Project II - Solution of Linear Equations

Lianna Santiago-Conty

Alonso Gutierrez

Angelo Scaria

Nicole Stanfill

Nicholas Riley

Andrew Roig

University of Central Florida
12/07/2020

2. Project II - Solution of linear equations. Consider the system of equations $Ax = b$, where A is a $N \times N$ matrix and b is a vector in $\mathbb{R}^n$.

2.1. Your task.

- Implement Gaussian Elimination with partial pivoting
- Implement the Jacobi iterative method to solve systems of equations (Section 8.4).
- Implement the Gauss-Seidel iterative method to solve systems of equations (Section 8.4).
- Implement the SOR iterative method to solve systems of equations (Section 8.4).
- Implement the Conjugate Gradient Method to solve system of linear equations (Section 8.4).
- Setting a fixed accuracy, use all methods to solve the system of equation $Ax = b$. Present the number of iterations necessary for each method. For the SOR method, use different values for the relaxation factor w. Use the system of equations on Exercise 3 of Section 8.4. Compare the results and justify the behavior.
- Repeat the previous item using the matrices in the computer exercise 10 on Section 8.4 of the book. Compare the results and justify the behavior.

Table of Contents

1. Introduction

Numerical analysis creates, analyzes, and implements algorithms to be able to solve problems of continuous math originating from real world applications of algebra, geometry and calculus that involve variables that vary continuously [2]. This paper focuses on utilizing iterative methods on problems that use a system of linear equations.

When one is given a system of linear equations with a single solution described as $Ax = b$ where A represents a known real matrix with $n \times n$ size, and b represents a known vector with size $n \times 1$, one must find the optimal solution for x, which represents the vector of unknowns with size $n \times 1$. There are multiple different ways to approach this problem, and this paper will focus specifically on five different methods to solve the problem. It is later showcased that one can use either one of these five methods to solve a problem given a system of linear equations and the output result should be the same amongst each method.

The five methods are Gaussian Elimination with partial pivoting, Jacobi iterative Method, Gauss-Seidel iterative method, Successive Overrelaxation (SOR) method, and Conjugate Gradient method. Although Gaussian Elimination with partial pivoting is not an iterative method, the solutions should give either a fairly similar result, or a more accurate one. Experiments have been conducted considering the system of equations Ax = b, where A is a N x N matrix and b is a vector in $\mathbb{R}^n$. The results of the experiments will be compared showcasing which method is most efficient to use given a scenario.

2. Methods

2.1 Gaussian Elimination Method

2.1.1 Gaussian Elimination Method Math

Gaussian elimination, also known as row reduction, is the method of solving a system of linear equations by performing a succession of elementary row reduction operations on the

system until the left hand side is in reduced row echelon form. This method is commonly taught to students in college linear algebra courses as a method for manually solving systems of linear equations. When used manually, the order and nature of the row reduction operations is determined by heuristic.

For a computer to use Gaussian elimination, a way of determining which row reductions operations to use must be decided. One such method is "Naive" Gaussian elimination. In this case, for a nxn matrix, first n forward elimination operations are performed. In this operation, A_{nn} is selected as a pivot, and then for each row below n, the row is multiplied by a scaling factor and subtracted from that row, such that all values below A_{nn} in its column are 0.

At this point, the matrix is then in row echelon form. Now, a backwards solve operation is performed. This step is essentially the same as the forward elimination, except starting with the bottom row instead of the top, and producing 0s above A_{nn} instead of below. After this step, each row can be scaled by a factor, so that the left hand side is the identity matrix (for a square input). The right hand side has now been transformed into **x**. (if our initial system was A**x** = **b**, x unknown).

This process can fail on some matrices. For instance, if a value on the diagonal is zero, this process will result in a divide by zero. To resolve this, we can use pivoting, where we swap rows and columns of the matrix before the forward pass. If we only swap rows and not columns, this is referred to as partial pivoting.

In partial pivoting, we order rows such that A_{nn} is greater than or equal to any value in its column or below it. The process of Gaussian elimination is then performed on this transformed matrix. This has the problem of changing the order of the values in the output vector. We might keep a permutation vector in order to keep track of which row in the solution corresponds to which row in the original matrix. Alternatively, we can avoid swapping rows in the matrix by using our permutation vector to keep track of what order to modify rows.

Partial pivoting can produce incorrect results, especially for systems where coefficients vary by many orders of magnitude. For instance, the equation
Produces x = 1.0001, y = 0.99990 when solved with partial pivoting. This produces 0.0001x + y = 1.00000001 =/= 1, which almost solves the system, but not exactly.

To remedy this, we perform scaled partial pivoting. Before performing the pivoting we scale each row by dividing it by the absolute value of the element of the row with the greatest absolute value.

2.1.2 Gaussian Elimination Method Code

$$\begin{cases} x + y = 2 \\ 0.0001x + y = 1 \end{cases}$$

```
def gauss(n, A, b):
    """Solve Ax=b using gaussian elimination with scaled partial pivoting

    A, b, must be numpy arrays
    """
    x = np.zeros(n)
    s = np.zeros(n)
    l = [i for i in range(n)]
    n = n-1
    A = A.copy()
    b = b.copy()

    for i in range(0, n+1):
        l[i] = i
        for j in range(0, n+1):
            s[i] = max(s[i], np.abs(A[i,j]))
    for k in range(0, n):
        r_max = 0
        for i in range(k, n+1):
            r = np.abs(A[l[i],k] / s[l[i]])
            if r > r_max:
                r_max = r
                j = i
        l[i], l[k] = l[k], l[i]
        for i in range(k+1, n+1):
            A[l[i],k] = A[l[i], k]/A[l[k],k]
            for j in range(k+1, n+1):
                A[l[i],j] = A[l[i],j] - A[l[i],k] * A[l[k],j]

    for k in range(0, n):
        for i in range(k+1, n+1):
            b[l[i]] = b[l[i]] - A[l[i],k] * b[l[k]]

    x[n] = b[l[n]]/A[l[n],n]
    for i in range(n-1, -1, -1):
        sum = b[l[i]]
        for j in range(i+1, n+1):
            sum = sum - A[l[i],j]*x[j]
        x[i] = sum/(A[l[i],i])
    return x
```

2.2 Jacobi Iterative Method

2.2.1 Jacobi Iterative Method Math

The Jacobi Iterative Method is an iterative algorithm to solve for the optimal solutions of large-scale systems of equations. Specifically, the process iterates by solving the diagonal elements of the matrices until it converges. This method is described as a simpler of the iterative methods, and an even simpler version of a matrix diagonalization and the Jacobi transformation.

This technique is used as an alternative to the Gauss-Jordan Method. The Gauss-Jordan Method uses the below formula as a more direct approach to how matrices are solved.

$$[A][x]=[b]$$

However, despite being direct, the Gauss-Jordan Method is not accurate, nor efficient, for large matrices. Thus, the Jacobi Iterative Method is utilized to solve for large matrices.

As mentioned before, this method is described as simpler and more comprenable for those not experienced with iterative manipulations due to its easy derivation. Thus, is used sometimes as a training tool to learn further complex iterations in the future. You also use this method when there is already an approximation or estimate for [x]. Thus, making the whole process easier for faster iterations and specific approximations. However, the Jacobi Method is no longer being used as frequently in practice; with more experienced mathematicians using more complex methods.

As mentioned earlier, the Jacobi method solves the diagonal elements of a matrix. Which means that a significant limitation to this method is that it requires the use of a diagonally dominant matrix. A diagonally dominant matrix presents itself when the magnitude of the diagonal element of a row is greater than or equal to the sum of the magnitudes of all other elements in that same row. These following formulas illustrate what constitutes the diagonal elements of a matrix and how we determine if a matrix is diagonally dominant:

$$\left|a_{ii}\right| \geq \sum_{j \neq i} \left|a_{ij}\right|$$

$$diag[A]=\begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \cdots & \\ & & & a_{MM} \end{bmatrix}$$

To first formulate the iteration, the usual setup equation of $[A][x]=[b]$ or $A\boldsymbol{x}=\boldsymbol{b}$ is written, and then replace the variables with the known values or matrices, as follows:

$$A\boldsymbol{x}=\boldsymbol{b} \text{ with } A=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \text{ and } \boldsymbol{b}=\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \text{ for } \boldsymbol{x}=\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

From there the *A* matrix of the equation is expanded into its diagonal elements (*D*), strict lower triangular (*L*), and strict upper triangular (*U*):

$$A=\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}-\begin{bmatrix} 0 & \cdots & 0 & 0 \\ -a_{21} & \cdots & 0 & 0 \\ \vdots & & \ddots & \vdots \\ -a_{n1} & \cdots & -a_{n,n-1} & 0 \end{bmatrix}-\begin{bmatrix} 0 & -a_{12} & \cdots & -a_{1n} \\ 0 & 0 & & \vdots \\ \vdots & \vdots & \ddots & -a_{n-1,n} \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

This manipulation is also better identifiable through its other equation: $A = D - L - U$ to be able to compute the xk+1 iteration.

Once the *A* matrix is expanded, it is placed it back into the original equation and simplified:

$$(D-L-U)\,\boldsymbol{x}=\boldsymbol{b}$$

$$D\boldsymbol{x}=(L+U)\,\boldsymbol{x}+\boldsymbol{b}$$

The next step is to find the inverse of the diagonal elements of the *A* matrix:

$$D^{-1} = \begin{bmatrix} \frac{1}{a_{11}} & 0 & \dots & 0 \\ 0 & \frac{1}{a_{22}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{a_{nn}} \end{bmatrix}$$

After the x is isolated in our equation, the following equation is left:

$$\boldsymbol{x} = D^{-1}(L+U)\boldsymbol{x} + D^{-1}\boldsymbol{b}$$

Finally, the last step in our deviation, the equation is converted to the matrix form of the Jacobi iteration method by giving a subscript (k) to the x as a way to identify that the x will encounter multiple iterations. Specifically, Where $x^{(k)}$ is the kth approximation of x:

$$\boldsymbol{x}^{(k)} = D^{-1}(L+U)\boldsymbol{x}^{(k-1)} + D^{-1}\boldsymbol{b} \qquad k = 1,2,3,\dots$$

Now that the equation of the Jacobi Iteration Method has been deviated, next is to show an example of how this method is applied.

A linear system of the form $Ax = b$ with initial estimate $x^{(0)}$ is given by

$$A = \begin{bmatrix} 2 & 1 \\ 5 & 7 \end{bmatrix}, \; b = \begin{bmatrix} 11 \\ 13 \end{bmatrix} \quad \text{and} \quad x^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

We use the equation $x^{(k+1)} = D^{-1}(b - (L+U)x^{(k)})$, described above, to estimate x. First, we rewrite the equation in a more convenient form $D^{-1}(b - (L+U)x^{(k)}) = Tx^{(k)} + C$, where $T = -D^{-1}(L+U)$ and $C = D^{-1}b$. From the known values

$$D^{-1} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/7 \end{bmatrix}, \; L = \begin{bmatrix} 0 & 0 \\ 5 & 0 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

we determine $T = -D^{-1}(L+U)$ as

$$T = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/7 \end{bmatrix} \left\{ \begin{bmatrix} 0 & 0 \\ -5 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix} \right\} = \begin{bmatrix} 0 & -1/2 \\ -5/7 & 0 \end{bmatrix}.$$

Further, C is found as

$$C = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/7 \end{bmatrix} \begin{bmatrix} 11 \\ 13 \end{bmatrix} = \begin{bmatrix} 11/2 \\ 13/7 \end{bmatrix}.$$

With T and C calculated, we estimate x as $x^{(1)} = Tx^{(0)} + C$:

$$x^{(1)} = \begin{bmatrix} 0 & -1/2 \\ -5/7 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 11/2 \\ 13/7 \end{bmatrix} = \begin{bmatrix} 5.0 \\ 8/7 \end{bmatrix} \approx \begin{bmatrix} 5 \\ 1.143 \end{bmatrix}.$$

The next iteration yields

$$x^{(2)} = \begin{bmatrix} 0 & -1/2 \\ -5/7 & 0 \end{bmatrix} \begin{bmatrix} 5.0 \\ 8/7 \end{bmatrix} + \begin{bmatrix} 11/2 \\ 13/7 \end{bmatrix} = \begin{bmatrix} 69/14 \\ -12/7 \end{bmatrix} \approx \begin{bmatrix} 4.929 \\ -1.714 \end{bmatrix}.$$

This process is repeated until convergence (i.e., until $\|Ax^{(n)} - b\|$ is small). The solution after 25 iterations is

$$x = \begin{bmatrix} 7.111 \\ -3.222 \end{bmatrix}.$$

Saad, Yousef (2003). *Iterative Methods for Sparse Linear Systems* (2nd ed.). SIAM. p. 414

Computer Project II - Solution of Linear Equations

The Jacobi Method can be used for a given system of linear equations. The way to approach this is by rewriting the system in terms of x1, x2, x3, x4, … xn, to get your system of equations.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots a_{nn}x_n &= b_n \end{aligned} \qquad \begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots a_{1n}x_n) \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots a_{2n}x_n) \\ &\vdots \\ x_n &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots a_{n,n-1}x_{n-1}) \end{aligned}$$

Choose your initial guess for the first iteration to be x(0), and plug it into your equations to solve for the remaining unknown values.

$$\boldsymbol{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, \ldots x_n^{(0)})$$

Then, for iteration 2, you increment up to x(1) and repeat the process. Continue to increment up by 1 each iteration until your system converges.

$$\left(x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, \ldots x_n^{(1)}\right). \; \boldsymbol{x}^{(k)} = \left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \ldots x_n^{(k)}\right)^t, \quad k = 1,2,3, \ldots$$

The system converges when the error is less than the tolerance, or when your x0 = x1 of up to about 5 or more decimal places.

2.2.2 Jacobi Iterative Method

```
# SOLVING MATRIX WAY#
import numpy as np
import time
import matplotlib.pyplot as plt
import scipy.sparse
from scipy.sparse.linalg import spsolve
import random

question_3 = 1
question_10a = 0
question_10b = 0

def Jacobi_Iterative_Method(A, b, x_0):



    D = np.diagflat(np.diag(A))    # D matrix with diagonal elements of 0


    lower_diag = np.tril(A)        # lower triangular matrix with diagonal elemen
ts of 0
    L = lower_diag - D             # L matrix
    U = A - lower_diag             # U matrix

    x_k = x_0                      # setting up first iteration to be

    # the D^(-1) * [ b - ((L + U)* x_0) ]
    x_k1 = np.dot(np.linalg.inv(D), b - np.dot(L + U, x_k))


    return x_k1

# starting tolerance value and acceptable tolerance (stoping criteria)
    tolerance = 1.0e-5
    starting_tol = 1.0
    i = 0


    print('''First find the optimal solution for x using the formulas specified i
n the Jacobi Iterative method: ''')
    t1_start = time.time()

    #for h in range(4):
    while (starting_tol > tolerance):
        print('Iteration # %s\t Value of x: %s' % (i, x))
```

```
    t2_start = time.process_time()
    x_using_imported_package = np.linalg.solve(A, b)
    t2_finish = time.process_time()
    print(x_using_imported_package)

    print('Time it took to do the numpy package: ', (t2_finish-t2_start), '\n')

    abs_error = la.norm(x - x_using_imported_package)
    print("Absolute error: " + str(abs_error))
    rel_error = (la.norm(x - x_using_imported_package)) / la.norm(x_using_importe
d_package)
    print("Percent error:" + "{:10.4f}".format(rel_error))
```

This is the algorithm for Jacobi. An input of an A matrix, b vector, and an initial guess for x with elements of 0 is needed for the code to run. The Jacobi_Iterative_Method function will run through a loop, and return the xk+1 value after each iteration. It will also print the previous xk and current xk+1 values after each iteration. The loop runs by having a starting tolerance of 1 (this value is just a placeholder for the iterations to begin). Every time the loop runs an iteration, it will calculate the current error and plug it into the starting_tol variable, replacing the previous value that's in it. The loop will continue to run until the starting_tol (current error value) variable is less than the acceptable tolerance. This is because of the Jacobi method stopping criteria. It will then print out the final x, along with its iteration number, how long it took to compute, and the absolute and relative error.

2.3 Gauss-Seidel Iterative Method

2.3.1 Gauss-Seidel Iterative Method Math

Closely related to the Jacobi method, The Gauss- Seidel method is too an iterative method used to solve a linear system of equations. The only difference being that the latter uses the most recently updated values of the unknown at each step, even if the updating occurs in the current step. For example, given the following matrix function:

$$\begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix}=\begin{bmatrix} 5 \\ 5 \end{bmatrix}$$

Sauer, T. (2012). *Numerical analysis*. Boston: Pearson.

Using the Gauss-Seidel method to approximate the solution in the following fashion:

$$\begin{bmatrix} u_0 \\ v_0 \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
$$\begin{bmatrix} u_1 \\ v_1 \end{bmatrix}=\begin{bmatrix} \frac{5-v_0}{3} \\ \frac{5-u_1}{2} \end{bmatrix}=\begin{bmatrix} \frac{5-0}{3} \\ \frac{5-5/3}{2} \end{bmatrix}=\begin{bmatrix} \frac{5}{3} \\ \frac{5}{3} \end{bmatrix}$$
$$\begin{bmatrix} u_2 \\ v_2 \end{bmatrix}=\begin{bmatrix} \frac{5-v_1}{3} \\ \frac{5-u_2}{2} \end{bmatrix}=\begin{bmatrix} \frac{5-5/3}{3} \\ \frac{5-10/9}{2} \end{bmatrix}=\begin{bmatrix} \frac{10}{9} \\ \frac{35}{18} \end{bmatrix}$$
$$\begin{bmatrix} u_3 \\ v_3 \end{bmatrix}=\begin{bmatrix} \frac{5-v_2}{3} \\ \frac{5-u_3}{2} \end{bmatrix}=\begin{bmatrix} \frac{5-35/18}{3} \\ \frac{5-55/54}{2} \end{bmatrix}=\begin{bmatrix} \frac{55}{54} \\ \frac{215}{108} \end{bmatrix}$$

Sauer, T. (2012). *Numerical analysis*. Boston: Pearson.

Here one can see the difference between the Jacobi and Gauss-Seidel method, that is, every v_{n+1} value uses the u_{n+1} value instead of the u_n

The Gauss-Seidel method of linear equations with unknown x is defined by the iteration

$$(L + D + U)x = b \text{ as}$$

$$(L + D)x_{k+1} = -Ux_k + b.$$

Where x_k is the kth approximation of x and x_{k+1} is the following iteration, and the matrix A is decomposed into a lower triangular component L_*, A's diagonal value D, and a strictly upper triangular component U. The Gauss-Seidel method now solves the left side x using a previous values for x on the right side:

$$x_{k+1} = D^{-1}(b - Ux_k - Lx_{k+1}) \quad \textbf{for } k = 0, 1, 2, \ldots.$$

One should use forward substitution to take advantage of the triangular form of L_*:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

The major gain of this algorithm over Jacobi's Method is it's accuracy at the same number of steps. However, because of its use of previously solved values, every new value in a vector approximation needs the previous value in order to be computed. This aspect makes it impossible to parallel compute the solution at each step, unlike the jacobi method which can be used in parallel computing.

An Example:

$$x_0 = \text{initial vector}$$
$$x_{k+1} = D^{-1}(b - Ux_k - Lx_{k+1}) \quad \textbf{for } k = 0, 1, 2, \ldots.$$

Apply the Gauss–Seidel Method to the system

$$\begin{bmatrix} 3 & 1 & -1 \\ 2 & 4 & 1 \\ -1 & 2 & 5 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 1 \end{bmatrix}.$$

The Gauss–Seidel iteration is

$$u_{k+1} = \frac{4 - v_k + w_k}{3}$$
$$v_{k+1} = \frac{1 - 2u_{k+1} - w_k}{4}$$
$$w_{k+1} = \frac{1 + u_{k+1} - 2v_{k+1}}{5}.$$

Starting with $x_0 = [u_0, v_0, w_0] = [0, 0, 0]$, we calculate

$$\begin{bmatrix} u_1 \\ v_1 \\ w_1 \end{bmatrix} = \begin{bmatrix} \frac{4-0-0}{3} = \frac{4}{3} \\ \frac{1-8/3-0}{4} = -\frac{5}{12} \\ \frac{1+4/3+5/6}{5} = \frac{19}{30} \end{bmatrix} \approx \begin{bmatrix} 1.3333 \\ -0.4167 \\ 0.6333 \end{bmatrix}$$

and

$$\begin{bmatrix} u_2 \\ v_2 \\ w_2 \end{bmatrix} = \begin{bmatrix} \frac{101}{60} \\ -\frac{3}{4} \\ \frac{251}{300} \end{bmatrix} \approx \begin{bmatrix} 1.6833 \\ -0.7500 \\ 0.8367 \end{bmatrix}.$$

The system is strictly diagonally dominant, and therefore the iteration will converge to the solution $[2, -1, 1]$. ◀

Sauer, T. (2012). *Numerical analysis*. Boston: Pearson.

2.3.2 Gauss-Seidel Iterative Method Code

Gauss-Seidel Algorithm defined function:

```
import time
import numpy as np
```

```
'''Defining our function as seidel which takes 3
   arguments an A matrix, Solution and B matrix '''

def seidel(a, x ,b):
    #Finding length of a(3)
    n = len(a)
    # for loop for 3 times as to calculate x, y , z
    for j in range(0, n):
        # temp variable d to store b[j]
        d = b[j]

        # to calculate respective xi, yi, zi
        for i in range(0, n):
            if(j != i):
                d-=a[j][i] * x[i]
        # updating the value of our solution
        x[j] = d / a[j][j]
    # returning our updated solution
    return x
```

2.4 SOR Iterative Method

2.4.1 SOR Iterative Method Math

Another iterative method for solving a linear system is the SOR method. The SOR method stands for Successive Over-Relaxation and builds on the foundations of Jacobi and Gauss-Seidel to achieve even faster convergence. This acceleration is achieved by introducing a relaxation parameter, ω. This relaxation parameter works as a weighted average by minimizing large step sizes of iterates in each successive iteration. So each iteration of SOR takes an approximation based on said parameter between x_i and x_{i+1}

The matrix form of the method is as follows:

$$\vec{x}_0 \text{ is the initial guess}$$
$$\vec{x}_{k+1} = (\omega L + D)^{-1}[(1-\omega)D\vec{x}_k - \omega U\vec{x}_k] + \omega(D+\omega L)^{-1}\vec{b}, \text{for } k = 0,1,2,\dots$$

The coefficient matrix A is decomposed into the diagonal matrix D, the strictly lower triangular matrix L (strictly means zeros on the diagonal) and the strictly upper triangular matrix U. ω is the relaxation parameter and a necessary condition for the method to converge is $0 < \omega < 2$. It is called over-relaxation when $\omega > 1$, whereas $\omega < 1$ is under-relaxation.

In practice, the inverse of the matrices above aren't computed because of computation complexity. They are merely for analytic exploration. Instead, here is an example of how the method works intuitively in comparison to Gauss-Seidel:

Example linear system

$$\begin{bmatrix} 3 & 1 & -1 \\ 2 & 4 & 1 \\ -1 & 2 & 5 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 1 \end{bmatrix}$$

Gauss-Seidel

$$\begin{aligned} u_{k+1} &= \frac{4 - v_k + w_k}{3} \\ v_{k+1} &= \frac{1 - 2u_{k+1} - w_k}{4} \\ w_{k+1} &= \frac{1 + u_{k+1} - 2v_{k+1}}{5}. \end{aligned}$$

SOR-Method

$$\begin{aligned} u_{k+1} &= (1-\omega)u_k + \omega\frac{4 - v_k + w_k}{3} \\ v_{k+1} &= (1-\omega)v_k + \omega\frac{1 - 2u_{k+1} - w_k}{4} \\ w_{k+1} &= (1-\omega)w_k + \omega\frac{1 + u_{k+1} - 2v_{k+1}}{5}. \end{aligned}$$

The above example shows how SOR is a variation on Gauss-Seidel. Notice that each variable at the k_{th} step is [(1-ω) * current variable value] + [ω * Gauss-Seidel part].
Let the relaxation parameter be 1. Then the left half becomes 0 and the SOR method devolves back into Gauss-Seidel.

The iterative form is as follows:

Gauss-Seidel Method

$$x_i{}^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_i^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} xj^{(k)} \right)$$

Successive Over-Relaxation Method

$$x_i{}^{(k+1)} = (1-w)\, x_i^{(k)} \frac{w}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} xj^{(k)} \right)$$

How can we be guaranteed that the SOR method converges? The SOR convergence theorem asserts that if you have a coefficient matrix with positive diagonal elements and relaxation factor $0 < \omega < 2$ then if the coefficient matrix is symmetric and positive definite, we can be assured that it converges for any initial starting solution vector.

The SOR Method doesn't come without its limitations. The SOR convergence theorem itself sheds light on the main limiting factor when considering choosing the SOR method in comparison to Jacobi or Gauss-Seidel. The guaranteed convergence of Jacobi or Gauss-Seidel only requires a diagonally dominant coefficient matrix while SOR requires a special type of symmetric matrix, a positive definite matrix. Also, the immensely fast convergence is reliant on choosing the most optimal relaxation factor before computing the approximate solution but this is currently an unsolved problem outside of special cases. Thus, poorer convergence than desired can arise with a poorly chosen relaxation factor. Generally, it has been found that utilizing under-relaxation converges slower than over-relaxation. Yet, the SOR method has its use cases when fast convergence is required.

Let's test the convergence of the SOR method with computer exercise 8.4.3 and 8.4.10. and see how the rate of convergence varies with the relaxation parameter.

Exercise 8.4.3				
Relaxation Factor w		**Iterations**		**Relative Error**
0.1		475		1.7702E-03
0.2		253		8.1182E-04
0.3		170		5.0016E-04
0.4		125		3.5801E-04
0.5		97		2.6434E-04
0.6		78		1.9181E-04
0.7		63		1.5834E-04
0.8		52		1.1919E-04
0.9		43		9.0688E-05
1		36		5.9017E-05
1.1		29		4.8139E-05
1.2		23		3.3702E-05
1.3		17		2.1248E-05
1.4		12		1.0818E-07
1.5		16		1.9418E-06
1.6		21		1.1322E-05
1.7		31		7.1032E-07
1.8		46		4.9243E-06
1.9		102		8.0115E-06

This exercise for the 3 x 3 matrix illuminates the general trend the SOR method follows even though the matrixs' size for practical applications is order of magnitudes smaller. The general trend is extrapolated to generalize dense matrices even though this experiment only utilized one small dense matrix.

The general trend for this dense matrix

When an under-relaxed factor is chosen, that is ω < 1 , The convergence rate is high and in turn this leads to a higher approximation error because for an iterative approach the inherent nature of roundoff error in the numerical calculations is higher as the iteration count rises.

The over-relaxed factors, that is ω > 1, show why the SOR method is a natural evolution of the Gauss-Seidel method. As the relaxation factor increases, the iterations and approximation error decrease until ω=1.4, and then increase again. This behavior mimics the graph of an absolute function about ω=1.4. Yet, even as the method moves above ω=1.4, it acts well-behaved and faster than the baseline gauss-seidel method.

The outliers, which will be more apparent with the sparse matrices exercise, are the relaxation factors within two steps of the range bound, that is ω= { 0.1, 0.2, 1.8, 1.9 }. The SOR method in theory is proven to converge at those values, but from the experiments, the numerical instability and the iterations make them a poor choice.

Relaxation Factor w = 0.3	Part A	Condition Number	Iterations	Relative Error
	8x8	225.3514979	1109	0.006162
	9x9	340.8330746	1584	0.00895111
	10x10	496.1081774	2182	0.0107432
	11x11	699.4799721	2886	0.01447201
	Part B			
	8x8	253.9256063	1128	0.0069168
	9x9	406.3917491	1714	0.01035703
	10x10	621.5915563	2192	0.01706198
	11x11	914.9055748	2845	0.03230088

Relaxation Factor w = 0.8	Part A	Condition Number	Iterations	Relative Error
	8x8	225.3514979	370	0.00160906
	9x9	340.8330746	538	0.00234244
	10x10	496.1081774	752	0.00281797
	11x11	699.4799721	1010	0.00380653
	Part B			
	8x8	253.9256063	367	0.00174535
	9x9	406.3917491	577	0.00264443
	10x10	621.5915563	787	0.00437734
	11x11	914.9055748	1056	0.00836334

Relaxation Factor w = 1.4	Part A	Condition Number	Iterations	Relative Error
	8x8	225.3514979	119	0.00039712
	9x9	340.8330746	178	0.00061391
	10x10	496.1081774	254	0.00076169
	11x11	699.4799721	346	0.00102747
	Part B			
	8x8	253.9256063	105	0.00038077
	9x9	406.3917491	180	0.00062523
	10x10	621.5915563	266	0.00108935
	11x11	914.9055748	364	0.00212806

Relaxation Factor w = 1.7	Part A	Condition Number	Iterations	Relative Error
	8x8	225.3514979	48	6.20E-05
	9x9	340.8330746	67	0.00012971
	10x10	496.1081774	89	0.00013314
	11x11	699.4799721	104	0.00047933
	Part B			
	8x8	253.9256063	MAX	FAILED
	9x9	406.3917491	166	1.17E-05
	10x10	621.5915563	157	1.36E-05
	11x11	914.9055748	125	0.00044914

This is a snapshot look at handpicked relaxation factors for the sparse matrices in exercise #10. The general trend is different from the 3x3 dense matrix.

The general trend for a sparse matrix

The condition number increases as the sparse matrix gets larger and as the condition number's magnitude gets farther and farther from 1.0, and with that the coefficient matrices become more ill-conditioned. This shows up in the error of the approximation, no matter which iterative method is used. As the sparse matrix grows in size, for any relaxation factor, the error grows at a rapid pace in comparison to the dense matrix.

The trend of the iterations and relative error are the same for under-relaxed factors. The trend of over-relaxed factors are surprisingly different though. As the relaxation moves from ω =

1.0 to the upper bound ω = 1.9, the convergence becomes quicker and the error gets smaller. So the iterations and errors follow a linear trend when over-relaxed.

An example of SOR method behavior at upper bound of ω

```
Iterations: 4999         Relaxation factor: 1.8
Solution: [ 2.87747662e+190 -4.34441949e+190  5.97519757e+190 -7.62026840e+190  8.85190186e+190 -9.3
1844549e+190  1.06006799e+191 -4.86223985e+190  2.08074204e+191]
Absolute error: inf
Relative error: inf
Time to compute: 0.4477859999999989 seconds


Iterations: 4999         Relaxation factor: 1.9
Solution: [nan nan nan nan nan nan nan nan nan]
Absolute error: nan
Relative error: nan
Time to compute: 0.45397399999999877 seconds
```

A problem encountered in the experiment for any size sparse matrix utilizing ω = 1.8 and ω = 1.9 is that they failed to converge and suffered from overflow. An interpretation of this is that they would still follow the linear trend and be better relaxation factors for these sparse matrices if the computer could hold all the bits required.

The literature states that the best relaxation factor varies across different coefficient matrices. So the general trends are only applicable for the sparse matrices and the dense 3x3 matrix used for this project. Even still, the experiments did show that over-relaxed is better than under-relaxed in the general case, that relaxation factors close to the upper and lower bounds are unstable and that a safe choice is ω = 1.4.

2.4.2 SOR Iterative Method Code

```
def sor_method(A_matrix, b_vector, x_vector, relax_factor, size):

  # Max Iterations set to a high ceiling to allow convergence
  k_max = 5000
  epsilon = 0.5 * pow(10,-4)

  # Main Loop - Loop through k_max iterations
  for k in range(k_max):
    # y_vector is copy of current unaltered x_vector, thus it is xi
    y_vector = np.copy(x_vector)

    # Loop through size(A) times
    for i in range(size):
      sum = b_vector[i]
      diag = A_matrix[i,i]

      for j in range(i):
        sum -= A_matrix[i,j] * x_vector[j]
      for j in range(i+1,size):
        sum -= A_matrix[i,j] * x_vector[j]

      x_vector[i] = sum / diag
      x_vector[i] = relax_factor * x_vector[i] + (1.0 - relax_factor) * y_vector[i]

    # Convergence Check
    # Test if the magnitude(norm) of the difference between xi+1 - xi is sufficiently small
    # That is, the required precision prescribed by the used epsilon value
    if np.linalg.norm(x_vector - y_vector) < epsilon:
      return k
  return k
```

2.5 Conjugate Gradient Method

2.5.1 Conjugate Gradient Method Math

The conjugate gradient method is a popular variant of the gradient method to solve for the optimal solutions of large-scale systems of equations of the form: $Ax = b$, where A is symmetric and positive definite. The method was written to combat the problem of the convergence of the optimum being slow for larger matrices. While the gradient method is relatively fast when far away from finding the optimal solution, the method becomes very slow if one is close to finding the optimal solution. The idea is that instead of moving along the gradient direction, leading to a slow process, one could employ the conjugate gradient method to move along the set conjugate directions to reach the optimum very fast. Another reason why the conjugate gradient method is a popular method to solve for the optimal solution is that one is guaranteed to converge in n steps, where n is the system size.

In the conjugate gradient method, one can find the current direction in which you must move using a linear combination of all of the previous gradient directions compared to the gradient method where you only use the gradient direction to move in the direction of the optimum solution.

To find the solution of $Ax = b$ where x is the optimal solution, x_*, one must find the set of k linearly independent conjugate directions, s_k, and the weight in which one must move towards s_k as the coefficient, β_k. Here is the k linear set of independent conjugate directions:

$$\left\{s_0,\ s_1,\ s_2,\ \ldots,\ s_k\right\}$$

s_k is the direction one must move towards and β_k is the coefficient of how much s_k should be weighed for each step in the iterations. You must consider both variables while taking steps toward the optimal solution, x_* until the error term e_k is less than the predetermined value where $e_k = x_k - x_*$. You must find the most optimal solution for x_k to be able to reach the optimum of ϕ whose global minimum is the solution to $Ax = b$ in the quadratic form:

$$\phi = \frac{1}{2} \cdot x^T Ax - x^T b$$

To begin to find the optimum solution for x_k, one must first address the required standard arbitrary values like setting the initial solution for x_k, and the initial conjugate direction, s_k to the residual, r_k:

$$x_0 = 0$$

$$s_0 = -r_0 = b - Ax_0$$

Once the required standard arbitrary values are set, one must then start taking n steps towards finding the final x_k where the stopping condition, $e_k = x_k - x_*$ is met where e_k is less than the predetermined error term.

For each iteration, new estimates of x_k (desired estimate of x_*), β_k (weight of each step taken in the direction of s_k), r_k (residual value employed in solving for s_k), ϑ_k (constant corresponding to s_{k-1} used to solve for s_k), and s_k (conjugate direction employed to make steps towards x_*). One must do complete n steps until the predetermined stopping condition set is met. The required equations for each step are as follows:

Weight of the step to be taken:

$$\beta_k = -\frac{s_k^T \cdot r_k}{s_k^T \cdot A \cdot s_k}$$

New estimate of x_k:

$$x_{k+1} = x_k + \beta_k \cdot s_k$$

New residual value:

$$r_{k+1} = r_k + \left(\beta_k \cdot A \cdot s_k\right)$$

New constant corresponding to s_{k-1}:

$$\vartheta_{k+1} = \frac{r_{k+1}^T \cdot r_{k+1}}{r_k \cdot r_k}$$

New s_k to make steps towards x_*:

$$s_{k+1} = \left(\vartheta_{k+1} \cdot s_k\right) - r_{k+1}$$

One may stop making steps towards x_* once the stopping condition of $e_k = x_k - x_*$ being less than the predetermined error term is met. Once the final optimal x_k is found, x_k now contains the global minimum of x_k is the solution to the quadratic form of ϕ and is the optimal solution to $Ax = b$.

An Example:

Where the matrix

$$A = \begin{bmatrix} 5 & -2 & 0 \\ -2 & 5 & 1 \\ 0 & 1 & 5 \end{bmatrix}$$

is positive definite and symmetric. For the cg-method, first set the starting point

$$x^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

and then calculate the initial assumptions:

$$r^{(0)} = b - Ax^{(0)} = b = \begin{bmatrix} 20 \\ 10 \\ -10 \end{bmatrix}$$

Rambo, M. (2017). *The Conjugate Gradient Method for Solving Linear Systems*. St. Mary's College.

Next, set $d^{(1)} = r^{(0)}$, so that

$$d^{(1)} = \begin{bmatrix} 20 \\ 10 \\ -10 \end{bmatrix}$$

Rambo, M. (2017). *The Conjugate Gradient Method for Solving Linear Systems*. St. Mary's College.

Then running through the first iteration, $k = 1$,

$$s_1 = \frac{\langle r^{(0)}, r^{(0)} \rangle}{\langle d^{(1)}, Ad^{(1)} \rangle} = \frac{3}{10}$$

$$x^{(1)} = x^{(0)} + s_1 d^{(1)} = \begin{bmatrix} 6 \\ 3 \\ -3 \end{bmatrix}$$

$$r^{(1)} = r^{(0)} - s_1 Ad^{(1)} = \begin{bmatrix} -4 \\ 10 \\ 2 \end{bmatrix}$$

$$c_1 = \frac{\langle r^{(1)}, r^{(1)} \rangle}{\langle r^{(0)}, r^{(0)} \rangle} = \frac{1}{5}$$

$$d^{(2)} = r^{(1)} + c_1 d^{(1)} = \begin{bmatrix} 0 \\ 12 \\ 0 \end{bmatrix}$$

Rambo, M. (2017). *The Conjugate Gradient Method for Solving Linear Systems*. St. Mary's College.

Next, for the second iteration $k = 2$,

$$s_2 = \frac{\langle r^{(1)}, r^{(1)} \rangle}{\langle d^{(2)}, Ad^{(2)} \rangle} = \frac{1}{6}$$

$$x^{(2)} = x^{(1)} + s_2 d^{(2)} = \begin{bmatrix} 6 \\ 5 \\ -3 \end{bmatrix}$$

$$r^{(2)} = r^{(1)} - s_2 Ad^{(2)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Since the residual $r^{(2)} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$, the solution is exactly $x^{(2)}$.

Rambo, M. (2017). *The Conjugate Gradient Method for Solving Linear Systems*. St. Mary's College.

2.5.2 Conjugate Gradient Method Code

Import the needed libraries and set the flags to direct the script to solve the questions:

```
import numpy as np
import time
import matplotlib.pyplot as plt
import scipy.sparse
from scipy.sparse.linalg import spsolve
import random

# Flags for the different questions
question3 = 1
question10a = 0
question10b = 0
```

Define the function and set the arbitrary initial conditions:

```
def conjugate_gradient(A, b, x=None):

    ''' Setting n as the matrix system size
    according to the size of array b '''
    n = len(b)

    ''' Set condtion where if this is the
    first step and there is no intial
    value of x, set an arbitary
    initial solution '''

    if not x:
        ''' Returns an arbitary array of
        zeros in the shape of n as the
        intial solution for x '''
        x = np.zeros(n)

    # Calculation of the residual, r
    r = np.dot(A, x) - b

    ''' The initial value of the conjugate
    step direction is the negative of
    the residual '''
    s = -r

    ''' Finding the dot product between the
    residual and residual and the residual
    to be used in the calculations to find
    the beta value '''
    r_dotProduct = np.dot(r, r)
```

Complete required number of steps until the stopping condition is satisfied:

```
    # Set the maximum number of iterations as 4*n
    for i in range(2*n):
        ''' A_dot_s is the dot product between the given matrix,
        A and the conjugate direction of the step, s '''
        A_dot_s = np.dot(A, s)

        ''' Calculates beta, the weight of the step to be
        taken each iteration '''
        beta_numerator = np.dot(np.transpose(s), r)
        beta_denomenator =(np.dot(np.transpose(s), A_dot_s))
        beta = -(beta_numerator/beta_denomenator)

        ''' The new estimate of x is the last estimate of x +
        (the weight constant, beta) * (conjugate direction, s) '''
        x += beta * s

        ''' The new residual value is the previous residual value -
        (weight of the step to be taken beta) * (matrix A) *
        (conjugate direction s) '''
        r += beta * A_dot_s

        ''' Find the dot product of the new residual value,
        r by it's transpose to be used to calculate theta '''
        rNext_dotProduct = np.dot(np.transpose(r), r)

        ''' Theta is the constant corresponding to the previous
        step's conjugate step direction, s '''
        theta = rNext_dotProduct / r_dotProduct

        ''' The new residual value for the next step, r becomes
        the r of the current step '''
        r_dotProduct = rNext_dotProduct

        ''' If the residual dot product is less thant the
        predetermined error term, stop doing iterations, you
        have found the optimal solution, x'''
```

Set the stopping condition and assign new value for the conjugate direction value:

```
        ''' If the residual dot product is less thant the
        predetermined error term, stop doing iterations, you
        have found the optimal solution, x'''
        if rNext_dotProduct < 1e-5:

            # Print the number of iterations
            print('The optimal solution for x was found after this many iterations: ', i)
            break

        # Find next step's conjugate step direction, s
        s = theta * s - r
    return x
```

This is the function used to do the conjugate gradient method. First, set the function to be used to then set the maximum number of iterations of the algorithm to be 2*n. A_dot_s is the dot product between the given matrix, A and the conjugate direction of the step s. Then calculate beta, the weight of the step to be taken each iteration. Then find the new estimate of x is the last estimate of x + (the weight constant, beta) * (conjugate direction, s). Then find the dot product of the new residual value r by its transpose to be used to calculate theta.

Theta is the constant corresponding to the previous step's conjugate step direction, s. Then the new residual value for the next step, r becomes the r of the current step. If the residual dot product is less than the predetermined error term, stop doing iterations, you have found the optimal solution, x.

After each iteration, calculator a new value for the conjugate direction s. Once the stopping condition is met, break out of the for loop.

3. Results

3.1 Gaussian Elimination Method

3.1.1 Computer Exercise 3

Script Input:

```
A = np.array([[ 7, 3,-1, 2],
              [ 3, 8, 1,-4],
              [-1, 1, 4,-1],
              [ 2,-4,-1, 6]])

b = np.array([-1, 0, -3, 1])

print('Q3 Solution:', gauss(4, A, b))
print('Q3 Expected:', np.linalg.solve(A,b))
print('Q3 error:', np.abs(np.linalg.solve(A,b)-gauss(4, A, b)))
```

Script Output:

```
Q3 Solution: [-0.36532738  0.171875   -0.70833333  0.16666667]
               Q3 Expected: [-1.  1. -1.  1.]
   Q3 error: [0.63467262 0.828125   0.29166667 0.83333333]
```

3.1.2 Computer Exercise 10a

Script Input:

```
m = 10 # A is mxm matrix
# Choose Diagonals
diagonals = [[1 for _ in range(m-2)],
             [-4 for _ in range(m-1)],
             [6 for _ in range(m)],
             [-4 for _ in range(m-1)],
             [1 for _ in range(m-2)]]
# Generate A
A = scipy.sparse.diags(diagonals, [-2, -1, 0, 1, 2]).toarray()
# Set diferent values at corners
A[0, 0:3] = [12., -6., (4/3)]
A[-1, -3:] = [(4/3), -6., 12.]

error = 0
for _ in range(n_tries):
    x = np.random.rand(m)
    b = A @ x

    ans = np.linalg.solve(A,b)
    ans2 = np.nan_to_num(gauss(m, A, b))
    error += np.square(ans - ans2).mean()
error = error/100
print('Q10A avg MSE:', error)

A[-2, -3:] = [-(93/25), (111/25), -(43/25)]
A[-1, -3:] = [(12/25), (24/25), (12/25)]
```

Script Output:

```
Q10A avg MSE: 5.245080949785693e-30
```

3.1.3 Computer Exercise 10b

Script Input:

```
A[-2, -3:] = [-(93/25), (111/25), -(43/25)]
A[-1, -3:] = [(12/25), (24/25), (12/25)]

error = 0
for _ in range(n_tries):
    x = np.random.rand(m)
    b = A @ x

    ans = np.linalg.solve(A,b)
    ans2 = np.nan_to_num(gauss(m, A, b))
    error += np.square(ans - ans2).mean()
error = error/100
print('Q10B avg MSE:', error)
```

Script Output:

```
Q10A avg MSE: 5.245080949785693e-30
```

3.2 Jacobi Iterative Method

3.2.1 Computer Exercise 3

Script Input:

```
if question_3 == 1:
    A = np.array([[7,  3,  -1,  2],  # matrix
                  [3,  8,   1, -4],
                  [-1, 1,   4, -1],
                  [2, -4,  -1,  6]])

    b = np.array([-1, 0, -3, 1])     # vector

    x = np.zeros(4)               # vector x with elements of 0 (initial setup)

    print("\nA :\n")
    print(A)

    print("\nb :\n")
    print(b)

    print("\nx :\n")
    print(x)

    # starting tolerance value and acceptable tolerance (stoping criteria)
    tolerance = 1.0e-5
    starting_tol = 1.0
    i = 0


    print('''First find the optimal solution for x using the formulas specified i
n the Jacobi Iterative method: ''')
    t1_start = time.time()

    #for h in range(4):
    while (starting_tol > tolerance):
        print('Iteration # %s\t Value of x: %s' % (i, x))
        i+=1
        # print each time the updated solution
        tempA = x - 0
        x = Jacobi_Iterative_Method(A, b, x)
        starting_tol = np.abs(np.max(tempA-x))

    t1_finish = time.time()
    print('Time it took to do the Jacobi Iterative method: ', (t1_finish-
t1_start), '\n')

    # Finding solution using the numpy package
    print('''Finally, find the exact optimal solution for x using the linlag.solv
e module from the numpy package: ''')
```

```
    print('''Finally, find the exact optimal solution for x using the linlag.solv
e module from the numpy package: ''')
    t2_start = time.process_time()
    x_using_imported_package = np.linalg.solve(A, b)
    t2_finish = time.process_time()
    print(x_using_imported_package)

    print('Time it took to do the numpy package: ', (t2_finish-t2_start), '\n')

    abs_error = la.norm(x - x_using_imported_package)
    print("Absolute error: " + str(abs_error))
    rel_error = (la.norm(x - x_using_imported_package)) / la.norm(x_using_importe
d_package)
    print("Percent error:" + "{:10.4f}".format(rel_error))
```

Script Output:

```
Question 3 answer:
A :
[[ 7  3 -1  2]
 [ 3  8  1 -4]
 [-1  1  4 -1]
 [ 2 -4 -1  6]]

b :
[-1  0 -3  1]

x :
[0. 0. 0. 0.]

First find the optimal solution for x using the formulas specified in the Jacobi Iterative method:
Iteration # 0   Value of x: [0. 0. 0. 0.]
Iteration # 1   Value of x: [-0.14285714  0.         -0.75        0.16666667]
Iteration # 2   Value of x: [-0.29761905  0.23065476 -0.74404762  0.08928571]
Iteration # 3   Value of x: [-0.3735119   0.24925595 -0.85974702  0.29563492]
Iteration # 4   Value of x: [-0.45696925  0.3953528  -0.83178323  0.3140501 ]
Iteration # 5   Value of x: [-0.52084883  0.43236142 -0.88456799  0.44392774]
Iteration # 6   Value of x: [-0.58135825  0.52785318 -0.87732063  0.48109589]
Iteration # 7   Value of x: [-0.63186742  0.56822237 -0.90702888  0.56613477]
Iteration # 8   Value of x: [-0.67770936  0.63339628 -0.90848876  0.60493257]
Iteration # 9   Value of x: [-0.71693468  0.67016839 -0.92654327  0.66341918]
Iteration # 10  Value of x: [-0.75198383  0.716378   -0.93092097  0.69799994]
```

```
Iteration # 61  Value of x: [-0.99969101  0.99964371 -0.99991651  0.99962762]
Iteration # 62  Value of x: [-0.99972899  0.9996875  -0.99992677  0.99967339]
Iteration # 63  Value of x: [-0.9997623   0.99972591 -0.99993577  0.99971354]
Iteration # 64  Value of x: [-0.99979151  0.9997596  -0.99994367  0.99974874]
Iteration # 65  Value of x: [-0.99981714  0.99978915 -0.99995059  0.99977963]
Iteration # 66  Value of x: [-0.99983961  0.99981506 -0.99995666  0.99980671]
Iteration # 67  Value of x: [-0.99985933  0.99983779 -0.99996199  0.99983047]
Iteration # 68  Value of x: [-0.99987662  0.99985773 -0.99996666  0.99985131]
Iteration # 69  Value of x: [-0.99989178  0.99987522 -0.99997076  0.99986958]
Iteration # 70  Value of x: [-0.99990508  0.99989055 -0.99997435  0.99988561]
Iteration # 71  Value of x: [-0.99991675  0.99990401 -0.99997751  0.99989967]
Iteration # 72  Value of x: [-0.99992698  0.9999158  -0.99998027  0.999912  ]
Time it took to do the Jacobi Iterative method:  0.0

Finally, find the exact optimal solution for x using the linlag.solve module from the numpy package:
[-1.  1. -1.  1.]
Time it took to do the numpy package:  0.0


Absolute error: 0.00012574414428218313
Percent error:   0.0001
```

After 72 iterations, the system converges. The final approximation is x = [-0.99992698, 0.9999158, -0.999998027, 0.999912], which is very close to the given exact value of x = [-1, 1, -1, 1]. The iterations stopped at an error of 5.8094468129155086e-05 which is smaller than the set tolerance of 1e-05 therefore fulfilling the criteria.

3.2.2 Computer Exercise 10a

Script Input:

```
if question_10a==1:
    error_list=[]
    # A is mxm matrix
    for m in range(8,12):
        # Choose Diagonals
        diagonals = [[1 for _ in range(m-2)],
                     [-4 for _ in range(m-1)],
                     [6 for _ in range(m)],
                     [-4 for _ in range(m-1)],
                     [1 for _ in range(m-2)]]

        # Generate A
        A = scipy.sparse.diags(diagonals, [-2, -1, 0, 1, 2]).toarray()
        # Set diferent values at corners
        A[0, 0:3] = [12., -6., (4/3)]
        A[-1, -3:] = [(4/3), -6., 12.]

        condition_number = la.cond(A)
        print("Condition Number: " + "{:10.8f}".format(condition_number))

        randomx = random.sample(range(m), int(m))
        b = np.matmul(A,randomx)

        x = np.zeros(m)              # vector x with elements of 0 (initial setu
p)


        print('The given linear system of equations: ', '\n', 'A= ', '\n', A)
        print('b= ', '\n', b, '\n')


        # relative error total and rel_sol values to stop while loop at certain e
rror margin
        tolerance = 1.0e-5
        starting_tol = 1.0
        i = 0

        print('''First find the optimal solution for x using the formulas specifi
ed in the jacobi iterative method: ''')
        t1_start = time.process_time()
        for h in range(10):
        #while (starting_tol > tolerance):
            print('Iteration # %s\t Value of x: %s' % (i, x))
            i+=1
            # print each time the updated solution
            tempA = x - 0
```

```
            print(x)
            x = Jacobi_Iterative_Method(A, b, x)
            starting_tol = np.abs(np.max(tempA-x))
            print("error at each iteration",starting_tol)

        print(randomx)
        t1_finish = time.process_time()
        print('Time it took to do the jacobi iterative method: ', (t1_finish-
t1_start), '\n')

        abs_error = la.norm(x - randomx)
        print("Absolute error: " + str(abs_error))
        rel_error = (la.norm(x - randomx)) / la.norm(randomx)
        print("Percent error:" + "{:10.4f}".format(rel_error))
```

Script Output:

```
Condition Number: 699.47997208

The given linear system of equations:

A=

[[12.     -6.      1.33333333 0.      0.      0.
  0.      0.      0.      0.      0.     ]
 [-4.      6.     -4.      1.      0.      0.
  0.      0.      0.      0.      0.     ]
 [ 1.     -4.      6.     -4.      1.      0.
  0.      0.      0.      0.      0.     ]
 [ 0.      1.     -4.      6.     -4.      1.
  0.      0.      0.      0.      0.     ]
 [ 0.      0.      1.     -4.      6.     -4.
  1.      0.      0.      0.      0.     ]
 [ 0.      0.      0.      1.     -4.      6.
 -4.      1.      0.      0.      0.     ]
 [ 0.      0.      0.      0.      1.     -4.
  6.     -4.      1.      0.      0.     ]
 [ 0.      0.      0.      0.      0.      1.
 -4.      6.     -4.      1.      0.     ]
 [ 0.      0.      0.      0.      0.      0.
  1.     -4.      6.     -4.      1.     ]
 [ 0.      0.      0.      0.      0.      0.
  0.      1.     -4.      6.     -4.     ]
 [ 0.      0.      0.      0.      0.      0.
  0.      0.      1.33333333 -6.     12.     ]]

b=

[-40.66666667  46.     -50.      37.      -5.
 -30.      37.     -13.     -17.      28.
 -11.33333333]
```

```
First find the optimal solution for x using the formulas specified in the jacobi iterative method:
Iteration # 0    Value of x: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
error at each iteration 8.333333333333332
Iteration # 1    Value of x: [-3.38888889  7.66666667 -8.33333333  6.16666667 -0.83333333 -5.
  6.16666667 -2.16666667 -2.83333333  4.66666667 -0.94444444]
[-3.38888889  7.66666667 -8.33333333  6.16666667 -0.83333333 -5.
  6.16666667 -2.16666667 -2.83333333  4.66666667 -0.94444444]
error at each iteration 8.842592592592592
Iteration # 2    Value of x: [ 1.37037037 -1.17592593  1.59259259 -0.38888889  0.30555556 -
2.11111111
  2.         0.11111111 -2.03703704  2.50925926  1.7037037 ]
[ 1.37037037 -1.17592593  1.59259259 -0.38888889  0.30555556 -2.11111111
  2.         0.11111111 -2.03703704  2.50925926  1.7037037 ]
error at each iteration 11.248456790123454
 Iteration # 8    Value of x: [  8.83827098 -15.94048291  23.68146416 -24.04114009  22.7384091
  -21.83463866  14.95856116  -6.69506491   1.24565393   4.75801598
    1.43533241]
 [  8.83827098 -15.94048291  23.68146416 -24.04114009  22.7384091
  -21.83463866  14.95856116  -6.69506491   1.24565393   4.75801598
    1.43533241]
 error at each iteration 63.93199283415473
 Iteration # 9    Value of x: [-13.99040414  33.35334678 -40.25052868  43.40910243 -37.85719005
   25.25401434 -16.85047955  11.48224718  -6.85701489   7.56983505
    1.29615755]
 [-13.99040414  33.35334678 -40.25052868  43.40910243 -37.85719005
   25.25401434 -16.85047955  11.48224718  -6.85701489   7.56983505
    1.29615755]
 error at each iteration 99.0821417723554
 [0, 7, 1, 8, 4, 2, 10, 6, 5, 9, 3]
 Time it took to do the jacobi iterative method:  0.0

 Absolute error: 125.98045345642355
 Percent error:   6.4206
```

For this complex of a matrix, the system does not converge if the loop does not have a fixed number of iterations. As the number of iterations go up, the error gets larger, so the loop will never come to a convergence based on the stopping criteria of the error being smaller than the tolerance. These results are showing a fixed number of 9 iterations, and the error shown is

already a significantly large value. The Jacobi Iterative method code is too rudimentary for this complex of a matrix.

3.2.3 Computer Exercise 10b

Script Input:

```
if question_10b==1:
    error_list=[]
    # A is mxm matrix
    for m in range(8,12):
        # Choose Diagonals
        diagonals = [[1 for _ in range(m-2)],
                     [-4 for _ in range(m-1)],
                     [6 for _ in range(m)],
                     [-4 for _ in range(m-1)],
                     [1 for _ in range(m-2)]]

        # Generate A
        A = scipy.sparse.diags(diagonals, [-2, -1, 0, 1, 2]).toarray()
        # Set diferent values at corners
        A[0, 0:3] = [12., -6., (4/3)]
        A[-1, -3:] = [(12/25), (24/25), (12/25)]
        A[-2, -3:] = [(-93/25), (111/25), (-43/25)]

        randomx = random.sample(range(m), int(m))
        b = np.matmul(A,randomx)


        x = np.zeros(m)               # vector x with elements of 0 (initial setu
p)

        print('The given linear system of equations: ', '\n', 'A= ', '\n', A)
        print('b= ', '\n', b, '\n')


        # relative error total and rel_sol values to stop while loop at certain e
rror margin
        tolerance = 1.0e-10
        starting_tol = 1.0
        i = 0


        print('''First find the optimal solution for x using the formulas specifi
ed in the jacobi iterative method: ''')
        t1_start = time.time()
        for h in range(10):
        #while (starting_tol > tolerance):
            print('Iteration # %s\t Value of x: %s' % (i, x))
            i+=1
            # print each time the updated solution
            tempA = x - 0
```

```
            i+=1
            # print each time the updated solution
            tempA = x - 0
            print(x)
            x = Jacobi_Iterative_Method(A, b, x)
            starting_tol = np.abs(np.max(tempA-x))
            print("error at each iteration",starting_tol)

        print(randomx)
        t1_finish = time.process_time()
        print('Time it took to do the jacobi iterative method: ', (t1_finish-
t1_start), '\n')

        abs_error = la.norm(x - randomx)
        print("Absolute error: " + str(abs_error))
        rel_error = (la.norm(x - randomx)) / la.norm(randomx)
        print("Percent error:" + "{:10.4f}".format(rel_error))
```

Script Output:

```
Condition Number: 914.90557479

The given linear system of equations:

A=

[[12.      -6.       1.33333333 0.       0.       0.
  0.       0.       0.       0.       0.      ]
 [-4.       6.      -4.       1.       0.       0.
  0.       0.       0.       0.       0.      ]
 [ 1.      -4.       6.      -4.       1.       0.
  0.       0.       0.       0.       0.      ]
 [ 0.       1.      -4.       6.      -4.       1.
  0.       0.       0.       0.       0.      ]
 [ 0.       0.       1.      -4.       6.      -4.
  1.       0.       0.       0.       0.      ]
 [ 0.       0.       0.       1.      -4.       6.
 -4.       1.       0.       0.       0.      ]
 [ 0.       0.       0.       0.       1.      -4.
  6.      -4.       1.       0.       0.      ]
 [ 0.       0.       0.       0.       0.       1.
 -4.       6.      -4.       1.       0.      ]
 [ 0.       0.       0.       0.       0.       0.
  1.      -4.       6.      -4.       1.      ]
 [ 0.       0.       0.       0.       0.       0.
  0.       1.      -3.72     4.44    -1.72    ]
 [ 0.       0.       0.       0.       0.       0.
  0.       0.       0.48     0.96     0.48    ]]

b=

[ 28.   21.  -13.  -20.   40.  -35.    6.   24.  -22.    2.84
  9.12]
```

Condition Number: 914.90557479

The given linear system of equations:

A=

```
[[12.      -6.       1.33333333  0.       0.       0.
  0.       0.       0.       0.       0.      ]
 [-4.       6.      -4.       1.       0.       0.
  0.       0.       0.       0.       0.      ]
 [ 1.      -4.       6.      -4.       1.       0.
  0.       0.       0.       0.       0.      ]
 [ 0.       1.      -4.       6.      -4.       1.
  0.       0.       0.       0.       0.      ]
 [ 0.       0.       1.      -4.       6.      -4.
  1.       0.       0.       0.       0.      ]
 [ 0.       0.       0.       1.      -4.       6.
 -4.       1.       0.       0.       0.      ]
 [ 0.       0.       0.       0.       1.      -4.
  6.      -4.       1.       0.       0.      ]
 [ 0.       0.       0.       0.       0.       1.
 -4.       6.      -4.       1.       0.      ]
 [ 0.       0.       0.       0.       0.       0.
  1.      -4.       6.      -4.       1.      ]
 [ 0.       0.       0.       0.       0.       0.
  0.       1.      -3.72     4.44    -1.72    ]
 [ 0.       0.       0.       0.       0.       0.
  0.       0.       0.48     0.96     0.48    ]]
```

b=

```
[ 28.   21.  -13.  -20.   40.  -35.    6.   24.  -22.    2.84
  9.12]
```

```
First find the optimal solution for x using the formulas specified in the jacobi iterative method:
Iteration # 0   Value of x: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
error at each iteration 5.833333333333333
Iteration # 1   Value of x: [ 2.33333333  3.5       -2.16666667 -3.33333333  6.66666667 -5.83333333
  1.        4.       -3.66666667  0.63963964 19.       ]
[ 2.33333333  3.5       -2.16666667 -3.33333333  6.66666667 -5.83333333
  1.        4.       -3.66666667  0.63963964 19.       ]
error at each iteration 5.916666666666666
Iteration # 2   Value of x: [ 4.32407407  4.16666667 -3.55555556  0.05555556  0.75      -0.83333333
 -0.72222222  3.08783784 -3.90690691  4.02702703 21.38738739]
[ 4.32407407  4.16666667 -3.55555556  0.05555556  0.75      -0.83333333
 -0.72222222  3.08783784 -3.90690691  4.02702703 21.38738739]
error at each iteration 6.534534534534538

 Iteration # 8   Value of x: [  0.09650425  20.37438908 -20.54489524  20.69759005 -23.5959237
   21.4969607  -22.84329916  22.23410237 -15.23107982  12.99342359
   15.49031947]
 [  0.09650425  20.37438908 -20.54489524  20.69759005 -23.5959237
   21.4969607  -22.84329916  22.23410237 -15.23107982  12.99342359
   15.49031947]
 error at each iteration 65.44505800575683
 Iteration # 9   Value of x: [ 14.80329401 -13.581859    29.13122266 -39.73910425  42.0277329
 -43.94809731  36.6252093  -27.1313167   21.04384725 -11.12846167
   8.24423264]
 [ 14.80329401 -13.581859    29.13122266 -39.73910425  42.0277329
 -43.94809731  36.6252093  -27.1313167   21.04384725 -11.12846167
   8.24423264]
 error at each iteration 102.11193926245532
 [7, 10, 3, 1, 6, 0, 5, 8, 2, 4, 9]
 Time it took to do the jacobi iterative method:  0.03125

 Absolute error: 148.28425405619757
 Percent error:   7.5573
```

It is the same case for part b of this problem. For this complex of a matrix, the system does not converge if the loop does not have a fixed number of iterations. As the number of iterations go up, the error gets larger, so the loop will never come to a convergence based on the stopping criteria of the error being smaller than the tolerance. These results are showing a fixed number of 9 iterations, and the error shown is already a significantly large value. The Jacobi Iterative method code is too rudimentary for this complex of a matrix.

3.3 Gauss-Seidel Iterative Method

3.3.1 Computer Exercise 3

Script Input:

```
'''Here are some sample inputs'''

# initial solution depending on n(here n=3)
x = [0, 0, 0]
a = [[4, 1, 2],[3, 5, 1],[1, 1, 3]]
b = [4,7,3]
xPrime = [0.5,1.0, 0.5]


# initial solution depending on n(here n=4)
# x = [0,0,0,0]
# a = [[7,3,-1,2],[3,8,1,-4],[-1,1,4,-1],[2,-4,-1,6]]
# b = [-1,0,-3,1]
# xPrime = [-1,1,-1,1]

# relative error total and rel_sol values to stop while loop at certain error margin
reltol=1.0e-5
rel_sol = 1.0

#iteration counter
i = 0
```

```
'''Here we actually iterate through solution
through the algorithm as it stumbles closer
and closer to the actual solution'''

tic=time.time()
#loop run for m times depending on the error value magnitude.
while (rel_sol > reltol):
    print('Iteration # %s\t Value of x: %s' % (i, x))
    i+=1
    #print each time the updated solution
    x = seidel(a, x, b)
    rel_sol = np.max(np.subtract(xPrime, x))

toc=time.time()
print("Gauss-Seidel solver time:\t\t{0:0.10f} seconds.".format(toc-tic))
```

```
# initial solution depending on n(here n=4)
x = [0,0,0,0]
a = [[7,3,-1,2],[3,8,1,-4],[-1,1,4,-1],[2,-4,-1,6]]
b = [-1,0,-3,1]
xPrime = [-1,1,-1,1]
```

Script Output:

```
Iteration # 0    Value of x: [0, 0, 0, 0]
Iteration # 1    Value of x: [-0.14285714285714285, 0.05357142857142857, -0.7991071428571428, 0.11681547619047616]
Iteration # 2    Value of x: [-0.31335034013605434, 0.2758025085034013, -0.8680843431122448, 0.3103043951955782]
Iteration # 3    Value of x: [-0.4737286655733722, 0.4413109900768343, -0.9011838151136571, 0.4685862460567374]
Iteration # 4    Value of x: [-0.5946127539225193, 0.5699208826385206, -0.9239868476260756, 0.5908203651288407]
Iteration # 5    Value of x: [-0.6879128893999027, 0.6688758720426433, -0.9414920990784263, 0.6849728613153254]
Iteration # 6    Value of x: [-0.7597236339767154, 0.7450693057837343, -0.9549550196112809, 0.7574615785795146]
Iteration # 7    Value of x: [-0.8150122991602161, 0.8037297789262485, -0.9653201248767375, 0.8132705981914481]
Iteration # 8    Value of x: [-0.8575786654340541, 0.8488923142430865, -0.9733000953714231, 0.8562377487448387]
Iteration # 9    Value of x: [-0.8903503622271943, 0.8836627721290451, -0.9794438464028501, 0.8893179944279531]
Iteration # 10   Value of x: [-0.9155811645208416, 0.9104324147096483, -0.9841738962006342, 0.9147863486132737]
Iteration # 11   Value of x: [-0.9350062625080181, 0.931042259772223, -0.9878155434167419, 0.9343943367813643]
Iteration # 12   Value of x: [-0.9499615708994485, 0.946909700405068, -0.9906192336307881, 0.9494904516313968]
Iteration # 13   Value of x: [-0.9614756054441121, 0.9591259820610889, -0.9927777839684512, 0.9611128925273548]
Iteration # 14   Value of x: [-0.9703402164580611, 0.9685312504315067, -0.9944396435905533, 0.970060965175266]
Iteration # 15   Value of x: [-0.9771650464622293, 0.9757723304597881, -0.9957191029366879, 0.9769500519711539]
Iteration # 16   Value of x: [-0.9824194568940514, 0.9813472101879321, -0.9967041537777075, 0.982253933460354]
Iteration # 17   Value of x: [-0.9864648073231731, 0.9856392886985803, -0.9974625406403498, 0.9863373714667195]
Iteration # 18   Value of x: [-0.989579307095647, 0.9889437434742712, -0.9980464197757997, 0.9894811947187633]
Iteration # 19   Value of x: [-0.9919771485194485, 0.9914878305261499, -0.9984959460817087, 0.9919016121769646]
Iteration # 20   Value of x: [-0.9938232374305839, 0.9934465133851649, -0.998842034659696, 0.9937650822903552]
Iteration # 21   Value of x: [-0.9952445341994144, 0.99495449580242, -0.9991084869278698, 0.9951997607801065]
Iteration # 22   Value of x: [-0.9963387851279062, 0.9961154856790018, -0.9993136275067004, 0.9963043142441864]
Iteration # 23   Value of x: [-0.9971812447188685, 0.9970093273300065, -0.9994715644511722, 0.9971547057177652]
Iteration # 24   Value of x: [-0.997829851125246, 0.9976974925872464, -0.9995931594986818, 0.9978094188501326]
Iteration # 25   Value of x: [-0.9983292107086694, 0.9982273083781525, -0.9996867750591723, 0.9983134799784629]
Iteration # 26   Value of x: [-0.9987136657357937, 0.9986352115225505, -0.9997588493199703, 0.9987015547069698]
Iteration # 27   Value of x: [-0.9990096561859373, 0.9989492545882077, -0.9998143390167937, 0.9990003319513185]
Gauss-Seidel solver time:              0.0092840195 seconds.
```

3.3.2 Computer Exercise 10a

Script Input:

```
for m in range(8,12):
    # Choose Diagonals
    diagonals = [[1 for _ in range(m-2)],
                 [-4 for _ in range(m-1)],
                 [6 for _ in range(m)],
                 [-4 for _ in range(m-1)],
                 [1 for _ in range(m-2)]]
    # Generate A
    A = scipy.sparse.diags(diagonals, [-2, -1, 0, 1, 2]).toarray()
    # Set diferent values at corners
    A[0, 0:3] = [12., -6., (4/3)]
    A[-1, -3:] = [(4/3), -6., 12.]


    '''We are creating a randomized x vector so that we may
    multiply Ax and get a b'''
    # create randomized x vector
    xtemp = random.sample(range(100), int(m))
    b = np.matmul(A,xtemp)

    # our intitial guess
    x = np.array(np.zeros(m))

    # relative error total and rel_sol values to stop while loop at certain err
    rel_sol = 1.0

    # list of errors
    errors = []

    #iteration counter
    i = 0

    '''Here we actually iterate through solution
    through the algorithm as it stumbles closer
    and closer to the actual solution of Q.CE 10 from 8.4'''
#     print(A)
#     print(b)

    tic=time.time()
    toc = 0
    # loop run for m times depending on the error value magnitude.
    while (i < 3):
        print('Iteration # %s\t Value of x: %s' % (i, x))
        i+=1

        if i == 3:
            toc=time.time()

        # print each time the updated solution
        tempA = x - 0
        x = seidel(A, x, b)
        rel_sol = np.abs(np.max(tempA-x))
        errors.append(rel_sol)


    print("For {}x{} Matrix".format(m,m))
    print('Actual Solution:',xtemp)
    condition_number = np.linalg.cond(A)
    print("Condition Number: " + "{:10.8f}".format(condition_number))
    print("\nGauss-Seidel solver time:\t\t{0:0.10f} seconds.".format(toc-tic))
    abs_error = np.linalg.norm(x - xtemp)
    print("Absolute error: " + str(abs_error))
    rel_error = (np.linalg.norm(x - xtemp)) / np.linalg.norm(xtemp)
    print("Percent error:" + "{:10.4f}".format(rel_error * 100))
    print("")
    print("\n\n\n")

print("*"*90)
```

```
[[12.         -6.          1.33333333  0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [-4.          6.         -4.          1.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 1.         -4.          6.         -4.          1.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          1.         -4.          6.         -4.          1.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.          1.         -4.          6.         -4.
   1.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          1.         -4.          6.
  -4.          1.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          1.         -4.
   6.         -4.          1.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          1.
  -4.          6.         -4.          1.          0.        ]
 [ 0.          0.          0.          0.          0.          0.
   1.         -4.          6.         -4.          1.        ]
 [ 0.          0.          0.          0.          0.          0.
   0.          1.         -4.          6.         -4.        ]
 [ 0.          0.          0.          0.          0.          0.
   0.          0.          1.33333333 -6.         12.       ]]
```

And

```
[ 640.         -108.         -201.          138.          195.
 -160.         -237.          374.         -147.          -72.
  314.66666667]
```

Script Output:

```
Iteration # 0    Value of x: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Iteration # 1    Value of x: [ 53.33333333  17.55555556 -30.68518519  -0.38271605  37.3590535
  -1.69684499 -46.85773891  31.37764822   4.2280553  -14.4109045
  18.54698605]
Iteration # 2    Value of x: [ 65.52057613   5.28737997 -47.37682899  15.72306051  57.55657166
 -27.38422962 -47.13515876  40.69445338  -2.21293863  -7.89304395
  22.52158232]
Iteration # 3    Value of x: [ 61.24111543 -11.37765246 -50.40267581  34.22957758  53.31987107
 -35.03086362 -44.24209557  38.51729513  -0.463747    -3.71432565
  24.41658684]
Iteration # 4    Value of x: [ 53.24480442 -21.81017719 -42.98117899  39.36596819  49.92728213
 -35.85708617 -45.97044988  37.97243738   1.93105166  -0.76364723
  25.62583731]
Iteration # 5    Value of x: [ 47.20393129 -21.7458265  -37.94177445  40.5908239   49.64119588
 -37.31337955 -47.65600271  38.19620377   4.12673193   1.4690122
  26.49820255]
Iteration # 737  Value of x: [56.98476382  7.94886094  2.90591077 83.86757992 91.84255503  5.8357956
  -0.15178194 87.87686423 46.91550928 27.95557843 34.98717707]
Iteration # 738  Value of x: [56.98488483  7.94926708  2.90665802 83.86863159 91.84380544  5.8370997
  -0.1505765  87.87784216 46.9161803  27.95593122 34.98727891]
Iteration # 739  Value of x: [56.98500487  7.94967     2.90739934 83.8696749  91.84504593  5.83839344
  -0.14938064 87.87881233 46.91684599 27.95628121 34.98737994]
Iteration # 740  Value of x: [56.98512396  7.95006971  2.90813476 83.87070994 91.84627656  5.83967691
  -0.14819427 87.8797748  46.91750639 27.95662842 34.98748017]
Actual Solution: [57, 8, 3, 84, 92, 6, 0, 88, 47, 28, 35]
```

3.3.3 Computer Exercise 10b

Script Input:

```
for m in range(8,12):
    # Choose Diagonals
    diagonals = [[1 for _ in range(m-2)],
                 [-4 for _ in range(m-1)],
                 [6 for _ in range(m)],
                 [-4 for _ in range(m-1)],
                 [1 for _ in range(m-2)]]
    # Generate A
    A = scipy.sparse.diags(diagonals, [-2, -1, 0, 1, 2]).toarray()
    # Set diferent values at corners
    A[0, 0:3] = [12., -6., (4/3)]
    A[-1, -3:] = [(12/25), (24/25), (12/25)]
    A[-2, -3:] = [(-93/25), (111/25), (-43/25)]


    '''We are creating a randomized x vector so that we may
    multiply Ax and get a b'''
    # create randomized x vector
    xtemp = random.sample(range(100), int(m))
    b = np.matmul(A,xtemp)

    # our intitial guess
    x = np.array(np.zeros(m))

    # relative error total and rel_sol values to stop while loop at certain error margin
    reltol = 1.0e-4
    rel_sol = 1.0

    # list of errors
    errors = []

    #iteration counter
    i = 0

    '''Here we actually iterate through solution
    through the algorithm as it stumbles closer
    and closer to the actual solution of Q.CE 10 from 8.4'''

#     print(A)
#     print(b)
    tic=time.time()
    toc = 0
    # loop run for m times depending on the error value magnitude.
    while (i < 3):
        #rel_sol > reltol or
        print('Iteration # %s\t Value of x: %s' % (i, x))
        i+=1


        if i == 3:
            toc=time.time()
        # print each time the updated solution
        tempA = x - 0
        x = seidel(A, x, b)
        rel_sol = np.abs(np.max(tempA-x))
        errors.append(rel_sol)

#     print('Iteration # %s\t Value of x: %s' % (i, x))
    print('Actual Solution:',xtemp)
    print("\nGauss-Seidel solver time:\t\t{0:0.10f} seconds.".format(toc-tic))
    condition_number = np.linalg.cond(A)
    print("Condition Number: " + "{:10.8f}".format(condition_number))
    abs_error = np.linalg.norm(x - xtemp)
    print("Absolute error: " + str(abs_error))
    rel_error = (np.linalg.norm(x - xtemp)) / np.linalg.norm(xtemp)
    print("Percent error:" + "{:10.4f}".format(rel_error * 100))
    print("")
    print("\n\n\n")

print("*"*90)
```

```
[[12.         -6.          1.33333333  0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [-4.          6.         -4.          1.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 1.         -4.          6.         -4.          1.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          1.         -4.          6.         -4.          1.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.          1.         -4.          6.         -4.
   1.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          1.         -4.          6.
  -4.          1.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          1.         -4.
   6.         -4.          1.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          1.
  -4.          6.         -4.          1.          0.        ]
 [ 0.          0.          0.          0.          0.          0.
   1.         -4.          6.         -4.          1.        ]
 [ 0.          0.          0.          0.          0.          0.
   0.          1.         -3.72        4.44       -1.72      ]
 [ 0.          0.          0.          0.          0.          0.
   0.          0.          0.48        0.96        0.48      ]]
```

And

```
[ 906.   -362.    423.   -487.    288.     -9.    -75.     43.    -17.
   -8.36   31.2 ]
```

Script Output:

```
Iteration # 0    Value of x: [0. 0. 0. 0. 0. 0. 0. 0.]
Iteration # 1    Value of x: [ 61.94444444 -26.2037037   39.20679012 -15.99485597  -2.69770233
 -11.63265889  17.87932686 191.87400518]
Iteration # 2    Value of x: [ 44.48628258  -9.03880887  43.34612673 -15.95580577 -14.09655204
 -39.29785002  71.597261   112.10332802]
Iteration # 3    Value of x: [ 52.60880371  -0.87074541  49.3635906  -16.29387505 -42.72129199
  -9.21759615  72.34449516  80.52860583]
Iteration # 4    Value of x: [ 56.02422834   5.4741918   57.56972177 -35.97714609 -37.28219764
   3.44962172  69.5008992   73.54857987]
Iteration # 5    Value of x: [ 58.28490459  15.73260859  50.00319051 -41.21637647 -30.59518479
   8.04844238  69.14390716  69.66374329]

                                        .
                                        .
                                        .

Iteration # 1138         Value of x: [93.99662921 61.98861128 84.9788896  20.97004914 65.96412277 66.96244643
 69.96559423 27.97340995 17.98513548 14.99962407 53.01561638]
Iteration # 1139         Value of x: [93.99665124 61.98868571 84.97902756 20.97024486 65.96435723 66.96269184
 69.96581907 27.97358372 17.98523262 14.99962653 53.01551432]
Iteration # 1140         Value of x: [93.99667312 61.98875964 84.97916461 20.97043931 65.96459015 66.96293565
 69.96604245 27.97375635 17.98532912 14.99962897 53.01541294]
Iteration # 1141         Value of x: [93.99669487 61.9888331  84.97930077 20.97063249 65.96482156 66.96317786
 69.96626436 27.97392785 17.985425   14.99963139 53.01531222]
Iteration # 1142         Value of x: [93.99671646 61.98890608 84.97943604 20.97082441 65.96505145 66.9634185
 69.96648482 27.97409823 17.98552024 14.9996338  53.01521215]
Actual Solution: [94, 62, 85, 21, 66, 67, 70, 28, 18, 15, 53]
```

3.4 SOR Iterative Method

3.4.1 Computer Exercise 3

Script Input Truncated:

```
'''
  ***Exercise 8.4.3 Setup and Data***
'''
time_entire = 0
time_entire_temp = time.process_time()

# Size of the square matrix nxn
size = 4

# Array for plotting
plot_result = []

# Array of Relaxation Factor's to compare
# Convergence is 0 < w < 2
# Fails to converge outside the relaxation range (0,2)
relax_factor = np.array([.1,.2,.3,.4,.5,.6,.7,.8,.9,1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9])

# Initial Data from computer exercise 8.4
b_vector = np.array([-1,0,-3,1])
A_matrix = np.array([[7,3,-1,2],[3,8,1,-4],[-1,1,4,-1],[2,-4,-1,6]])


# Step through all relax_factors, call sor_method and print the results
for i in range(len(relax_factor)):
  total_time = 0
  # Starting solution vector, the standard initial guess is the zero vector
  x_vector = np.zeros(size)
  time_temp = time.process_time()
  # SOR method call. The iterative solution vector is pass-by-reference so only returns iterations
  result = sor_method(A_matrix, b_vector, x_vector, relax_factor[i], size)
  total_time += time.process_time() - time_temp
  abs_error = (la.norm(x_vector) - la.norm(exact_solution))
  rel_error = (la.norm(x_vector)-la.norm(exact_solution)) / la.norm(exact_solution)
```

Script Output:

```
SOR Method — Computer Exercise 8.4.3
-------------------------------------
Iterations: 475  Relaxation Factor: 0.1
Solution: [−0.99803757  0.99774643 −0.99947863  0.99765557]
Absolute error: −0.0035403592071714396
Relative error: −0.0017701796035857204
Time to compute: 0.168914 seconds

Iterations: 253  Relaxation Factor: 0.2
Solution: [−0.99909463  0.99896504 −0.99976396  0.99892885]
Absolute error: −0.0016236466263652716
Relative error: −0.000811823313182636
Time to compute: 0.04112300000000002 seconds

Iterations: 170  Relaxation Factor: 0.3
Solution: [−0.99943844  0.99936133 −0.99985668  0.99934281]
Absolute error: −0.0010003244859100313
Relative error: −0.0005001622429550159
Time to compute: 0.01370300000000002 seconds

Iterations: 125  Relaxation Factor: 0.4
Solution: [−0.99959497  0.99954201 −0.99989911  0.9995318 ]
Absolute error: −0.0007160286752558864
Relative error: −0.000358014337627943 3
Time to compute: 0.011066999999999938 seconds

Iterations: 97   Relaxation Factor: 0.5
Solution: [−0.99969835  0.99966115 −0.99992694  0.99965617]
Absolute error: −0.0005286806544158029
Relative error: −0.0002643403272079015 7
Time to compute: 0.007978999999999958 seconds

Iterations: 78   Relaxation Factor: 0.6
Solution: [−0.99977893  0.99975355 −0.99994818  0.99975207]
Absolute error: −0.0003836296874963896
Relative error: −0.0001918148437481948 5
Time to compute: 0.006408999999999887 seconds

Iterations: 63   Relaxation Factor: 0.7
Solution: [−0.99981538  0.99979601 −0.99995838  0.99979686]
Absolute error: −0.0003166818233875101 6
Relative error: −0.0001583409116937551 4
Time to compute: 0.004712999999999967 seconds

Iterations: 52   Relaxation Factor: 0.8
Solution: [−0.9998591   0.99984598 −0.99996971  0.99984846]
Absolute error: −0.00023837912521762483
Relative error: −0.0001191895626088124 6
Time to compute: 0.004480999999999957 seconds

Iterations: 43   Relaxation Factor: 0.9
Solution: [−0.99989098  0.99988238 −0.99997791  0.99988598]
Absolute error: −0.0001813761221496346
Relative error: −9.068806107481733e−05
Time to compute: 0.0037689999999999113 seconds

Iterations: 36   Relaxation Factor: 1.0
Solution: [−0.99992754  0.99992312 −0.99998642  0.99992686]
Absolute error: −0.0001180346942835086 6
Relative error: −5.901734714175435e−05
Time to compute: 0.0030049999999999244 seconds

Iterations: 29   Relaxation Factor: 1.1
Solution: [−0.99993923  0.99993694 −0.99998977  0.9999415 ]
Absolute error: −9.627863660188751e−05
Relative error: −4.813931830094377e−05
Time to compute: 0.002266999999999797 seconds

Iterations: 23   Relaxation Factor: 1.2
Solution: [−0.99995571  0.99995554 −0.99999371  0.99996023]
Absolute error: −6.740492751799998e−05
Relative error: −3.3702463759e−05
Time to compute: 0.001811000000000007 seconds

Iterations: 17   Relaxation Factor: 1.3
Solution: [−0.99997003  0.99997167 −0.99999701  0.9999763 ]
Absolute error: −4.2495199175540677e−05
Relative error: −2.1247599587770345e−05
Time to compute: 0.001587000000000005 seconds

Iterations: 12   Relaxation Factor: 1.4
Solution: [−1.00000439  0.99999997 −0.99999445  1.00000163]
Absolute error: 2.1636558655835358e−07
Relative error: 1.0818279327917683e−07
Time to compute: 0.0011713000000000756 seconds

Iterations: 16   Relaxation Factor: 1.5
Solution: [−1.00000161  0.99999137 −1.00000426  0.99999499]
Absolute error: −3.8833656292683855e−06
Relative error: −1.9417826463419283e−06
Time to compute: 0.0015080000000000648 seconds

Iterations: 21   Relaxation Factor: 1.6
Solution: [−0.99998779  0.99998657 −0.99998512  0.99999523]
Absolute error: −2.2643307541647673e−05
Relative error: −1.132165377082384e−05
Time to compute: 0.001875000000000071 seconds

Iterations: 31   Relaxation Factor: 1.7
Solution: [−1.00000173  0.99999991 −1.00001007  0.99999113]
Absolute error: 1.4206364629121282e−06
Relative error: 7.103182314560643e−07
Time to compute: 0.0023920000000001718 seconds

Iterations: 46   Relaxation Factor: 1.8
Solution: [−0.99999147  1.00002477 −0.99999073  1.00001273]
Absolute error: 9.848564510361868e−06
Relative error: 4.924282255180936e−06
Time to compute: 0.0034799999999999276 seconds

Iterations: 102  Relaxation Factor: 1.9
Solution: [−1.00000709  1.00001365 −1.00001716  0.99999415]
Absolute error: 1.6023068005077 69e−05
Relative error: 8.011534002538848e−06
Time to compute: 0.008315999999999999 seconds
```

3.4.2 Computer Exercise 10a

Script Input Truncated:

```
'''
  ***Part A Setup***
'''
# Initial size of sparse matrix m x m
m = 8
# Main loop - Grow sparse matrix on each loop upto 16 x 16
while m <= 11:
  # Setup Diagonals
  diagonals = [[1 for _ in range(m-2)],
               [-4 for _ in range(m-1)],
               [6 for _ in range(m)],
               [-4 for _ in range(m-1)],
               [1 for _ in range(m-2)]]

  # Generate A
  A = scipy.sparse.diags(diagonals, [-2, -1, 0, 1, 2]).toarray()

  # Initialize the nonrepeating values in the corners
  A[0, 0:3] = [12., -6., (4/3)]
  A[-1, -3:] = [(4/3), -6., 12.]
  condition_number = la.cond(A)

  # Create Random Exact solution x_vector
  exact_solution = np.random.rand(m)

  # Get the b_vector by multiplying A by x , i.e. b = Ax
  b_vector = np.matmul(A, exact_solution)

  # Step through all relax_factors, call sor_method and print the results
  for i in range(len(relax_factor)):
    total_time = 0
    # Starting solution vector, the standard initial guess is the zero vector
    # Is pass by reference so don't need to return the x_vector back
    x_vector = np.zeros(m)
    time_temp = time.process_time()
    result = sor_method(A, b_vector, x_vector, relax_factor[i], m)
    total_time += time.process_time() - time_temp
    abs_error = la.norm(x_vector - exact_solution)
    rel_error = (la.norm(x_vector - exact_solution)) / la.norm(exact_solution)
```

Script Output Truncated :

```
SOR Method - Computer Exercise 8.4.10
-------------------------------------
Part A

8x8 sparse matrix        Condition Number: 225.35149787107684
[[12.         -6.          1.33333333  0.          0.          0.          0.          0.        ]
 [-4.          6.         -4.          1.          0.          0.          0.          0.        ]
 [ 1.         -4.          6.         -4.          1.          0.          0.          0.        ]
 [ 0.          1.         -4.          6.         -4.          1.          0.          0.        ]
 [ 0.          0.          1.         -4.          6.         -4.          1.          0.        ]
 [ 0.          0.          0.          1.         -4.          6.         -4.          1.        ]
 [ 0.          0.          0.          0.          1.         -4.          6.         -4.        ]
 [ 0.          0.          0.          0.          0.          1.33333333 -6.         12.        ]]

Iterations: 2703         Relaxation factor: 0.1
Solution: [0.78048589 0.49799823 0.3881466  0.31495663 0.7031614  0.3798651  0.61509537 0.80702706]
Absolute error: 0.03849360900336782
Relative error: 0.0227152025457891
Time to compute: 0.17685400000000007 seconds

Iterations: 1554         Relaxation factor: 0.2
Solution: [0.78203896 0.50292411 0.39650772 0.32542249 0.71362718 0.38822603 0.62002106 0.80858004]
Absolute error: 0.018190259080355792
Relative error: 0.010734130419792818
Time to compute: 0.10807499999999992 seconds

Iterations: 1085         Relaxation factor: 0.3
Solution: [0.78255537 0.50456207 0.39928803 0.32890272 0.71710737 0.39100625 0.62165894 0.80909642]
Absolute error: 0.011438929541379893
Relative error: 0.006750149132982606
Time to compute: 0.07572200000000029 seconds

Iterations: 823  Relaxation factor: 0.4
Solution: [0.78281553 0.50538722 0.40068862 0.3306558  0.71886031 0.39240648 0.62248374 0.80935642]
Absolute error: 0.00803837965466343
Relative error: 0.004743473701820344
Time to compute: 0.055620000000000225 seconds

Iterations: 652  Relaxation factor: 0.5
Solution: [0.7829689  0.5058738  0.40151469 0.33168994 0.71989449 0.39323266 0.62297043 0.80950984]
Absolute error: 0.006032407531534495
Relative error: 0.0035597430967192803
Time to compute: 0.04567299999999985 seconds


Iterations: 261  Relaxation factor: 1.0
Solution: [0.78327966 0.50686013 0.40318957 0.3337867  0.72199086 0.39490661 0.62395589 0.80982024]
Absolute error: 0.0019674747232901963
Relative error: 0.0011610131655711237
Time to compute: 0.018100000000000005 seconds

Iterations: 220  Relaxation factor: 1.1
Solution: [0.78331006 0.50695656 0.4033531  0.33399099 0.72219454 0.39506869 0.62405093 0.80985005]
Absolute error: 0.0015724856684677588
Relative error: 0.0009279288532408229
Time to compute: 0.015826000000000118 seconds

Iterations: 184  Relaxation factor: 1.2
Solution: [0.78333424 0.50703348 0.40348383 0.33415457 0.72235777 0.39519863 0.6241271  0.80987392]
Absolute error: 0.0012564623080704304
Relative error: 0.0007414424513033436
Time to compute: 0.013939999999999841 seconds

Iterations: 152  Relaxation factor: 1.3
Solution: [0.78335473 0.50709883 0.40359505 0.3342938  0.72249664 0.39530901 0.62419167 0.8098941 ]
Absolute error: 0.0009880193437190744
Relative error: 0.0005830333941868856
Time to compute: 0.011051000000000144 seconds

Iterations: 123  Relaxation factor: 1.4
Solution: [0.78337303 0.5071573  0.40369461 0.33441828 0.72262046 0.39540703 0.62424872 0.80991182]
Absolute error: 0.0007489860774190847
Relative error: 0.0004419790945313047
Time to compute: 0.008439999999999781 seconds

Iterations: 96   Relaxation factor: 1.5
Solution: [0.78339148 0.507216   0.40379387 0.33454122 0.72274136 0.3955015  0.62430294 0.80992841]
Absolute error: 0.0005149615131309401
Relative error: 0.00030388044605097754
Time to compute: 0.006888000000000005 seconds

Iterations: 67   Relaxation factor: 1.6
Solution: [0.78340869 0.50727228 0.40389041 0.33466127 0.72285881 0.39559215 0.62435401 0.80994368]
Absolute error: 0.00029060577170526115
Relative error: 0.00017148740105618095
Time to compute: 0.004666000000000281 seconds
```

3.4.3 Computer Exercise 10b

Script Input Truncated:

```
'''
  *** Part B Setup ***
'''
# Initial size of sparse matrix mxm
m = 8
# Main loop - Grow sparse matrix on each loop
while m <= 11:
  # Setup Diagonals
  diagonals = [[1 for _ in range(m-2)],
               [-4 for _ in range(m-1)],
               [6 for _ in range(m)],
               [-4 for _ in range(m-1)],
               [1 for _ in range(m-2)]]

  # Generate A
  A = scipy.sparse.diags(diagonals, [-2, -1, 0, 1, 2]).toarray()

  # Initialize the nonrepeating values in the corners and in 2nd to last row
  A[0, 0:3] = [12., -6., (4/3)]
  A[-2, -4:] = [1, -(93/25), (111/25), -(43/25)]
  A[-1, -3:] = [(12/25), (24/25), (12/25)]
  condition_number = la.cond(A)
  print(str(m)+ "x"+ str(m) + " sparse matrix" + "\t" + " Condition Number: " + str(condition_number))
  print(A)

  # Create Random Exact solution x_vector
  exact_solution = np.random.rand(m)

  # Get the b_vector by multiplying A by x , i.e. b = Ax
  b_vector = np.matmul(A, exact_solution)
  # Step through all relax_factors, call sor_method and print the results
  for i in range(len(relax_factor)):
    total_time = 0

    # Starting solution vector, the standard initial guess is the zero vector
    # Is pass by reference so don't need to return the x_vector back
    x_vector = np.zeros(m)
    time_temp = time.process_time()
    result = sor_method(A, b_vector, x_vector, relax_factor[i], m)
    total_time += time.process_time() - time_temp
    abs_error = la.norm(x_vector - exact_solution)
    rel_error = (la.norm(x_vector - exact_solution)) / la.norm(exact_solution)
```

Script Output Truncated:

```
Part B

8x8 sparse matrix        Condition Number: 253.92560629603
[[12.         -6.          1.33333333  0.          0.          0.          0.          0.        ]
 [-4.          6.         -4.          1.          0.          0.          0.          0.        ]
 [ 1.         -4.          6.         -4.          1.          0.          0.          0.        ]
 [ 0.          1.         -4.          6.         -4.          1.          0.          0.        ]
 [ 0.          0.          1.         -4.          6.         -4.          1.          0.        ]
 [ 0.          0.          0.          1.         -4.          6.         -4.          1.        ]
 [ 0.          0.          0.          0.          1.         -3.72        4.44       -1.72      ]
 [ 0.          0.          0.          0.          0.          0.48        0.96        0.48      ]]
Iterations: 2635         Relaxation factor: 0.1
Solution: [0.69599056 0.06412408 0.45755978 0.42414541 0.06569642 0.73821924 0.73059521 0.30264673]
Absolute error: 0.039361765786827534
Relative error: 0.027120410013676554
Time to compute: 0.17237399999999958 seconds


Iterations: 1526         Relaxation factor: 0.2
Solution: [0.69772419 0.06956905 0.46664685 0.43515075 0.07589662 0.74468462 0.73092504 0.29543514]
Absolute error: 0.01852191389333651
Relative error: 0.012761670849466847
Time to compute: 0.10186599999999935 seconds


Iterations: 1067         Relaxation factor: 0.3
Solution: [0.69829887 0.07137409 0.46965938 0.43879934 0.07927833 0.74682812 0.73103438 0.29304427]
Absolute error: 0.01161306407805389
Relative error: 0.008001446404046209
Time to compute: 0.07731700000000075 seconds


Iterations: 809  Relaxation factor: 0.4
Solution: [0.69858871 0.07228449 0.47117877 0.44063943 0.08098372 0.74790897 0.73108948 0.29183882]
Absolute error: 0.008129018572625423
Relative error: 0.005600925474033769
Time to compute: 0.06696499999999972 seconds


Iterations: 250  Relaxation factor: 1.0
Solution: [0.69910856 0.07391817 0.47390616 0.4439429  0.084045   0.74984852 0.73118807 0.28967678]
Absolute error: 0.0018776301855120564
Relative error: 0.001293694514644556
Time to compute: 0.016079999999998762 seconds


Iterations: 208  Relaxation factor: 1.1
Solution: [0.69914005 0.07401746 0.47407224 0.44414424 0.08423152 0.74996651 0.73119399 0.28954557]
Absolute error: 0.001497666857056276
Relative error: 0.0010318983006817536
Time to compute: 0.013246999999999787 seconds


Iterations: 172  Relaxation factor: 1.2
Solution: [0.6991702  0.07411201 0.47422951 0.44433368 0.08440581 0.75007595 0.73119924 0.28942466]
Absolute error: 0.0011414797184198205
Relative error: 0.0007864839741565031
Time to compute: 0.011739000000000388 seconds


Iterations: 139  Relaxation factor: 1.3
Solution: [0.69919266 0.07418313 0.47434873 0.44447817 0.08453933 0.75016002 0.7312033  0.28933171]
Absolute error: 0.0008703659675290383
Relative error: 0.0005996855433055078
Time to compute: 0.01007900000000106 seconds


Iterations: 109  Relaxation factor: 1.4
Solution: [0.69921521 0.07425409 0.47446671 0.44461971 0.08466856 0.75024028 0.73120684 0.28924411]
Absolute error: 0.0006069109615941773
Relative error: 0.0004181640175740552
Time to compute: 0.006906999999999965 seconds


Iterations: 79   Relaxation factor: 1.5
Solution: [0.69923555 0.07431876 0.4745746  0.44474876 0.08478536 0.75031179 0.73120964 0.28916729]
Absolute error: 0.00037014423306089126
Relative error: 0.000255030818972268
Time to compute: 0.005866000000001037 seconds


Iterations: 67   Relaxation factor: 1.6
Solution: [0.69929276 0.07444831 0.47476931 0.44493801 0.08494091 0.75038515 0.73121287 0.28907951]
Absolute error: 2.101907781060347e-05
Relative error: 1.448222651951513e-05
Time to compute: 0.004267000000000465 seconds
```

3.5 Conjugate Gradient Method

3.5.1 Computer Exercise 3

Script Input:

```
# Here is the execution for question 3
if question3==1:
    A_row1 = [7, 3, -1, 2]
    A_row2 = [3, 8, 1, -4]
    A_row3 = [-1, 1, 4, -1]
    A_row4 = [2, -4, -1, 6]
    A_combined = [A_row1, A_row2, A_row3, A_row4]

    A = np.array(A_combined)
    b = np.array([-1, 0, -3, 1])

    print('The given linear system of equations: ', '\n', 'A= ', '\n', A)
    print('b= ', '\n', b, '\n')

    # Finding solution using the conjugate gradient method
    print('''First find the optimal solution for x using the formulas specified in the conjugate gradient method: ''')
    t1_start = time.process_time()
    x_using_formulas = conjugate_gradient(A, b)
    t1_finish = time.process_time()
    print(x_using_formulas)

    print('Time it took to do the conjugate gradient method: ', (t1_finish-t1_start), '\n')

    # Finding solution using the numpy package
    print('''Finally, find the exact optimal solution for x using the linlag.solve module from the numpy package: ''')
    t2_start = time.process_time()
    x_using_imported_package = np.linalg.solve(A, b)
    t2_finish = time.process_time()
    print(x_using_imported_package)

    # Finding absolute and relative error
    abs_error = la.norm(x_using_formulas -  x_using_imported_package)
    print("Absolute error: " + str(abs_error))
    rel_error = (la.norm(x_using_formulas -  x_using_imported_package)) / la.norm(x_using_imported_package)
    print("Percent error:" + "{:10.4f}".format(rel_error))
    print("")

    print('Time it took to do the numpy package: ', (t2_finish-t2_start), '\n')
    print('----------------------------------------------------------------------')
```

Once the given A matrix and the b vector are declared on top, find the optimal solution using the created conjugate_gradient function. To be able to compare methods, also print the absolute error and relative error between the actual and calculated solution. Also, print the processing time it took to do the conjugate gradient method.

Script Output:

```
The given linear system of equations:
 A=
 [[ 7  3 -1  2]
 [ 3  8  1 -4]
 [-1  1  4 -1]
 [ 2 -4 -1  6]]
b=
 [-1  0 -3  1]

First find the optimal solution for x using the formulas specified in the conjugate gradient method:
Iteration:  0
Iteration:  1
Iteration:  2
Iteration:  3
The optimal solution for x was found after this many iterations:  3
[-1.  1. -1.  1.]
Time it took to do the conjugate gradient method:  0.0

Finally, find the exact optimal solution for x using the linlag.solve module from the numpy package:
[-1.  1. -1.  1.]
Absolute error: 2.7194799110210365e-16
Percent error:    0.0000

Time it took to do the numpy package:  0.0

------------------------------------------------------------------
```

After 4 iterations, the system converges to x = [-1, 1, -1, 1], which is the same answer and the exact result given initially in the problem. The error is significantly less than the acceptable tolerance, and the system converged very quickly at a time of approximately 0.0 seconds.

3.5.2 Computer Exercise 10a

Script Input:

```
if question10a==1:
    # A is mxm matrix
    time_list = []

    for m in range(8,12):
        print('Matrix Size: ', m)
        # Choose Diagonals
        diagonals = [[1 for _ in range(m-2)],
                     [-4 for _ in range(m-1)],
                     [6 for _ in range(m)],
                     [-4 for _ in range(m-1)],
                     [1 for _ in range(m-2)]]
        # Generate A
        A = scipy.sparse.diags(diagonals, [-2, -1, 0, 1, 2]).toarray()
        # Set diferent values at corners
        A[0, 0:3] = [12., -6., (4/3)]
        A[-1, -3:] = [(4/3), -6., 12.]

        condition_number = la.cond(A)
        print("Condition Number: " + "{:10.8f}".format(condition_number))

        # Setting a random x solution to be able to generate a b to evaluate a the method
        randomx = random.sample(range(m), int(m))
        print('Set a random vector x: ', randomx)
        b = np.matmul(A, randomx)

        print('The given linear system of equations: ', '\n', 'A= ', '\n', A)
        print('b= ', '\n', b, '\n')

        # Finding solution using the conjugate gradient method
        print('This was the initial random solution given: ', randomx, '\n')
        print('''First find the optimal solution for x using the formulas specified in the conjugate gradient method: ''')
        t1_start = time.process_time()
        x_using_formulas = conjugate_gradient(A, b)
        t1_finish = time.process_time()
        print(x_using_formulas)
        print('Time it took to do the conjugate gradient method: ', (t1_finish-t1_start), '\n')

        # Find the absolute and relative error
        abs_error = la.norm(x_using_formulas - randomx)
        print("Absolute error: " + str(abs_error))
        rel_error = (la.norm(x_using_formulas - randomx)) / la.norm(randomx)
        print("Percent error:" + "{:10.4f}".format(rel_error))
        print("")

        # Append the times per size matrix into the empty list
        time_list.append(t1_finish-t1_start)
        print('-----------------------------------------------------------------------')

    print('Here is the final list of processing times as the matrix gets larger: \n', time_list)
```

To build the matrix A as per the problem's conditions of a banded matrix, the diagonals in the matrix were built first using for loops. Then the diagonals were injected into a sparse matrix using the scipy package. Once the sparse matrix is built, change values in the first and last rows to match the problem's specifications. Due to time constraints, the b vector could not be found as per the Euler-Bernoulli beam equation. So, to find the b vector, a random x vector was found and then multiplied the random x vector by the matrix X using numpy package's matmul function.

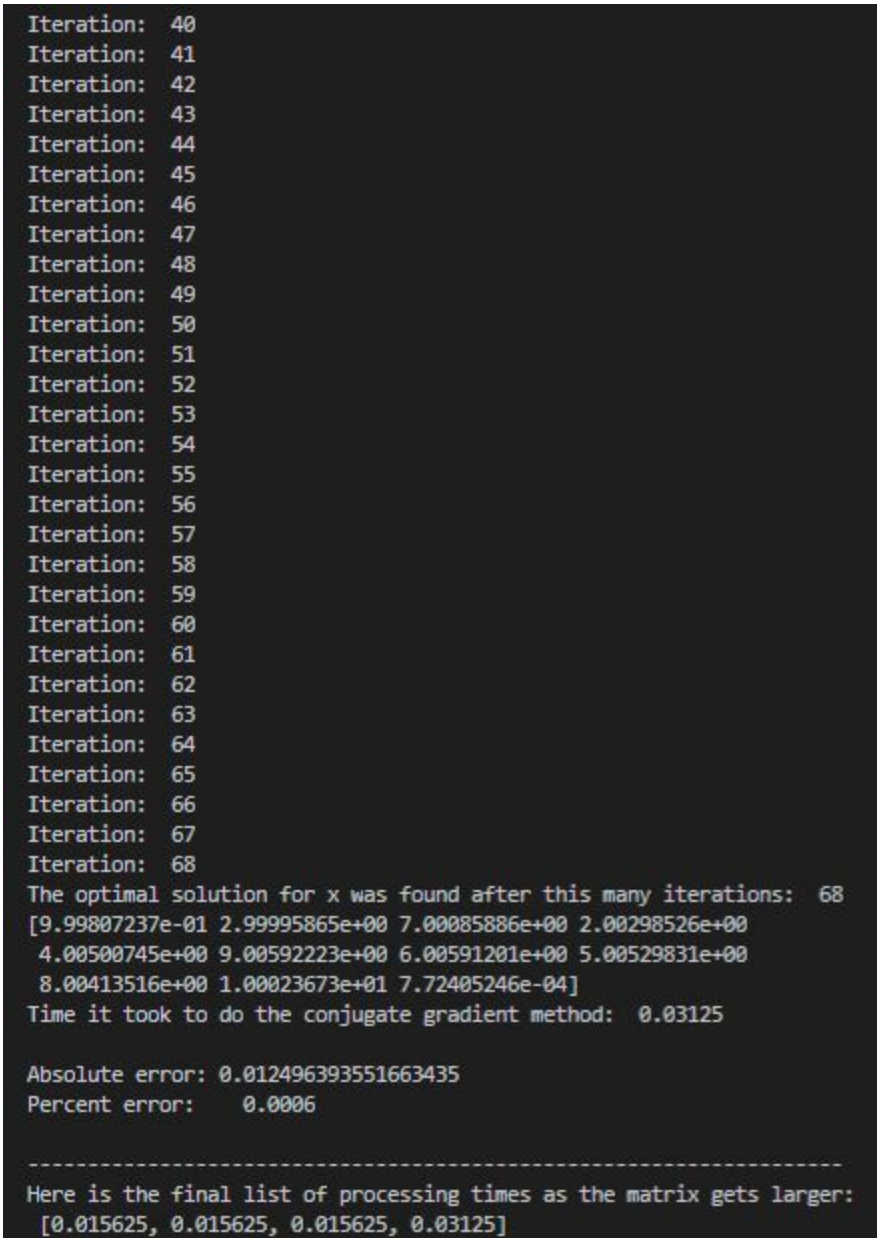
Once the given A matrix and the b vector are declared on top, find the optimal solution using the created conjugate_gradient function in a loop as the matrix gets larger. To be able to compare methods, also keep track of the absolute and relative error between the initial random x solution and calculated solution in a list to be able to evaluate the errors as the matrix gets larger. Also, keep a track of the processing times per mxm size matrix to be able to compare the 5 different methods.

Script Output:

```
----------------------------------------------------------------------
Matrix Size:  11
Condition Number: 699.47997208
Set a random vector x:  [1, 3, 7, 2, 4, 9, 6, 5, 8, 10, 0]
The given linear system of equations:
 A=
 [[12.         -6.          1.33333333  0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [-4.          6.         -4.          1.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 1.         -4.          6.         -4.          1.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          1.         -4.          6.         -4.          1.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.          1.         -4.          6.         -4.
   1.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          1.         -4.          6.
  -4.          1.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          1.         -4.
   6.         -4.          1.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          1.
  -4.          6.         -4.          1.          0.        ]
 [ 0.          0.          0.          0.          0.          0.
   1.         -4.          6.         -4.          1.        ]
 [ 0.          0.          0.          0.          0.          0.
   0.          1.         -4.          6.         -4.        ]
 [ 0.          0.          0.          0.          0.          0.
   0.          0.          1.33333333 -6.         12.        ]]
b=
 [  3.33333333 -12.          27.         -20.          -7.
  21.          -8.          -7.          -6.          33.
 -49.33333333]

This was the initial random solution given:  [1, 3, 7, 2, 4, 9, 6, 5, 8, 10, 0]
```

Here is the printed matrix A, vector b, and the initial random solution used to find the vector b.



For this complex of a matrix like for size 11x11, after 69 iterations the system converges. As the size of the matrix becomes larger, the absolute and relative error also gets larger. This tells the readers that as the matrix gets larger, the error also becomes larger. Also, one can notice that the processing times also become larger as the matrix becomes larger as expected.

3.5.3 Computer Exercise 10b

Script Input:

```
if question10b==1:
    # A is mxm matrix
    time_list = []

    for m in range(8,12):
        print('Matrix Size ', m)
        # Choose Diagonals
        diagonals = [[1 for _ in range(m-2)],
                     [-4 for _ in range(m-1)],
                     [6 for _ in range(m)],
                     [-4 for _ in range(m-1)],
                     [1 for _ in range(m-2)]]
        # Generate A
        A = scipy.sparse.diags(diagonals, [-2, -1, 0, 1, 2]).toarray()
        # Set diferent values at corners
        A[0, 0:3] = [12., -6., (4/3)]
        A[-1, -3:] = [(12/25), (24/25), (12/25)]
        A[-2, -3:] = [(-93/25), (111/25), (-43/25)]

        condition_number = la.cond(A)
        print("Condition Number: " + "{:10.8f}".format(condition_number))

        # Setting a random x solution to be able to generate a b to evaluate a the method
        randomx = random.sample(range(m), int(m))
        print('Set a random vector x: ', randomx)
        b = np.matmul(A, randomx)

        print('The given linear system of equations: ', '\n', 'A= ', '\n', A)
        print('b= ', '\n', b, '\n')

        # Finding solution using the conjugate gradient method
        print('This was the initial random solution given: ', randomx, '\n')
        print('First find the optimal solution for x using the formulas specified in the conjugate gradient method: ')
        t1_start = time.process_time()
        x_using_formulas = conjugate_gradient(A, b)
        t1_finish = time.process_time()
        print(x_using_formulas)
        print('Time it took to do the conjugate gradient method: ', (t1_finish-t1_start), '\n')


        abs_error = la.norm(x_using_formulas - randomx)
        print("Absolute error: " + str(abs_error))
        rel_error = (la.norm(x_using_formulas - randomx)) / la.norm(randomx)
        print("Percent error:" + "{:10.4f}".format(rel_error))
        print("")

        # Append the times per size matrix into the empty list
        time_list.append(t1_finish-t1_start)
        print('-----------------------------------------------------------------------')

    print('Here is the final list of processing times as the matrix gets larger: \n', time_list)
```

The difference between 10a and 10b is that the A matrix is slightly different. To build the matrix A as per the problem's conditions of a banded matrix, the diagonals in the matrix were built first using for loops. Then the diagonals were injected into a sparse matrix using the scipy package. Once the sparse matrix is built, change values in the first and last rows to match the problem's specifications. Due to time constraints, the b vector could not be found as per the Euler-Bernoulli beam equation. So, to find the b vector, a random x vector was found and then multiplied the random x vector by the matrix X using numpy package's matmul function.

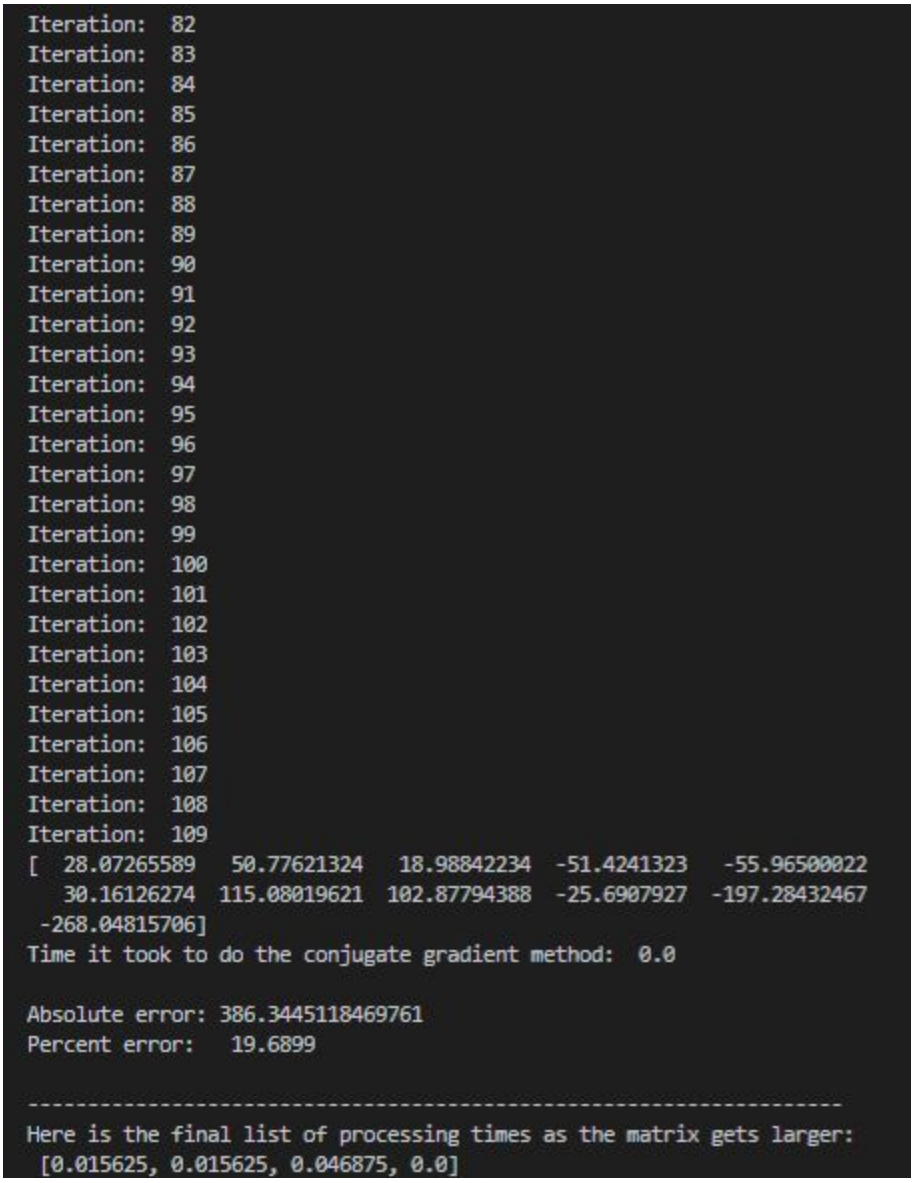
Once the given A matrix and the b vector are declared on top, find the optimal solution using the created conjugate_gradient function in a loop as the matrix gets larger. To be able to compare methods, also keep track of the absolute error and relative between the initial random x solution and calculated solution in a list to be able to evaluate the errors as the matrix gets larger. Also, keep a track of the processing times per mxm size matrix to be able to compare the 5 different methods.

Script Output:

```
----------------------------------------------------------------------
Matrix Size  11
Condition Number: 914.90557479
Set a random vector x:  [1, 4, 8, 3, 5, 9, 7, 2, 6, 0, 10]
The given linear system of equations:
 A=
 [[12.         -6.          1.33333333  0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [-4.          6.         -4.          1.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 1.         -4.          6.         -4.          1.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          1.         -4.          6.         -4.          1.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.          1.         -4.          6.         -4.
   1.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          1.         -4.          6.
  -4.          1.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          1.         -4.
   6.         -4.          1.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          1.
  -4.          6.         -4.          1.          0.        ]
 [ 0.          0.          0.          0.          0.          0.
   1.         -4.          6.         -4.          1.        ]
 [ 0.          0.          0.          0.          0.          0.
   0.          1.         -3.72        4.44       -1.72      ]
 [ 0.          0.          0.          0.          0.          0.
   0.          0.          0.48        0.96        0.48      ]]
b=
 [ -1.33333333  -9.          26.         -21.          -3.
  11.           9.         -31.          45.         -37.52
   7.68      ]

This was the initial random solution given:  [1, 4, 8, 3, 5, 9, 7, 2, 6, 0, 10]
```

Here is the printed matrix A, vector b, and the initial random solution used to find the vector b.

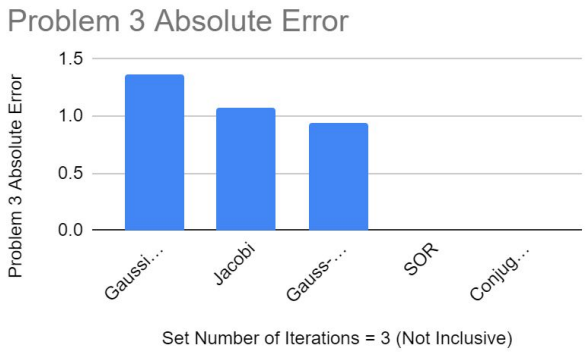


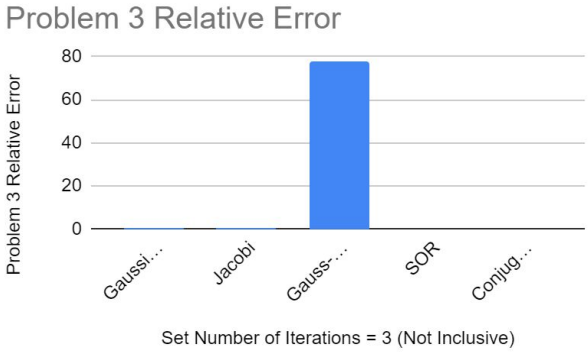
Considering the new matrix A, regardless of how many iterations are set, the system does not converge. As the size of the matrix becomes larger, both the absolute and relative error also gets large. This tells the readers that as the matrix gets larger, the error also becomes larger. Also, one can notice that the processing times also become larger as the matrix becomes larger as expected.

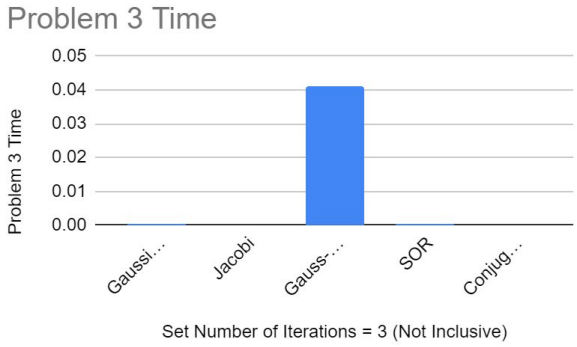
3.6 Final Comparisons

3.6.1 Computer Exercise 3

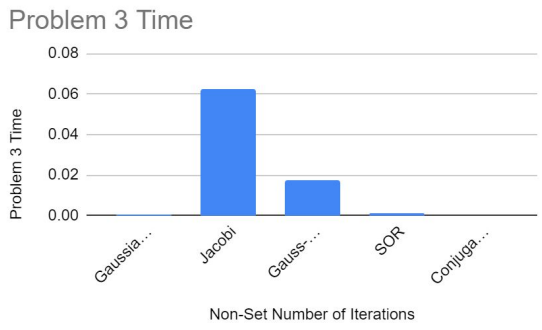
Below are the results for a set of only 3 iterations. This will show how each method runs from the start versus how they finish. In comparing only the first 3 iterations of each method, one can see which method is the quickest and has the least amount of error which is the SOR method.

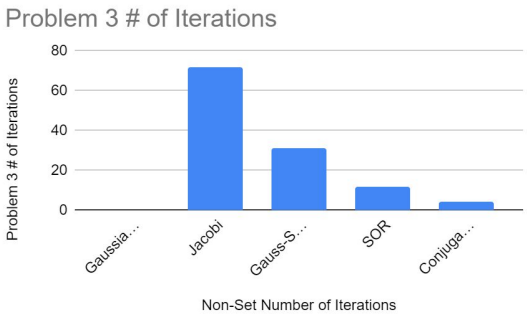


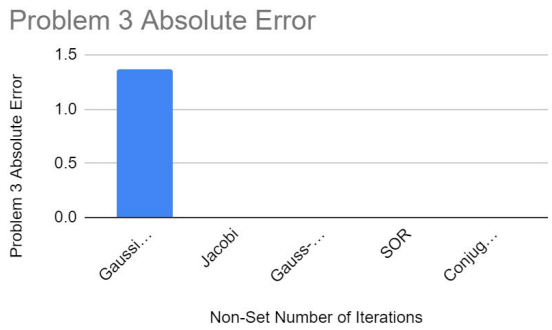


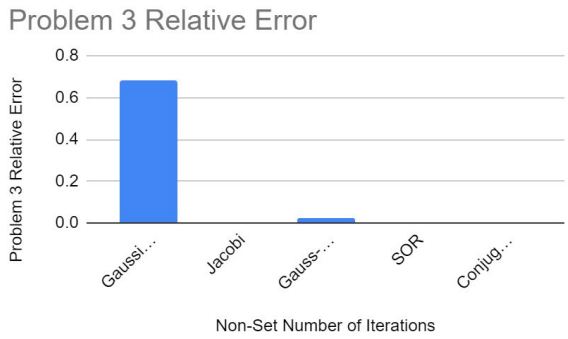


Below are the exact same subjects of comparison, except it is comparing the results for when the system converges. Although Gaussian Elimination takes the least amount of time and iterations, it does have the largest amount of errors. Also, Jacobi has the longest amount of time and iterations, but has very small amounts of error. Although, the Conjugate Gradient method seems to be the best method to choose for this example, for it has the least amount of time, iterations and errors.



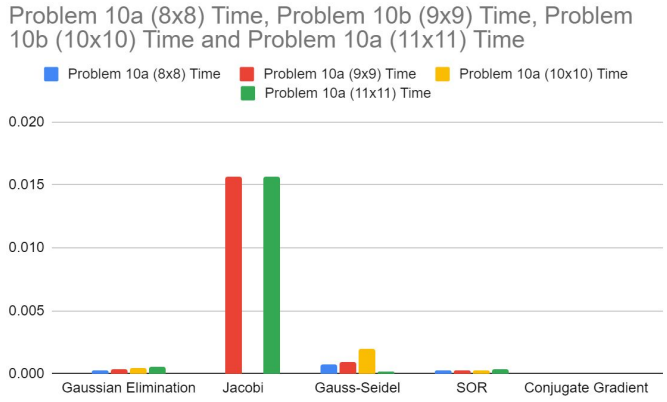


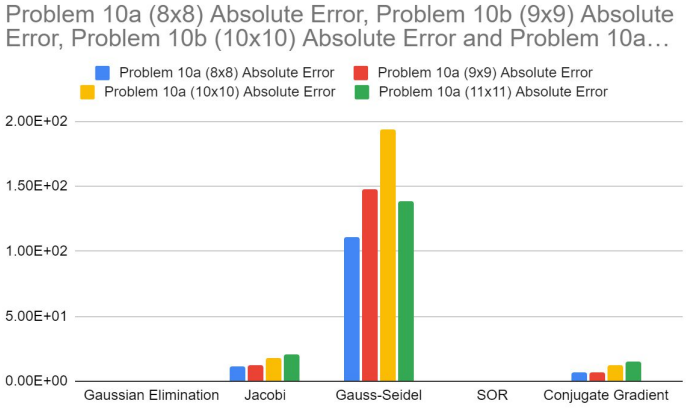


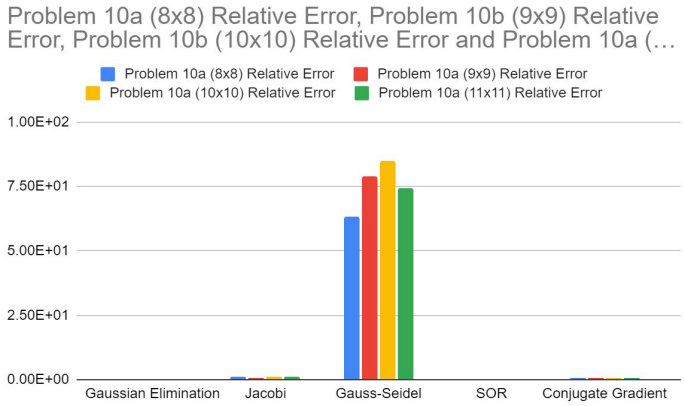


3.6.2 Computer Exercise 10a

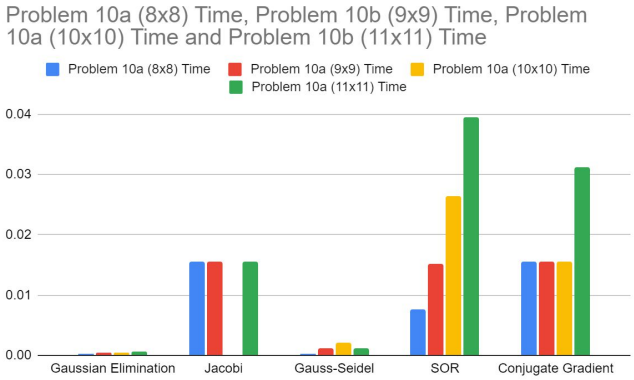
Below are the results for a set of only 3 iterations. This will show how each method runs from the start versus how they finish for a much larger and complex matrix, and there are 4 different matrices being compared. An 8x8 matrix, 9x9 matrix, 10x10 matrix, and an 11x11 matrix. In comparing only the first 3 iterations, one can see that Gaussian Elimination and SOR method are the ones with the better results in time, and error.

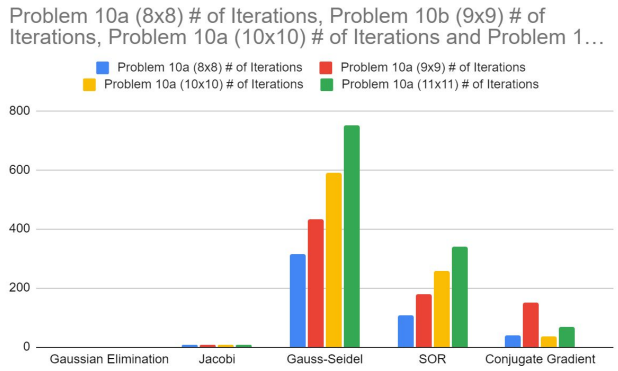


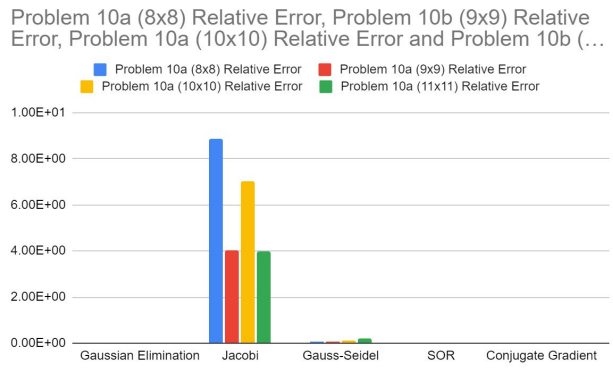


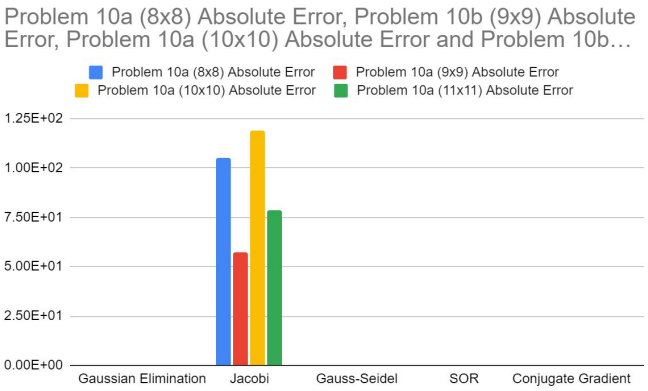


Below are the exact same subjects of comparison, except it is comparing the results for when the system converges. There are 4 different matrices being compared starting at an 8x8 matrix, 9x9 matrix, 10x10 matrix, and an 11x11 matrix. The Gaussian Elimination method seems to be the best method to choose for all 4 size matrices in terms of time, and errors.



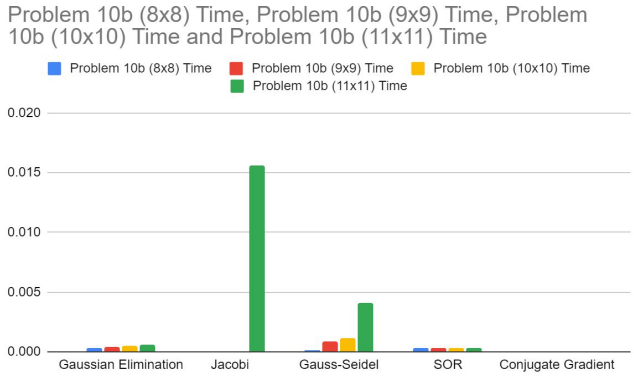


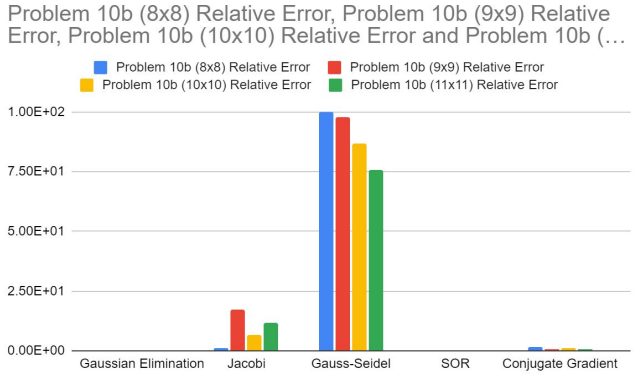


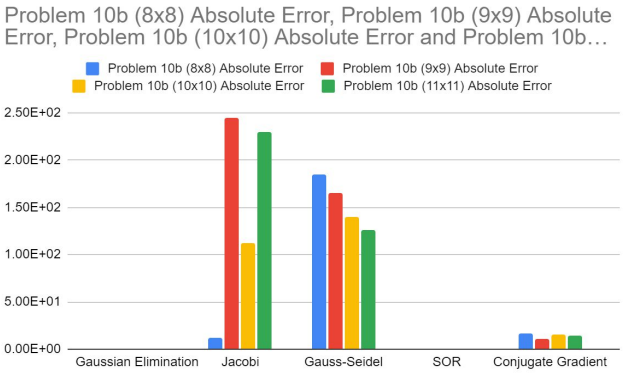


3.6.3 Computer Exercise 10b

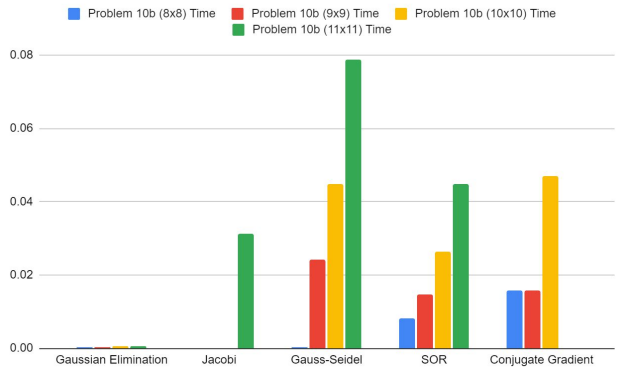
Below are the results for a set of only 3 iterations. This will show how each method runs from the start versus how they finish for a much larger and complex matrix, and there are 4 different matrices being compared. An 8x8 matrix, 9x9 matrix, 10x10 matrix, and an 11x11 matrix. It seems that SOR and Gaussian Elimination are the best methods to use for all 4 size matrices.

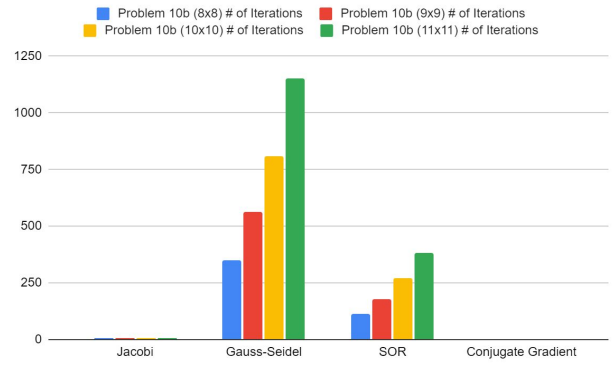


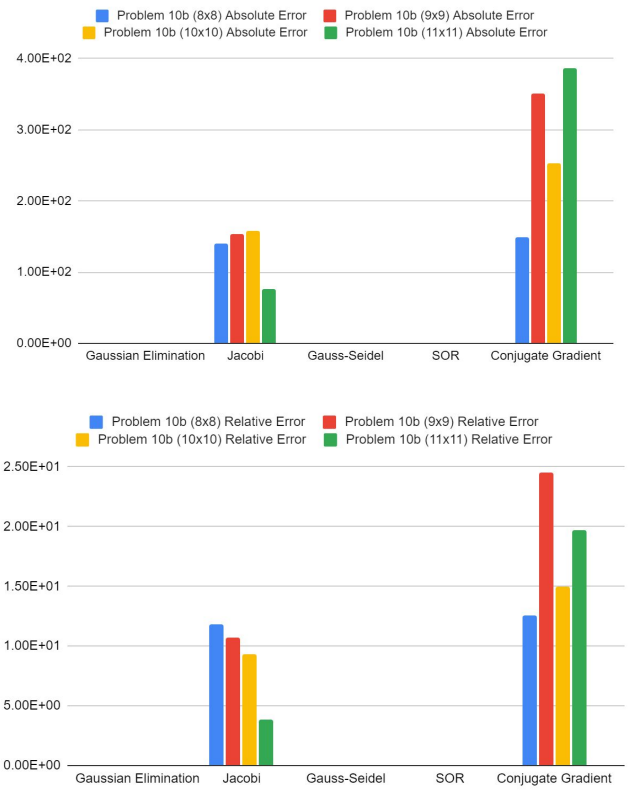




Below are the exact same subjects of comparison, except it is comparing the results for when the system converges. There are 4 different matrices being compared starting at an 8x8 matrix, 9x9 matrix, 10x10 matrix, and an 11x11 matrix. Although SOR and Gauss-Seidel have a large number of iterations and take a long time to compute, it does have a very small amount of error. Therefore, Gaussian Elimination, Gauss-Seidel, and SOR would be the best methods to choose for this example.







One can see that the results varied considerably. This can be that each method performs better depending on the system of linear equations given. Each method outputs relatively the same result, so one can choose which method they prefer to use depending on their scenario.

4. Conclusion

It has been proven that all of five of these distinct methods are able to solve the same problem, with differences that help each method standout in different scenarios. Whether that is a difference per input or difference of system performing the operations. The Gaussian Elimination method with partial pivoting breaks the ground with the simpler version of solving a system of equations, and performs no iterations. Also, according to the comparison results, Gaussian seems to work better on more complex matrices.

The Jacobi method breaks ground with a more efficient approach for matrices that fulfill the requirement of being square matrices that are diagonally dominant. It has also been shown that using either the system of linear equations or matrix form approach, you should still get a close approximation regardless of which approach was chosen. Also, according to the comparison results, this method seems to work best for less complex matrices.

Furthermore, The Gauss-Seidel Method is almost entirely similar to The Jacobi Method, except for one aspect; when the first element of the kth vector iteration is solved for, the Gauss-Seidel Method will use this new element to solve for the next in the column vector of the kth iteration. This, in turn, sacrifices the parallel nature of the Jacobi method in order to achieve greater accuracy per iteration. This result is demonstrated in the outputs of each respective code segment in sections 3.2.1 and 3.3.1 in the number of iterations for Jacobi and Gauss Seidel for problem 8.4-3CE (84 and 27, respectively). As for the comparison results, this method seems to work well with both complex and less complex matrices, the only thing is that it is not as quick to compute its results.

Moreover, as with the Gauss-Seidel and Jacobi methods, the SOR algorithm approaches convergence to a solution by using previously approximated values. As for the comparison results, this method works great with complex and less complex matrices. All-in-all the SOR method is a copy of Gauss-Seidel (and another variation of Jacobi) that introduces a new parameter: the ω variable which seeks to amplify the rate at which each iteration closes on the actual solution. It is only reasonable to assume that if each iteration of the previous methods only gets closer and closer to the actual solution, then we could speed things up by multiplying each iteration by a scalar such that each new vector's magnitude is amplified once calculated.

This is an early demonstration of the use of a modernly called **tuning parameter**. One that is left to be changed at the user's discretion (whether that is simply by opinion or calculation of exactitude; between a factor 0 and 2). The speed gains from this tuning parameter come at a greater number of points of criteria that the matrix system in question must fulfill, i.e. the matrix, aside from fulfilling the Gauss-Seidel criteria, the matrix must contain positive diagonal elements. Moreover, the accuracy of the answer becomes one of the costs of implementing this method, as fine tuning the omega parameter too high or too low might lead the algorithm to overshoot or undershoot the closest iteration in some elements.

Compared to the other 4 methods of solving systems of linear equations, the Conjugate Gradient Method was written to combat the problem of the convergence of the optimum being slow for larger matrices. The idea is that instead of moving along the gradient direction, leading to a slow process, one could employ the conjugate gradient method to move along the set conjugate directions to reach the optimum very quickly. In the conjugate gradient method, one can find the current direction in which you must move using a linear combination of all of the previous gradient directions. To find the solution of $Ax = b$ where x is the optimal solution, x_*, one must find the set of k linearly independent conjugate directions, s_k, and the weight in which one must move towards s_k as the coefficient, β_k. Also, according to the comparison results, this method works great for less complex matrices.

As per the data collected the results varied considerably. This may be the result of variations within each machine and methods used at processing times. Although in theory these algorithms stand out from the others within different parameters of a problem at hand, it is important to keep in mind that by coding these methods one can introduce inefficiencies depending on conditions such as coding language, imported support modules, computer hardware and number of other software running at compilation time that may take priority over these scripts. However, one clear trend is noticeable in the comparison of performance of some of the exercises, e.g. problem 3 (non-set number of operations). Although the non-iterative nature of gaussian elimination (with partial pivoting) may lead to an answer in less steps than the other methods, it may lead to greater errors in a smaller system than the other methods.

In conclusion, as with the previous example, some of our other results did highlight the potential benefit of each algorithm at different times remarking again the importance of taking the parameters of the problem at hand into consideration in order to best match against the criteria under which each method works best. Moreover, further research would need to be made into the differences of these methods for a more robust understanding of their respective performances.

5. Bibliography

[1] Black, Noel; Moore, Shirley; and Weisstein, Eric W. "Jacobi Method." From *MathWorld*--A Wolfram Web Resource. https://mathworld.wolfram.com/JacobiMethod.html

[2] Karunanithi, S., Gajalakshmi, N., Malarvizhi, M., & Saileshwari, M. (2018). *A Study on Comparison of Jacobi, GaussSeidel and Sor Methods for the Solution in System of Linear Equations, 56*(4), 1-9.

[3] Lynch, Deirdre. "2.5.2/Gauss-Seidel Method and SOR." *Numerical Analysis*, by Tim Sauer, Pearson, Boston, Massachusetts, 2012, pp. 108–109.

[4] Saad, Yousef (2003). *Iterative Methods for Sparse Linear Systems* (2nd ed.). SIAM. p. 414

[5] Strong, David M. "Iterative Methods for Solving Ax = b - Jacobi's Method." *Iterative Methods for Solving Ax = b - Jacobi's Method | Mathematical Association of America*, July 2005, www.maa.org/press/periodicals/loci/joma/iterative-methods-for-solving-iaxi-ibi-jacobis-method.

[6] Rambo, Mikeq. "The Conjugate Gradient Method for Solving Linear Systems of Equations." *Http://Math.stmarys-Ca.edu/*, 2016, math.stmarys-ca.edu/wp-content/uploads/2017/07/Mike-Rambo.pdf.

[7] Cheney, E. W., and David Kincaid. *Numerical Mathematics and Computing*. Brooks/Cole, Cengage Learning, 2013.

[8] Dongarra, Jack. " The Successive Overrelaxation Method." *The Successive Overrelaxation Method*, 1995, www.netlib.org/linalg/html_templates/node15.html.

[9] Ford, William. "Relaxation Parameter." *Relaxation Parameter - an Overview*, www.sciencedirect.com/topics/computer-science/relaxation-parameter.

[10] Katopodes, Nikolas D. "Over-Relaxation." *Over-Relaxation - an Overview | ScienceDirect Topics*, www.sciencedirect.com/topics/engineering/over-relaxation.

[11] Moler, Cleve. "What Is the Condition Number of a Matrix?" *Cleve's Corner: Cleve Moler on Mathematics and Computing*, The MathWorks, Inc., 17 July 2017, blogs.mathworks.com/cleve/2017/07/17/what-is-the-condition-number-of-a-matrix/.

[12] Petersdorff, Tobias Von. *Errors for Linear Systems*. terpconnect.umd.edu/~petersd/460/linsysterrn.pdf.

[13] Rapp, Bastian E. “Numerical Methods for Linear Systems of Equations.” *Microfluidics: Modelling, Mechanics and Mathematics*, Elsevier, 6 Dec. 2016, www.sciencedirect.com/science/article/pii/B9781455731411500253.

[14] “Relaxation Method.” *Relaxation Method - Encyclopedia of Mathematics*, encyclopediaofmath.org/wiki/Relaxation_method.

[15] Strong, David M. “Iterative Methods for Solving Ax = b - The SOR Method.” *Iterative Methods for Solving Ax = b - The SOR Method | Mathematical Association of America*, www.maa.org/press/periodicals/loci/joma/iterative-methods-for-solving-iaxi-ibi-the-sor-method.

[16] “Successive Overrelaxation Method.” *From Wolfram MathWorld*, mathworld.wolfram.com/SuccessiveOverrelaxationMethod.html.