Jacqueline K. Garcia
25070118

March 11, 2017
NucE 155, Slaybaugh

Assignment 5

1. (10 points) For $n = 100$ we will use this tridiagonal system of equations

$$\begin{aligned} 2x_0 - x_1 &= 0 \\ -x_{j-1}+2x_j - x_{j+1} &= j\,, \qquad j = 1, \ldots, n-2 \\ -x_{n-2}+2x_{n-1} &= n-1 \end{aligned}$$

in a few different ways. You may use Python, MATLAB or another package or code language of your choice. Note: `NumPy` and `SciPy` are libraries that can be imported into Python and would be useful for this assignment.

(a) (2.5 points) Use built in Python or MATLAB commands to construct $\mathbf{A}$ and $\vec{b}$.

(b) (2.5 points) What is the condition number of $\mathbf{A}$?

(c) (2.5 points) Solve this problem by explicitly inverting $\mathbf{A}$ and multiplying $\vec{b}$.

(d) (2.5 points) Use `scipy.linalg.solve` (Python) or the backslash operator (MATLAB) to solve the system.

Plot both your explicit (c) and numerical (d) solutions on the same plot. Include axis labels, a title, and a legend.

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)

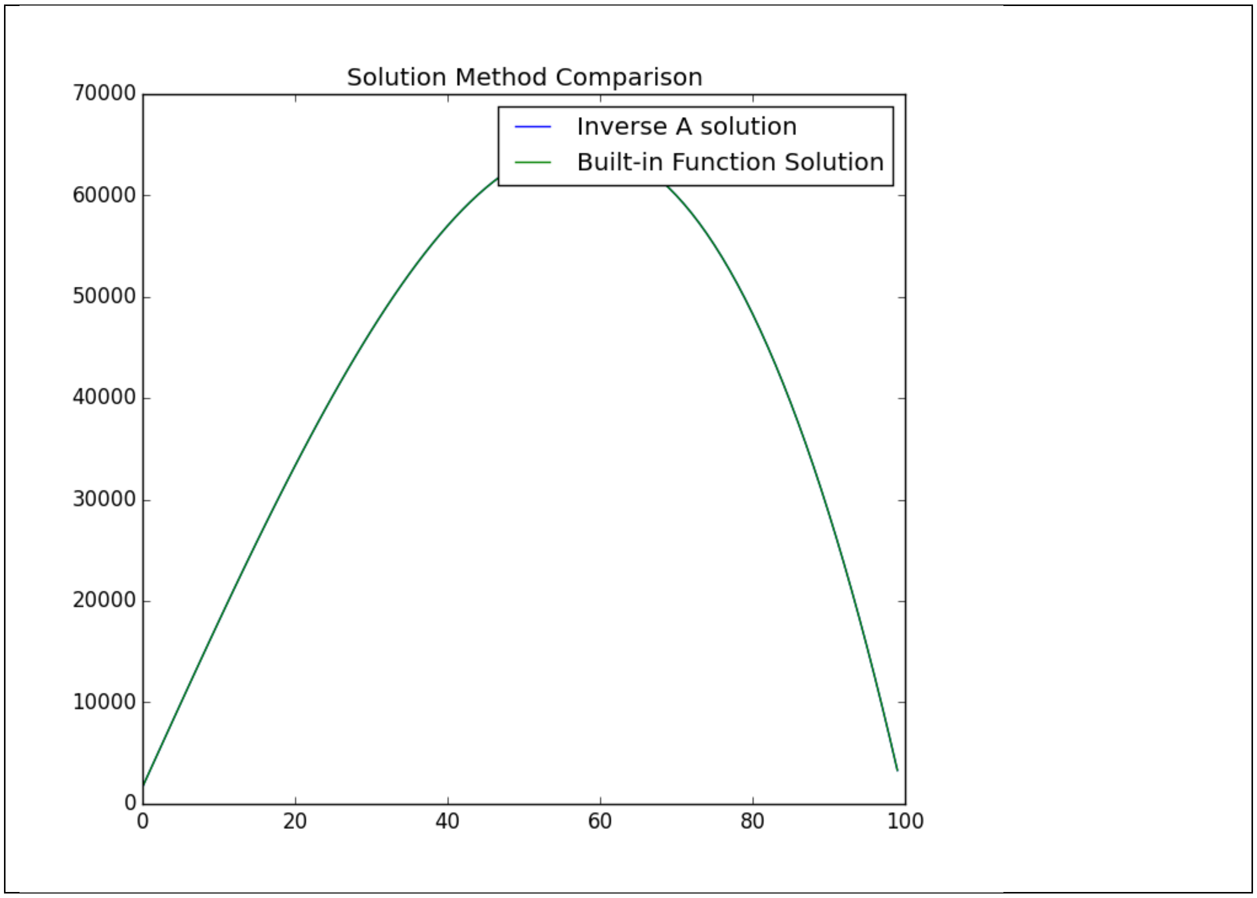
Ne155_hw5_p1

```
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()
```



2. (5 points) Discuss the significance of the spectral radius for the iterative solution of $\mathbf{A}\vec{x} = \vec{b}$, including how it is used to determine convergence and how it is related to rate of convergence.

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

$$Px = (P - A)x + b$$

And an iteration defined by:

$$Px_{k+1} = (P - A)x_k + b$$

The error is then defined by: $e_k = x - x_k \quad \therefore \quad Pe_{k+1} = (P - A)e_k + b$

After combining equations:

$$e_{k+1} = (I - P^{-1}A)e_k, \qquad I - P^{-1}A = M, \qquad then\ e_{k+1} = Me_k$$

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.

3. (10 points) By hand, find the first two iterations (report the vectors $\vec{x}^{(1)}$ and $\vec{x}^{(2)}$) of the

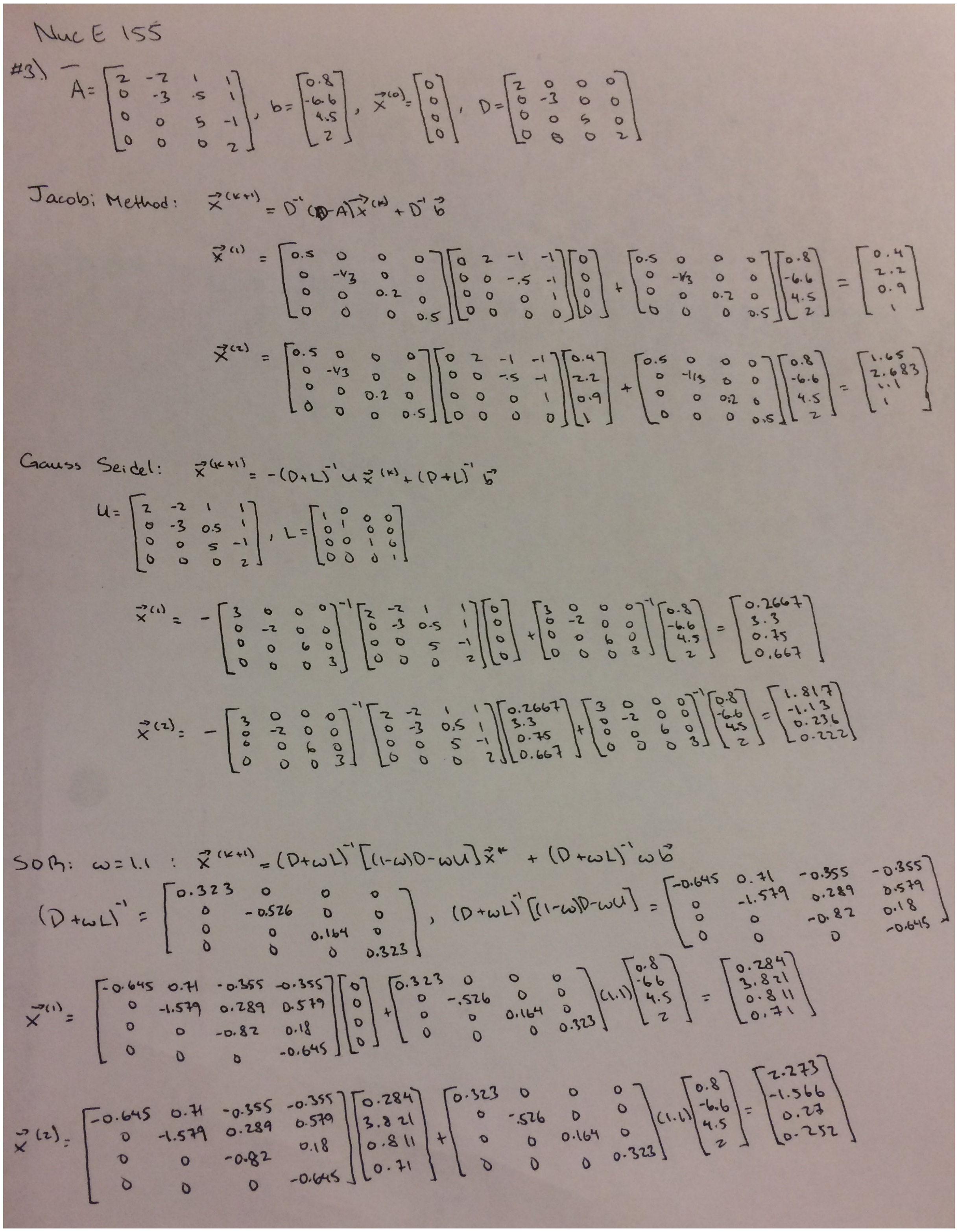
- Jacobi method
- Gauss Seidel method
- SOR (with $\omega = 1.1$) method

for the following system of equations. Use $\vec{x}^{(0)} = \vec{0}$. Briefly compare the results between methods.

$$\text{(b).}\quad \begin{aligned} &2x_1 + -2x_2 + 1x_3 + 1x_4 = 0.8 \\ &0x_1 + -3x_2 + 0.5x_3 + 1x_4 = -6.6 \\ &0x_1 + 0x_2 + 5x_3 + -1x_4 = 4.5 \\ &0x_1 + 0x_2 + 0x_3 + 2x_4 = 2 \end{aligned}$$

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.

Jacqueline K. Garcia	March 11, 2017
25070118	NucE 155, Slaybaugh



NucE 155

#3)

$$\bar{A}=\begin{bmatrix}2&-2&1&1\\0&-3&.5&1\\0&0&5&-1\\0&0&0&2\end{bmatrix},\ b=\begin{bmatrix}0.8\\-6.6\\4.5\\2\end{bmatrix},\ \vec{x}^{(0)}=\begin{bmatrix}0\\0\\0\\0\end{bmatrix},\ D=\begin{bmatrix}2&0&0&0\\0&-3&0&0\\0&0&5&0\\0&0&0&2\end{bmatrix}$$

Jacobi Method: $\vec{x}^{(k+1)}=D^{-1}(D-A)\vec{x}^{(k)}+D^{-1}\vec{b}$

$$\vec{x}^{(1)}=\begin{bmatrix}0.5&0&0&0\\0&-1/3&0&0\\0&0&0.2&0\\0&0&0&0.5\end{bmatrix}\begin{bmatrix}0&2&-1&-1\\0&0&-.5&-1\\0&0&0&1\\0&0&0&0\end{bmatrix}\begin{bmatrix}0\\0\\0\\0\end{bmatrix}+\begin{bmatrix}0.5&0&0&0\\0&-1/3&0&0\\0&0&0.2&0\\0&0&0&0.5\end{bmatrix}\begin{bmatrix}0.8\\-6.6\\4.5\\2\end{bmatrix}=\begin{bmatrix}0.4\\2.2\\0.9\\1\end{bmatrix}$$

$$\vec{x}^{(2)}=\begin{bmatrix}0.5&0&0&0\\0&-1/3&0&0\\0&0&0.2&0\\0&0&0&0.5\end{bmatrix}\begin{bmatrix}0&2&-1&-1\\0&0&-.5&-1\\0&0&0&1\\0&0&0&0\end{bmatrix}\begin{bmatrix}0.4\\2.2\\0.9\\1\end{bmatrix}+\begin{bmatrix}0.5&0&0&0\\0&-1/3&0&0\\0&0&0.2&0\\0&0&0&0.5\end{bmatrix}\begin{bmatrix}0.8\\-6.6\\4.5\\2\end{bmatrix}=\begin{bmatrix}1.65\\2.683\\1.1\\1\end{bmatrix}$$

Gauss Seidel: $\vec{x}^{(k+1)}=-(D+L)^{-1}U\vec{x}^{(k)}+(D+L)^{-1}\vec{b}$

$$U=\begin{bmatrix}2&-2&1&1\\0&-3&0.5&1\\0&0&5&-1\\0&0&0&2\end{bmatrix},\ L=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

$$\vec{x}^{(1)}=-\begin{bmatrix}3&0&0&0\\0&-2&0&0\\0&0&6&0\\0&0&0&3\end{bmatrix}^{-1}\begin{bmatrix}2&-2&1&1\\0&-3&0.5&1\\0&0&5&-1\\0&0&0&2\end{bmatrix}\begin{bmatrix}0\\0\\0\\0\end{bmatrix}+\begin{bmatrix}3&0&0&0\\0&-2&0&0\\0&0&6&0\\0&0&0&3\end{bmatrix}^{-1}\begin{bmatrix}0.8\\-6.6\\4.5\\2\end{bmatrix}=\begin{bmatrix}0.2667\\3.3\\0.75\\0.667\end{bmatrix}$$

$$\vec{x}^{(2)}=-\begin{bmatrix}3&0&0&0\\0&-2&0&0\\0&0&6&0\\0&0&0&3\end{bmatrix}^{-1}\begin{bmatrix}2&-2&1&1\\0&-3&0.5&1\\0&0&5&-1\\0&0&0&2\end{bmatrix}\begin{bmatrix}0.2667\\3.3\\0.75\\0.667\end{bmatrix}+\begin{bmatrix}3&0&0&0\\0&-2&0&0\\0&0&6&0\\0&0&0&3\end{bmatrix}^{-1}\begin{bmatrix}0.8\\-6.6\\4.5\\2\end{bmatrix}=\begin{bmatrix}1.817\\-1.13\\0.236\\0.222\end{bmatrix}$$

SOR: $\omega=1.1$: $\vec{x}^{(k+1)}=(D+\omega L)^{-1}[(1-\omega)D-\omega U]\vec{x}^{k}+(D+\omega L)^{-1}\omega\vec{b}$

$$(D+\omega L)^{-1}=\begin{bmatrix}0.323&0&0&0\\0&-0.526&0&0\\0&0&0.164&0\\0&0&0&0.323\end{bmatrix},\ (D+\omega L)^{-1}[(1-\omega)D-\omega U]=\begin{bmatrix}-0.645&0.71&-0.355&-0.355\\0&-1.579&0.289&0.579\\0&0&-0.82&0.18\\0&0&0&-0.645\end{bmatrix}$$

$$\vec{x}^{(1)}=\begin{bmatrix}-0.645&0.71&-0.355&-0.355\\0&-1.579&0.289&0.579\\0&0&-0.82&0.18\\0&0&0&-0.645\end{bmatrix}\begin{bmatrix}0\\0\\0\\0\end{bmatrix}+\begin{bmatrix}0.323&0&0&0\\0&-.526&0&0\\0&0&0.164&0\\0&0&0&0.323\end{bmatrix}(1.1)\begin{bmatrix}0.8\\-66\\4.5\\2\end{bmatrix}=\begin{bmatrix}0.284\\3.821\\0.811\\0.71\end{bmatrix}$$

$$\vec{x}^{(2)}=\begin{bmatrix}-0.645&0.71&-0.355&-0.355\\0&-1.579&0.289&0.579\\0&0&-0.82&0.18\\0&0&0&-0.645\end{bmatrix}\begin{bmatrix}0.284\\3.821\\0.811\\0.71\end{bmatrix}+\begin{bmatrix}0.323&0&0&0\\0&-.526&0&0\\0&0&0.164&0\\0&0&0&0.323\end{bmatrix}(1.1)\begin{bmatrix}0.8\\-6.6\\4.5\\2\end{bmatrix}=\begin{bmatrix}2.273\\-1.566\\0.27\\0.252\end{bmatrix}$$

Jacqueline K. Garcia March 11, 2017
25070118 NucE 155, Slaybaugh

4. (30 points) We will use the following system of n equations:

$$\mathbf{A}\vec{x} \equiv \begin{pmatrix} 3 & -1 & 0 & \cdots & 0 \\ -1 & 3 & -1 & \ddots & \vdots \\ 0 & -1 & 3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} 100 \\ 100 \\ 100 \\ \vdots \\ 100 \end{pmatrix} \equiv \vec{b}$$

Write a program to implement the

(a) Jacobi method

(b) Gauss Seidel method

(c) SOR method

for a matrix with n unknowns. Turn in your source code electronically; include instructions for how to run it, input files, etc. if necessary.

Solve the above system of equations with each program.
Use $\omega = 1.15$ for SOR; use $\vec{x}^{(0)} = \vec{0}$ and $n = 5$.
Print the solution vector from each method converged to an **absolute** tolerance of 10^{-6}.
Indicate the number of iterations required to meet this tolerance for each method.

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.

5. (20 points) Use the programs you just wrote with the same matrix and using the same settings to answer the following.

   (a) (10 points) How many iterations are required for each method to reach the stopping criterion (*relative error*):

$$\frac{||x^{(k+1)} - x^{(k)}||}{||x^{(k+1)}||} < \epsilon$$

   for $\epsilon = 10^{-6}$ and $\epsilon = 10^{-8}$?
   Also:

   - For each method, how does the absolute error (from the previous question) with $\epsilon = 10^{-6}$ compare to the relative error?
   - Which method required the fewest iterations?
   - What do you observe about reaching a tighter convergence tolerance?

   (b) (10 points) Perform an experiment to determine ω_{opt} for SOR. Explain your procedure and include the results.

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:
```

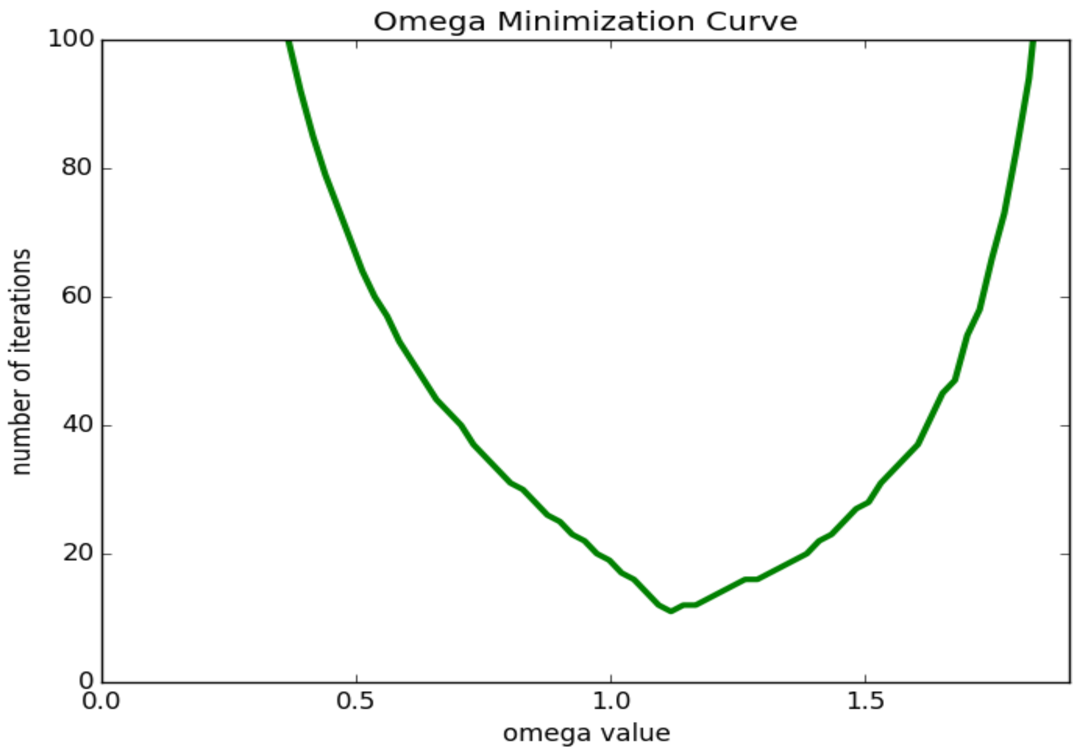
```
9.77935936983e-09
iterations required for error less than 10E-6 using Gauss Seidel:
14
relative error:
9.14040253479e-07
iterations required for error less than 10E-8 using Gauss Seidel:
19
relative error:
3.76149446683e-09
iterations required for error less than 10E-6 using SOR
9
relative error:
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.