles solving batches of 5 equations instead of 2. Notice that the *for i* loop calculates values for a 5 by 5 matrix AA, and 5-element vectors xx and bb. These variables make up the data for solving sets of 5 linear equations using the statement xx=AA\bb. The solutions are temporarily copied into vector x2. When the for i loop terminates the code copies the values of vector x2 back into vector x. This assignment implements the late-assignment which is a feature of the Jacobi method.

```
% MVSE with N variables for Jacobi method
nvars=10;
m=nvars-1;
n=nvars*100; % must be a multiple of nvars
A=1+n*rand(n,n);
for i=1:n
    A(i,i)=2*sum(A(i,:));
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(1,nvars);
AA=zeros(nvars,nvars);
Arow=zeros(1,nvars);
lastr=norm(A*x-b)/n;
for k=1:100*n
    x2=x;
    for i=1:nvars:n
        for colOffset=0:m
          xx(colOffset+1)=x(i+colOffset);
        end
        AA=[];
        for rowOffset=0:m
          Arow=zeros(nvars,1);
          for colOffset=0:m
            Arow(colOffset+1)=A(i+rowOffset,i+colOffset);
          end
          AA=[AA;Arow'];
        end

        for rowOffset=0:m
          sumx=dot(A(i+rowOffset,:),x);
```



```
          for colOffset=0:m
            sumx = sumx -
            A(i+rowOffset,i+colOffset)*xx(colOffset+1);
          end
          bb(rowOffset+1)=b(i+rowOffset)-sumx;
        end

        xx=AA\bb';
        for colOffset=0:m
          x2(i+colOffset)=xx(colOffset+1);
        end
    end % for i
    x=x2;
    r=norm(A*x-b)/n;
    if abs(lastr-r)/lastr>0.1
      lastr=r;
      fprintf('k= %i, r = %e\n', k,r)
    end
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iteratins=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 4. MATLAB code for the MVSE for Jacobi using N variables.

Enhanced Gauss-Seidel Method

The Gauss-Seidel method also benefits from the MVSE method. Listing 5 has MATLAB code for the basic Gauss-Seidel method. Listing 6 has MATLAB code for the MVSE using 2 variables. Listing 7 has MATLAB code for the MVSE using 5 variables. Listing 8 has MATLAB code for the MVSE using N variables. These listings resemble those for the Jacobi method, except that the values for vector **x** are updated right after solving each subset number of linear equations.

```
% Gauss-Seidel method
n=1000;
toler=0.00001;
rtoler=0.00001;
A=1+n*rand(n,n);
for i=1:n
  A(i,i)=sum(A(i,:));
end
```



```
x0=1:n;
b=A*x0';

D=diag(A);
L=tril(A);
U=A-L;
Linv=inv(L);
T=-Linv*U;
C=Linv*b;
maxIter=100*n;
x=zeros(n,1);
lastr=norm(A*x-b)/n;
r2=lastr;
for iter=1:maxIter
  x=T*x + C;
  r=norm(A*x-b)/n;
  if abs(r-lastr)/lastr>=0.1
    lastr=r;
    fprintf('k = %i, r = %e\n', iter, r);
  end
  if abs(r2-r)<=rtoler, break; end
  r2=r;
  if r<=toler, break; end
  if isnan(r), break; end
end
if n<50
  disp(x)
else
  for i=1:20
    fprintf('%i : %f\n', i, x(i));
  end
    for i=n-19:n
    fprintf('%i : %f\n', i, x(i));
    end
end

fprintf('Iters = %i\n', iter);
fprintf('r = %e\n', r);
```

Listing 5. MATLAB code for the Gauss-Seidel method.

```
% MVSE method for Gauss-Seidel with 2 variables
n=1000;
A0=1+n*rand(n,n);
A=A0;
for i=1:n
    A(i,i)=sum(A(i,:));
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(2,1);

for k=1:100*n
    for i=1:2:n-1
```



```
        xx(1)=x(i);
        xx(2)=x(i+1);
        dotx=dot(A(i,:),x);
        AA=[A(i,i) A(i,i+1);A(i+1,i) A(i+1,i+1)];
        sumx=dotx-A(i,i)*xx(1)-A(i,i+1)*xx(2);
        bb(1)=b(i)-sumx;
        dotx=dot(A(i+1,:),x);
        sumx=dotx-A(i+1,i)*xx(1)-A(i+1,i+1)*xx(2);
        bb(2)=b(i+1)-sumx;
        xx=AA\bb;
        x(i)=xx(1);
        x(i+1)=xx(2);
    end
    r=norm(A*x-b)/n;
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iterations=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 6. MATLAB code for the MVSE for Gauss-Seidel using 2 variables.

```
% MVSE method for Gauss-Seidel with 5 variables
n=1000; % must be a multiple of 5
A=1+n*rand(n,n);
for i=1:n
    A(i,i)=sum(A(i,:));
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(1,5);
AA=zeros(5,5);

for k=1:100*n
    for i=1:5:n
        for j=1:5
            xx(j)=x(i+j-1);
        end
        AA=[A(i,i) A(i,i+1) A(i,i+2) A(i,i+3) A(i,i+4); ...
            A(i+1,i) A(i+1,i+1) A(i+1,i+2) A(i+1,i+3) A(i+1,i+4); ...
            A(i+2,i) A(i+2,i+1) A(i+2,i+2) A(i+2,i+3) A(i+2,i+4); ...
            A(i+3,i) A(i+3,i+1) A(i+3,i+2) A(i+3,i+3) A(i+3,i+4); ...
            A(i+4,i) A(i+4,i+1) A(i+4,i+2) A(i+4,i+3) A(i+4,i+4)];
        dotx=dot(A(i,:),x);
        sumx=dotx-A(i,i)*xx(1)-A(i,i+1)*xx(2)-A(i,i+2)*xx(3) - ...
```



```
                A(i,i+3)*xx(4) -A(i,i+4)*xx(5);
        bb(1)=b(i)-sumx;
        dotx=dot(A(i+1,:),x);
        sumx=dotx-A(i+1,i)*xx(1)-A(i+1,i+1)*xx(2)-A(i+1,i+2)*xx(3) - ...
                A(i+1,i+3)*xx(4) -A(i+1,i+4)*xx(5);
        bb(2)=b(i+1)-sumx;
        dotx=dot(A(i+2,:),x);
        sumx=dotx-A(i+2,i)*xx(1)-A(i+2,i+1)*xx(2)-A(i+2,i+2)*xx(3) - ...
                A(i+2,i+3)*xx(4) -A(i+2,i+4)*xx(5);
        bb(3)=b(i+2)-sumx;
        dotx=dot(A(i+3,:),x);
        sumx=dotx-A(i+3,i)*xx(1)-A(i+3,i+1)*xx(2)-A(i+3,i+2)*xx(3) - ...
                A(i+3,i+3)*xx(4) -A(i+3,i+4)*xx(5);
        bb(4)=b(i+3)-sumx;
        dotx=dot(A(i+4,:),x);
        sumx=dotx-A(i+4,i)*xx(1)-A(i+4,i+1)*xx(2)-A(i+4,i+2)*xx(3) - ...
                A(i+4,i+3)*xx(4) -A(i+4,i+4)*xx(5);
        bb(5)=b(i+4)-sumx;
        xx=AA\bb';
        for j=1:5
            x(i+j-1)=xx(j);
        end
    end
    r=norm(A*x-b)/n;
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iteratins=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 7. MATLAB code for the MVSE for Gauss-Seidel using 5 variables.

```
% MVSE method for Gauss-Seidel with N variables
nvars=100;
m=nvars-1;
n=nvars*100; % must be a multiple of nvars
A=1+n*rand(n,n);
for i=1:n
    A(i,i)=2*sum(A(i,:));
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(1,nvars);
AA=zeros(nvars,nvars);
Arow=zeros(1,nvars);
```



```
for k=1:100*n
    for i=1:nvars:n
        for colOffset=0:m
          xx(colOffset+1)=x(i+colOffset);
        end
        AA=[];
        for rowOffset=0:m
          Arow=zeros(nvars,1);
          for colOffset=0:m
            Arow(colOffset+1)=A(i+rowOffset,i+colOffset);
          end
          %Arow(1,1:nvars)=A(i+rowOffset,i+rowOffset:i+rowOffset+m);
          AA=[AA;Arow'];
        end

        for rowOffset=0:m
          sumx=dot(A(i+rowOffset,:),x);
          for colOffset=0:m
            sumx=sumx-A(i+rowOffset,i+colOffset)*xx(colOffset+1);
          end
          %sumx=dot(A(i+rowOffset,:),x)-
dot(A(i+rowOffset,i+rowOffset:i+rowOffset+m),xx);
          bb(rowOffset+1)=b(i+rowOffset)-sumx;
        end

        xx=AA\bb';
        for colOffset=0:m
          x(i+colOffset)=xx(colOffset+1);
        end
    end % for i
    %disp(x);
    r=norm(A*x-b)/n;
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iterations=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 8. MATLAB code for the MVSE for Gauss-Seidel using N variables.

Enhanced SOR Method

The SOR method also benefits from the MVSE method, especially when using larger subsets of linear equations. Listing 9 has MATLAB code for the basic SOR

method. Listing 10 has MATLAB code for the MVSE using 2 variables. Listing 11 has MATLAB code for the MVSE using 5 variables. Listing 12 has MATLAB code for the MVSE using N variables. These listings use variable w to represent the over-relaxation factor ω. I set the value for w to be 0.95 which seems to be optimum for the kind of random matrices A.

```
% SOR method
n=1000;
toler=0.00001;
rtoler=0.00001;
w=0.95;
A=1+n*rand(n,n);
for i=1:n
  A(i,i)=sum(A(i,:));
end
x0=1:n;
b=A*x0';

v=diag(A);
D=diag(v);
L=A;
U=A;
for i=1:n
  for j=1:n
    if i>=j, L(i,j)=0; end
    if i<=j, U(i,j)=0; end
  end
end

M1 = inv(D+w*L);
V1 = w*b;
M2 = w*U + (w-1)*D;
maxIter=10*n;
x=zeros(n,1);
Linv=inv(L+D);
lastr=norm(A*x-b)/n;
r2=lastr;
for iter=1:maxIter
  x=M1 * (V1 - M2*x);
  %x=(1-w)*x + w*Linv*(b-U*x);
  r=norm(A*x-b)/n;
  if abs(r-lastr)/lastr>=0.1
    lastr=r;
    fprintf('k = %i, r = %e\n', iter, r);
  end
  if abs(r2-r)<=rtoler, break; end
  r2=r;
  if r<=toler, break; end
  if isnan(r), break; end
end
if n<50
  disp(x)
else
```



```
  for i=1:20
    fprintf('%i : %f\n', i, x(i));
  end
    for i=n-19:n
    fprintf('%i : %f\n', i, x(i));
    end
end

fprintf('Iters = %i\n', iter);
fprintf('r = %e\n', r);
```

Listing 9. MATLAB code for the SOR method.

```
% MSVE method for SOR using 2 variables
n=1000;
w=0.95;
A0=1+n*rand(n,n);
A=A0;
for i=1:n
    A(i,i)=sum(A(i,:));
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(2,1);
lastr=norm(A*x-b)/n;
for k=1:100*n
    for i=1:2:n-1
        xx(1)=x(i);
        xx(2)=x(i+1);
        dotx=dot(A(i,:),x);
        AA=[A(i,i) A(i,i+1);A(i+1,i) A(i+1,i+1)];
        sumx=dotx-A(i,i)*xx(1)-A(i,i+1)*xx(2);
        bb(1)=b(i)-sumx;
        dotx=dot(A(i+1,:),x);
        sumx=dotx-A(i+1,i)*xx(1)-A(i+1,i+1)*xx(2);
        bb(2)=b(i+1)-sumx;
        xx=AA\bb;
        x(i)=(1-w)*x(i)+w*xx(1);
        x(i+1)=(1-w)*x(i+1)+w*xx(2);
    end
    r=norm(A*x-b)/n;
    if abs(r-lastr)/lastr>=0.1
      lastr=r;
      fprintf('k = %i, r = %e\n', k, r);
    end
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
```



```
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iterations=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 10. MATLAB code for the MVSE for SOR using 2 variables.

```
% MSVE method for SOR using 5 variables
n=1000; % must be a multiple of 5
w=0.95;
A=1+n*rand(n,n);
for i=1:n
    A(i,i)=sum(A(i,:));
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(1,5);
AA=zeros(5,5);

for k=1:100*n
    for i=1:5:n
        for j=1:5
            xx(j)=x(i+j-1);
        end
        AA=[A(i,i) A(i,i+1) A(i,i+2) A(i,i+3) A(i,i+4); ...
            A(i+1,i) A(i+1,i+1) A(i+1,i+2) A(i+1,i+3) A(i+1,i+4); ...
            A(i+2,i) A(i+2,i+1) A(i+2,i+2) A(i+2,i+3) A(i+2,i+4); ...
            A(i+3,i) A(i+3,i+1) A(i+3,i+2) A(i+3,i+3) A(i+3,i+4); ...
            A(i+4,i) A(i+4,i+1) A(i+4,i+2) A(i+4,i+3) A(i+4,i+4)];

        dotx=dot(A(i,:),x);
        sumx=dotx-A(i,i)*xx(1)-A(i,i+1)*xx(2)-A(i,i+2)*xx(3) - ...
                  A(i,i+3)*xx(4) -A(i,i+4)*xx(5);
        bb(1)=b(i)-sumx;
        dotx=dot(A(i+1,:),x);
        sumx=dotx-A(i+1,i)*xx(1)-A(i+1,i+1)*xx(2)-A(i+1,i+2)*xx(3) - ...
                  A(i+1,i+3)*xx(4) -A(i+1,i+4)*xx(5);
        bb(2)=b(i+1)-sumx;
        dotx=dot(A(i+2,:),x);
        sumx=dotx-A(i+2,i)*xx(1)-A(i+2,i+1)*xx(2)-A(i+2,i+2)*xx(3) - ...
                  A(i+2,i+3)*xx(4) -A(i+2,i+4)*xx(5);
        bb(3)=b(i+2)-sumx;
        dotx=dot(A(i+3,:),x);
        sumx=dotx-A(i+3,i)*xx(1)-A(i+3,i+1)*xx(2)-A(i+3,i+2)*xx(3) - ...
                  A(i+3,i+3)*xx(4) -A(i+3,i+4)*xx(5);
        bb(4)=b(i+3)-sumx;
        dotx=dot(A(i+4,:),x);
        sumx=dotx-A(i+4,i)*xx(1)-A(i+4,i+1)*xx(2)-A(i+4,i+2)*xx(3) - ...
                  A(i+4,i+3)*xx(4) -A(i+4,i+4)*xx(5);
        bb(5)=b(i+4)-sumx;
        xx=AA\bb';
        x(i)=(1-w)*x(i)+w*xx(1);
        x(i+1)=(1-w)*x(i+1)+w*xx(2);
```



```
        x(i+2)=(1-w)*x(i+2)+w*xx(3);
        x(i+3)=(1-w)*x(i+3)+w*xx(4);
        x(i+4)=(1-w)*x(i+4)+w*xx(5);
    end
    r=norm(A*x-b)/n;
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iterations=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 11. MATLAB code for the MVSE for SOR using 5 variables.

```
% MSVE method for SOR using N variables
nvars=10;
w=0.95;
m=nvars-1;
n=nvars*100; % must be a multiple of nvars
A=1+n*rand(n,n);
for i=1:n
    A(i,i)=sum(A(i,:));
end
x0=1:n;
b=A*x0';
x=zeros(n,1);
bb=zeros(1,nvars);
AA=zeros(nvars,nvars);
Arow=zeros(1,nvars);
for k=1:100*n
    for i=1:nvars:n
        for colOffset=0:m
          xx(colOffset+1)=x(i+colOffset);
        end
        AA=[];
        for rowOffset=0:m
          Arow=zeros(nvars,1);
          for colOffset=0:m
            Arow(colOffset+1)=A(i+rowOffset,i+colOffset);
          end
          %Arow(1,1:nvars)=A(i+rowOffset,i+rowOffset:i+rowOffset+m);
          AA=[AA;Arow'];
        end

        for rowOffset=0:m
          sumx=dot(A(i+rowOffset,:),x);
          for colOffset=0:m
```



```
            sumx=sumx-A(i+rowOffset,i+colOffset)*xx(colOffset+1);
          end
          %sumx=dot(A(i+rowOffset,:),x)-
dot(A(i+rowOffset,i+rowOffset:i+rowOffset+m),xx);
          bb(rowOffset+1)=b(i+rowOffset)-sumx;
        end

        xx=AA\bb';
        for colOffset=0:m
          x(i+colOffset)=(1-w)*x(i+colOffset)+w*xx(colOffset+1);
        end
    end % for i
    r=norm(A*x-b)/n;
    if isnan(r), break, end
    if r<=0.00001, break; end
end
if n<=100
  disp(x')
else
    for i=1:20
        fprintf('%i : %f\n', i, x(i));
    end
    for i=n-19:n
        fprintf('%i : %f\n', i, x(i));
    end
end
fprintf('Iterations=%i\n', k)
fprintf('Last r = %e\n', r);
```

Listing 12. MATLAB code for the MVSE for SOR using N variables.

Comparison with basic Conjugate Gradient Method

I would like to compare the above versions of the MVSE method with the basic Conjugate Gradient method. The pseudo-code for this method is:

```
r = b-Ax
p = r
k = 0;
Do
  k = k+1
  alpha = (r^T r)/(p^T Ap)
  x = x + alpha*p
  rnew = r - alpha*p
  If norm(rnew)/n<=toler Then Exit loop
  beta = (rnew^T rnew) / (r^T r)
  r = rnew
  p = r + beta*p
Loop
Return r and k
```


Listing 13 contains the MATLAB code for the program that implements a basic Conjugate Gradient method.

```
% Conjugate Gradient method
n=1000;
toler=0.00001;
A=1+n*rand(n,n);
for i=1:n
  A(i,i)=sum(A(i,:));
end
x=1:n;
b=A*x';
x=zeros(n,1);
r=b-A*x;
p=r;
k=0;
while 1
  k=k+1;
  alpha=dot(r,r)/dot(p,A*p);
  x=x+alpha*p;
  rnew=r-alpha*A*p;
  if norm(r)/n<=toler, break; end
  beta=dot(rnew,rnew)/dot(r,r);
  r=rnew;
  p=r+beta*p;
end

if n<50
  disp(x)
else
  for i=1:20
    fprintf('%i : %f\n', i, x(i));
  end
    for i=n-19:n
    fprintf('%i : %f\n', i, x(i));
    end
end

fprintf('Iters = %i\n', k);
fprintf('r = %e\n', norm(r)/n);
```

Listing 13. MATLAB code for the Conjugate Gradient method.

Matrices with Positive Coefficients

The next sections discuss the results for running various MATLAB programs. Keep in mind that the values of the norm, r, are not reproducible since the code uses random values for the matrix **A**. We start with matrices with coefficients that have uniformly distributed values in the range of (1, n).

Testing 100 Linear Equations

Method	*Iters*	*r=||Ax-b||/n*	*NumEqns*
Jacobi	2364	9.9747300E-06	1
Jacobi	701	9.8640440E-06	2
Jacobi	446	9.7903420E-06	3
Jacobi	240	9.7973820E-06	5
Jacobi	27	9.0120710E-06	10
Jacobi	15	2.7967990E-06	50
Gauss-Seidel	15	5.4688380E-06	1
Gauss-Seidel	15	3.8962180E-06	2
Gauss-Seidel	15	3.6859340E-06	3
Gauss-Seidel	15	3.1127700E-06	5
Gauss-Seidel	10	4.8007050E-06	10
Gauss-Seidel	8	2.3085020E-06	50
SOR	13	8.3507450E-06	1
SOR	14	4.0413700E-06	2
SOR	14	3.1249080E-06	3
SOR	14	2.1181200E-06	5
SOR	10	3.3007730E-06	10
SOR	11	5.1206290E-06	50
Conjugate Grad.	18 or 19	7.170738e-06	

Table 1. Results for 100 linear equations.

Table 1 shows that the Jacobi method benefits greatly from using the MVSE enhancement. The Gauss-Seidel and SOR methods benefit from the MVSE enhancement when using higher batches of linear equations. Using 50 sets of

equations puts the Gauss-Seidel at a better advantage that SOR! Both the Gauss-Seidel and SOR methods do slightly better than the Conjugate Gradient method.

Testing 1000 Linear Equations

Method	*Iters*	*r=||Ax-b||/n*	*NumEqns*
Jacobi	28101	9.9900970E-06	1
Jacobi	9503	9.9997700E-06	2
Jacobi	5665	9.9623800E-06	3
Jacobi	3113	9.9236900E-06	5
Jacobi	41	5.9364300E-06	10
Jacobi	38	4.8753830E-06	50
Gauss-Seidel	19	5.4991510E-06	1
Gauss-Seidel	19	5.0482230E-06	2
Gauss-Seidel	19	4.9085950E-06	3
Gauss-Seidel	19	4.4350880E-06	5
Gauss-Seidel	14	9.6146900E-07	10
Gauss-Seidel	13	3.1047390E-06	50
SOR	17	3.0215750E-06	1
SOR	17	5.8738710E-06	2
SOR	17	5.8776450E-06	3
SOR	17	5.2698450E-06	5
SOR	12	5.5567790E-06	10
SOR	12	3.5951050E-06	50
Conjugate Grad.	14	5.135925e-06	

Table 2. Results for 1000 linear equations.

Table 2 shows that the Jacobi method benefits greatly from using the MVSE enhancement. The Gauss-Seidel and SOR methods benefit from the MVSE enhancement when using higher batches of linear equations. The SOR method maintains a lead over the Gauss-Seidel method. Both the Gauss-Seidel and SOR methods do slightly better than the Conjugate Gradient method.

Testing 5000 Linear Equations

Method	*Iters*	*r=||Ax-b||/n*	*NumEqns*
Jacobi	19904	9.9990070E-06	5
Jacobi	47	8.2809030E-06	10
Jacobi	46	9.5265400E-06	50
Jacobi	45	9.8166110E-06	100
Jacobi	39	6.1525860E-06	500
Gauss-Seidel	22	2.7090660E-06	1
Gauss-Seidel	22	2.2113710E-06	2
Gauss-Seidel	22	2.2295160E-06	3
Gauss-Seidel	22	2.1843970E-06	5
Gauss-Seidel	16	9.2737010E-07	10
Gauss-Seidel	15	8.3735430E-06	50
Gauss-Seidel	15	6.3182670E-06	100
Gauss-Seidel	14	5.1037130E-06	500
SOR	19	6.4757720E-06	1
SOR	20	2.9925280E-06	2
SOR	20	2.9902730E-06	3
SOR	20	2.9317730E-06	5
SOR	14	2.0755470E-06	10
SOR	14	1.9425890E-06	50

Method	Iters	r=||Ax-b||/n	NumEqns
SOR	14	1.6515960E-06	100
SOR	14	2.4018240E-06	500
Conjugate Grad.	13	1.165085e-06	

Table 3. Results for 5000 linear equations.

Table 3 shows that the Jacobi method benefits greatly from using the MVSE enhancement. The Gauss-Seidel and SOR methods benefit from the MVSE enhancement when using higher batches of linear equations. The SOR method maintains a lead over the Gauss-Seidel method, except for when using 500 equations. Both the Gauss-Seidel and OR methods fail to outperform the Conjugate Gradient method.

Testing 10000 Linear Equations

Method	Iters	r=||Ax-b||/n	NumEqns
Jacobi	50	6.7694330E-06	3
Jacobi	50	6.7711940E-06	5
Jacobi	50	6.6063720E-06	10
Jacobi	49	9.8291580E-06	50
Jacobi	49	7.6078470E-06	100
Jacobi	46	9.1749530E-06	500
Gauss-Seidel	23	1.3113640E-05	1
Gauss-Seidel	23	4.1369770E-06	2
Gauss-Seidel	23	4.1809690E-06	3
Gauss-Seidel	23	4.1250800E-06	5
Gauss-Seidel	15	1.7849650E-06	10
Gauss-Seidel	15	3.0312840E-06	50
Gauss-Seidel	16	5.3179950E-06	100

Method	*Iters*	*r=||Ax-b||/n*	*NumEqns*
Gauss-Seidel	16	8.6212090E-06	500
SOR	21	1.5416330E-05	1
SOR	21	2.4661790E-06	2
SOR	21	2.4908430E-06	3
SOR	21	2.4804840E-06	5
SOR	15	1.7185190E-06	10
SOR	15	2.9382770E-06	50
SOR	15	4.0885080E-06	100
SOR	15	8.7813930E-06	500
Conjugate Grad.	13	6.891991e-07	

Table 4. Results for 10000 linear equations.

Table 4 shows that the Jacobi method benefits greatly from using the MVSE enhancement. The Gauss-Seidel and SOR methods benefit from the MVSE enhancement when using higher batches of linear equations. The SOR method maintains a lead over the Gauss-Seidel method. Both the Gauss-Seidel and OR methods fail to outperform the Conjugate Gradient method.

I studied the relationship between the number of iterations needed to reach an average norm of residuals (calculated as ||**Ax-b**||/n) of 1.00e-5. The following exponential decay equation describes the relation between iteration, k, and the average norm of residuals r:

$$\ln(r) = a_0 - a_1 k \quad (6)$$

The regression coefficients a_0 and a_1 depend on the number of equations, configuration of the coefficients of matrix **A**, number of sublinear systems used by the MSVE method, and the tolerance value for the average norm of residuals.

I have used the exponential decay equation to estimate the value for k in difficult cases. In such cases, convergence for a solution (where r reaches values below 1.00e-5) is painfully slow. The number of iterations is then tens of thousands if not more. Obtaining such results often required a PC to run for over a day, and perhaps two days! The method I use to estimate the number of iterations needed for r to fall below 1e-5 is:

1. Within a loop that solves the system of linear equations (for whatever algorithm), iterate for, say, 5000 times. Store the values of r and the iteration number k in arrays, call them rdata and kdata, respectively.
2. Perform a curve fit (using the Matlab function *polyfit(kdata,log(rdata),1)*) to estimate the values for the regression coefficients a_0 and a_1.
3. Select a target value of r, slightly below 1.00e-5, such as 9.998e-6.
4. Use equation 6 to calculate the value of k.
5. Round up the value of k and then calculate r based on the adjusted value of k.
6. Optionally plot the array values of ln(r) and k to make sure that the plot shows a straight line. This step serves as a visual confirmation for the validity of the estimated final value of the k.

To obtain reliable values for the regression coefficients a_0 and a_1 you need to use large arrays of r and k. Smaller arrays generate increasing error in projecting the value of k for a small value of r. Such error generates smaller projected values of k.

Matrices with Positive and Negative Values

The conclusion discussed the effect of the MSVE method with matrices having all positive values (and relatively high condition numbers). What about matrices with

both positive and negative values? We can initialize such matrices using random normal functions using the following code:

```
A = n*randn(n,n)
for i=1:n
  A(i,i) = sum(abs(A(i,:)));
end
```

Matrices with normally distributed coefficient values include positive and negative values and positive-value dominant diagonal elements. These matrices are much easier to solve with the stationary methods. In this case, the MSVE method contributes little improvement on the number of iterations. Table 5 shows sample results for solving 10,000 linear equations. The MSVE method has virtually no effect on improving the stationary algorithms!

Method	*Iters*	*r=||Ax-b||/n*	*NumEqns*	*Omega*
Jacobi			1	
Jacobi	8	6.5242510E-06	2	
Jacobi	8	6.5373270E-06	3	
Jacobi	8	6.6157030E-06	5	
Jacobi	8	6.7353960E-06	10	
Jacobi	8	6.8959890E-06	50	
Jacobi	8	6.9723120E-06	100	
Jacobi	8	8.6108370E-06	500	
Gauss-Seidel	8	1.1109120E-05	1	
Gauss-Seidel	7	3.4432740E-06	2	
Gauss-Seidel	7	3.3799170E-06	3	
Gauss-Seidel	7	3.4900830E-06	5	
Gauss-Seidel	7	3.5005170E-06	10	
Gauss-Seidel	7	3.8862110E-06	50	
Gauss-Seidel	7	4.2099080E-06	100	

Method	Iters	r=||Ax-b||/n	NumEqns	Omega
Gauss-Seidel	7	6.6414060E-06	500	
SOR	13	1.5823760E-05	1	0.95
SOR	12	4.9789780E-06	2	0.95
SOR	12	5.0540190E-06	3	0.95
SOR	12	4.9781920E-06	5	0.95
SOR	12	5.1099050E-06	10	0.95
SOR	12	5.4362880E-06	50	0.95
SOR	12	5.5972230E-06	100	0.95
SOR	12	7.8666610E-06	500	0.95
SOR	10	5.2505430E-06	10	0.975
SOR	10	5.5447490E-06	50	0.975
SOR	10	5.8530650E-06	100	0.975
SOR	10	7.7765090E-06	500	0.975

Table 5. Results for 10000 linear equations with normally-distributed matrix coefficients with positive and negative values.

The relationship between the number of iterations needed to reach an average norm of residuals (calculated as ||**Ax-b**||/n) of 1.00e-5 for normally distributed matrix coefficients also follow the empirical model in equation 6.

Matrices with Negative Values

In this section I discuss the results of applying the MVSE with matrices having negative coefficients. I will handle two general cases:

1. All the matrix coefficients are negative. The diagonal elements have negative and dominant values.
2. All the matrix coefficients, except the diagonal ones, are negative. The diagonal elements have positive and dominant values.

Case 1 uses the following statements to generate the values for the matrix coefficients:

```
A = -1-n*rand(n,n)
for i=1:n
  A(i,i) = sum(A(i,:));
end
```

Case 2 uses the following statements to generate the values for the matrix coefficients:

```
A = -1-n*rand(n,n)
for i=1:n
  A(i,i) = sum(abs(A(i,:)));
end
```

Tables 6, 7, 8, and 9 show the results for cases 1 and 2 applied to 100, 1000, 5000, and 10000 equations, respectively.

Testing 100 Linear Equations

Table 6 shows the Results for 100 linear equations with negative matrix coefficients for cases 1 and 2. What stands out in that table is that the matrices in case 2 yield solutions that are much slower to converge than those of case 1. For matrices of case 1, the MVSE method is more effective in improving the Jacobi method. In the case of Gauss-Seidel and SOR, the improvement is minor. The table entries with a yellow background refer to results obtained by estimation.

Method	***Case 1***		***Case 2***		***NumEqns***
	Iters	***r***	***Iters***	***r***	
Jacobi	2095	9.9537500E-06	1766	9.9378200E-06	1
Jacobi	247	9.5697600E-06	1583	9.9536300E-06	5
Jacobi	28	6.4262000E-06	29	8.6518000E-06	10
Jacobi	15	2.8936900E-06	20	8.2948200E-06	50
Gauss-Seidel	15	7.9311960E-06	884	9.8742060E-06	1
Gauss-Seidel	15	3.4244100E-06	832	9.8897200E-06	5
Gauss-Seidel	14	3.6224000E-06	807	9.8792700E-06	10
Gauss-Seidel	11	4.1963600E-06	456	9.8235600E-06	50

SOR	13	6.5703400E-06	Error		1
SOR	14	2.4690700E-06	933	9.8964500E-06	5
SOR	13	5.9529100E-06	877	9.8798100E-06	10
SOR	14	4.4172800E-06	498	9.8318700E-06	50

Table 6. Results for 100 linear equations with negative matrix coefficients for cases 1 and 2.

Testing 1000 Linear Equations

Table 7 shows the Results for 1000 linear equations with negative matrix coefficients for cases 1 and 2. The comments on this table are similar to those of Table 6, except the number of iterations needed to reach the solution is higher.

Method	***Case 1***		***Case 2***		***NumEqns***
	Iters	***r***	***Iters***	***r***	
Jacobi	28251	9.9916500E-06	20718	9.9927700E-06	1
Jacobi	3125	9.9631100E-06	40	6.4012000E-06	5
Jacobi	41	5.9590900E-06	40	5.7811200E-06	10
Jacobi	38	4.9080500E-06	39	5.1458700E-06	50
Gauss-Seidel	19	5.3157610E-06	10554	9.9900180E-06	1
Gauss-Seidel	19	4.5422800E-06	10361	9.9898100E-06	5
Gauss-Seidel	19	3.6622900E-06	10328	9.9891500E-06	10
Gauss-Seidel	18	5.5075300E-06	9972	9.9893800E-06	50
SOR	17	2.9984440E-06	11169	9.9989000E-06	1
SOR	17	5.3482500E-06	11631	9.9937300E-06	5
SOR	17	4.7380300E-06	11314	9.9936700E-06	10
SOR	17	3.0320000E-06	10973	9.9930400E-06	50

Table 7. Results for 1000 linear equations with negative matrix coefficients for cases 1 and 2.

Testing 5000 Linear Equations

Table 8 shows the Results for 5000 linear equations with negative matrix coefficients for cases 1 and 2. The comments on this table are similar to those of Table 6, except the number of iterations needed to reach the solution is higher.

Method	***Case 1***		***Case 2***		***NumEqns***
	Iters	***r***	***Iters***	***r***	
Jacobi	162506	9.9970800E-06	114783	9.9978900E-06	1
Jacobi	47	8.8241900E-06	46	6.1411900E-06	5
Jacobi	47	8.2599400E-06	46	5.9909100E-06	10
Jacobi	46	9.5494200E-06	46	7.0016100E-06	50
Jacobi	45	9.8070500E-06	45	7.9671900E-06	100
Jacobi	39	1.9811300E-08	43	5.0578600E-06	500
Gauss-Seidel	22	2.6834290E-06	57493	9.9964940E-06	1
Gauss-Seidel	22	2.1897200E-06	57869	9.9969900E-06	5
Gauss-Seidel	22	2.1373900E-06	58320	9.9946000E-06	10
Gauss-Seidel	21	9.7193700E-06	58415	9.9955700E-06	50
Gauss-Seidel	21	6.0028100E-06	57279	9.9970900E-06	100
Gauss-Seidel	20	2.4869600E-06	52806	9.9856400E-06	500
SOR	19	6.4381070E-06	62216	9.9972610E-06	1
SOR	20	2.9329500E-06			5
SOR	20	2.8546600E-06	63287	9.9949300E-06	10
SOR	19	9.2091600E-06	64003	9.9967100E-06	50
SOR	19	8.8018600E-06	63307	9.9977300E-06	100
SOR	18	9.4892700E-06	58506	9.9960000E-06	500

Table 8. Results for 5000 linear equations with negative matrix coefficients for cases 1 and 2.

Testing 10000 Linear Equations

Table 9 shows the Results for 10000 linear equations with negative matrix coefficients for cases 1 and 2. The comments on this table are similar to those of Table 6, except the number of iterations needed to reach the solution is higher. The table entries with the red background indicate that the calculations have reached a minimum value of r, below which no values can be obtained. This effect has manifested itself only in one case in this study. I assume that decreasing the tolerance value much below1.00e-5 would yield more cases, since a limit will be reached due to rounding calculation errors.

Method	***Case 1***		***Case 2***		***NumEqns***
	Iters	***r***	***Iters***	***r***	
Jacobi	336658	9.9971700E-06	236635	9.9978500E-06	1
Jacobi	50	6.5686800E-06	46	8.8447000E-06	10
Jacobi	49	9.9183700E-06	48	8.2782500E-06	50
Jacobi	49	7.6804400E-06	48	7.6854000E-06	100
Jacobi	46	9.3248700E-06	47	8.5348900E-06	500
Gauss-Seidel	23	1.2937480E-05	119757	9.9969160E-06	1
Gauss-Seidel	23	4.1155800E-06	119781	9.9967300E-06	5
Gauss-Seidel	23	4.0692700E-06	119681	9.9966800E-06	10
Gauss-Seidel	23	3.8181700E-06	119777	9.9979000E-06	50
Gauss-Seidel	23	3.6590600E-06	118318	9.9969500E-06	100
Gauss-Seidel	23	3.6590600E-06	113438	9.9959700E-06	500
SOR	31	1.5664290E-05	129181	9.9966170E-06	1
SOR	21	2.5028500E-06	131233	9.9961600E-06	5
SOR	21	2.5326200E-06	131745	9.9963600E-06	10
SOR	21	2.8574200E-06	132328	9.9967200E-06	50

SOR	21	3.2280800E-06	131222	9.9970300E-06	100
SOR	20	7.2814800E-06	51762	9.9963000E-06	500

Table 9. Results for 10000 linear equations with negative matrix coefficients for cases 1 and 2.

Conclusion

Table 10 contains statistics for the four categories of random matrices. The table shows the mean and standard deviation for the condition numbers of the random matrices. The statistics are calculated based on sample sizes of 35. The matrices with normally distributed positive and negative values have the lowest condition number in the range of (1, 1.62)—that is, close to 1.0. As such, they are the easiest to solve in systems of linear equations. The matrices with all-positive and all-negative values have condition numbers in the range of (2, 2.4) and are therefore relatively easy to solve. The matrix values with negative non-diagonals and positive diagonals (case 2) have very high condition numbers. In fact, the condition numbers increase with the number of equations. This category of matrix is more difficult to solve and gets even more difficult (i.e. slow convergence) as the number of equations increases. The variation between the matrix condition number and the number of equations is close to linear. The condition number statistics in Table 10 confirm the results shown in earlier tables as they relate to how easy or difficult it is to solve linear systems with different types of random matrices.

n	*Mean CN*	*Sdev CN*	*Type*
100	2.39297	0.056761	All positive coefficients
1000	2.13806	0.021338	
5000	2.06533	0.004953	
10000	2.0477	0.003392	
100	2.38314	0.052551	All negative coefficients: Case 1
1000	2.13587	0.014099	
5000	2.06469	0.005155	
10000	2.04762	0.003019	
100	117.262	5.9837	Negative coefficients Case 2
1000	1063.39	18.5218	
5000	5157.99	34.1682	
10000	10233.1	56.77	
100	1.61075	0.038993	Negative and positive
1000	1.19578	0.011926	
5000	1.0911	0.004295	
10000	1.06623	0.002767	

Table 10. Matrix condition number statistics.

Table 11 summaries the results. The results are based on:

1. Using diagonally dominant matrix coefficients. This is a condition when the absolute value of a diagonal element is greater than the sum of the absolute values of non-diagonal elements in the same matrix row.
2. In the case of the SOR, the study uses over-relaxation factor of 0.95. The optimality of this factor is not paramount to this study. Instead, this study seeks to examine the variation in the number of iterations as the number of linear sub-systems varies.
3. The study uses a tolerance value of 1.00e-5 to compare with the average norm of residuals r (=‖**Ax-b**‖/n) values. Remember that the values of r decay exponentially with the number of iterations. In the case where the desired convergence of the average norm of residuals is reached, the difference between that value and the tolerance 1.00e-5 can serve as an indication for convergence speed. Values of r that fall barely below 1.00e-5 indicate that the convergence is very slow, and vice versa.

The MVSE method works best when all the matrix coefficients are positive. I tested four schemes for assigning matrix coefficients. You are welcome to modify the MATLAB code to test more complex value distribution for the matrix coefficients. The improvement brought by the MVSE method is not linearly proportional to the number of linear sub-systems used. Rather, the improvement is stepwise or threshold-wise.

I also tested the Conjugate Gradient method. It seems to perform better than the Gauss-Seidel and SOR methods when dealing with larger number of linear equations.

I did attempt to use a version of MVSE where the updated values of vector x overlap by a single element. While the number of iterations remained the same (at a higher computational effort), the values of the norm, r, where a bit smaller than their counterparts shown in the above tables. I did not pursue this overlapping subset scheme since I did not see any tangible benefits.

Matrix Type	***Comments***
All coefficients are positive	MVSE method improves all three stationary methods.

Matrix Type	*Comments*
All coefficients are negative	MVSE method modestly improves all three stationary methods when the number of linear sub-systems increase to 100 and/or 500.
All non-diagonal coefficients are negative. Diagonal elements are positive.	Solution of this type of matrices is very slow. The number of iterations is significantly higher than for other matrix types. Relative improvement of MVSE appears when the number of linear sub-systems increase to 100 and/or 500.
Mixed positive and negative coefficients.	MSVE method has virtually little effect on improvising the three stationary methods.

Table 11. Results Summary.

Document History

Date	*Version*	*Updates*
November 1, 2016	1.00.00	Initial release.
November 1, 2016	1.01.00	Minor bug adjustment in Listing 13.
November 10, 2016	1.10.00	Added Prologue section.
November 25, 2016	2.00.00	Rearranged document sections and expanded on the types of matrices studied.
November 26, 2016	2.10.00	Added table of condition numbers for various categories of matrix **A**.

