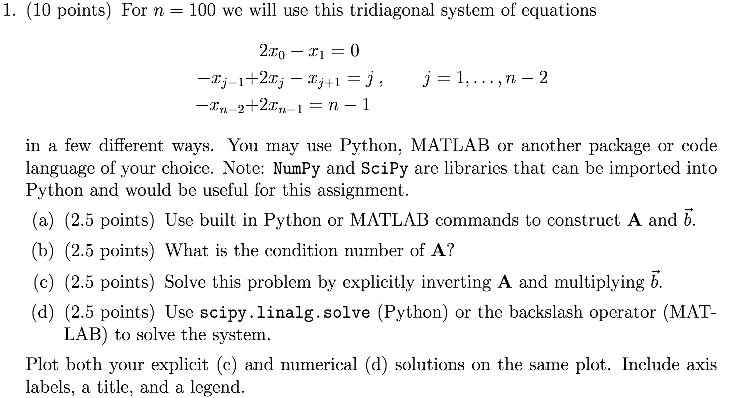
**Assignment 5**



a)

|  |
| --- |
| Ne155\_hw5\_p1 (code that generates matrix A and array b) |
| import numpy as np  import scipy  def tridiag(a, b, c, k1=-1, k2=0, k3=1):  return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)  #A function that creates tridiagonal matrices  a =[]  i = 0  while i < 99:  a.append(-1)  i+=1  #Assigns value to lower diagonal and upper diagonal  b =[]  j = 0  while j < 100:  b.append(2)  j+=1  #Assigns value to center diagonal  A = tridiag(a, b, a)  b = np.zeros((100, 1))  num = np.linspace(0, 99, 100)  for i in num:  b[i][0] = i |

b)

|  |
| --- |
| Ne155\_hw4\_p1 |
| np.linalg.cond(A)  #Condition number of matrix A |
| Returns: 4133.643 |

The condition number for matrix A is therefore 4133.643.

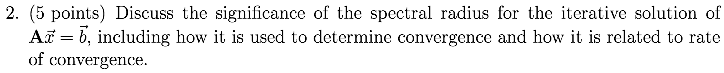
c)

|  |
| --- |
| Ne155\_hw5\_p1 |
| inv\_A = np.linalg.inv(A)  #Computes the inverse of matrix A  inv\_sol = np.dot(inv\_A, b)  #Computes x by multiplying the inverse of A with array b  print(inv\_sol) |
| [[ 1650.]  [ 3300.]  [ 4949.]  [ 6596.]  [ 8240.]  [ 9880.]  [ 11515.]  [ 13144.]  [ 14766.]  [ 16380.]  [ 17985.]  [ 19580.]  [ 21164.]  [ 22736.]  [ 24295.]  [ 25840.]  [ 27370.]  [ 28884.]  [ 30381.]  [ 31860.]  [ 33320.]  [ 34760.]  [ 36179.]  [ 37576.]  [ 38950.]  [ 40300.]  [ 41625.]  [ 42924.]  [ 44196.]  [ 45440.]  [ 46655.]  [ 47840.]  [ 48994.]  [ 50116.]  [ 51205.]  [ 52260.]  [ 53280.]  [ 54264.]  [ 55211.]  [ 56120.]  [ 56990.]  [ 57820.]  [ 58609.]  [ 59356.]  [ 60060.]  [ 60720.]  [ 61335.]  [ 61904.]  [ 62426.]  [ 62900.]  [ 63325.]  [ 63700.]  [ 64024.]  [ 64296.]  [ 64515.]  [ 64680.]  [ 64790.]  [ 64844.]  [ 64841.]  [ 64780.]  [ 64660.]  [ 64480.]  [ 64239.]  [ 63936.]  [ 63570.]  [ 63140.]  [ 62645.]  [ 62084.]  [ 61456.]  [ 60760.]  [ 59995.]  [ 59160.]  [ 58254.]  [ 57276.]  [ 56225.]  [ 55100.]  [ 53900.]  [ 52624.]  [ 51271.]  [ 49840.]  [ 48330.]  [ 46740.]  [ 45069.]  [ 43316.]  [ 41480.]  [ 39560.]  [ 37555.]  [ 35464.]  [ 33286.]  [ 31020.]  [ 28665.]  [ 26220.]  [ 23684.]  [ 21056.]  [ 18335.]  [ 15520.]  [ 12610.]  [ 9604.]  [ 6501.]  [ 3300.]] |

d)

|  |
| --- |
| Ne155\_hw5\_p1 |
| x =np.linalg.solve(A, b)  #solves Ax = b |
| [[ 1650.]  [ 3300.]  [ 4949.]  [ 6596.]  [ 8240.]  [ 9880.]  [ 11515.]  [ 13144.]  [ 14766.]  [ 16380.]  [ 17985.]  [ 19580.]  [ 21164.]  [ 22736.]  [ 24295.]  [ 25840.]  [ 27370.]  [ 28884.]  [ 30381.]  [ 31860.]  [ 33320.]  [ 34760.]  [ 36179.]  [ 37576.]  [ 38950.]  [ 40300.]  [ 41625.]  [ 42924.]  [ 44196.]  [ 45440.]  [ 46655.]  [ 47840.]  [ 48994.]  [ 50116.]  [ 51205.]  [ 52260.]  [ 53280.]  [ 54264.]  [ 55211.]  [ 56120.]  [ 56990.]  [ 57820.]  [ 58609.]  [ 59356.]  [ 60060.]  [ 60720.]  [ 61335.]  [ 61904.]  [ 62426.]  [ 62900.]  [ 63325.]  [ 63700.]  [ 64024.]  [ 64296.]  [ 64515.]  [ 64680.]  [ 64790.]  [ 64844.]  [ 64841.]  [ 64780.]  [ 64660.]  [ 64480.]  [ 64239.]  [ 63936.]  [ 63570.]  [ 63140.]  [ 62645.]  [ 62084.]  [ 61456.]  [ 60760.]  [ 59995.]  [ 59160.]  [ 58254.]  [ 57276.]  [ 56225.]  [ 55100.]  [ 53900.]  [ 52624.]  [ 51271.]  [ 49840.]  [ 48330.]  [ 46740.]  [ 45069.]  [ 43316.]  [ 41480.]  [ 39560.]  [ 37555.]  [ 35464.]  [ 33286.]  [ 31020.]  [ 28665.]  [ 26220.]  [ 23684.]  [ 21056.]  [ 18335.]  [ 15520.]  [ 12610.]  [ 9604.]  [ 6501.]  [ 3300.]] |

|  |
| --- |
| Ne155\_hw5\_p1 |
| plt.plot(inv\_sol)  plt.plot(x)  plt.title('Solution Method Comparison')  plt.legend(("Inverse A solution", "Built-in Function Solution"), loc = 'upper right')  plt.show() |
|  |



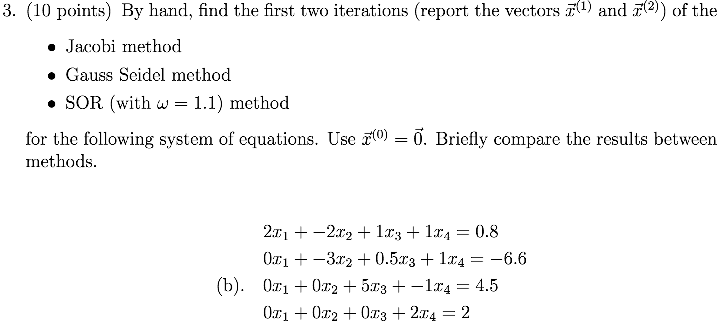
Suppose we have a linear system Ax = b and a preconditioner P where P is close to A but does not equal A (for fast iterations), then we have a linear system as such:

And an iteration defined by:

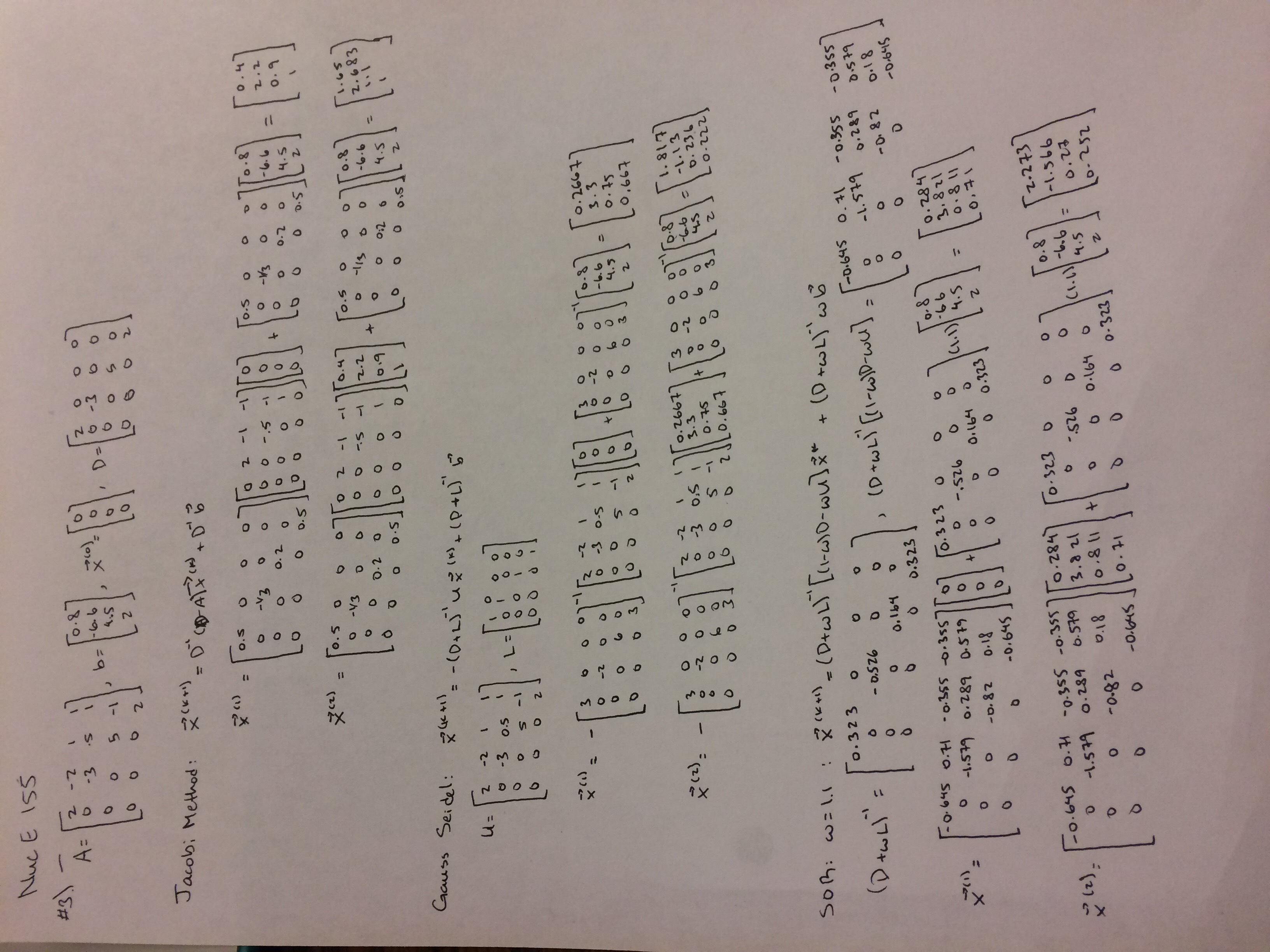
The error is then defined by:

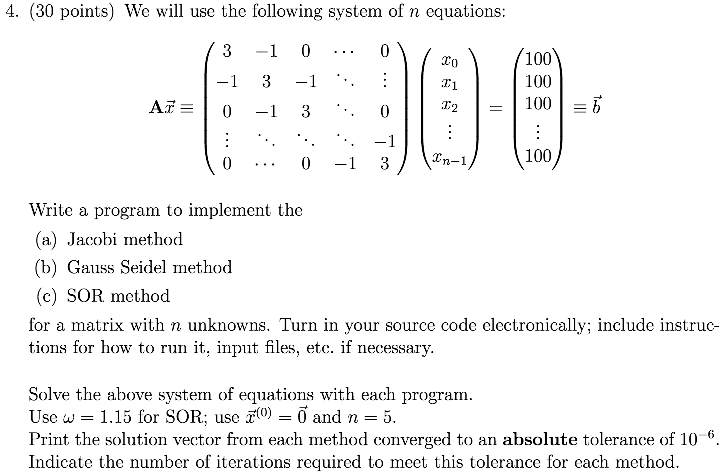
After combining equations:

In order for x to converge, then the spectral radius of M must be less than 1. **The spectral radius sets the convergence rate.** If the initial error is an eigenvector of matrix M, then for every step in the iteration, that error will be multiplied by the corresponding eigenvalue, hence if the eigenvalue is larger than one, then the error will grow excessively large and x will not converge.



The simplest to calculate was Jacobi since the algorithm for Jacobi is rather small compared to Gauss Seidel and SOR. However, Gauss Seidel allows for more information in the algorithm and therefore can converge twice as fast. SOR is similar to Gauss Seidel, however, since it has a component that can be controlled (“omega”), it allows for a faster iteration.

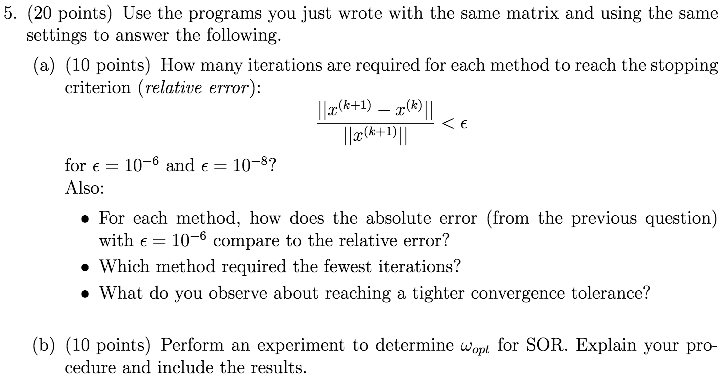




In order to run the code, one must edit the file at line 35 and put in the desired number of rows and columns for matrix A into the matrix\_gen argument. Then, simply run the file through ipython using the terminal.

|  |
| --- |
| Code: Ne155\_hw5\_4 |
| import numpy as np  import scipy.linalg  import matplotlib.pyplot as plt  #The following function creates our tridiagonal matrix  def matrix\_gen(n):  a =[]  i = 0  while i < n - 1:  a.append(-1)  i+=1  #Assigns value to lower diagonal and upper diagonal  b =[]  j = 0  while j < n:  b.append(3)  j+=1  def tridiag(a, b, c, k1=-1, k2=0, k3=1):  return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)  #A function that creates tridiagonal matrices  A = tridiag(a, b, a)  b\_array = np.zeros((n, 1))  num = np.linspace(0, n - 1, n)  for i in num:  b\_array[i][0] = 100  x\_0 = np.zeros((n, 1))  return A, b\_array, x\_0  A, b\_array, x\_0 = matrix\_gen(5)  #The following function calculates an approximation for x via Jacobi Method  D = np.diag(A)  p\_j = np.diagflat(D) - A  def Jacobi\_method(x):  return np.dot(np.linalg.inv(np.diagflat(D)),np.dot(p\_j, x)+b\_array)  exact\_sol = np.dot(np.linalg.inv(A), b\_array)  def absolute\_tolerance(x):  return np.linalg.norm(x- exact\_sol)  abs\_tol = 1  i = 0  x = x\_0  #Iterates using Jacobi until desired absolute tolerance is achieved  while abs\_tol > 10 \*\* (-6):  x = Jacobi\_method(x)  abs\_tol = absolute\_tolerance(x)  i += 1  print("solutions using Jacobi Method:")  print(x)  print("required iterations using Jacobi method:")  print(i)  print(abs\_tol)  #The following function will use Gauss Seidel Iteration to solve x  L = np.tril(A) #np.tril gives D+L  U = A - L #A = L+U  coef\_1 = np.linalg.inv(L)  coef\_2 = np.dot(coef\_1, -U)  addition = np.dot(coef\_1, b\_array)  def Gauss\_Seidel(x):  return np.dot(coef\_2, x) + addition  abs\_tol = 1  i = 0  x = x\_0  #Iterates using Gauss Seidel until desired absolute tolerance is achieved  while abs\_tol > 10 \*\* (-6):  x = Gauss\_Seidel(x)  abs\_tol = absolute\_tolerance(x)  i += 1  print("solutions using Gauss Seidel Method:")  print(x)  print("required iterations using Gauss Seidel method:")  print(i)  print(abs\_tol)  D = np.diagflat(D)  #The following function will use SOR Iteration  def SOR(x, w=1.15):  L\_prime = L - D  coef1 = np.linalg.inv(D+(w\*L\_prime))  coef2 = np.dot(coef1, (((1-w)\*D) - (w\*U)))  addition = np.dot(coef1, w\*b\_array)  return np.dot(coef2, x) + addition  abs\_tol = 1  i = 0  x = x\_0  #Iterates using SOR until desired absolute tolerance is achieved  while abs\_tol > 10 \*\* (-6):  x = SOR(x)  abs\_tol = absolute\_tolerance(x)  i += 1  print("solutions using SOR Method:")  print(x)  print("required iterations using SOR method:")  print(i)  print(abs\_tol) |
| Output:  solutions using Jacobi Method:  [[ 61.1111109 ]   [ 83.33333295]   [ 88.88888846]   [ 83.33333295]   [ 61.1111109 ]]  required iterations using Jacobi method:  35  7.59877296935e-07  solutions using Gauss Seidel Method:  [[ 61.11111051]   [ 83.33333273]   [ 88.88888849]   [ 83.33333313]   [ 61.11111104]]  required iterations using Gauss Seidel method:  18  9.65124548512e-07  solutions using SOR Method:  [[ 61.11111105]   [ 83.33333328]   [ 88.88888886]   [ 83.33333332]   [ 61.1111111 ]]  required iterations using SOR method:  12  8.83675321692e-08 |

The output of the code shows that number of iterations required to reach the desired absolute tolerance for Jacobi is 35, for Gauss Seidel it takes 18 iterations (pretty much half of Jacobi, which makes sense because Gauss Seidel is twice as fast as Jacobi as indicated in the lecture notes), and for SOR it takes 12 iterations.



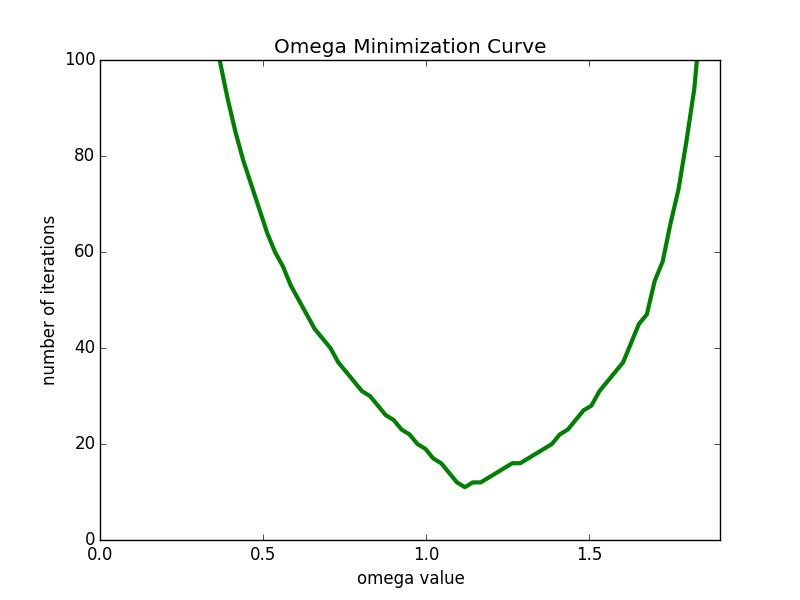
|  |
| --- |
| Code: Ne155\_hw5\_5 |
| import numpy as np  import scipy.linalg  import matplotlib.pyplot as plt  def matrix\_gen(n):  a =[]  i = 0  while i < n - 1:  a.append(-1)  i+=1  #Assigns value to lower diagonal and upper diagonal  b =[]  j = 0  while j < n:  b.append(3)  j+=1  def tridiag(a, b, c, k1=-1, k2=0, k3=1):  return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)  #A function that creates tridiagonal matrices  A = tridiag(a, b, a)  b\_array = np.zeros((n, 1))  num = np.linspace(0, n - 1, n)  for i in num:  b\_array[i][0] = 100  x\_0 = np.zeros((n, 1))  return A, b\_array, x\_0  def rel\_error(x\_k, x\_k1):  return np.linalg.norm(x\_k1 - x\_k) / np.linalg.norm(x\_k1)  ###  A, b\_array, x\_0 = matrix\_gen(5)  D = np.diag(A)  p\_j = np.diagflat(D) - A  def Jacobi\_method(x):  return np.dot(np.linalg.inv(np.diagflat(D)),np.dot(p\_j, x)+b\_array)  e = 1  i = 0  x = x\_0  while e > 10 \*\* (-6):  x\_1 = Jacobi\_method(x)  e = rel\_error(x, x\_1)  x = x\_1  i += 1  print("iterations required for error less than 10E-6 using Jacobi:")  print(i)  print("relative error:")  print(e)  e = 1  i = 0  x = x\_0  while e > 10 \*\* (-8):  x\_1 = Jacobi\_method(x)  e = rel\_error(x, x\_1)  x = x\_1  i += 1  print("iterations required for error less than 10E-8 using Jacobi:")  print(i)  print("relative error:")  print(e)  ###  L = np.tril(A) #np.tril gives D+L  U = A - L #A = L+U  coef\_1 = np.linalg.inv(L)  coef\_2 = np.dot(coef\_1, -U)  addition = np.dot(coef\_1, b\_array)  def Gauss\_Seidel(x):  return np.dot(coef\_2, x) + addition  e = 1  i = 0  x = x\_0  while e > 10 \*\* (-6):  x\_1 = Gauss\_Seidel(x)  e = rel\_error(x, x\_1)  x = x\_1  i += 1  print("iterations required for error less than 10E-6 using Gauss Seidel:")  print(i)  print("relative error:")  print(e)  e = 1  i = 0  x = x\_0  while e > 10 \*\* (-8):  x\_1 = Gauss\_Seidel(x)  e = rel\_error(x, x\_1)  x = x\_1  i += 1  print("iterations required for error less than 10E-8 using Gauss Seidel:")  print(i)  print("relative error:")  print(e)  ###  D = np.diagflat(D)  #The following function will use SOR Iteration  def SOR(x, w=1.15):  L\_prime = L - D  coef1 = np.linalg.inv(D+(w\*L\_prime))  coef2 = np.dot(coef1, (((1-w)\*D) - (w\*U)))  addition = np.dot(coef1, w\*b\_array)  return np.dot(coef2, x) + addition  e = 1  i = 0  x = x\_0  while e > 10 \*\* (-6):  x\_1 = SOR(x)  e = rel\_error(x, x\_1)  x = x\_1  i += 1  print("iterations required for error less than 10E-6 using SOR")  print(i)  print("relative error:")  print(e)  e = 1  i = 0  x = x\_0  while e > 10 \*\* (-8):  x\_1 = SOR(x)  e = rel\_error(x, x\_1)  x = x\_1  i += 1  print("iterations required for error less than 10E-8 using SOR:")  print(i)  print("relative error:")  print(e) |
| Output:  iterations required for error less than 10E-6 using Jacobi:  25  relative error:  7.92128941434e-07  iterations required for error less than 10E-8 using Jacobi:  33  relative error:  9.77935936983e-09  iterations required for error less than 10E-6 using Gauss Seidel:  14  relative error:  9.14040253479e-07  iterations required for error less than 10E-8 using Gauss Seidel:  19  relative error:  3.76149446683e-09  iterations required for error less than 10E-6 using SOR  9  relative error:  8.97373953024e-07  iterations required for error less than 10E-8 using SOR:  13  relative error:  4.44324055073e-10 |

|  |  |  |
| --- | --- | --- |
| **Method** | **Iterations required to meet absolute tolerance** | **Iterations required to meet relative error (10E-6)** |
| Jacobi | 35 | 25 |
| Gauss Seidel | 18 | 14 |
| SOR | 12 | 9 |

The general observable trend is that it takes less iterations to achieve the relative error than it takes to reach the absolute tolerance. SOR appears the be the fastest method in both cases and the quickest process was using SOR to reach the desired relative error; it only took 9 iterations. A tighter convergence tolerance requires more iterations since we are trying to get closer and closer to get to the real solution, hence requiring the extra steps.

Speed of method can be reflected through iterations required to reach absolute tolerance (which explains why it takes half the iterations for the Gauss Seidel than it takes using Jacobi). Absolute tolerance measures the difference between the estimated answer and the actual answer. The relative error, not so much since this is a reflection of convergence between points, hence the iterations for Gauss Seidel isn’t exactly half of Jacobi.

b)



|  |
| --- |
| Code: Ne155\_hw5\_5b |
| import numpy as np  import scipy.linalg  import matplotlib.pyplot as plt  def matrix\_gen(n):  a =[]  i = 0  while i < n - 1:  a.append(-1)  i+=1  #Assigns value to lower diagonal and upper diagonal  b =[]  j = 0  while j < n:  b.append(3)  j+=1  def tridiag(a, b, c, k1=-1, k2=0, k3=1):  return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)  #A function that creates tridiagonal matrices  A = tridiag(a, b, a)  b\_array = np.zeros((n, 1))  num = np.linspace(0, n - 1, n)  for i in num:  b\_array[i][0] = 100  x\_0 = np.zeros((n, 1))  return A, b\_array, x\_0  A, b\_array, x\_0 = matrix\_gen(5)  D = np.diag(A)  p\_j = np.diagflat(D) - A  exact\_sol = np.dot(np.linalg.inv(A), b\_array)  def absolute\_tolerance(x):  return np.linalg.norm(x- exact\_sol)  num = np.linspace(1.15, 2.5, 10)  L = np.tril(A) #np.tril gives D+L  U = A - L #A = L+U  D = np.diagflat(D)  #The following function will use SOR Iteration  def SOR(x, w):  L\_prime = L - D  coef1 = np.linalg.inv(D+(w\*L\_prime))  coef2 = np.dot(coef1, (((1-w)\*D) - (w\*U)))  addition = np.dot(coef1, w\*b\_array)  return np.dot(coef2, x) + addition  num = np.linspace(.1, 2.5, 100)  values = []  for numbers in num:  abs\_tol = 1  i = 0  x = x\_0  #Iterates using SOR until desired absolute tolerance is achieved  while abs\_tol > 10 \*\* (-6):  x = SOR(x, numbers)  abs\_tol = absolute\_tolerance(x)  i += 1  values.append(i)  print(values)  print(min(values))  print(num[values.index(min(values))])  plt.plot(num, values, linewidth=3, color = 'g')  plt.xlim(0, 1.90)  plt.ylim(0, 100)  plt.xlabel("omega value")  plt.ylabel("number of iterations")  plt.title("Omega Minimization Curve")  plt.show() |
| Output:  11  1.11818181818 |

The optimal omega is omega = 1.118 and this produces a total of 11 iterations. The method for finding these values is the first creating a list of values for omega, then creating a plot of the number of iterations produced from a while loop that plugged in the omega values for the SOR method. Once the plot is generated, one can see the minimum point in the plot.