Jacobi Iteration Method

Mohamed Ali Ashfaq Ahamed[†]

[†] *Department of Mathematics and Statistics, Memorial University of Newfoundland, St. John's (NL) A1C 5S7, Canada*

E-mails: aamohamedali@mun.ca

Abstract
A class of issues are solved using the Jacobi method for solving a system of linear equations. With special attention paid to the norms, a thorough investigation of the relationship between theoretical convergence results, and empirical monotonic & non-monotonic convergence is made.

1 Introduction

1.1 Background

Computer scientists find that a high-dimensional system of equations requires a lot of algorithms. When we have few equations, it is easy to find the exact solutions to the linear system. As the linear system grows, mathematicians can only find approximate solutions. That's when the scientist came up with iterative methods to find the estimated solution. Iterative techniques are methods used to convert nonlinear algebraic equations to linear algebraic equations. It consists of mathematical simulations that generate rough approximation. These techniques are mostly performed by computers. There are two types of iterative methods stationary iterative methods and non-stationary iterative methods. Stationary iterative methods are simple and easy to implement compared to non-stationary methods. The non-stationary method is developed recently based on the idea of the sequence of orthogonal vectors. Some of the well-known stationary iterative methods are Jacobi method, gauss-seidel method, successive overrelaxation method & symmetry successive overrelaxation method and the non-stationary iterative methods are conjugate gradient method, minres SYMMLQ, generalized minimal residual, biconjugate gradient, quasi-minimal residual, conjugate gradient squared method, biconjugate gradient stabilized & Chebyshev iteration.

1.2 Context

The Jacobi iteration algorthm is devloped in 1804. At that time it was very efficient algorithm. The method is named after Carl Gustav Jacob Jacobi. The method is widely used in numerical algebra to determine the solution that is diagonally dominant. In this

research, we focus on the number of iterations it takes to converge for different dimensions of A matrix .

1.3 Outline

The matrix A that we are doing the research is an NxN tridiagonal matrix where each value on the main diagonal are $-2N^2$ and the sub diagonals are N^2.

$$A = \begin{bmatrix} -2N^2 & N^2 & 0 & \dots & 0 \\ N^2 & -2N^2 & N^2 & \ddots & 0 \\ 0 & N^2 & -2N^2 & N^2 & 0 \\ 0 & \dots & \dots & \ddots & N^2 \\ 0 & \dots & \dots & N^2 & -2N^2 \end{bmatrix}$$

Figure 1: The general form of matrix A. For each N values the matrix builds NxN dimensional matrix

b is a column vector with ith element defined by the equation.

$$b_i = 2\sigma(2\sigma x_i^2 - 1)e^{-\sigma x_i^2}$$

Where $x_i = (i+1)/(N+1) - 1/2$ for $i = 0, 1, 2.....N-1$.
The results produced in the research is by considering $\sigma = 500$

1.4 Hypothesis

Using the Jacobi iteration technique solving the linear system Ax=B may take many iterations and produce an approximate solution vector that is very close to the actual solution vector.

2 Methods

we used jacobi iteration technique when solving the linear system $Ax = B$. We split the matrix A as M and N, where M and N are two matrices.

$$A = M + N;$$

Ax can be written as

$$Ax = Mx + Nx = B;$$
$$Mx = B - Nx;$$
$$M^{-1}Mx = M^{-1}(B - Nx);$$
$$x = M^{-1}(B - Nx).$$

If we take a random choice x_k (k as an index) then, the formulae generates a new choice x_{k+1} such that

$$x_{k+1} = M^{-1}(B - Nx_K);$$

If the process keeps on iterating. We hope that we may reach an approximate solution that is very close to the exact solution.
The formula can further be modified such that

$$x_{k+1} = M^{-1}(B - Nx_k);$$

$$x_{k+1} = M^{-1}(B - Mx_k - Nx_k + Mx_k);$$

$$x_{k+1} = M^{-1}(B - Ax_k + Mx_k);$$

$$x_{k+1} = M^{-1}(B - Ax_k) + M^{-1}Mx_k;$$

$$x_{k+1} = x_k + M^{-1}(B - Ax_k). \tag{1}$$

To use the above-modified form we only need to know M, B and A Matrices. When splitting the matrix A, we make sure that M is taken as a diagonal matrix. So that it would be easier to find the M^{-1} that is the reciprocal of the diagonal elements.
By plugging in the initial assumption vector x_k and the diagonal matrix M. The next vector x_{k+1} is generated which is closer to the exact solution. Now by plugging in the new vector as x_k, the formula returns the newest vector which is closer to the exact solution than the previous one. The iteration can be modified by the user as they wish. The more iteration it goes, the more exact value it gets. The user can stop the iteration by defining a residual error which will be comparing the previous solution vector and the new solution vector and if the difference is below the residual error then the iteration stops.

3 Results

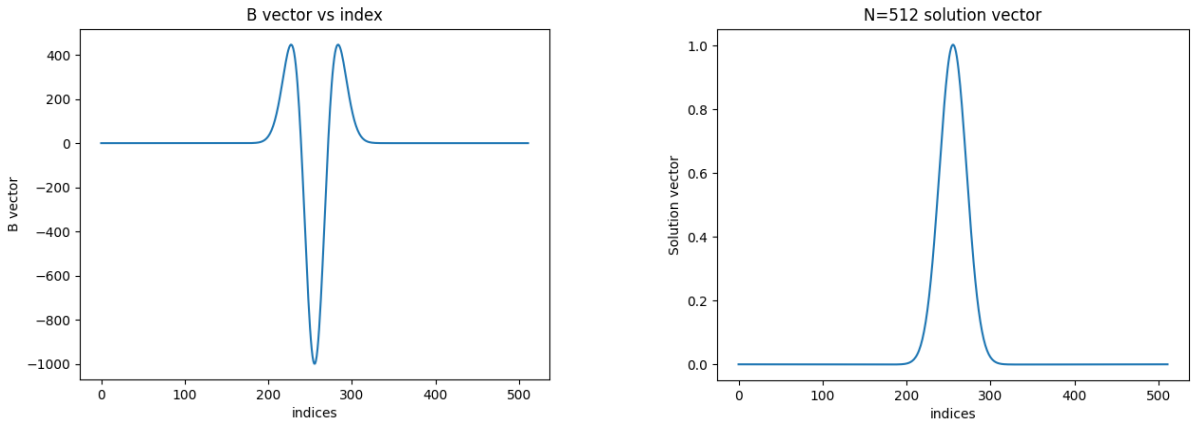

Figure 2: The graphs are drawn between the values of b vector & its indices and between the solution vector its indices for N=512

While comparing these two graphs, the b vector is almost the mirror image of the solution vector. If we closely look at the solution vector graph, the rows of the solution vectors are zero until the 200th row and beyond 300th row. But between the 200th row and 300th row, there is a symmetrical increase and decrease.

N	Number of iterations
32	913
64	3839
128	16281
256	69281
512	294315

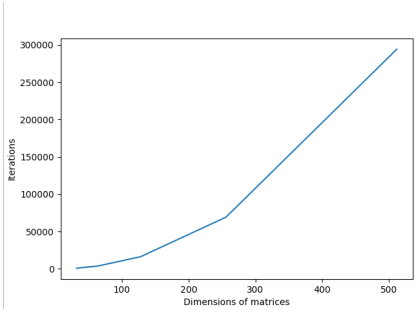

Figure 3: This graph illustrates the number of iteration it reaches when the dimension of A matrix increases

As we can see, the number of iterations increases irrationally as the dimension of A matrix changes and also when increasing the σ value, the number of iterations reduces gradually.

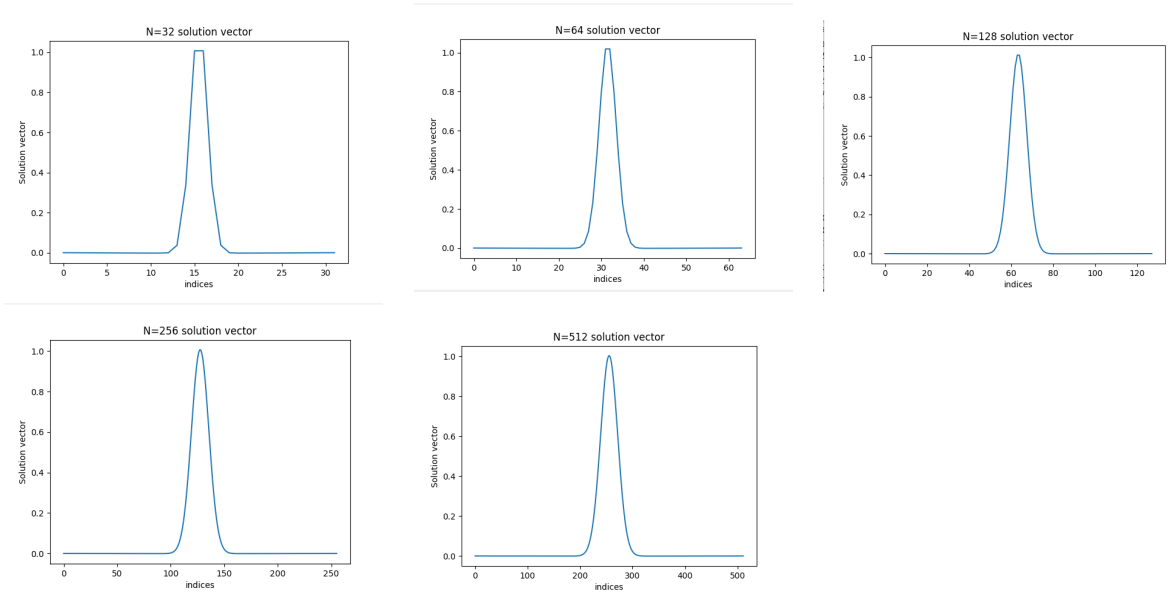

Figure 4: The graph is plotted between the values of solution vector and corresponding index for different N values

For different n-dimension matrices, the graphs are always symmetric. As the dimension increases the graphs get steeper. The maximum value in the solution vector is always one, no matter how large the dimension gets.

4 Analysis

The usual method to solve the linear system is determine the product of the matrices A^{-1} and B. Though the steps seem simpler, mathematicians find this method inefficient when determining the A^{-1} where the dimension of A is larger. That is when iterative techniques come into play. We only need the diagonal matrix M which is a split from matrix A and any initial assumption for the solution vector. In most cases, we assume the initial solution as a zero vector. With this matrix and vector, the Jacobi formula iterates until we find a vector that is closer to the exact solution. But this iteration may take a longer time than we expected. In finding a closer solution to matrix A and vector B, N=512 took more than an hour to stop the iteration. So in the real world, the matrix might be billions x billions of dimensions, which may take days to find a solution. This technique can only be applied to a diagonally dominant matrix. But one thing mathematicians are able to guarantee is that it gives a more accurate value. There are more efficient iterative techniques compared to Jacobi. My future research might be the other efficient techniques. When I ran the same code again, the iteration moved by 1 unit for every N value, that is one of the things that we are concerned about.

5 Appendix 1

The diagonal of a matrix is always greater than the sum of the other remaining elements in the row is called as a diagonally dominant matrix.

6 Appendix 2

```
#Ashfaq Ahamed Mohamed Ali
#determining appriximate solution using jacobi iteration technique
import numpy as np
import math
import matplotlib.pyplot as plt


#building a tri-diagnal matrix
def tridiagonal(size):
    diagonal=[-2*size**2 for i in range(size)]
    diagonal_cons=[size**2 for i in range(size-1)]

    A = [[0 for j in range(size)]
              for i in range(size)]

    for k in range(size-1):
        A[k][k] = diagonal[k]
        A[k][k+1] = diagonal_cons[k]
        A[k+1][k] = diagonal_cons[k]

    A[size-1][size - 1] = diagonal[size-1]
    return np.array(A)
```

```
#building the b matrix
def Bmatrix(size):
    alp=500
    B=[]
    for i in range(size):
        x=(i+1)/(size+1)-1/2
        b=[2*alp*(2*alp*(x**2)-1)*math.exp(-1*alp*(x**2))]
        B.append(b)
    return np.array(B)


# defining a function for Jacobi with matrix A and B as parameter
def myJacobiIteration(A,b):
    # get the size of A
    n = A.shape[0]
    M = np.diag(A)*np.eye(n,n)
    x = np.zeros_like(b)
    count=0
    y=0
    while y==0:
        count+=1
        x = x + np.linalg.inv(M)@(b-A@x)
        if np.linalg.norm(b-A@x)<0.1:
            break
    print(count)
    return np.array(x)

#the indices of the vector that is to be plotted
def xaxis(size):
    xvalues=np.arange(size)
    return xvalues

#calling the matrix A,B and the Solution Vectors for each N value
a1=tridiagonal(32)
b1=Bmatrix(32)
solnx1 = myJacobiIteration(a1,b1)
x1=xaxis(32)

a2=tridiagonal(64)
b2=Bmatrix(64)
solnx2 = myJacobiIteration(a2,b2)
x2=xaxis(64)

a3=tridiagonal(128)
b3=Bmatrix(128)
solnx3= myJacobiIteration(a3,b3)
x3=xaxis(128)
```

```
a4=tridiagonal(256)
b4=Bmatrix(256)
solnx4 = myJacobiIteration(a4,b4)
x4=xaxis(256)

a5=tridiagonal(512)
b5=Bmatrix(512)
solnx5 = myJacobiIteration(a5,b5)
x5=xaxis(512)


#Drawing the graphs for each N value
#when genarating the graphs from the code i plotted only one graph for each value
#and then took screenshot to have it in the article.
plt.figure(figsize=(10,10))

#b vector for n=512
plt.subplot(2,4,1)
plt.plot(x5,b5)
plt.xlabel('indices')
plt.ylabel('B vector')
plt.title('N=512 B vector')

#solution vector for n=32
plt.subplot(2,4,2)
plt.plot(x5,solnx5)
plt.xlabel('indices')
plt.ylabel('solution vector')
plt.title('N=512 solution vector')

#solution vector for n=512
plt.subplot(2,4,3)
plt.plot(x1,solnx1)
plt.xlabel('indices')
plt.ylabel('solution vector')
plt.title('N=32 solution vector')

#solution vector for n=64
plt.subplot(2,4,4)
plt.plot(x2,solnx2)
plt.xlabel('indices')
plt.ylabel('solution vector')
plt.title('N=64 solution vector')

#solution vector for n=128
plt.subplot(2,4,5)
plt.plot(x3,solnx3)
```

```
plt.xlabel('indices')
plt.ylabel('solution vector')
plt.title('N=128 solution vector')


#solution vector for n=256
plt.subplot(2,4,6)
plt.plot(x4,solnx4)
plt.xlabel('indices')
plt.ylabel('solution vector')
plt.title('N=256 solution vector')

#solution vector for n=512
plt.subplot(2,4,7)
plt.plot(x5,solnx5)
plt.xlabel('indices')
plt.ylabel('solution vector')
plt.title('N=512 solution vector')


plt.show()
```