Damian Franco
dfranco24@unm.edu
101789677
CS-575

Homework 9

1) For the first problem we were given the SVD of matrix A. Were we then asked to use the SVD to find out what is the range and null-space of A^T in terms of the columns of matrices U and V. To find the range and null space of A^T using its SVD, I used the columns of matrices U and V. The columns of U show the direction of the output of A, while the columns of V show the direction of the output of A^T. If we want to find vectors that give a zero output when multiplied with A, I used the columns of V that correspond to the zero singular values σ of A. If we want to find vectors that give an output when multiplied with A^T, we can use the columns of U that correspond to the non-zero singular values σ of A. This way, we can use the SVD of A to find the range and null space of A^T using the columns of U and V. My typed "proof" or more like explanation is shown below.
    i) We are given that the SVD of a matrix A is given by $A = U\Sigma V^T$, where U and V are orthonormal matrices and Σ is a diagonal matrix of singular values.
    ii) From the SVD decomposition, we know that the columns of U form an orthonormal basis for the range of A and the columns of V form an orthonormal basis for the range of A^T.
    iii) To see why the columns of U can be used to find the null space of A. Let us first consider the equation $Ax = 0$, where x is a vector in the null space of A. This equation implies that x lies in the orthogonal complement of the range of A. But since the columns of U form an orthonormal basis for the range of A, we know that x can be expressed as a linear combination of the columns of V corresponding to the zero singular values of A.

iv) Similarly, to see why the columns of V can be used to find the range of A^T, consider the equation $y = A^T z$, where y is a vector in the range of A^T and z is some vector. Using the SVD of A, we have $A^T = V\Sigma^T U^T$. Therefore, we can write $y = V\Sigma^T U^T z$, which means that y can be expressed as a linear combination of the columns of U corresponding to the non-zero singular values σ of A.

v) Combining both points in (iii) and (iv), we can see that the columns of U and V can be used to find the range and null space of A^T in terms of the singular values σ of A. Therefore, the SVD of A provides the range and null space of A^T in terms of the columns of matrices U and V.

2) For this question, we are given the matrix *A* as a pseudo input for the Gauss-Seidel method. We have multiple parts to this problem that involve finding the iteration matrix, discussing if the Gauss-Seidel method will converge and such. For parts (a) through (c), I will be using Python to compute the answers, as instructed in our homework.

   a) The first question asks us to see if the matrix *A* is invertible. I first set up the matrix and found the determinant of the matrix with a NumPy function. I checked if the determinant is non-zero. If it is, then the matrix is invertible, otherwise, it is singular and not invertible. I found that the matrix is invertible since the determinant was non-zero. Below is my code for the setup of *A* and how I found that it is indeed invertible.

```
# Set up matrix A
A_GS = np.array([[1, -1, 0], [-1, 2, -1], [0, -1, 2]])
A_GS
```

```
array([[ 1, -1,  0],
       [-1,  2, -1],
       [ 0, -1,  2]])
```

```
# Compute the determinant of A
det_A_GS = np.linalg.det(A_GS)

if det_A_GS != 0:
    print("A is invertible")
else:
    print("A is not invertible")
```

```
A is invertible
```

b) Next we were tasked to find the spectral radius of the iteration matrix *M*. First, I had to find the iteration matrix *M* which I did first by getting the *LU* decomposition of *A* and then by multiplying the inverse of the lower-triangular part of *A* by the upper-triangular part of *A*. I also used the diagonal part of *A* and subtracted it by *L* before taking the inverse and multiplying to *U*. I used multiple NumPy methods and found that the spectral radius of *M* is 0.75. Below is my code for finding *p(M)*.

```
# Compute the iteration matrix
D = np.diag(np.diag(A_GS))
L = np.tril(A_GS, -1)
U = np.triu(A_GS, 1)
M_GS = np.linalg.inv(D - L) @ U
print(M_GS)
```

```
[[ 0.   -1.    0.  ]
 [ 0.    0.5  -0.5 ]
 [ 0.   -0.25  0.25]]
```

```
# Compute the eigenvalues and p(M)
eigVals = np.linalg.eigvals(M_GS)
spectral_radius = np.max(np.abs(eigVals))
print("Spectral radius =", spectral_radius)
```

```
Spectral radius = 0.75
```

c) This part asks us to find the 1-norm, 2-norm and infinity-norm of our iteration matrix *M*. I once again used NumPy functions to do this for me and found the following.

```
# 1-norm
one_norm = np.linalg.norm(M_GS, ord=1)
print("1-norm =", one_norm)
```

```
1-norm = 1.75
```

```
# 2-norm
two_norm = np.linalg.norm(M_GS, ord=2)
print("2-norm =", two_norm)
```

```
2-norm = 1.1841130945790346
```

```
# Infinity norm
inf_norm = np.linalg.norm(M_GS, ord=np.inf)
print("Infinity norm =", inf_norm)
```

```
Infinity norm = 1.0
```

d) For the last part, we were asked some questions about this system. I believe that this matrix *A* will converge to a true solution with the Gauss-Seidel method. I believe that because for the Gauss-Seidel method, the algorithm will converge if and only if the spectral radius of its iteration matrix is strictly less than 1. We see that each norm for the iteration matrix is computed to be 1 or above which goes against what we studied in class that the Gauss-Seidel method will converge if the norm of *M* is less than one for some norm. In our case, we do not have a norm lower than 1, but we see that the spectral radius is low and if the spectral radius is strictly less than 1, then the method is guaranteed to converge, regardless of the norms of the iteration matrix. I believe this is because the spectral radius controls the overall behavior of the iteration process.

3) This question asks us various parts but mainly involves us implementing two methods that solve a linear system $Ax = b$ which are the Jacobi Method and Gauss-Seidel Algorithm. Since there are multiple parts, I will explain my reasoning in each section below.
    a) First, we were asked to construct a linear system (similar to one in some example code that Professor Zeb gave us in *poisson_1d.ipynb*). This linear system had to have the same features as the tridiagonal matrix generated in the example code, but this one will have to properly store it as a sparse matrix instead of a dense one, meaning no zeros in our matrix. To do this, I simply used the SciPy library to construct the matrix A and did the same construction for vector b as the example code. My code for this is shown below with an example of what my output looks like.

Form sparse system and testing:

```
def form_sparse_lin_sys(N):
    # Form matrix A
    A = scipy.sparse.diags([-1, 2, -1], [-1, 0, 1], shape=(N, N))

    # Form vector b
    h = 1/(N+1)
    b = np.zeros((N,))
    for i in range(N):
        b[i] = h**2 * f( (i+1)*h )

    return A, b
```

```
# Testing out functionality of the function
N_test = 10
A_test, b_test = form_sparse_lin_sys(N_test)
A_test
```

```
<10x10 sparse matrix of type '<class 'numpy.float64'>'
        with 28 stored elements (3 diagonals) in DIAgonal format>
```

b) Next, we were simply asked to solve the system with the SciPy sparse solve function. I did this with the 10x10 example matrix above and as well as in my for loop for iterations. Below is my code for this and my approximate solution for the 10x10 example.

Testing out sparse solving:

```
x_star = scipy.sparse.linalg.spsolve(A_test, b_test)
x_star
```

```
array([ 0.55558324,  0.93477272,  1.01717847,  0.77663724,  0.28951918,
       -0.28951918, -0.77663724, -1.01717847, -0.93477272, -0.55558324])
```

c) Next we were asked to implement both Jacobi and Gauss Seidel's algorithms but for each algorithm to take advantage of the tridiagonal property of the *A* matrix and avoid the sparse areas when computing. To do so, in both algorithms I first converted the input matrix *A* to CSR format using the *csr_matrix* function from SciPy. This format is optimized for sparseness and only stores the nonzero elements of the matrix. Both the Jacobi method and Gauss-Seidel were implemented off some pseudocode that I found through various resources online and in textbooks. I added a few more elements that have my own coding flare like that checking for strictly diagonally dominant matrices as the input. I tested out with various matrices of different sizes and saw that each algorithm converged at an average of under 500 iterations and that the errors generated were very low that it would exit out of the tolerance level I set, which is 10^{-10}. On the next couple of pages is my code for both the Jacobi and Gauss-Seidel algorithms and some testing for it.

Jacobi Method:

```python
def jacobi(A, b, n, max_iter):
    # Define tolerance for error
    tol = 1e-10

    # Check if input matrix A is strictly diagonally dominant
    if not strictDiagDom(A):
        print('A is not strictly diagonally dominant, may not converge')

    # Convert A to CSR format
    A = scipy.sparse.csr_matrix(A)
    x = np.zeros_like(n)
    x_star = scipy.sparse.linalg.spsolve(A, b)

    for i in range(max_iter):
        # Compute residual
        r = b - A.dot(x)
        D = A.diagonal()

        # Compute the Jacobi update
        delta = np.divide(r, D)
        x_new = x + delta

        # Check for convergence
        if np.linalg.norm(delta) < tol:
            break

        x = x_new

    # Compute iteration matrix and spectral radius
    M = scipy.sparse.diags(1/D, 0).dot(A)
    spectral_radius = np.abs(scipy.sparse.linalg.eigs(M, k=1, return_eigenvectors=False))[0]

    # Compute the error
    err_st1 = abs(x_star - x)
    err = scipy.linalg.norm(err_st1)

    return x, spectral_radius, err
```

Gauss-Seidel's Method:

```python
def gaussSeidel(A, b, n, max_iter):
    # Define tolerance for error
    tol = 1e-10

    # Check if input matrix A is strictly diagonally dominant
    if not strictDiagDom(A):
        print('A is not strictly diagonally dominant, may not converge')

    # Convert A to CSR format
    A = scipy.sparse.csr_matrix(A)
    x = np.zeros(n)
    x_star = scipy.sparse.linalg.spsolve(A, b)

    for i in range(max_iter):
        for j in range(n):
            # Compute the residual for the jth equation
            r_j = b[j] - A[j,:].dot(x)
            D_jj = A[j,j]

            # Compute the Gauss-Seidel update for the jth variable
            x[j] += r_j / D_jj

        # Check for convergence
        if np.linalg.norm(r_j) < tol:
            break

    # Compute iteration matrix and spectral radius
    M = scipy.sparse.tril(A, k=-1).dot(scipy.sparse.linalg.spsolve(scipy.sparse.triu(A, k=0), scipy.sparse.eye(n)))
    spectral_radius = np.abs(scipy.sparse.linalg.eigs(M, k=1, return_eigenvectors=False))[0]

    # Compute the error
    err_st1 = abs(x_star - x)
    err = scipy.linalg.norm(err_st1)

    return x, spectral_radius, err
```

Testing with different matrices of various sizes:

```
Current Matrix Size N: 10
[ 0.55558324  0.93477272  1.01717847  0.77663724  0.28951918 -0.28951918
 -0.77663724 -1.01717847 -0.93477272 -0.55558324]
[ 0.55558324  0.93477272  1.01717847  0.77663725  0.28951918 -0.28951918
 -0.77663724 -1.01717847 -0.93477272 -0.55558324]
Jacobi Method Spectral Radius: 1.9594929736144946
Gauss-Seidel Spectral Radius: 0.9206267664155896

Current Matrix Size N: 20
[ 0.29696393  0.56754131  0.78769016  0.93784928  1.00467637  0.98223356
  0.87251499  0.68526963  0.43713506  0.15015912 -0.15015912 -0.43713506
 -0.68526963 -0.87251499 -0.98223356 -1.00467637 -0.93784928 -0.78769016
 -0.56754131 -0.29696393]
[ 0.29696393  0.56754132  0.78769017  0.93784929  1.00467638  0.98223358
  0.872515    0.68526965  0.43713508  0.15015913 -0.1501591  -0.43713504
 -0.68526962 -0.87251498 -0.98223355 -1.00467636 -0.93784927 -0.78769015
 -0.56754131 -0.29696393]
Jacobi Method Spectral Radius: 1.9888308262251309
Gauss-Seidel Spectral Radius: 0.9777864028930712
```

```
 Current Matrix Size N: 40
 [ 0.15294722  0.30230949  0.44458588  0.57644156  0.69478593  0.79684511
   0.88022692  0.94297695  0.9836244   1.00121653  0.995341    0.96613551
   0.91428463  0.84100369  0.74801033  0.63748425  0.51201608  0.37454668
   0.22829823  0.07669866 -0.07669866 -0.22829823 -0.37454668 -0.51201608
  -0.63748425 -0.74801033 -0.84100369 -0.91428463 -0.96613551 -0.995341
  -1.00121653 -0.9836244  -0.94297695 -0.88022692 -0.79684511 -0.69478593
  -0.57644156 -0.44458588 -0.30230949 -0.15294722]
 [ 0.15296815  0.30235109  0.44464775  0.57652316  0.69488661  0.79696407
   0.88036325  0.94312965  0.98379235  1.00139852  0.99553575  0.96634168
   0.9145008   0.84122842  0.74824215  0.63772164  0.51225753  0.37479068
   0.22854327  0.07694327 -0.07645593 -0.22805879 -0.37431189 -0.51178724
  -0.63726261 -0.74779707 -0.8407999  -0.91409135 -0.9659537  -0.99517152
  -1.00106018 -0.98348188 -0.94284888 -0.88011384 -0.79674747 -0.69470411
  -0.57637584 -0.44453648 -0.30227653 -0.15293076]
 Jacobi Method Spectral Radius: 1.9970658011837445
 Gauss-Seidel Spectral Radius: 0.9941402118901763
```

```
Current Matrix Size N: 80
[ 0.07371571  0.14698809  0.21937646  0.29044547  0.35976771  0.42692627
  0.49151724  0.55315217  0.61146038  0.66609119  0.71671606  0.7630305
  0.80475599  0.84164159  0.87346545  0.90003619  0.92119401  0.93681165
  0.9467952   0.95108462  0.9496541   0.94251224  0.92970201  0.91130044
  0.8874182   0.85819893  0.82381834  0.78448322  0.74043012  0.69192398
  0.63925654  0.58274453  0.52272782  0.45956737  0.39364303  0.32535127
  0.25510281  0.18332014  0.11043495  0.03688559 -0.03688559 -0.11043495
 -0.18332014 -0.25510281 -0.32535127 -0.39364303 -0.45956737 -0.52272782
 -0.58274453 -0.63925654 -0.69192398 -0.74043012 -0.78448322 -0.82381834
 -0.85819893 -0.8874182  -0.91130044 -0.92970201 -0.94251224 -0.9496541
 -0.95108462 -0.9467952  -0.93681165 -0.92119401 -0.90003619 -0.87346545
 -0.84164159 -0.80475599 -0.7630305  -0.71671606 -0.66609119 -0.61146038
 -0.55315217 -0.49151724 -0.42692627 -0.35976771 -0.29044547 -0.21937646
 -0.14698809 -0.07371571]
[ 0.07770877  0.15495263  0.23126877  0.30619994  0.37929719  0.45012261
  0.51825189  0.58327695  0.64480833  0.70247756  0.75593938  0.80487377
  0.84898793  0.88801801  0.9217307   0.94992463  0.97243156  0.98911743
  0.99988316  1.00466519  1.00343593  0.99620387  0.98301359  0.96394542
  0.939115    0.90867258  0.87280209  0.83172007  0.78567434  0.73494252
  0.67983035  0.62066986  0.55781737  0.49165136  0.42257016  0.35098959
  0.27734041  0.20206582  0.12561867  0.04845886 -0.02894952 -0.10614093
 -0.1826512  -0.25802031 -0.33179517 -0.40353232 -0.47280064 -0.53918389
 -0.60228325 -0.66171972 -0.71713636 -0.76820049 -0.81460563 -0.85607341
 -0.89235518 -0.92323354 -0.94852365 -0.96807431 -0.98176891 -0.98952609
 -0.99130026 -0.98708185 -0.97689737 -0.96080927 -0.93891554 -0.91134912
 -0.87827712 -0.83989978 -0.7964493  -0.74818841 -0.69540881 -0.6384294
 -0.57759438 -0.51327115 -0.44584812 -0.37573235 -0.30334713 -0.2291294
 -0.15352716 -0.07699672]
Jacobi Method Spectral Radius: 1.9992479525042306
Gauss-Seidel Spectral Radius: 0.9984964705839006
```

```
Current Matrix Size N: 160
[ 0.02080335  0.04157501  0.06228336  0.08289687  0.10338414  0.12371397
  0.1438554   0.16377777  0.18345073  0.20284432  0.22192902  0.24067576
  0.25905598  0.27704171  0.29460554  0.31172075  0.32836125  0.34450171
  0.36011755  0.375185    0.3896811   0.40358378  0.41687186  0.42952512
  0.44152429  0.45285108  0.46348826  0.47341962  0.48263004  0.4911055
  0.49883308  0.50580102  0.51199872  0.51741672  0.52204678  0.52588185
  0.52891609  0.53114487  0.53256482  0.53317375  0.53297074  0.53195611
  0.5301314   0.52749939  0.52406408  0.51983071  0.51480572  0.50899677
  0.5024127   0.49506354  0.48696048  0.47811585  0.46854314  0.45825691
  0.44727283  0.43560763  0.42327907  0.41030593  0.39670796  0.38250587
  0.36772128  0.35237672  0.33649554  0.32010194  0.30322087  0.28587805
  0.26809988  0.24991345  0.23134643  0.21242711  0.1931843   0.17364731
  0.15384587  0.13381016  0.11357067  0.09315824  0.07260394  0.05193907
  0.03119512  0.01040365 -0.01040365 -0.03119512 -0.05193907 -0.07260394
 -0.09315824 -0.11357067 -0.13381016 -0.15384587 -0.17364731 -0.1931843
 -0.21242711 -0.23134643 -0.24991345 -0.26809988 -0.28587805 -0.30322087
 -0.32010194 -0.33649554 -0.35237672 -0.36772128 -0.38250587 -0.39670796
 -0.41030593 -0.42327907 -0.43560763 -0.44727283 -0.45825691 -0.46854314
 -0.47811585 -0.48696048 -0.49506354 -0.5024127  -0.50899677 -0.51480572
 -0.51983071 -0.52406408 -0.52749939 -0.5301314  -0.53195611 -0.53297074
 -0.53317375 -0.53256482 -0.53114487 -0.52891609 -0.52588185 -0.52204678
 -0.51741672 -0.51199872 -0.50580102 -0.49883308 -0.4911055  -0.48263004
 -0.47341962 -0.46348826 -0.45285108 -0.44152429 -0.42952512 -0.41687186
 -0.40358378 -0.3896811  -0.375185   -0.36011755 -0.34450171 -0.32836125
 -0.31172075 -0.29460554 -0.27704171 -0.25905598 -0.24067576 -0.22192902
 -0.20284432 -0.18345073 -0.16377777 -0.1438554  -0.12371397 -0.10338414
 -0.08289687 -0.06228336 -0.04157501 -0.02080335]
[ 0.03032731  0.06062215  0.09083859  0.1209308   0.15085309  0.18056006
  0.21000658  0.23914791  0.26793977  0.29633837  0.32430053  0.35178369
  0.37874603  0.40514648  0.43094484  0.4561018   0.48057899  0.50433911
  0.52734589  0.54956423  0.5709602   0.59150111  0.61115559  0.62989357
  0.64768639  0.66450682  0.68032909  0.69512895  0.7088837   0.72157222
  0.73317503  0.74367425  0.75305373  0.76129898  0.76839726  0.77433755
  0.77911061  0.78270896  0.78512692  0.7863606   0.7864079   0.78526856
  0.78294408  0.77943781  0.77475486  0.76890217  0.76188844  0.75372415
  0.74442152  0.73399452  0.72245883  0.70983182  0.69613254  0.68138166
  0.66560145  0.64881577  0.63105001  0.61233106  0.59268725  0.57214834
  0.55074546  0.52851106  0.50547885  0.48168378  0.45716195  0.43195059
  0.40608798  0.37961339  0.35256704  0.32499003  0.29692427  0.26841242
  0.23949784  0.21022448  0.18063689  0.15078006  0.12069943  0.09044079
  0.06005017  0.02957386 -0.00094174 -0.03145017 -0.06190494 -0.09225966
 -0.12246808 -0.15248416 -0.18226214 -0.21175664 -0.24092267 -0.26971575
 -0.29809195 -0.32600799 -0.35342127 -0.38028992 -0.40657295 -0.4322302
 -0.45722249 -0.48151163 -0.5050605  -0.5278331  -0.54979461 -0.57091143
 -0.59115124 -0.61048306 -0.62887729 -0.64630573 -0.66274168 -0.67815992
 -0.69253678 -0.70585018 -0.71807965 -0.72920637 -0.73921321 -0.74808471
 -0.75580717 -0.76236862 -0.76775887 -0.77196949 -0.77499388 -0.77682721
 -0.7774665  -0.77691056 -0.77516003 -0.7722174  -0.76808693 -0.76277472
 -0.75628867 -0.74863848 -0.73983562 -0.7298933  -0.71882651 -0.70665192
 -0.69338794 -0.6790546  -0.66367359 -0.64726821 -0.62986331 -0.61148529
 -0.59216202 -0.57192286 -0.55079852 -0.52882113 -0.50602408 -0.48244205
 -0.45811093 -0.43306775 -0.40735064 -0.38099879 -0.35405235 -0.32655241
 -0.2985409  -0.27006055 -0.24115484 -0.2118679  -0.18224444 -0.15232974
 -0.12216951 -0.09180987 -0.06129725 -0.03067833]
Jacobi Method Spectral Radius: 1.999809627498074
Gauss-Seidel Spectral Radius: 0.9996192912378363
```

d) Next, we were asked to compute the spectral radius of our iteration matrices with the *wigs* functions and asked what it implies. From my understanding, the spectral radius of the iteration matrix provides important information about its convergence behavior. Specifically, the spectral radius determines the rate at which the iterative method converges towards the true solution.I found that for Jacobi's method the spectral radius was around 1 and Gauss-Seidel's algorithm also had a spectral radius around 1, but below one. I mention that it is below one because if the spectral radius is less than 1, then the iterative method will converge to the true solution. The smaller the spectral radius, the faster the convergence. This is because a smaller spectral radius means that the magnitudes of the eigenvalues of the iteration matrix are smaller, which implies that the errors introduced in each iteration become smaller. I am going to say that Gauss-Seidel's algorithm is much faster in convergence and has a likelihood of converging to a true solution compared to Jacobi's which has a likelihood of divergence instead of convergence since the spectral radius is larger than the Gauss-Seidel's spectral radius. As my dimensions of each matrix I notice that my spectral radius is getting closer and closer to 1 which indicates less of a convergence to me. Below are my findings for the spectral radius of both algorithms.

Iteration	Jacobi Spectral Radius	Gauss-Seidel Spectral Radius
1	0.9970658011837406	0.9941402118901742
10	0.9970658011837443	0.9941402118901723
20	0.9970658011837377	0.9941402118901739
30	0.9970658011837443	0.9941402118901734
40	0.997065801183743	0.9941402118901738
50	0.9970658011837434	0.9941402118901761
60	0.99706580118374	0.9941402118901793
70	0.9970658011837386	0.9941402118901742
80	0.9970658011837428	0.9941402118901743
90	0.9970658011837392	0.9941402118901747

e) We were then asked to tabulate our results with some iterations and see how each method converges to the solution. The absolute error will show how close the solution is to the exact solution while the relative error or η_k is comparable to the rate of convergence the algorithm has. I will discuss these results in the next part of the problem. My table is shown below.

Iteration	Jacobi Absolute Error	Gauss-Seidel Absolute Error	Jacobi Relative Error	Gauss-Seidel Relative Error
1	4.483397476812058	4.433605189224998	1.0	1.0
10	4.032078494457569	3.6032739754310357	0.8993354961970935	0.8127187292608015
20	3.5836938880328084	2.8592588938131285	0.799325490672536	0.6449060689395602
30	3.18517159345417	2.267929417788103	0.7104370312754428	0.5115316589983829
40	2.8309667055063206	1.7990152941449997	0.6314333538679004	0.4057680414388613
50	2.516150936469359	1.4279967321726026	0.5612152278451298	0.3220847755328026
60	2.2363440455804744	1.1351204094202614	0.4988056618104353	0.256026497844
70	1.9876528938366993	0.9045264266480328	0.44333631004539653	0.20401600684840085
80	1.7666172761677774	0.7235124330010749	0.3940353906394976	0.16318828630917093
90	1.570161777306207	0.5819200962597871	0.350216947176113	0.13125212359323943

f) These results show that the Gauss-Seidel method is the better method for faster convergence to an appropriate solution. As seen above in the table, the absolute error for Jacobi's method is much larger than the Gauss-Seidel algorithm and convergence rate is much slower in the Jacobi method. Each algorithm had comparable performance where both did not outperform the other by more than microseconds. I notice the relative error does not tend towards the spectral radius for both algorithms. Overall, I found that Gauss-Seidel was the algorithm to choose in this case. I feel that way because for solving systems with large and sparse systems of equations, it utilizes more information than Jacobi about the matrix in each iteration. I believe each situation is different and it depends on the specific problem being solved, but for this case, I think Gauss-Seidel is the more appropriate algorithm to use.