MATH 400 Assignment 2 Documentation
Fall 2020

Group 8 – Java:
Peter Boukhalil (6502073558)
Luong Dang (916116778)
Kimberly Nivon (8188242113)
Julia Beatriz Ramos (9164124314)
Nhi Vu (4152037089)

Section 1 - Problem 3
Section 2 - Problem 1
Section 3 - Problem 1

Presentation:

Slide:
https://docs.google.com/presentation/d/1ea8TAlXWkdNwaO5A7-SXzOfJRV2JI6LHhXXzVbskquw/edit?usp=sharing

Record video:
https://drive.google.com/file/d/1TYBeEqxDNKs5bV-hptMjI_mef-Jm7OhD/view?usp=sharing

Table of Contents:

1. Introduction

1.1 Project Overview

The project is to solve algorithmic problems related to linear equations and polynomial equations, using Elimination Operation (Gauss-Seidel's method, Jacobi's method, Partial fraction coefficient, etc.)

1.2 Technical Overview

The project is our demonstration of understanding of iterative methods that solve a system of linear equations. This project covered 3 sections. We implemented the Gaussian Elimination and back substitution technique in Section 1 - Problem 3. In Section 2, we use regular (no pivoting) Gauss elimination (GE), partial pivoting GE, Scaled partial pivoting GE, and LU factorization to estimate the solution. In Section 3, we have a chance to do an experiment to find out which method, between Jacobi and Gauss-Seidel, requires less iteration.

1.3 Summary of Work Completed

The project allowed us to review the second-half of the semester and learned how to use the knowledge of it.

Section1 - Problem 3:

- We wrote an algorithm that implements pivoting GE and back substitution to solve the linear equation.
- We also need utility methods to have the code running. This includes printMatrix and scanner

Section2

Part 1:

- First step is to have an initialization method for the descriptive matrix. The problem is difficult to understand so we spent a long time figuring if we understood it correctly.
- Part 1 requires 3 elimination methods, ones require some extra steps to another. Pivoting GE method reuse non-pivoting GE method's code with an extra method that finds pivot. Scaled-Pivoting GE needs a scaling method. All three use the back substitute method to solve the processed matrix.

Part 2:

- LU decomposition can reuse code from part 1. We made the GE method to keep track of the scaler, then store it as a Lower triangular matrix. Upper triangular matrix is the row-reduced matrix we found from part 1.
- To find inverse of matrix A, where inverse of matrix A = U^-1 * L^-1. So we wrote functions to find the inverse matrices. We made the back substitute method to find solution of L|I and U|I, which is the inverse of L and U. We also wrote a method that multiply matrices to get the inverse of the matrix A. This multiplication method also helps us find estimated x, where x~ = A^-1 *b.

Section 3

- Wrote Jacobi's method and Gauss-Seidel's method algorithm.
- Wrote printing-method for the solution.
- Wrote iteration counter for comparison.

We uploaded the source code into Github for future use.

There is a 15 minutes presentation about the 3 problems in this project we made and were included in the submitted file folder. The presentation includes our explanation of the problems, explaining the steps we did to solve the problems, our codes, and a demonstration of the programs.
The project took us 3 meetings and 30 minutes recording session. Everybody learned many things during the making of this project.

2. Development Environment

Java version: JDK 11.0.2
IDE: JetBrains IntelliJ IDEA Ultimate

3. How to Build/Import your Project

1. Import file:
   - In terminal, enter :
     " git clone https://github.com/luongdang0701/Project2_Group8"
     or, download the files which were submitted.
2. Compile/Run:
   - Section 1 (Problem 1):
     Compile file:
     ">> cd Sec1-Prob3"
     ">> javac Section1Problem3.java"
     Run file

“>> java Section1Problem3”

- Section 2 (Problem 1):

  Compile file:

  “>> cd Sec2”

  “>> javac Section2.java”

  Run file

  “>> java Section2”

- Section 3:

  Compile file:

  “>> cd Sec3”

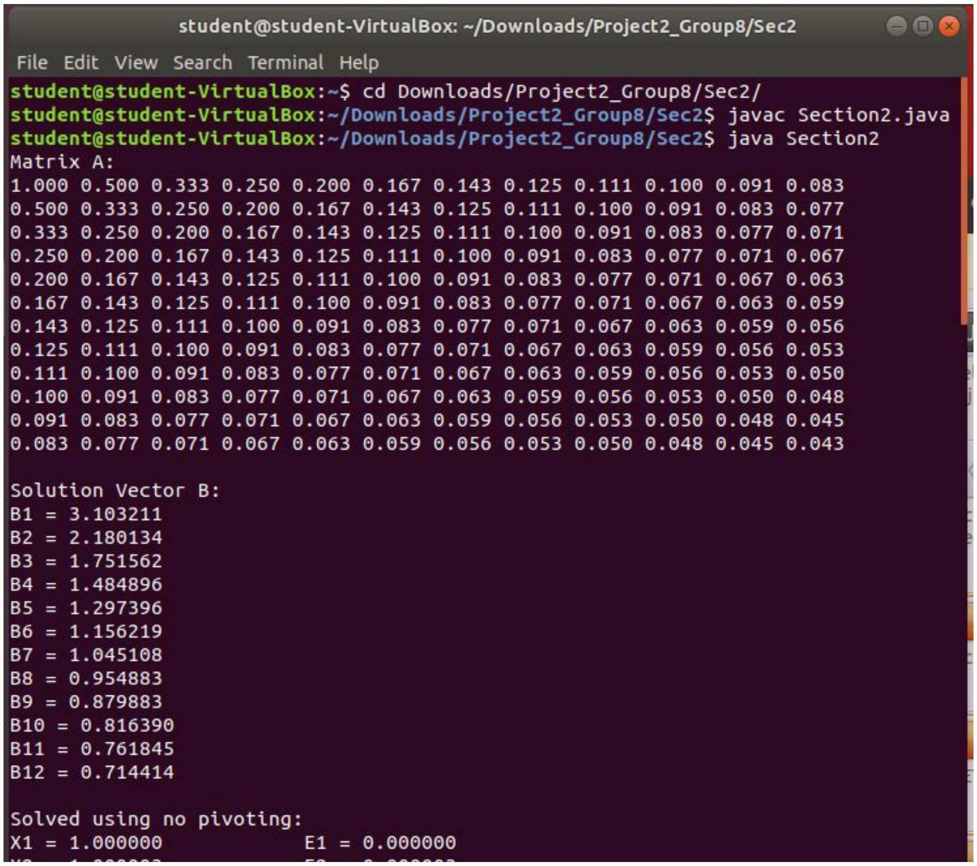
  “>> javac Section3.java”

  Run file

  “>> java Section3”

git clone "@address-of-github"
cd : change directory
javac: java compile
java: run java file

demo:



4. Assumption Made:

Gaussian Elimination:

The goals of Gaussian elimination are to make the upper-left corner element a 1, use elementary row operations to get 0s in all positions underneath that first 1, get 1s for leading coefficients in every row diagonally from the upper-left to lower-right corner, and get 0s beneath all leading coefficients. Basically, you eliminate all variables in the last row except for one, all variables except for two in the equation above that one, and so on and so forth to the top equation, which has all the variables. Then you can use back substitution to solve for one variable at a time by plugging the values you know into the equations from the bottom up.

You accomplish this elimination by eliminating the x (or whatever variable comes first) in all equations except for the first one. Then eliminate the second variable in all equations except for the first two. This process continues, eliminating one more variable per line, until only one variable is left in the last line. Then solve for that variable.

You can multiply any row by a constant (other than zero).

You can switch any two rows.
You can add two rows together.

Back-Substitution

The process of solving a linear system of equations that has been transformed into row-echelon form or reduced row-echelon form. The last equation is solved first, then the next-to-last

Gaussian Elimination with Partial Pivoting

Pivoting helps reduce rounding errors; you are less likely to add/subtract with very small number (or very large) numbers.

Partial Pivoting: Exchange only rows
Exchanging rows does not affect the order of the xi
For increased numerical stability, make sure the largest possible pivot element is used. This requires searching in the partial column below the pivot element.
Partial pivoting is usually sufficient.

LU Factorization

Consider the system Ax = b with LU factorization A = LU. Then we have

$$L \underbrace{U\boldsymbol{x}}_{=\boldsymbol{y}} = \boldsymbol{b}.$$

Therefore we can perform (a now familiar) 2-step solution procedure:
1. Solve the lower triangular system Ly = b for y by forward substitution.
2. Solve the upper triangular system Ux = y for x by back substitution.

Moreover, consider the problem AX = B (i.e., many different right-hand sides that are associated with the same system matrix). In this case we need to compute the factorization A = LU only once, and then

AX = B ⇐⇒ LUX = B,

and we proceed as before:
1. Solve LY = B by many forward substitutions (in parallel).

2. Solve UX = Y by many back substitutions (in parallel). In order to appreciate the usefulness of this approach note that the operations count for the matrix factorization is O((2 /3)m3), while that for forward and back substitution is O(m2).

Inverse:

The inverse operator has the following property:

A = LU => A −1 = U^-1 * L^-1

So here is twp-step procedure to find the inverse of a matrix A:

Step 1.. Find the LU decomposition A = LU (Gaussian form or the Crout form whichever you are told to find)

Step 2.. Find the inverse of A−1 = U^−1 * L^−1 by inverting the matrices U and L.

The Jacobi Method

***Two assumptions made on Jacobi Method*:**

1. The system given by

$$\begin{gathered} a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots a_{nn}x_n = b_n \end{gathered}$$

Has a unique solution.

2. The coefficient matrix A has no zeros on its main diagonal, namely, $a_{11}, a_{22}, \dots, a_{nn}$ are nonzeros.

Main idea of Jacobi

To begin, solve the 1st equation for x_1, the 2nd equation for x_2 and so on to obtain the rewritten equations:

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

Then make an initial guess of the solution $\boldsymbol{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, \dots x_n^{(0)})$. Substitute these values into the right hand side the of the rewritten equations to obtain the *first approximation*, $\left(x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, \dots x_n^{(1)}\right)$.

This accomplishes one **iteration**.

In the same way, the *second approximation* $\left(x_1^{(2)}, x_2^{(2)}, x_3^{(2)}, \dots x_n^{(2)}\right)$ is computed by substituting the first approximation's x-vales into the right hand side of the rewritten equations.

By repeated iterations, we form a sequence of approximations $\boldsymbol{x}^{(k)} = \left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \dots x_n^{(k)}\right)^t, \quad k = 1,2,3, \dots$

The Jacobi Method. For each $k \geq 1$, generate the components $x_i^{(k)}$ of $\mathbf{x}^{(k)}$ from $\mathbf{x}^{(k-1)}$ by

$$x_i^{(k)} = \frac{1}{a_{ii}}\left[\sum_{\substack{j=1,\\ j\neq i}}^{n}(-a_{ij}x_j^{(k-1)}) + b_i\right], \qquad \text{for } i = 1,2,,\ldots n$$

5. Procedure:

Section 1: First step: Find the matrix A and matrix B by the system equations of variable A1, A2, A3, A4, A5, A6 on the technique of partial fractions. After deriving the system of equations, we can assemble the Matrix A from the variable coefficients and identify vector b to solve for the partial fraction coefficients. After obtaining this, we implement the Gaussian elimination algorithm with partial pivoting in the code. This requires finding the pivot row and swapping accordingly, using multipliers to row reduce the matrix into upper triangular form, then solving for the solution vector using back substitution.

```
import java.util.Scanner;

public class Section1Problem3 {

  // Performs Gaussian Elimination using matrix A and solution matrix B
  public static void pivotingGE(double[][] A, double[] B) {

      int N = B.length;

      // For every row
      for (int k = 0; k < N; k++) {

          // Find pivot row and swap
          int max = k;
          for (int i = k + 1; i < N; i++)
              if (Math.abs(A[i][k]) > Math.abs(A[max][k]))
                  max = i;
          double[] temp = A[k];
          A[k] = A[max];
          A[max] = temp;
          double t = B[k];
          B[k] = B[max];
          B[max] = t;

          // Identify row below
          for (int i = k + 1; i < N; i++) {

              // Use the multiplier to row reduce
              double multiplier = A[i][k] / A[k][k];
              B[i] -= multiplier * B[k];
```

```
            for (int j = k; j < N; j++)
                A[i][j] -= multiplier * A[k][j];
        }
    }

    // Solve using back substitution
    double[] solution = new double[N];
    for (int i = N - 1; i >= 0; i--) {
        double sum = 0.0;
        for (int j = i + 1; j < N; j++)
            sum += A[i][j] * solution[j];
        solution[i] = (B[i] - sum) / A[i][i];
    }

    // Output the solution
    printSolution(solution);
  }

  // Prints the solution
  public static void printSolution(double[] sol) {
    int N = sol.length;
    System.out.println("\nSolutions: ");
    for (int i = 0; i < N; i++)
        System.out.printf("A%d = %.3f\n", i+1, sol[i]);
    System.out.println();
  }

  // Main function
  public static void main(String[] args) {

    Scanner scan = new Scanner(System.in);

    System.out.println("\nEnter number of equations:");
    int N = scan.nextInt();

    double[] B = new double[N];
    double[][] A = new double[N][N];

    System.out.println("\nEnter "+ N +" equations coefficients:");
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = scan.nextDouble();

    System.out.println("\nEnter "+ N +" solutions:");
    for (int i = 0; i < N; i++)
        B[i] = scan.nextDouble();

    pivotingGE(A,B);
  }
}
```

Output:

```
Enter number of equations:
6

Enter 6 equations coefficients:
1 1 0 1 1 0
-8 -7 1 -6 -9 1
22 16 -3 12 29 -9
-26 -16 3 -12 -39 29
21 15 -3 11 18 -39
-18 -9 2 -6 0 18

Enter 6 solutions:
0
0
0
1
1
1

Solutions:
A1 = -0.375
A2 = 1.400
A3 = 0.650
A4 = -1.015
A5 = -0.010
A6 = -0.030
```

Section 2:

```
public class Section2Problemi {

  // Performs Gaussian Elimination with no pivoting
  public static void noPivotingGE(double[][] A, double[] B) {
```

```
    int N = B.length;

    // For every row
    for (int k = 0; k < N; k++) {

        // Identify row below
        for (int i = k + 1; i < N; i++) {

            // Use the multiplier to row reduce
            double multiplier = A[i][k] / A[k][k];
            B[i] -= multiplier * B[k];
            for (int j = k; j < N; j++)
                A[i][j] -= multiplier * A[k][j];
        }
    }

    // Solve using back substitution
    double[] solution = new double[N];
    for (int i = N - 1; i >= 0; i--) {
        double sum = 0.0;
        for (int j = i + 1; j < N; j++)
            sum += A[i][j] * solution[j];
        solution[i] = (B[i] - sum) / A[i][i];
    }

    //Find Error with x_i=[1]
    double [] e= new double [N];
    for(int i=0; i<N; i++){
        e[i]=Math.abs(solution[i]-1);
    }

    // Output the solution
    printSolution(solution,e,N);
}

// Performs Gaussian Elimination using partial pivoting
public static void pivotingGE(double[][] A, double[] B) {
    int N = B.length;
    for (int k = 0; k < N; k++) { // For every row
        // Find pivot row and swap
        int max = k;
        for (int i = k + 1; i < N; i++)
            if (Math.abs(A[i][k]) > Math.abs(A[max][k]))
                max = i;
        double[] temp = A[k];
        A[k] = A[max];
        A[max] = temp;
        double t = B[k];
        B[k] = B[max];
        B[max] = t;

        // Identify row below
        for (int i = k + 1; i < N; i++) {
            // Use the multiplier to row reduce
```

```
            double multiplier = A[i][k] / A[k][k];
            B[i] -= multiplier * B[k];
            for (int j = k; j < N; j++)
                A[i][j] -= multiplier * A[k][j];
        }
    }

    // Solve using back substitution
    double[] solution = new double[N];
    for (int i = N - 1; i >= 0; i--) {
        double sum = 0.0;
        for (int j = i + 1; j < N; j++)
            sum += A[i][j] * solution[j];
        solution[i] = (B[i] - sum) / A[i][i];
    }

    //Find Error with x_i=[1]
    double [] e= new double [N];
    for(int i=0; i<N; i++){
        e[i]=Math.abs(solution[i]-1);
    }
    printSolution(solution,e,N); // Output the solution
}

// Performs Gaussian Elimination using scaled partial pivoting
// Algorithm from: https://www.youtube.com/watch?v=4YzIfcSFVCU&ab_channel=ThomasBingham
public static void scaledPivotingGE(double[][] A, double[] B) {

    int N = B.length;

    double[] S = new double[N];
    for (int i = 0; i < N; i++) {
        S[i] = arrayMax(A[i],false);
    }

    // For every row
    for (int k = 0; k < N; k++) {

        // Scale the rows using highest magnitude elements and find the max row
        int max = k;
        double[] RV = initRV(N);
        for (int i = k; i < N; i++) {
            RV[i] = Math.abs(A[i][k])/S[i];
        }
        max = (int)arrayMax(RV, true);

        // Pivot the rows
        double[] temp = A[k];
        A[k] = A[max];
        A[max] = temp;
        double t = B[k];
        B[k] = B[max];
        B[max] = t;

        // Identify row below
```

```
        for (int i = k + 1; i < N; i++) {

            // Use the multiplier to row reduce
            double multiplier = A[i][k] / A[k][k];
            B[i] -= multiplier * B[k];
            for (int j = k; j < N; j++)
                A[i][j] -= multiplier * A[k][j];
        }
    }

    // Solve using back substitution
    double[] solution = new double[N];
    for (int i = N - 1; i >= 0; i--) {
        double sum = 0.0;
        for (int j = i + 1; j < N; j++)
            sum += A[i][j] * solution[j];
        solution[i] = (B[i] - sum) / A[i][i];
    }

    //Find Error with x_i=[1]
    double [] e= new double [N];
    for(int i=0; i<N; i++){
        e[i]=Math.abs(solution[i]-1);
    }

    // Output the solution
    printSolution(solution,e,N);

}

// Returns the max from an array
public static double arrayMax(double[] A, boolean index) {
    int maxInd = 0;
    for (int i = 0; i < A.length; i++) {
        if (Math.abs(A[i]) > Math.abs(A[maxInd])) {
            maxInd = i;
        }
    }

    return ((index) ? maxInd : Math.abs(A[maxInd]));
}

// Initialize the ratio vector for scaled partial pivoting
public static double [] initRV (int N) {
    double[] RV = new double [N];
    for (int i = 0; i < N; i++)
        RV[i] = 0;
    return RV;
}

// Initialize matrix A
public static double [][] initA (int N) {
    double[][] A = new double [N][N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
```

```
            A[i][j] = 1/(i+j+1.0);
        }
    }
    return A;
}

// Initialize matrix B
public static double [] initB (double[][] A) {
    int N = A.length;
    double[] B = new double [N];
    for (int i = 0; i < N; i++) {
        B[i] = 0;
        for (int j = 0; j < N; j++) {
            B[i] += A[i][j];
        }
    }
    return B;
}

// Prints the solution with error
public static void printSolution(double[] sol, double []e,int n ){
    for (int i = 0; i < n; i++) {
        if (i < 9) {
            System.out.printf("%s%d = %-20.6f" + "%s%d = %.6f\n", "X", i + 1, sol[i], "E", i + 1, e[i]);
        } else {
            System.out.printf("%s%d = %-19.6f" + "%s%d = %.6f\n", "X", i + 1, sol[i], "E", i + 1, e[i]);
        }
    }
    System.out.println("||x-x~|| = " + arrayMax(e, false));
    System.out.println();
}

// Prints a vector
public static void printVector(double[] sol, String letter) {
    int N = sol.length;
    for (int i = 0; i < N; i++)
        System.out.printf("%s%d = %.6f\n", letter, i+1, sol[i]);
    System.out.println();
}

// Print matrix
public static void printMatrix(double[][] A) {
    int N = A.length;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            System.out.printf("%.3f ", A[i][j]);
        }
        System.out.println();
    }
    System.out.println();
}



//########################################PART 2 ############################
```

```java
public static void LUDecomposition(double[][] A) {
   int n = A.length;
   double[][] upper= new double[n][n];
   double[][] lower= new double[n][n];

   //LU Decomposition
---------------------------------------------------------------------
   double[][] temp = A;
   int N = A.length;
   // For every row
   for (int k = 0; k < N; k++) {
       // Identify row below
       for (int i = k + 1; i < N; i++) {
           // Use the multiplier to row reduce
           double multiplier = temp[i][k] / temp[k][k];
           lower[i][k] = multiplier; // <= Lower --------------------
           for (int j = k; j < N; j++) {
               temp[i][j] -= multiplier * temp[k][j];
           }
       }
   }
   for (int i = 0; i < n; i++) {
       for (int j = 0; j < n; j++) {
           if (i > j) {
           } else if (i == j) {
               lower[i][j] = 1;
           } else {
               lower[i][j] = 0;
           }
       }
   }
   for (int i = 0; i < n; i++) {
       for (int j = 0; j < n; j++) {
           if (i <= j) {
               upper[i][j] = temp[i][j];
           } else {
               upper[i][j] = 0.0;
           }
       }
   }

   // -------------------------------------------------------------------

   System.out.println("-----------------LU Factorial --------------");
   System.out.println("Upper - U");
   printMatrix(upper);
   double[][] inverseU = findInverseUpper(upper);

   System.out.println("Lower - L");
   printMatrix(lower);
   double[][] inverseL = findInverseLower(lower);

   System.out.println("---------Inverse of U and L----------------");
   System.out.println("InverseUpper - U^-1");
```

```java
    printMatrix(inverseU);
    System.out.println("InverseLower - L^-1");
    printMatrix(inverseL);

    System.out.println("-----------------Inverse of A---------------------");
    System.out.println("InverseA - A^-1");
    double[][] inverseA = multiplyMatrices(inverseU,inverseL);
    printMatrix(inverseA);


    System.out.println("------- Estimated X ( x~ = A^-1 * b) -----------");
    double[] B = initB(A);
    double[] xsenor = multiplyMatrixWithVector(inverseA,B);
    printVector(xsenor,"x~ ");
}

static double[] multiplyMatrixWithVector(double A[][], double B[]) {
    int n = A.length;
    double C[] = new double[n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
                C[i] += A[i][j] * B[j];
        }
    }
    return C;
}

static double[][] multiplyMatrices(double A[][], double B[][]) {
    int n = A.length;
    double C[][] = new double[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C;

}

public static double[][] findInverseLower(double[][] lower){
    int n = lower.length;
    double[][] inverseL = new double[n][n];
    for(int u = 0; u < n; u ++) {
        double[] B = new double[n];

        for (int i = 0; i < n; i++) {
            if(i==u){
                B[i]=1;
            }else{
                B[i] = 0;}
        }

        double[] solution = new double[n];
        for (int i = 0; i <n; i++) {
```

```
            double sum = 0.0;
            for (int j = 0; j < n; j++)
                sum += lower[i][j] * solution[j];
            solution[i] = (B[i] - sum) / lower[i][i];
        }
        for(int a = 0; a < n; a++){
            inverseL[a][u] = solution[a];
        }
    }

    return inverseL;
}




public static double[][] findInverseUpper(double[][] upper){
    int n = upper.length;
    double[][] inverseU = new double[n][n];
    for(int u = 0; u < n; u ++) {
        double[] B = new double[n];

        for (int i = 0; i < n; i++) {
            if(i==u){
                B[i]=1;
            }else{
                B[i] = 0;}
        }

        double[] solution = new double[n];
        for (int i = n - 1; i >= 0; i--) {
            double sum = 0.0;
            for (int j = i + 1; j < n; j++)
                sum += upper[i][j] * solution[j];
            solution[i] = (B[i] - sum) / upper[i][i];
        }
        for(int a = 0; a < n; a++){
            inverseU[a][u] = solution[a];
        }
    }
    return inverseU;
}


  // Main function
  public static void main(String[] args) {

    int N = 12;
    double[][] A;
    double[] B;

    A = initA(N);
    B = initB(A);

    System.out.println("Matrix A: ");
```

```java
        printMatrix(A);

        System.out.println("Solution Vector B: ");
        printVector(B,"B");

        A = initA(N);
        B = initB(A);

        System.out.println("Solved using no pivoting:");
        noPivotingGE(A,B);

        A = initA(N);
        B = initB(A);

        System.out.println("Solved using partial pivoting:");
        pivotingGE(A,B);

        A = initA(N);
        B = initB(A);

        System.out.println("Solved using scaled partial pivoting:");
        scaledPivotingGE(A,B);

// ########################  Part2 ########################

        LUDecomposition(A);

    }
}
```

Part 1:

Initializing matrix A which contains coefficients: Although we get given information aij = 1/(i+j−1) for 1 ≤ i, j, ≤ n, but in java, array method begins at 0, so the loop of i and j, begin at 0 to n, and we have to plus 1 for i and j in all calculations.

Initializing matrix B, solution vector: B=A.X with Xi={1}, then Bi={ sum of Aij} with j is from 0 to n.

Using the Gaussian Elimination with partial pivoting algorithm developed in Section 1 Problem 3, we modified the function to produce 3 functions: GE with no pivoting, GE with partial pivoting, and GE with scaled partial pivoting. Using the scaled partial pivoting algorithm presented by Thomas Bingham at Oregon Institute of Technology (https://www.youtube.com/watch?v=4YzIfcSFVCU&ab_channel=ThomasBingham), we were able to develop the function for scaled partial pivoting. Then, we developed code to perform error analysis for each method.

Part 2:

LU Factorization includes row reducing operation. To LU decompose a matrix, we need to know that Upper triangular matrix (U) is the row-reduced echelon form of matrix A. Lower triangular matrix (L) is the matrix containing all magnitude from the row-reducing operation. Therefore, with modification, we can reuse codes from part 1.

First, Do row-reducing operations with non-pivoting GE, store all magnitudes from the operation as Lower Triangular matrix in the process. Finally, store the row-reduced echelon matrix as the Upper Triangular matrix.

Now, to find Inverse of A, we can use the identity below:

A = LU => A^-1 = U^-1 * L^-1

To find the inverse, we can do back-substitution algorithm from part 1, with modification.

```
for-loop(go through each vector of Identity matrix) {
        backSubstitution(U,I); // Where I is the identity
}
```

Now, the array of all solutions from the backSubstitution method is the inverse of U. Lower triangular matrix is a little different, we modify the backSubstitution method to loop in the reverse direction. The array of all solutions from frontSubstitution is the inverse of L matrix.

Write a method to multiply matrices. Then apply the above identity of A^-1.

Last, estimate x using A^-1 which we just found. Use B from part 1, we can estimate x using the below identity.

X~ = A^-1*B where B is the array of solutions of matrix A.

Output:

Part 1:

```
Matrix A:
1.000 0.500 0.333 0.250 0.200 0.167 0.143 0.125 0.111 0.100 0.091 0.083
0.500 0.333 0.250 0.200 0.167 0.143 0.125 0.111 0.100 0.091 0.083 0.077
0.333 0.250 0.200 0.167 0.143 0.125 0.111 0.100 0.091 0.083 0.077 0.071
0.250 0.200 0.167 0.143 0.125 0.111 0.100 0.091 0.083 0.077 0.071 0.067
0.200 0.167 0.143 0.125 0.111 0.100 0.091 0.083 0.077 0.071 0.067 0.063
0.167 0.143 0.125 0.111 0.100 0.091 0.083 0.077 0.071 0.067 0.063 0.059
0.143 0.125 0.111 0.100 0.091 0.083 0.077 0.071 0.067 0.063 0.059 0.056
0.125 0.111 0.100 0.091 0.083 0.077 0.071 0.067 0.063 0.059 0.056 0.053
0.111 0.100 0.091 0.083 0.077 0.071 0.067 0.063 0.059 0.056 0.053 0.050
0.100 0.091 0.083 0.077 0.071 0.067 0.063 0.059 0.056 0.053 0.050 0.048
0.091 0.083 0.077 0.071 0.067 0.063 0.059 0.056 0.053 0.050 0.048 0.045
0.083 0.077 0.071 0.067 0.063 0.059 0.056 0.053 0.050 0.048 0.045 0.043

Solution Vector B:
B1 = 3.103211
B2 = 2.180134
B3 = 1.751562
B4 = 1.484896
B5 = 1.297396
B6 = 1.156219
B7 = 1.045108
B8 = 0.954883
B9 = 0.879883
B10 = 0.816390
B11 = 0.761845
B12 = 0.714414
```

```
Solved using no pivoting:
X1 = 1.000000            E1 = 0.000000
X2 = 1.000003            E2 = 0.000003
X3 = 0.999916            E3 = 0.000084
X4 = 1.001125            E4 = 0.001125
X5 = 0.991859            E5 = 0.008141
X6 = 1.035385            E6 = 0.035385
X7 = 0.902315            E7 = 0.097685
X8 = 1.175419            E8 = 0.175419
X9 = 0.795768            E9 = 0.204232
X10 = 1.148663           E10 = 0.148663
X11 = 0.938525           E11 = 0.061475
X12 = 1.011022           E12 = 0.011022
||x-x~|| = 0.20423224371501658
```

```
Solved using partial pivoting:
X1 = 1.000000            E1 = 0.000000
X2 = 1.000004            E2 = 0.000004
X3 = 0.999862            E3 = 0.000138
X4 = 1.001880            E4 = 0.001880
X5 = 0.986242            E5 = 0.013758
X6 = 1.060376            E6 = 0.060376
X7 = 0.831939            E7 = 0.168061
X8 = 1.303973            E8 = 0.303973
X9 = 0.643862            E9 = 0.356138
X10 = 1.260684           E10 = 0.260684
X11 = 0.891667           E11 = 0.108333
X12 = 1.019511           E12 = 0.019511
||x-x~|| = 0.3561379938120136
```

```
Solved using scaled partial pivoting:
X1 = 1.000000            E1 = 0.000000
X2 = 1.000003            E2 = 0.000003
X3 = 0.999922            E3 = 0.000078
X4 = 1.001044            E4 = 0.001044
X5 = 0.992461            E5 = 0.007539
X6 = 1.032709            E6 = 0.032709
X7 = 0.909838            E7 = 0.090162
X8 = 1.161705            E8 = 0.161705
X9 = 0.811938            E9 = 0.188062
X10 = 1.136765           E10 = 0.136765
X11 = 0.943491           E11 = 0.056509
X12 = 1.010125           E12 = 0.010125
||x-x~|| = 0.18806224504046665
```

Part 2:

```
-----------------LU Factorial ----------------------------------------------------
Upper - U
1.000 0.500 0.333 0.250 0.200 0.167 0.143 0.125 0.111 0.100 0.091 0.083
0.000 0.083 0.083 0.075 0.067 0.060 0.054 0.049 0.044 0.041 0.038 0.035
0.000 0.000 0.006 0.008 0.010 0.010 0.010 0.010 0.009 0.009 0.009 0.008
0.000 0.000 0.000 0.000 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.002
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000

Lower - L
1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.500 1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.333 1.000 1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.250 0.900 1.500 1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.200 0.800 1.714 2.000 1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.167 0.714 1.786 2.778 2.500 1.000 0.000 0.000 0.000 0.000 0.000 0.000
0.143 0.643 1.786 3.333 4.091 3.000 1.000 0.000 0.000 0.000 0.000 0.000
0.125 0.583 1.750 3.712 5.568 5.654 3.500 1.000 0.000 0.000 0.000 0.000
0.111 0.533 1.697 3.960 6.853 8.615 7.467 4.000 1.000 0.000 0.000 0.000
0.100 0.491 1.636 4.112 7.930 11.631 12.600 9.529 4.500 1.000 0.000 0.000
0.091 0.455 1.573 4.196 8.811 14.538 18.529 17.647 11.842 5.000 1.000 0.000
0.083 0.423 1.511 4.231 9.519 17.247 24.912 28.096 23.882 14.404 5.496 1.000
```

```
-----------------Inverse of U and L----------------------------------------------------------
InverseUpper - U^-1
1.000 -6.000 30.000 -140.000 630.000 -2772.000 12012.000 -51480.001 218789.130 -923630.235 3864521.779 -14846154.393
0.000 12.000 -180.000 1680.000 -12600.000 83160.000 -504504.002 2882880.052 -15752818.423 83127218.732 -425169315.197 1966968751.995
0.000 0.000 180.000 -4200.000 56700.000 -582120.000 5045040.021 -38918880.581 275674338.236 -1828807263.062 11481065140.154 -64110572830.028
0.000 0.000 0.000 2800.000 -88200.000 1552320.000 -20180160.081 216216002.730 -2021611910.191 17068930541.684 -132683790896.200 899615123104.549
0.000 0.000 0.000 0.000 44100.000 -1746360.000 37837800.150 -594594006.451 7581044967.041 -83211285655.304 812756538755.638 -6759839530390.263
0.000 0.000 0.000 0.000 0.000 698544.000 -33297264.130 856215368.082 -15768574068.356 232992179605.120 -2926127715335.261 30331758417574.540
0.000 0.000 0.000 0.000 0.000 0.000 11099088.043 -618377765.128 18396670284.400 -388321114927.605 6502888857920.057 -86054685266559.330
0.000 0.000 0.000 0.000 0.000 0.000 0.000 176679361.297 -11263267806.753 380396877653.955 -9024871566593.120 158240818061213.620
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 2815817014.190 -202086155408.567 7615067081287.886 -188098046280462.100
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 44908095538.747 -3572636582153.354 139453504120440.450
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 714551286274.742 -58615669198153.336
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 10664718303203.338

InverseLower - L^-1
1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
-0.500 1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.167 -1.000 1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
-0.050 0.600 -1.500 1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.014 -0.286 1.286 -2.000 1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
-0.004 0.119 -0.833 2.222 -2.500 1.000 0.000 0.000 0.000 0.000 0.000 0.000
0.001 -0.045 0.455 -1.818 3.409 -3.000 1.000 0.000 0.000 0.000 0.000 0.000
-0.000 0.016 -0.220 1.224 -3.365 4.846 -3.500 1.000 0.000 0.000 0.000 0.000
0.000 -0.006 0.098 -0.718 2.692 -5.600 6.533 -4.000 1.000 0.000 0.000 0.000
-0.000 0.002 -0.041 0.380 -1.853 5.188 -8.647 8.471 -4.500 1.000 0.000 0.000
0.000 -0.001 0.016 -0.186 1.137 -4.095 9.101 -12.630 10.657 -5.000 1.000 0.000
-0.000 0.000 -0.006 0.084 -0.634 2.844 -8.068 14.837 -17.637 13.076 -5.496 1.000
```

Inverse of A (For more detailed image, see the presentation) :

```
----------------Inverse of A-----------------------------------------------------------
InverseA - A^-1
141.558 -9986.670 230515.687 -2570261.643 16326820.932 -64350513.075 164562680.414 -277829365.178 307395474.267 -214370527.393 85461165.120 -14846154.393
-9986.581 942351.202 -24533411.750 292405565.562 -1938278017.327 7869835047.085 -20571928444.531 35323524327.223 -39611544703.252 27928538275.371 -11235918245.151 1966968751.995
230509.642 -24532996.485 682688025.804 -8490230355.528 57970440691.849 -240569995662.774 639511099924.488 -1112836727786.559 1261552833073.852 -897527468344.136 363842182172.637 -64110572830.028
-2570134.733 292394198.467 -8490050133.788 108755486988.147 -758933084044.575 3202666927699.829 -8627545100978.269 15176123347712.297 -17359009387125.654 12443628258758.914 -5077100533464.463 899615123104.549
16325579.266 -1938154722.380 57967855310.188 -758915823136.151 5385326220987.964 -23029126810316.543 62708889293528.010 -111295479451039.670 128264247897395.060 -92536981043353.020 37965826660334.820 -6759839530390.263
-64343784.039 7869129648.192 -240553451787.893 3202520207869.763 -23028606537375.605 99541672608634.860 -273479195649980.060 489021592568536.600 -567197348575277.000 411474310503833.440 -169633923415610.780 30331758417574.540
164540663.715 -20569542672.548 639451677180.323 -8626953921250.545 62706099756723.750 -273473303399766.800 756984395502949.900 -1362273360452339.500 1588779753674609.200 -1158133155887938.800 479472068322202.900 -8605468526655
-277784089.176 35318507793.453 -1112707019992.404 15174749975501.135 -111288200534274.690 489001116630666.300 -1362246010986804.000 2465042609984094.500 -2888731768027791.000 2114624167176623.000 -878739630784391.500 158240818
307336573.547 -39604913974.192 1261376975377.165 -17357073293643.645 128253316467117.050 -567162787173315.000 1588718248276979.000 -2888678453103799.000 3399476340655684.500 -2497803391949726.000 1041429534388116.800 -18809804
-214323485.929 27923179192.848 -897382703515.624 12441991388308.354 -92527364182183.020 411441904002014.500 -1158068316462167.200 2114549264102001.000 -2497761342337377.000 1841369776092772.800 -770029561187216.500 13945350412
85440118.271 -11233498308.348 363775900249.183 -5076336420709.667 37961213233037.160 -169617740339068.880 479437548297008.500 -878694786881987.600 1041395979448671.200 -770017787758622.500 322874871535260.400 -58615669198153.3
-14842115.902 1966500984.917 -64097621352.201 899463607160.415 -6758906385510.336 30328393511010.016 -86047211858886.580 158230466045797.100 -188089334566781.800 139449429540441.360 -58614856930328.250 10664718303203.338
```

```
----------------- Estimated X ( x~ = A^-1 * b) ----------------------------
x~ 1 = 44.255617
x~ 2 = -4456.445084
x~ 3 = 33650.806093
x~ 4 = 754269.469548
x~ 5 = -11927647.907219
x~ 6 = 72269853.471216
x~ 7 = -240906296.433417
x~ 8 = 488170436.717795
x~ 9 = -617637110.436587
x~ 10 = 477418119.347765
x~ 11 = -206500085.203225
x~ 12 = 38329272.877260
```

Section 3:

```
public class App {
  public static void main(String[] args) throws Exception {

      // First, define the tolerance
      double epsilon = 5E-6;

      // Define maximum number of iterations
      int maxIters = 500;

      // Define the initial values/seed (x0^0)
      double[] x0 = {0,0,0,0,0,0};

      // Define the matrix of coefficients
      double[][] M = {
         {4, -1, 0, -2, 0, 0}, // coefficients for first equation
         {-1, 4, -1, 0, -2, 0}, // second equation
         {0, -1, 4, 0, 0, -2}, // third
         {-1, 0, 0, 4, -1, 0}, // fourth
         {0, -1, 0, -1, 4, -1}, // ...
         {0, 0, -1, 0, -1, 4} // last equation
         };
```

```java
    // Define vector b (equalities)
    double[] b = {-1, 0, 1, -2, 1, 2};

    // Call method
    Jacobi(M, b, x0, epsilon, maxIters);
    System.out.println("");
    GaussSeidel(M, b, x0, epsilon, maxIters);
}

public static void Jacobi(double[][] M, double[] b, double[] x0, double epsilon, int maxIters)
{
    System.out.println(" *** JACOBI METHOD *** ");
    System.out.println("Solving the following system:");
    System.out.println("Number of variables: " + b.length);
    System.out.println("Toleance: " + epsilon);
    System.out.println("Max. number of iterations: " + maxIters);
    System.out.println("The system is:\n");

    // Define initial error
    double err = 1E+10; // a rally big number

    // Define number of equations/variables
    int N = b.length;

    // Print equations so user can see them
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            System.out.print(M[i][j]+"*x" + (j+1) + " ");
        }
        System.out.print("= " + b[i] + "\n");
    }
    System.out.println("");


    System.out.println("Starting iterations... ");
    // Start iterations
    double[] xold = x0; // varray to store old values (from previous iteration)
    double[] xnew = xold.clone(); // array to store the result of the new iteration
    int iterCounter = 0; // variable to count the iterations

    double xi, erri; // helper variables to be used in the method

    while(err > epsilon)
    {
        iterCounter++;

        // Jacobi Method
        for(int i = 0; i < N ;i++)
        {
            xi = b[i];
            for(int j = 0; j < N; j++)
            {
```

```
                if(i != j)
                    xi -= M[i][j] * xold[j];
            }
            xi = xi/M[i][i];
            xnew[i] = xi;
        }

        // Calculate errors and pick maximum

        /*
            For this method the error is calculated as the difference between the old solution and
            the new one.
        */
        double maxErr = -1;
        for(int i = 0; i < N; i++)
        {
            erri = Math.abs((xnew[i]-xold[i]));
            if(erri > maxErr)
                maxErr = erri;
        }
        xold = xnew.clone(); // For the next iteration, the old_values are the new_values in this one
        err = maxErr;
        System.out.println("Iteration " + iterCounter + ", Err: " + err);
        if(iterCounter == maxIters) // reached maximum number of iterations
        {
            System.out.println("Maximum number of iterations reached. Last error: " + err);
            break;
        }
    }

    // Display results only if the system converged
    if(iterCounter < maxIters && err <= epsilon)
    {
        System.out.println("");
        System.out.println("Convergence achieved in " + iterCounter + " iterations. Min error was: " +
err);
        System.out.println("The solution is:");
        for(int i = 0; i < N; i++)
        {
            System.out.println("x[" + (i+1) + "] = " + xnew[i]);
        }
    }
}

public static void GaussSeidel(double[][] M, double[] b, double[] x0, double epsilon, int maxIters)
{
    System.out.println(" *** GAUSS-SEIDEL METHOD *** ");
    System.out.println("Solving the following system:");
    System.out.println("Number of variables: " + b.length);
    System.out.println("Toleance: " + epsilon);
    System.out.println("Max. number of iterations: " + maxIters);
    System.out.println("The system is:\n");

    // Define initial error
    double err = 1E+10; // a rally big number
```

```
// Define number of equations/variables
int N = b.length;

// Display equations
for(int i = 0; i < N; i++)
{
    for(int j = 0; j < N; j++)
    {
        System.out.print(M[i][j]+"*x" + (j+1) + " ");
    }
    System.out.print("= " + b[i] + "\n");
}
System.out.println("");
System.out.println("Starting iterations... ");

// Start iterations
double[] x = x0; // vector of solutions is setted to the initial values
int iterCounter = 0; // variable to count interations

double xi, erri; // helper variables
double sigma; // helper variable for gauss-Seidel method

// Start iterations
while(err > epsilon)
{
    iterCounter++;

    // Gauss-Seidel method
    for(int i = 0; i < N ;i++)
    {
        sigma = 0;
        for(int j = 0; j < N; j++)
        {
            if(i != j)
                sigma += M[i][j]*x[j];
        }
        xi = (b[i]-sigma)/M[i][i];
        x[i] = xi;
    }

    // Calculate errors and pick maximum
    /*
        For this method the error is calculated as the value of the functions for the current solution.
        The method converges when the value of the function f(xi)-bi = 0 is less than epsilon
    */
    double maxErr = -1;
    for(int i = 0; i < N; i++)
    {
        erri = 0;
        for(int j = 0; j < N; j++)
        {
            erri += M[i][j]*x[j];
        }
        erri -= b[i];
```

```
            erri = Math.abs(erri);
            if(erri > maxErr)
                maxErr = erri;
        }
        //xold = xnew.clone(); // the results of this iterations are the old for the next iteration
        err = maxErr;
        System.out.println("Iteration " + iterCounter + ", Err: " + err);
        if(iterCounter == maxIters) // reached max number of iters
        {
            System.out.println("Maximum number of iterations reached. Last error: " + err);
            break;
        }
    }

    if(iterCounter < maxIters && err <= epsilon) // display results only if system converged
    {
        System.out.println("");
        System.out.println("Convergence achieved in " + iterCounter + " iterations. Min error was: " +
err);
        System.out.println("The solution is:");
        for(int i = 0; i < N; i++)
        {
            System.out.println("x[" + (i+1) + "] = " + x[i]);
        }
    }
  }
}
```

There are two main iterative methods which are Jacobi and Gauss Seidel. Started by implementing 5×10^{-6} and defining it as epsilon for the termination of the iteration. The initializing the coefficient matrix to M and vector b to b.

For the Jacobi method:

So within the code, epsilon is defined for the termination of the iteration. Here we have the coefficient matrix and b values defined. Then there are call methods to separately call each method.

Within the code: it defines the initial error which is one times ten to the tenth. It then defines the number of variables or equations. Then prints the equation for the user using a nested for loop followed by displaying the contents within the matrix. The code then starts iteration by defining what the old value and new value would be. Then we have helper variables such as xi and erri that are to be used in the method.

There is a while loop that tells us if the initial error is greater than epsilon, then it will increase the iteration count and then perform the Jacobi method. The code then calculates the error and picks the maximum which is calculated as the difference of the new value. The maximum error is defined as negative 1. The code then uses a for loop and an if statement to find the maximum error. Then we are defining what x_old is, for the next iteration which is where the old values become new ones.

Below that defines what the overall error is. Then prints out the iterations and the errors.

The code then uses an if statement with the conditions of iteration count and maximum. Iterations must equal each other in order to determine when the maximum number of iterations was truly reached. Within that statement we print the last error. We then display the results if the system converges. By using an if statement with the conditions stating: if the iteration count is less than maximum iteration and error is less than or equal to epsilon. Then it prints out the amount of iterations needed and the minimum error along with the count of each x such as x1, x2 and so on. From there we present the final outputs.

For the Gauss Seidel method:

The code for the Gauss Seidel method is much like the Jacobi however we went through each iteration updating values of x1, x2, x3, x4, x5, and x6 to our most recently calculated value. We iterate until we reach a point where our error is lower than our tolerance value/epsilon value of 5×10^{-6}. At this point we know we have reached convergence, and we can record the number of iterations needed and display that number in our output.

Output:

Jacobi:

```
 *** JACOBI METHOD ***
Solving the following system:
Number of variables: 6
Toleance: 5.0E-6
Max. number of iterations: 500
The system is:

4.0*x1 -1.0*x2 0.0*x3 -2.0*x4 0.0*x5 0.0*x6 = -1.0
-1.0*x1 4.0*x2 -1.0*x3 0.0*x4 -2.0*x5 0.0*x6 = 0.0
0.0*x1 -1.0*x2 4.0*x3 0.0*x4 0.0*x5 -2.0*x6 = 1.0
-1.0*x1 0.0*x2 0.0*x3 4.0*x4 -1.0*x5 0.0*x6 = -2.0
0.0*x1 -1.0*x2 0.0*x3 -1.0*x4 4.0*x5 -1.0*x6 = 1.0
0.0*x1 0.0*x2 -1.0*x3 0.0*x4 -1.0*x5 4.0*x6 = 2.0
```

```
The solution is:
x[1] = -0.4464295251028877
x[2] = 0.2499980926513672
x[3] = 0.6964276177542549
x[4] = -0.5178580965314592
x[5] = 0.3749990463256836
x[6] = 0.7678561891828264
```

```
Iteration 1, Err: 0.5
Iteration 2, Err: 0.25
Iteration 3, Err: 0.09375
Iteration 4, Err: 0.0625
Iteration 5, Err: 0.03515625
Iteration 6, Err: 0.03125
Iteration 7, Err: 0.01611328125
Iteration 8, Err: 0.015625
Iteration 9, Err: 0.00787353515625
Iteration 10, Err: 0.0078125
Iteration 11, Err: 0.00391387939453125
Iteration 12, Err: 0.00390625
Iteration 13, Err: 0.0019540786743164062
Iteration 14, Err: 0.001953125
Iteration 15, Err: 9.766817092895508E-4
Iteration 16, Err: 9.765625E-4
Iteration 17, Err: 4.882961511611938E-4
Iteration 18, Err: 4.8828125E-4
Iteration 19, Err: 2.4414248764514923E-4
Iteration 20, Err: 2.44140625E-4
Iteration 21, Err: 1.2207054533064365E-4
Iteration 22, Err: 1.220703125E-4
Iteration 23, Err: 6.103518535383046E-5
Iteration 24, Err: 6.103515625E-5
Iteration 25, Err: 3.051758176297881E-5
Iteration 26, Err: 3.0517578125E-5
Iteration 27, Err: 1.525878951724735E-5
Iteration 28, Err: 1.52587890625E-5
Iteration 29, Err: 7.629394588093419E-6
Iteration 30, Err: 7.62939453125E-6
Iteration 31, Err: 3.814697272730427E-6
Iteration 32, Err: 3.814697265625E-6
Iteration 33, Err: 1.9073486337006784E-6
Iteration 34, Err: 1.9073486328125E-6
Iteration 35, Err: 9.536743165172723E-7

Convergence achieved in 35 iterations. Min error was: 9.536743165172723E-7
```

Gauss-Seidel Method:

```
 *** GAUSS-SEIDEL METHOD ***
Solving the following system:
Number of variables: 6
Toleance: 5.0E-6
Max. number of iterations: 500
The system is:

4.0*x1 -1.0*x2 0.0*x3 -2.0*x4 0.0*x5 0.0*x6 = -1.0
-1.0*x1 4.0*x2 -1.0*x3 0.0*x4 -2.0*x5 0.0*x6 = 0.0
0.0*x1 -1.0*x2 4.0*x3 0.0*x4 0.0*x5 -2.0*x6 = 1.0
-1.0*x1 0.0*x2 0.0*x3 4.0*x4 -1.0*x5 0.0*x6 = -2.0
0.0*x1 -1.0*x2 0.0*x3 -1.0*x4 4.0*x5 -1.0*x6 = 1.0
0.0*x1 0.0*x2 -1.0*x3 0.0*x4 -1.0*x5 4.0*x6 = 2.0
```

```
The solution is:
x[1] = -0.44643071719578326
x[2] = 0.24999785423278809
x[3] = 0.6964274985449654
x[4] = -0.5178582157407488
x[5] = 0.37499892711639404
x[6] = 0.7678566064153398
```

```
Starting iterations...
Iteration 1, Err: 1.1875
Iteration 2, Err: 0.580078125
Iteration 3, Err: 0.230712890625
Iteration 4, Err: 0.1307373046875
Iteration 5, Err: 0.0690765380859375
Iteration 6, Err: 0.03500175476074219
Iteration 7, Err: 0.017558813095092773
Iteration 8, Err: 0.008786648511886597
Iteration 9, Err: 0.004394229501485825
Iteration 10, Err: 0.002197227906435728
Iteration 11, Err: 0.001098628097679466
Iteration 12, Err: 5.493158168974333E-4
Iteration 13, Err: 2.7465812945592916E-4
Iteration 14, Err: 1.3732909235386614E-4
Iteration 15, Err: 6.866454963017077E-5
Iteration 16, Err: 3.4332275246740096E-5
Iteration 17, Err: 1.7166137677326887E-5
Iteration 18, Err: 8.58306884521376E-6
Iteration 19, Err: 4.291534423828125E-6

Convergence achieved in 19 iterations. Min error was: 4.291534423828125E-6
```

6. Conclusion:

The three parts in this project reuse many methods. We learned to use Gauss-Seidel Elimination Operation on different types of problems. Section 1 is the basic of GE operation, where we later reuse its code as the basic to solve Section 2 and 3. In Section 2, we did the comparison and found that Scaled pivoting GE has the smallest error. In part 2 of section 2, we learned how to estimate x using LU factorization. LU factorization doesn't work well with small magnitude matrix. In Section 3, we did a comparison with Jacobi's method and Gauss-Seidel's method and found that Gauss-Seidel's method is almost double in efficiency (19 iterations vs 35 iterations).

7. Contribution/Evaluation:

Julia: Implemented the Gaussian Elimination algorithms for Section 1 Problem 3 and Section 2.

Nhi: Section 1, finding the matrix A and B. Section 2(i), writing the code for Gaussian Elimination with scaled partial pivoting and error analysis algorithm.

Luong: Section 2 - part 2, wrote code, edited and finalized the project's document. Recorded and edited the presentation.

Kimberly: Section 3 Problem 1. Wrote code to solve the problem. Presented the Jacobi method section in our presentation.

Peter: Section 3 Problem 1. Wrote code to solve the problem. Presented the Gauss-Seidel method section in our presentation.