CFP Project Report

Muhammad Umer Saeed (Sec. B)
Ch Muhammad Musa (Sec. B)
Hussain Raza (Sec. A)
Abdul Wassay (Sec. A)

December 23, 2024

—

Department of Electrical Engineering

—

First Semester

—

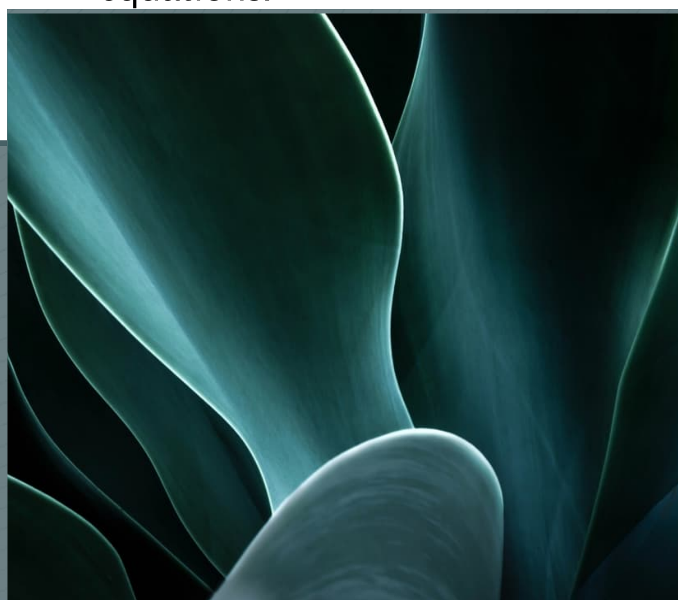
Dr. Muhammad Tufail

Matrices Operations & System of Linear Equations

<u>Objective</u>:

The main objectives of our project were as follows:

- To calculate the determinant of a matrix.
- To compute the inverse of a matrix (if it exists).
- To find the Row Echelon Form.
- To find the Row Reduced Echelon Form (RREF).
- To determine the rank of a matrix.
- To solve systems of linear equations.



Methodology

Algorithm

1) Matrix Categorization:
   The input matrix is categorized based on its order as either a square or rectangular matrix. It is noted that the inverse of a rectangular matrix cannot be computed.
2) Determinant Calculation:
   The determinant is evaluated to classify the matrix as either singular or non-singular.
3) Row Echelon Form:
   The matrix provided by the user is transformed into its Row Echelon Form using the Gauss-Jordan elimination method.
4) Reduced Row Echelon Form:
   The matrix provided by the user is transformed into its Reduced Row Echelon Form using the Gauss-Jordan elimination method.
5) Matrix Inversion:
   For square, non-singular matrices, the inverse is calculated using the Gauss-Jordan elimination method.
6) Rank Determination:
   The rank of the matrix is determined after transforming it into its Reduced Row Echelon Form.
7) System of Linear Equations Solution:
   The system of linear equations is represented as an augmented matrix. The solution is derived by applying the Gauss-Jordan elimination method to the augmented matrix.

Implementation

The program is structured with multiple functions, each developed for a specific purpose. For example, the inverse function takes a matrix as its parameter, constructs an identity matrix, and then computes the inverse of the given matrix using the Gauss-Jordan method.

These functions are interdependent. For instance, the function designed to solve a system of linear equations utilizes the RREF function, which is also implemented using the Gauss-Jordan method, to determine the solutions.

Output

1) Display the title using Title() func:

```
"Program for Matrix Operations and System of Linear Equations"
```

2) Input the Order of Matrix using OrderOfMatrix() func:

```
Enter the order of your matrix: -

Enter the number of rows for the matrix: 5
Enter the number of columns for the matrix: 5
```

- Input cannot be a char or a string.

```
Enter the order of your matrix: -

Enter the number of rows for the matrix: t
Invalid input! Please enter a valid number of rows between 1 and 8.
Enter the order of your matrix: -

Enter the number of rows for the matrix: 3
Enter the number of columns for the matrix: 5
```

3) Categorize the matrix into square & rectangular.

```
Enter the number of rows for the matrix: 3
Enter the number of columns for the matrix: 3

Enter the elements of your square matrix: -
```

or

```
Enter the number of rows for the matrix: 2
Enter the number of columns for the matrix: 5

Enter the elements of your rectangular matrix: -
```

4) Ask for matrix elements using InputMatrix() func:

```
Enter the elements of your square matrix: -

Enter element number 1x1 : 2
Enter element number 1x2 : 7
Enter element number 1x3 : 9
Enter element number 1x4 : 0
Enter element number 1x5 : -7
Enter element number 2x1 : 9
Enter element number 2x2 : 0
Enter element number 2x3 : 5
Enter element number 2x4 : 0
Enter element number 2x5 : 2
Enter element number 3x1 : -4
Enter element number 3x2 : 11
Enter element number 3x3 : 4
Enter element number 3x4 : 22
Enter element number 3x5 : 4
Enter element number 4x1 : 3
Enter element number 4x2 : 65
Enter element number 4x3 : 1
Enter element number 4x4 : 7
Enter element number 4x5 : 1
Enter element number 5x1 : 0
Enter element number 5x2 : 6
Enter element number 5x3 : -5
Enter element number 5x4 : 2
Enter element number 5x5 : 12
```

- Input cannot be a char or a string.

```
Enter the elements of your square matrix: -

Enter element number 1x1 : 5
Enter element number 1x2 : 6
Enter element number 1x3 : u
Invalid input! Please enter a valid number.
Enter element number 1x3 : r
Invalid input! Please enter a valid number.
Enter element number 1x3 : 3
Enter element number 2x1 :
```

5) Display the user's matrix using OutputMatrix() func:

```
Your matrix is:

|  2.0    7.0    9.0    0.0   -7.0  |
|  9.0    0.0    5.0    0.0    2.0  |
| -4.0   11.0    4.0   22.0    4.0  |
|  3.0   65.0    1.0    7.0    1.0  |
|  0.0    6.0   -5.0    2.0   12.0  |
```

6) If square matrix, then check if it is invertible (inverse exists) by calculating it's determinant by Determinant() func:

```
The determinant of this square matrix is -686393.0 (non-singular matrix).
Therefore, the inverse of the matrix exists (invertible).

The Inverse of this Matrix is:

| -0.136    0.134    0.002    0.024   -0.104  |
|  0.011   -0.010   -0.005    0.014    0.008  |
|  0.207   -0.035   -0.001   -0.034    0.130  |
| -0.085    0.038    0.050    0.007   -0.073  |
|  0.095   -0.016   -0.006   -0.023    0.145  |
```

- If rectangular matrix, then inverse does not exist

```
Your matrix is:

|  1.0    2.0  |
|  3.0    1.0  |
|  2.0    3.0  |

This is a rectangular matrix.
Therefore, the Determinant and Inverse of this matrix does not exist.
```

7) Convert the matrix to Echelon form using Echelon() func & display the REF matrix using OutputMatrix():

```
The Echelon form of this Matrix is:

|  1.0    3.5    4.5    0.0   -3.5  |
|  0.0    1.0    0.4    2.0    0.4  |
|  0.0    0.0    1.0    0.0    0.4  |
|  0.0    0.0    0.0    1.0    0.1  |
|  0.0    0.0    0.0    0.0    1.0  |
```

8) Convert the matrix to Reduced Echelon form using RREF() func & then display the RREF matrix using OutputMatrix() func:

```
The RREF of this Matrix is:

|  1.0    0.0    0.0    0.0    0.0  |
|  0.0    1.0    0.0    0.0    0.0  |
|  0.0    0.0    1.0    0.0    0.0  |
|  0.0    0.0    0.0    1.0    0.0  |
|  0.0    0.0    0.0    0.0    1.0  |
```

9) Find & display the Rank of the matrix using Rank() func:

```
The rank of this Matrix is 5
```

10) Ask user to enter the elements of Aug. matrix using InputMatrix() func & display the Aug. Matrix:

```
Enter element number 4x3 : 4
Enter element number 4x4 : 8
Enter element number 4x5 : 1
Enter element number 4x6 : 6
Enter element number 5x1 : 5
Enter element number 5x2 : 2
Enter element number 5x3 : 7
Enter element number 5x4 : 6
Enter element number 5x5 : 88
Enter element number 5x6 : 12

Your augumented matrix is:
|  2.0    7.0    9.0    3.0    7.0   :  1.0  |
|  9.0    0.0    5.0    0.0    2.0   : 11.0  |
|  4.0    1.0    4.0    2.0    4.0   :  3.0  |
|  1.0    6.0    4.0    8.0    1.0   :  6.0  |
|  5.0    2.0    7.0    6.0   88.0   : 12.0  |
```

11) Calculate and display the solution of the matrix using SystemOfLinearEq() & OutputMatrix() func:

```
The solutions are:
| 2.55|
| 2.52|
| -2.45|
| -0.25|
| 0.14|

* ------------------------------------------------- *
```

```cpp
// Libraries
#include <iostream>
#include <iomanip>
#include <limits>
using namespace std;

const int MaxRows = 8;
const int MaxCols = 8;

// Function Prototypes
// Function to display a title at the start of the program
void inline Title();
// Funtion to get the order of the matrix (keep asking for input if the input entered is not an int)
void inline OrderOfMatrix(unsigned int& rows, unsigned int& cols);
// Function to take elements of a matrix as input (Takes in only the constant vectors if the bool "NewInputForAug" is false)
void InputMatrix(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols);
// Function to display the matrix (display an Augmented matrix if the bool "Augmented" is true)
void OutputMatrix(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols, int Precision, bool Augmented, bool DisplayLastColumnOnly);
// Function to find the determinant of the matrix using recursion
float Determinant(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols);
// Function to send the rows with leading entery zeros to the end
void Swap(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols);
// Function to normalize the values below pivot
void Normalizer(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols);
// Program to convert a matrix to Echelon form (used Swap and Normalizer func)
void Echelon(float OriginalMatrix[MaxRows][MaxCols], int r, int c);
// Program to convert a matrix to Reduced Echelon form (used Swap and Normalizer func)
void RREF(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols);
// Program to find the rank of a matrix (uses the RREF func)
int Rank(float matrix[MaxRows][MaxCols], int rows, int cols);
// Function to find the inverse of a matrix (uses RREF func)
void Inverse(float OriginalMatrix[MaxRows][MaxCols], float InverseMatrix[MaxRows][MaxCols], int rows, int cols);
// Function to solve a system of linear eq (uses RREF func)
void SystemOfLinearEq(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols);
// Function to display the solution of system of linear eq
void inline EndLine();

int main()
{
    // Displays the Title
    Title();
    // Variables required in main func
```

```
    float OriginalMatrix[MaxRows][MaxCols], InverseMatrix[MaxRows][MaxCols];
    unsigned int rows, cols;
    bool InverseExists = 0;
    // Input the order of the matrix
    OrderOfMatrix(rows, cols);
    cout << endl;

    // Categorize the matrix as square or rectangular
    // SQUARE MATRIX
    if (rows == cols)
    {
        // Input the matrix elements
        cout << "Enter the elements of your square matrix: -\n\n";
        InputMatrix(OriginalMatrix, rows, cols);
        cout << endl;
        // Display the square matrix
        cout << "Your matrix is: \n\n";
        OutputMatrix(OriginalMatrix, rows, cols, 1, 0, 0);
        cout << endl;

        float det = Determinant(OriginalMatrix, rows, cols);
        if (det == 0)   // Det is zero
        {
            cout << "The determinant of this square matrix is zero (singular matrix).\nTherefore, the inverse of the matrix does not exist (non-invertible).\n";
            InverseExists = false;
        }
        else    // Det is non-zero. The function of det returns -1 to avoid unnecessary calculation of inverse.
        {
            cout << "The determinant of this square matrix is " << det << " (non-singular matrix). \nTherefore, the inverse of the matrix exists (invertible).\n";
            InverseExists = true;
        }
    }

    // RECTANGULAR MATRIX
    else
    {
        // Input the matrix elements
        cout << "Enter the elements of your rectangular matrix: -\n\n";
        InputMatrix(OriginalMatrix, rows, cols);
        cout << endl;
        // Display the rectangular Matrix
        cout << "Your matrix is: \n\n";
        OutputMatrix(OriginalMatrix, rows, cols, 1, 0, 0);
        cout << endl;
        cout << "This is a rectangular matrix. \nTherefore, the Determinant and Inverse of this matrix does not exist.\n";
        InverseExists = false;
```

```cpp
    }
    cout << endl;

    // ROW ECHELON
    cout << "The Echelon form of this Matrix is: \n\n";
    Echelon(OriginalMatrix, rows, cols);
    OutputMatrix(OriginalMatrix, rows, cols, 1, 0, 0);
    cout << endl;

    // INVERSE
    if (InverseExists)
    {
        // Output the Inverse
        cout << "The Inverse of this Matrix is: \n\n";
        Inverse(OriginalMatrix, InverseMatrix, rows, cols);
        OutputMatrix(InverseMatrix, rows, cols, 3, 0, 0);
    }
    cout << endl;

    // REDUCED ROW ECHELON
    cout << "The RREF of this Matrix is: \n\n";
    RREF(OriginalMatrix, rows, cols);
    OutputMatrix(OriginalMatrix, rows, cols, 1, 0, 0);
    cout << endl;

    // RANK
    int rank = Rank(OriginalMatrix, rows, cols);
    cout << "The rank of this Matrix is " << rank << "\n\n";

    // SYSTEM OF LINEAR EQ.
    cout << "Now, solving the system of linear equations:\n\n";
    cout << "Enter the new elements for your Augumented matrix (including the constants): \n\n";
    InputMatrix(OriginalMatrix, rows, cols + 1);
    cout << endl;
    // Display the Augmented Matrix
    cout << "Your augumented matrix is:\n";
    OutputMatrix(OriginalMatrix, rows, cols + 1, 1, 1, 0);
    // Solve the Augmented Matrix
    SystemOfLinearEq(OriginalMatrix, rows, cols + 1);
    cout << endl;
    cout << "The solutions are: \n";
    // Display the Solution
    OutputMatrix(OriginalMatrix, rows, cols + 1, 3, 1, 1);
    cout << endl;
    // End the program
    EndLine();
    return 0;
}

void inline Title()
{
    cout << "\"Program for Matrix Operations and System of Linear
```

```
        Equations\"\n\n";
}

void inline OrderOfMatrix(unsigned int& rows, unsigned int& cols)
{
    // Validate the rows input
    while (true)
    {
        cout << "Enter the order of your matrix: -\n\n";
        cout << "Enter the number of rows for the matrix: ";
        cin >> rows;

        if (cin.fail() || rows <= 0 || rows > MaxRows)
        {
            cout << "Invalid input! Please enter a valid number of rows between 1 and " << MaxRows << ".\n";
            cin.clear();  // Clear the error flag
            cin.ignore(numeric_limits<streamsize>::max(), '\n');  // Ignore the invalid input
        }
        else
        {
            break;  // Valid input, exit the loop
        }
    }

    // Validate the columns input
    while (true)
    {
        cout << "Enter the number of columns for the matrix: ";
        cin >> cols;

        if (cin.fail() || cols <= 0 || cols > MaxCols)
        {
            cout << "Invalid input! Please enter a valid number of columns between 1 and " << MaxCols << ".\n";
            cin.clear();  // Clear the error flag
            cin.ignore(numeric_limits<streamsize>::max(), '\n');  // Ignore the invalid input
        }
        else
        {
            break;  // Valid input, exit the loop
        }
    }
}

void InputMatrix(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols)
{
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
```

```cpp
        {
            while (true)  // Loop to keep asking until valid input is given
            {
                cout << "Enter element number " << i + 1 << "x" << j + 1 << " : ";
                cin >> OriginalMatrix[i][j];

                if (cin.fail())
                {
                    // Handle invalid input
                    cout << "Invalid input! Please enter a valid number.\n";
                    cin.clear();  // Clears the error flag on cin
                    cin.ignore(numeric_limits<streamsize>::max(), '\n');  // Discards the invalid input
                }
                else
                {
                    break;  // Valid input entered, break out of the loop
                }
            }
        }
    }
}

void OutputMatrix(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols, int Precision, bool Augmented, bool DisplayLastColumnOnly)
{
    for (int i = 0; i < rows; i++)
    {
        cout << "| ";

        // If DisplayLastColumnOnly is true, only print the last column
        if (DisplayLastColumnOnly)
        {
            // Handling the last column
            if (Augmented && cols > 1)
            {
                cout << fixed << setprecision(Precision) << OriginalMatrix[i][cols - 1];
            }
            else
            {
                cout << fixed << setprecision(Precision) << OriginalMatrix[i][cols - 1];
            }
        }
        else
        {
            // Otherwise, display the entire row
            for (int j = 0; j < cols; j++)
```

```cpp
            {
                // Add colon before the last column if Augmented is true (system of linear equations case)
                if (Augmented && j == cols - 1)
                {
                    cout << ": ";
                }

                // Handling the zeros negative signs
                if (OriginalMatrix[i][j] == 0)
                {
                    cout << " " << fixed << setprecision(Precision) << abs(OriginalMatrix[i][j]) << "   "; // Adding extra space between elements
                }
                else
                {
                    // Negative or two-digit numbers
                    if (OriginalMatrix[i][j] < 0 || OriginalMatrix[i][j] > 9)
                    {
                        cout << fixed << setprecision(Precision) << OriginalMatrix[i][j] << "   "; // Normal space for negatives or large numbers
                    }
                    // Single digit positive numbers
                    else
                    {
                        cout << " " << fixed << setprecision(Precision) << OriginalMatrix[i][j] << "   "; // Add extra space for alignment
                    }
                }
            }
        }

        cout << "\b|";  // Backspace to remove the last space
        cout << endl;   // Move to the next row
    }
}


float Determinant(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols)
{
    float det = 0;

    if (rows != cols) //Det does not exist for rectangular matrix
    {               // No need to calculate the det of a rectangular matrix
        return -1;
    }

```

```
    if (rows == 1 && cols == 1) // Base case: 1x1 matrix
    {
        return OriginalMatrix[0][0];
    }

    if (rows == 2 && cols == 2) // Base case: 2x2 matrix
    {
        return OriginalMatrix[0][0] * OriginalMatrix[1][1] - OriginalMatrix[0][1] * OriginalMatrix[1][0];
    }

    // Recursive case: Cofactor expansion along the first row
    for (int i = 0; i < rows; ++i)
    {
        // Create a submatrix by excluding the current row and column
        float subMatrix[MaxRows][MaxCols];  // A submatrix of the original matrix
        int subRow = 0;

        // Exclude the current row (i) and create the submatrix
        for (int j = 1; j < rows; ++j)
        {
            int subCol = 0;
            for (int k = 0; k < cols; ++k)
            {
                if (k == i) continue; // Skip the column of the current element
                subMatrix[subRow][subCol] = OriginalMatrix[j][k];
                ++subCol;
            }
            ++subRow;
        }
        // Add or subtract the cofactor
        float sign = (i % 2 == 0) ? static_cast<float>(1) : static_cast<float>(-1);  // Alternate signs for cofactors
        det += sign * OriginalMatrix[0][i] * Determinant(subMatrix, rows - 1, cols - 1);
    }
    return det;
}

void Swap(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols)
{
    int i, j, k;

    for (i = 0; i < rows; i++)
    {
        if (OriginalMatrix[i][i] == 0)
        {
            for (j = i + 1; j < rows; j++)
            {
                for (k = 0; k < cols; k++)
                {
```

```
                    swap(OriginalMatrix[i][k], OriginalMatrix[j][k]);
                }
                break;
            }
        }
    }
}

void Normalizer(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols)
{
    int i, k;

    for (i = 0; i < rows; i++)
    {
        if (OriginalMatrix[i][i] != 0)
        {

            float pivot = OriginalMatrix[i][i];

            for (k = 0; k < cols; k++)
            {
                OriginalMatrix[i][k] = OriginalMatrix[i][k] / pivot;

            }
        }
    }
}
void Echelon(float OriginalMatrix[MaxRows][MaxCols], int r, int c)
{
    // Declare loop variables i, j, and k
    int i, j, k;

    // Call Swap() function (likely swaps rows or pivots to ensure correct pivoting)
    Swap(OriginalMatrix, r, c);

    // Call Normalizer() function (likely scales rows by their pivot to make pivots equal to 1)
    Normalizer(OriginalMatrix, r, c);

    // Outer loop: Iterate through each row (i) of the matrix
    for (i = 0; i < r; i++)
    {
        // Inner loop: Iterate through the rows below the current row (j)
        for (j = i + 1; j < r; j++)
        {
            // If the element in position [j][i] is non-zero, proceed to eliminate it
            if (OriginalMatrix[j][i] != 0)
            {
                // Loop through each column (k) in row i to modify row
```

```
                j
                for (k = 0; k < c; k++)
                {
                    // Calculate the multiplier: the element at position [j][i] multiplied by the pivot row element [i][k]
                    float Multiplier = OriginalMatrix[j][i] * OriginalMatrix[i][k];

                    // Subtract the appropriate multiple of the pivot row from row j to eliminate element [j][i]
                    OriginalMatrix[j][k] = OriginalMatrix[j][k] - Multiplier;
                }
            }
        }
    }
}

void RREF(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        // Ensure the pivot is non-zero
        if (OriginalMatrix[i][i] == 0)
        {
            for (int j = i + 1; j < rows; j++)
            {
                if (OriginalMatrix[j][i] != 0)
                {
                    // Swap rows
                    for (int k = 0; k < cols; k++)
                    {
                        swap(OriginalMatrix[i][k], OriginalMatrix[j][k]);
                    }
                    break;
                }
            }
        }

        // Normalize the pivot row
        float pivot = OriginalMatrix[i][i];
        if (pivot != 0)
        {
            for (int k = 0; k < cols; k++)
            {
                OriginalMatrix[i][k] /= pivot;
            }
        }

        // Eliminate below and above the pivot
        for (int j = 0; j < rows; j++)
```

```cpp
        {
            if (j != i && OriginalMatrix[j][i] != 0)
            {
                float multiplier = OriginalMatrix[j][i];
                for (int k = 0; k < cols; k++)
                {
                    OriginalMatrix[j][k] -= multiplier * OriginalMatrix[i][k];
                }
            }
        }
    }
}

int Rank(float matrix[MaxRows][MaxCols], int rows, int cols)
{
    int rank = 0;

    for (int i = 0; i < rows; i++)
    {
        bool nonZeroRow = false;
        for (int j = 0; j < cols; j++)
        {
            if (matrix[i][j] != 0)
            {
                nonZeroRow = true;
                break;
            }
        }
        if (nonZeroRow)
        {
            rank++;
        }
    }
    return rank;
}

void Inverse(float OriginalMatrix[MaxRows][MaxCols], float InverseMatrix[MaxRows][MaxCols], int rows, int cols)
{
    // Creating an identity matrix to find and store inverse
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            if (i == j)
            {
                InverseMatrix[i][j] = 1;   // Set diagonal elements to 1
            }
            else
            {
                InverseMatrix[i][j] = 0;   // Set off-diagonal elements
```

```
                  to 0
            }
        }
    }

    for (int i = 0; i < rows; i++)
    {
        // Ensure the pivot is non-zero
        if (OriginalMatrix[i][i] == 0)
        {
            for (int j = i + 1; j < rows; j++)
            {
                if (OriginalMatrix[j][i] != 0)
                {
                    // Swap rows
                    for (int k = 0; k < cols; k++)
                    {
                        swap(OriginalMatrix[i][k], OriginalMatrix[j][k]);
                        swap(InverseMatrix[i][k], InverseMatrix[j][k]);
                    }
                    break;
                }
            }
        }

        // Normalize the pivot row
        float pivot = OriginalMatrix[i][i];
        if (pivot != 0)
        {
            for (int k = 0; k < cols; k++)
            {
                OriginalMatrix[i][k] /= pivot;
                InverseMatrix[i][k] /= pivot;
            }
        }

        // Eliminate below and above the pivot
        for (int j = 0; j < rows; j++) {
            if (j != i && OriginalMatrix[j][i] != 0)
            {
                float multiplier = OriginalMatrix[j][i];
                for (int k = 0; k < cols; k++)
                {
                    OriginalMatrix[j][k] -= multiplier * OriginalMatrix[i][k];
                    InverseMatrix[j][k] -= multiplier * InverseMatrix[i][k];
                }
            }
        }
    }
}
```

```cpp

void SystemOfLinearEq(float OriginalMatrix[MaxRows][MaxCols], int rows, int cols)
{
    RREF(OriginalMatrix, rows, cols);
}

void inline EndLine()
{
    cout << "* -------------------------------------------------- * \n";
}
```