Project 2

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

Group 8 – Java:
Peter Boukhalil
Luong Dang
Kimberly Nivon
Julia Beatriz Ramos
Nhi Vu

Section 1 - Problem 3

3. In calculus the following integral would be found by the technique of partial fractions:

$$\int \frac{x^2 + x + 1}{(x-1)(x-2)(x-3)^2(x^2+1)} dx.$$

This would require finding the coefficients A_i, for $i = 1, 2, \ldots, 6$, in the expression

$$\frac{x^2 + x + 1}{(x-1)(x-2)(x-3)^2(x^2+1)} = \frac{A_1}{x-1} + \frac{A_2}{x-2} + \frac{A_3}{(x-3)^2} + \frac{A_4}{x-3} + \frac{A_5 x + A_6}{x^2+1}.$$

Find the partial fraction coefficients.

Section 1 - Problem 3 - Finding Matrix A

$$\frac{x^2+x+1}{(x-1)(x-2)(x-3)^2(x^2+1)} = \frac{A_1}{x-1} + \frac{A_2}{x-2} + \frac{A_3}{(x-3)^2} + \frac{A_4}{x-3} + \frac{A_5x+A_6}{x^2+1}$$

Then we get:

$$\begin{aligned}
x^2+x+1 =& A_1(x-2)(x-3)^2(x^2+1)+\\
& A_2(x-1)(x-3)^2(x^2+1)+\\
& A_3(x-1)(x-2)((x^2+1)+\\
& A_4(x-1)(x-2)(x-3)(x^2+1)+\\
& (A_5x+A_6)(x-1)(x-2)(x-3)^2\\
x^2+x+1 =& x^5(A_1+A_2+A_4+A_5)+\\
& x^4(-8A_1-7A_2+A_3-6A_4-9A_5+A_6)+\\
& x^3(22A_1+16A_2-3A_3+12A_4+29A_5-9A_6)+\\
& x^2(-26A_1-16A_2+3A_3-12A_4-39A_5+29A_6)+\\
& x(21A_1+15A_2-3A_3+11A_4+18A_5-39A_6)+\\
& (-18A_1-9A_2+2A_3-6A_4+18A_6)
\end{aligned}$$

Section 1 - Problem 3 - Finding Matrix A

By comparison, we get:

$$A_1 + A_2 + A_3 + A_5 = 0$$
$$-8A_1 - 7A_2 + A_3 - 6A_4 - 9A_5 + A_6 = 0$$
$$22A_1 + 16A_2 - 3A_3 + 12A_4 + 29A_5 - 9A_6 = 0$$
$$-26A_1 - 16A_2 + 3A_3 - 12A_4 - 39A_5 + 29A_6 = 1$$
$$21A_1 + 15A_2 - 3A_3 + 11A_4 + 18A_5 - 39A_6 = 1$$
$$-18A_1 - 9A_2 + 2A_3 - 6A_4 + 18A_6 = 1$$

Shown in the matrix:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 \\ -8 & -7 & 1 & -6 & -9 & 1 \\ 22 & 16 & -3 & 12 & 29 & -9 \\ -26 & -16 & 3 & -12 & -39 & 29 \\ 21 & 15 & -3 & 11 & 18 & -39 \\ -18 & -9 & 2 & -6 & 0 & 18 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Section 1 - Problem 3 GE Implementation

```java
// Performs Gaussian Elimination using matrix A and solution matrix B (algorithm from notes)
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];
        }
    }
```

```java
    // 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);
}
```

Section 1 - Problem 3 Main Method

```java
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);
}
```

Section 1 - Problem 3 Output

$$
\begin{bmatrix}
1 & 1 & 1 & 0 & 1 & 0 \\
-8 & -7 & 1 & -6 & -9 & 1 \\
22 & 16 & -3 & 12 & 29 & -9 \\
-26 & -16 & 3 & -12 & -39 & 29 \\
21 & 15 & -3 & 11 & 18 & -39 \\
-18 & -9 & 2 & -6 & 0 & 18
\end{bmatrix}
\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \end{bmatrix}
=
\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}
$$

```
"F:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...

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

Section 2

For this problem, we will implement LU factorization, Gaussian elimination with no pivoting, with partial pivoting, and with scaled partial pivoting. You can find extensive literature online on scaled partial pivoting. Please cite the sources you used.

(i) Let A be the $n \times n$ matrix whose entries are given by $a_{ij} = 1/(i+j-1)$ for $1 \leq i, j, \leq n$. Solve the system $A\mathbf{x} = \mathbf{b}$ for $n = 12$. Take $\mathbf{b}$ as the vector that corresponds to an exact solution of $x_i = 1$ for each $i = 1, 2, 3, \ldots, n$. Compare the solutions obtained using Gaussian elimination without pivoting, with partial pivoting and with scaled partial pivoting. Which technique provided the most accurate solution?

(ii) Find an LU factorization of A and use the factorization to find A^{-1} and $\tilde{\mathbf{x}} = A^{-1}\mathbf{b}$.

Section 2 - Part (i) Main Method

```java
// 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: ");
    printMatrix(A);

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

Section 2 - Part (i) - Finding A and b

```java
// 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;
}
```

```
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
```

Section 2 - Part (i)
No Pivoting + Output

```java
// 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);
}
```

```
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
```

Section 2 - Part (i)
Partial Pivoting + Output

```java
// 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
}
```

```
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
```

```java
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], index: 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, index: 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];
        }
    }
```

Back substitution and print output

Section 2 - Part (i) Scaled Partial Pivoting + Output

Algorithm from:
https://www.youtube.com/watch?v=4YzIfcSFVCU&ab_channel=ThomasBingham

```
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
```

Section 2 - Part (ii)

(ii) Find an LU factorization of A and use the factorization to find A^{-1} and $\tilde{\mathbf{x}} = A^{-1}\mathbf{b}$.

Reused methods from Part 1:

- initA() , initB()
- printMatrix()
- printVector()
- Gaussian elimination algorithm

New methods:

- multiplyMatrices()
- multiplyMatrixWithVector()
- findInverseLowerTri()
- findInverseUpperTri()

```
// 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: ");
    printMatrix(A);

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

    LUDecomposition(A); // Part2
```

1) Find L and U matrices

Reused Gauss Elimination Algorithm with modification

a)Upper Matrix = row reduced matrix of A

b)Lower Matrix = multipliers from the Gauss Elimination Operation

```
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;
        }
    }
}
```

Easier to load data into each matrix separately, so we create a temporary matrix, which equals to matrix A. Then have a for-loop to copy tempMatrix's data into upperMatrix after the elimination.

L and U matrix

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

Find Inverse of Matrix A:

```java
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;

}
```

```java
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;
}
```

Assumption:

A x= LUx = B
=> A= LU
=> A^-1= U^-1 * L^-1

X = A^-1 * B

To find Inverse of a matrix, reuse back substitution algorithm with modification

Inverse of a matrix is the solution of Back-substitution Operation where B is Identity and B = {b1, b2, b3, ..bn}

Ex:

$$A|I = \begin{bmatrix} 2 & 3 & 1 & | & 1 \\ 3 & 3 & 1 & | & 0 \\ 2 & 4 & 1 & | & 0 \end{bmatrix}$$ = I|a1^-1

$$A|I = \begin{bmatrix} 2 & 3 & 1 & | & 0 \\ 3 & 3 & 1 & | & 1 \\ 2 & 4 & 1 & | & 0 \end{bmatrix}$$ = I|a2 ^-1

$$A|I = \begin{bmatrix} 2 & 3 & 1 & | & 0 \\ 3 & 3 & 1 & | & 0 \\ 2 & 4 & 1 & | & 1 \end{bmatrix}$$ = I|a3 ^-1

A^-1 = {a1 ^-1,a2 ^-1,a3 ^-1}

```java
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;
}
```

L and U matrix

```
-----------------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 L and U matrix

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

```
-----------------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, where x~ = A^-1 * b

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

Section 3

Use both the Jacobi method and the Gauss-Seidel method to solve the indicated linear system of equations. Your code should efficiently use the "sparseness" of the coefficient matrix. Take $\mathbf{x}^{(0)} = \mathbf{0}$, and terminate the iteration when $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_\infty$ falls below 5×10^{-6}. Record the number of iterations required to achieve convergence.

$$
\begin{array}{rcrcrcrcrcrcr}
4x_1 & - & x_2 & & & - & 2x_4 & & & & & = & -1 \\
-x_1 & + & 4x_2 & - & x_3 & & & - & 2x_5 & & & = & 0 \\
& - & x_2 & + & 4x_3 & & & & & - & 2x_6 & = & 1 \\
-x_1 & & & & & + & 4x_4 & - & x_5 & & & = & -2 \\
& - & x_2 & & & - & x_4 & + & 4x_5 & - & x_6 & = & 1 \\
& & & - & x_3 & & & - & x_5 & + & 4x_6 & = & 2
\end{array}
$$

Section 3

Use both the Jacobi method and the Gauss-Seidel method to solve the indicated linear system of equations. Your code should efficiently use the "sparseness" of the coefficient matrix. Take $\mathbf{x}^{(0)} = \mathbf{0}$, and terminate the iteration when $||\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}||_\infty$ falls below 5×10^{-6}. Record the number of iterations required to achieve convergence.

$$\begin{array}{rcrcrcrcrcrcr}
4x_1 & - & x_2 & & & - & 2x_4 & & & & & = & -1 \\
-x_1 & + & 4x_2 & - & x_3 & & & - & 2x_5 & & & = & 0 \\
& - & x_2 & + & 4x_3 & & & & & - & 2x_6 & = & 1 \\
-x_1 & & & & & + & 4x_4 & - & x_5 & & & = & -2 \\
& - & x_2 & & & - & x_4 & + & 4x_5 & - & x_6 & = & 1 \\
& & & - & x_3 & & & - & x_5 & + & 4x_6 & = & 2
\end{array}$$

```java
// 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
};

// 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);
```

Jacobi Method

Approximation of x$^{(k)} = (x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \ldots, x_n^{(k)})$ to find new values: $\boldsymbol{x}^{(k+1)} = (x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_n^{(k+1)})$

```java
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
```

```java
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
    the new one.
*/
    double maxErr = -1;
    for(int i = 0; i < N; i++)
    {
        erri = Math.abs((xnew[i]-xold[i]));
        if(erri > maxErr)
```

```java
    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 i
    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 err
    System.out.println("The solution is:");
    for(int i = 0; i < N; i++)
    {
        System.out.println("x[" + (i+1) + "] = " + xnew[i]);
    }
}
```

$$
\begin{array}{ccccccccc}
a_{11}x_1^{(k+1)} & + & a_{12}x_2^{(k)} & + & \cdots & + & a_{1n}x_n^{(k)} & = & b_1 \\
a_{21}x_1^{(k)} & + & a_{22}x_2^{(k+1)} & + & \cdots & + & a_{2n}x_n^{(k)} & = & b_2 \\
\vdots & & \vdots & & \ddots & & \vdots & & \vdots \\
a_{n1}x_1^{(k)} & + & a_{n2}x_2^{(k)} & + & \cdots & + & a_{nn}x_n^{(k+1)} & = & b_n
\end{array}
$$

- Starts with x (0) = 0
- Use old values to calculate the new value within equation
- We then calculate the absolute error
- Iteration occurs to find a sequence of increasingly better approximation

Jacobi Method - Output

```
 *** 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
```

```
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
```

The new values are:

```
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
```

35 Iterations required to achieve convergence

Gauss-Seidel Method

$$
\begin{array}{ccccccccc}
a_{11}x_1^{(k+1)} & + & a_{12}x_2^{(k)} & + & \cdots & + & a_{1n}x_n^{(k)} & = & b_1 \\
a_{21}x_1^{(k)} & + & a_{22}x_2^{(k+1)} & + & \cdots & + & a_{2n}x_n^{(k)} & = & b_2 \\
\vdots & & \vdots & & \ddots & & \vdots & & \vdots \\
a_{n1}x_1^{(k)} & + & a_{n2}x_2^{(k)} & + & \cdots & + & a_{nn}x_n^{(k+1)} & = & b_n
\end{array}
$$

That is..

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

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

Where **true** solution is $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$

- Similar to Jacobi Method except we solve this using the new and old values to get the new value

```java
// 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];
```

```java
        }
        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;
    }
```

```java
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]);
    }
}
```

Gauss-Seidel Method- Output

```
 *** 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 new values are:

```
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
```

19 Iterations required to achieve convergence

The End