Jacobi

March 7, 2023

Peter Chu, all code written between 8:05 PM March 6th to 1:20 AM March 7th

0.1 Jacobi Method

[1]:
```
# import necessary libraries
import math as math
import numpy as np
```

0.1.1 Part a

[2]:
```
# Input A = matrix, b = sol vector, x = guess vector, tol = tolerance, n = max␣
↪iterations, start writing code 8:05

def jacobi(A, b, x, tol, n):

    # Create necessary variables for formula
    # D = diag matrix
    # D_1 = inverse of D
    # U = strictly upper matrix * -1 to make entries negative
    # L = strictly lower matrix * -1 to make entries negative
    # T = D^-1 * (L + U)
    # count is a ticker variable to keep track of number of iterations

    D = np.diag(np.diag(A))
    D_1 = np.linalg.inv(D)
    U = (np.triu(ls1) - D).dot(-1)
    L = (np.tril(ls1) - D).dot(-1)
    T = D_1.dot(L + U)
    count = 0

    # Do first iteration
    x_k = x
    x_k1 = (T.dot(x_k.T)) + (D_1.dot(b.T))

    count += 1

    # If x_0 = 0, ||X_0||_inf = 0 we will have a divide by zero error.
    # Iterate one more time to avoid this and then enter loop
```

```
    if(x_k.all() == 0):
        x_k = x_k1.T
        x_k1 = (T.dot(x_k.T)) + (D_1.dot(b.T))

        count += 1

    # Create stop value for exit condition
    StopVal = np.linalg.norm(abs(x_k1 - x_k), np.inf) / np.linalg.norm(x_k, np.
↪inf)

    # Loop until we hit exit condition or we do n interations
    while(StopVal > tol and count < n):
        #Do jacobi method
        x_k = x_k1.T
        x_k1 = (T.dot(x_k.T)) + (D_1.dot(b.T))

        #Update stop value and add 1 to count
        StopVal = np.linalg.norm(abs(x_k1 - x_k), np.inf) / np.linalg.norm(x_k,␣
↪np.inf)
        count += 1

    # Expect x^k vector and the number of iterations it took

    return x_k1, 'iterations:', count
```

0.1.2 Part b

[3]:
```
# Use np.newaxis on vectors to make them transposable correctly
# Python does not know how to transpose 1 dimensional arrays, need to make 2␣
↪dimensional to do so
# ex. example = np.array([1,1,1])
# example.T = np.array([1,1,1]) != np.array([1],[1],[1]) <- desired transposed␣
↪vector

# linear system 1
ls1 = np.array([[1,2,-2], [1,1,1], [2,2,1]])
b1 = np.array([7,2,5])[np.newaxis]

# Linear system 2

ls2 = np.array([[2,-1,1], [2,2,2], [-1,-1,2]])
b2 = np.array([-1,4,-5])[np.newaxis]

# initial guess
x_0 = np.array([0,0,0])[np.newaxis]
```

```
# tolerance
tolVal = 0.00005
```

[4]:
```
# Run Jacobi function using defined variables

aprox1 = jacobi(ls1, b1, x_0, tolVal, 25)

aprox2 = jacobi(ls2, b2, x_0, tolVal, 25)

print(aprox1, '\n', aprox2)
```

```
(array([[ 1.],
       [ 2.],
       [-1.]]), 'iterations:', 25)
 (array([[-8262.5],
       [ 7400. ],
       [ 6387.5]]), 'iterations:', 25)
```

0.1.3 Part c

[5]:
```
# Calculate spectral radius for ls1 and store value as lambda1

D = np.diag(np.diag(ls1))
U = (np.triu(ls1) - D).dot(-1)
L = (np.tril(ls1) - D).dot(-1)
D_1 = np.linalg.inv(D)

T = D_1.dot(L + U)

lambda1 = max(abs(np.linalg.eig(T)[0]))

# Calculate spectral radius for ls2 and store value as lambda2

D = np.diag(np.diag(ls2))
U = (np.triu(ls2) - D).dot(-1)
L = (np.tril(ls2) - D).dot(-1)
D_1 = np.linalg.inv(D)

T = D_1.dot(L + U)

lambda2 = max(abs(np.linalg.eig(T)[0]))

print('Spectral radius of linear system 1:', lambda1, '\n')
print('Spectral radius of linear system 2:', lambda2)
```

```
Spectral radius of linear system 1: 1.0813325779705841e-05

Spectral radius of linear system 2: 1.1180339887498956
```

The Jacobi method gave a good approximation for linear system 1. Looking at $\rho(T_j)$ for linear system 1, we have $\rho(T_g) < 1$. Thus T_j is convergent and converges to certain values. As a result we get a good approximation because x^k converges as well. For linear system 2, we have that $\rho(T_j) > 1$ which results in T_j being divergent and thus x^k does not converge to some value. This results in a bad approximation.

0.2 Gauss-Seidel Method

0.2.1 Part a

[6]:
```
# Input A = matrix, b = sol vector, x = guess vector, tol = tolerance, n = max␣
↪iterations

def GaussSeidel(A, b, x, tol, n):

    # Create necessary variables for formula
    # D = diag matrix
    # U = strictly upper matrix * -1 to make entries negative
    # L = strictly lower matrix * -1 to make entries negative
    # DL_1 = (D - L)^-1, T = (D - l)^-1 * U
    # count is a ticker variable to keep track of number of iterations

    D = np.diag(np.diag(A))
    U = (np.triu(A) - D).dot(-1)
    L = (np.tril(A) - D).dot(-1)
    DL_1 = np.linalg.inv(D - L)
    T = DL_1.dot(U)
    count = 0

    # Do first iteration

    x_k = x
    x_k1 = T.dot(x_k.T) + DL_1.dot(b.T)

    # Update number of iterations
    count += 1

    #If x_0 = 0, ||X_0||_inf = 0 we will have a divide by zero error.
    #Iterate one more time to avoid this and then enter loop

    if(x_k.all() == 0):
        x_k = x_k1.T
        x_k1 = T.dot(x_k.T) + DL_1.dot(b.T)
        count += 1

    # Create stop value for exit condition
    StopVal = np.linalg.norm(abs(x_k1 - x_k), np.inf) / np.linalg.norm(x_k, np.
↪inf)
```

```
    # Loop until we hit exit condition or we do n interations
    while(StopVal > tol and count < n):
        #Do Gauss-Seidel method
        x_k = x_k1.T
        x_k1 = T.dot(x_k.T) + DL_1.dot(b.T)

        #Update stop value and add 1 to count
        StopVal = np.linalg.norm(abs(x_k1 - x_k), np.inf) / np.linalg.norm(x_k,␣
↪np.inf)
        count += 1

    # Expect x^k vector and the number of iterations it took

    return x_k1, 'iterations:',  count
```

0.2.2 Part b

[7]:
```
# Use np.newaxis on vectors to make them transposable correctly
# Python does not know how to transpose 1 dimensional arrays, need to make 2␣
↪dimensional to do so
# ex. example = np.array([1,1,1])
# example.T = np.array([1,1,1]) != np.array([1],[1],[1]) <- desired transposed␣
↪vector

# linear system 1

ls1 = np.array([[1,2,-2], [1,1,1], [2,2,1]])
b1 = np.array([7,2,5])[np.newaxis]

# Linear system 2

ls2 = np.array([[2,-1,1], [2,2,2], [-1,-1,2]])
b2 = np.array([-1,4,-5])[np.newaxis]

# initial guess
x_0 = np.array([0,0,0])[np.newaxis]

# tolerance
tolVal = 0.00005
```

[8]:
```
# Run Gauss Seidel using defined variables

aprox1 = GaussSeidel(ls1, b1, x_0, tolVal, 25)
```

```
aprox2 = GaussSeidel(ls2, b2, x_0, tolVal, 25)

print(aprox1, '\n', aprox2)
```

```
(array([[ 1.30862285e+09],
       [-1.32540006e+09],
       [ 3.35544310e+07]]), 'iterations:', 25)
 (array([[ 1.00000063],
       [ 1.99999931],
       [-1.00000003]]), 'iterations:', 25)
```

0.2.3 Part c

[9]:
```
# Calculate spectral radius for ls1 and store value as lambda1, finish code 1:20

D = np.diag(np.diag(ls1))
U = (np.triu(ls1) - D).dot(-1)
L = (np.tril(ls1) - D).dot(-1)
DL_1 = np.linalg.inv(D - L)
T = DL_1.dot(U)

lambda1 = max(abs(np.linalg.eig(T)[0]))

# Calculate spectral radius for ls2 and store value as lambda2

D = np.diag(np.diag(ls2))
U = (np.triu(ls2) - D).dot(-1)
L = (np.tril(ls2) - D).dot(-1)
DL_1 = np.linalg.inv(D - L)
T = DL_1.dot(U)

lambda2 = max(abs(np.linalg.eig(T)[0]))

print('Spectral radius of linear system 1:', lambda1, '\n')
print('Spectral radius of linear system 2:', lambda2)
```

```
Spectral radius of linear system 1: 2.0

Spectral radius of linear system 2: 0.5
```

The Gauss-Seidel method gave a good approximation for linear system 2. Looking at $\rho(T_g)$ for linear system 2, we have $\rho(T_g) = 0.5 < 1$. Thus T_g is convergent and converges to certain values. As a result we get a good approximation because x^k converges as well. For linear system 1, we have that $\rho(T_j) = 2 > 1$ which results in T_j being divergent and thus x^k does not converge to some value. This results in a bad approximation.