DLITHE PROJECT REPORT

TECHNICAL TRAINING

**PROJECT ID:** CP058.

**PROJECT TITLE:** MATRIX CALCULATOR.

**TEAM MEMBERS:**

1. SANJANA R RAO

(4mt21ai045)

1. PA ABHISHEK

(4mt21ai036)

1. AHMED RAHAN

(4mt21ai005)

1. SHRUTHI POOJARY

(4mt21ai052)

1. GREESHMA BANGERA

(4mt21ai020)

REPORT

**Abstract:**

The Matrix Calculator Program is a software application designed to perform various operations on matrices. This report provides an abstract of the key components and functionality of the program.

**Introduction:**

The program is developed in the C programming language.

It serves as a utility for performing matrix-related calculations, such as addition, subtraction, scalar multiplication, vector multiplication, transposition, determinant calculation, and matrix inversion.

**Background:**

Matrix calculations are fundamental in various fields, including mathematics, engineering, physics, computer science, and data analysis. Matrices are used to represent and manipulate data in a structured manner. Recognizing the importance of matrix operations, the Matrix Calculator Program was developed to provide a versatile tool for performing these operations efficiently.

**Objectives:**

**Matrix Operation Facilitation:** To provide a user-friendly platform for performing common matrix operations such as addition, subtraction, scalar multiplication, vector multiplication, transposition, determinant calculation, and matrix inversion.

**Education and Learning Aid**: To assist students and learners in understanding and practicing matrix operations, especially in the context of mathematics, engineering, and other STEM disciplines.

**Efficiency and Accuracy:** To automate matrix calculations, reducing the likelihood of errors that may occur when performing such operations manually. To enhance efficiency by allowing users to perform calculations on matrices of varying sizes quickly.

**Modularity and Code Reusability:** To implement matrix operations using modular functions, making the codebase organized and promoting reusability. To enable easy maintenance and future enhancements by structuring the program in a modular fashion.

**Error Handling:** To incorporate error-checking mechanisms that ensure valid matrix operations are performed, such as checking matrix compatibility and square matrix requirements for determinant and inversion calculations. To provide informative error messages to guide users when issues arise.

**Versatility:** To cater to a diverse user base, including students, educators, professionals, and researchers, by offering a broad range of matrix operations in one program.

**User Interaction and Clarity:** To create a user-friendly interface that allows users to interact with the program through clear prompts and a menu system. To provide feedback and display results in a structured and understandable format.

**Educational and Practical Utility**: To serve as both an educational tool for learning matrix operations and a practical utility for professionals and researchers needing to perform matrix calculations in their work.

**Potential for Future Enhancements**: To lay the foundation for future improvements and extensions, such as adding support for more advanced matrix operations or implementing a graphical user interface (GUI) for improved user experience.

**Robustness and Error Handling:** To handle various edge cases gracefully and provide clear error messages to guide users in case of input errors or invalid operations.

Efficient Memory Management: To manage memory efficiently, especially when dealing with large matrices, to ensure the program's performance and prevent memory-related issues.

**Program Architecture:**

The program is structured with modular functions for each matrix operation.

It utilizes 2D arrays to represent matrices and float data types for numeric values.

The user specifies the dimensions of matrices.

**System Architecture:**

**1. User Interface (UI):**

The UI is the front-end of the program, responsible for user interaction. It presents a menu-driven interface to the user, allowing them to select the desired matrix operation. The UI collects user input, such as matrix dimensions, matrix elements, scalar values, and vectors.

**2. Menu System:**

The menu system is part of the user interface and guides the user in selecting the operation they want to perform. Users input their choice by entering a numeric option .The menu system routes the user's choice to the corresponding function for execution.

**3. Business Logic Layer:**

This layer contains the core logic of the program, including the matrix operations and error handling. It consists of modular functions for each matrix operation, such as addition, subtraction, scalar multiplication, vector multiplication, transposition, determinant calculation, adjacent calculation, and matrix inversion. Error-checking and validation routines are embedded within these functions to ensure the validity of operations.

**4. Matrix Data Structures:**

The program utilizes 2D arrays to represent matrices. These arrays are dynamically allocated based on user-defined dimensions. Scalar values and vectors are also stored as appropriate data structures.

**5. Error Handling:**

The error handling component detects and manages errors related to user input and matrix operations. It provides informative error messages to guide users in resolving issues.

**6. Mathematical Computations:**

The core mathematical computations, such as matrix transposition, determinant calculation, adjacent calculation, and matrix inversion, are abstracted into separate functions. These functions are part of the business logic layer and perform the necessary calculations based on user input.

**7. Output Generation:**

The program generates output to display the results of matrix operations and error messages. Output is presented in a structured format to enhance user understanding.

**8. Future Enhancements Layer:**

While not part of the current program, this layer is mentioned in the architecture to indicate the potential for future improvements and extensions. Future enhancements might include additional matrix operations, GUI integration, or support for more advanced mathematical concepts.

**9. Memory Management:**

The program employs memory management techniques to allocate and deallocate memory efficiently, especially when working with matrices of varying sizes.

**10. Input Validation:**

- Input validation is performed to ensure that user-provided data is of the correct format and within acceptable ranges.

**File handling in the program:**

* **Saving Matrices to a File**
* **Loading Matrices from a File**
* Add menu options for saving and loading matrices, making them accessible to the user along with other matrix operations.
* Saving matrices to external text files.
* Loading matrices from text files, enhancing usability for large matrices and data persistence

**Program Features:**

**. User-Friendly Interface:**

A menu-driven user interface that guides users in selecting matrix operations. Clear and informative prompts for entering matrix elements, scalar values, and vectors. Structured output format to present results in an understandable manner.

**Error Handling:**

Robust error handling mechanisms to detect and manage potential issues. Checks for matrix compatibility before addition and subtraction. Ensures square matrices for determinant and inversion calculations. Handles invalid inputs and provides user-friendly error messages.

**Modular Architecture:**

Modular functions for each matrix operation, promoting code organization and reusability. Abstraction of mathematical computations into separate functions for maintainability.

**Dynamic Memory Management:**

Efficient allocation and deallocation of memory for matrices based on user-defined dimensions. Prevents memory leaks and optimizes program performance.

**Educational and Learning Aid:**

A valuable tool for students, educators, and professionals working with matrices. Supports learning and practice of matrix operations, enhancing understanding of fundamental concepts.

**Versatile Application:**

Suitable for a diverse user base, including: Students learning matrix operations in mathematics or engineering courses. Educators demonstrating matrix concepts. Professionals and researchers performing practical matrix calculations in various fields.

**Potential for Future Enhancements:**

A foundation for future improvements, including: Support for additional matrix operations or mathematical concepts. Integration of a graphical user interface (GUI) for enhanced user experience. Expanded error checking and validation for edge cases.

**Design and Implementation:**

**Design Phase**:

**1. Requirements Analysis**:

- Define the program's objectives, including the supported matrix operations, user interface, and potential enhancements.

**2. User Interface Design:**

- Plan the user interface, including menu options and prompts.

- Decide on the input format (e.g., user input or file input) and the output format (e.g., displaying matrices and results).

**3. Architecture Design:**

- Determine the program's overall structure. For example, you can have functions for each matrix operation.

- Decide on data structures for representing matrices and other data (e.g., scalar values, vectors).

- Plan error handling mechanisms, including how to handle invalid inputs and provide informative error messages.

**4. Algorithm Design:**

- Develop algorithms for each matrix operation, such as matrix addition, multiplication, and determinant calculation.

- Consider using modular functions to encapsulate mathematical computations.

**5. Memory Management Design:**

- Plan how memory will be allocated and deallocated for matrices.

- Ensure efficient memory usage and prevent memory leaks.

**Implementation Phase:**

**1. Coding:**

- Write the code for the Matrix Calculator Program based on the design. Start with the main function that presents the menu to the user and call other functions as needed.

- Implement functions for matrix operations (addition, subtraction, etc.) following the algorithms designed earlier.

- Incorporate user input and output handling, including prompts and error messages.

**2. Testing:**

- Test the program thoroughly by running various test cases for each matrix operation.

- Verify that the program produces correct results and handles invalid inputs gracefully.

- Perform boundary testing, testing edge cases, and verifying that the program meets all requirements.

**3. Error Handling:**

- Ensure that error handling mechanisms are in place and effectively communicate issues to the user.

- Validate user inputs and check for potential errors during file operations (if file handling is included).

**4. Optimization:**

- Optimize the program for performance, especially when dealing with large matrices.

- Ensure efficient memory management to prevent memory leaks.

**5. Documentation:**

- Create user documentation, including instructions on how to use the program.

- Include comments in the code to explain its functionality and any complex algorithms.

**6. Optional File Handling:**

- If including file handling features:

- Implement functions to save matrices to files and load matrices from files.

- Test these functions thoroughly to ensure they work as expected.

**7. Optional Future Enhancement**:

- If planning for future enhancements:

- Leave hooks in the code for adding new matrix operations or extending functionality.

- Consider adding a graphical user interface (GUI) for improved user experience.

**8. Final Testing:**

- Conduct final testing to ensure that all features work seamlessly together.

**9. User Training:**

- If applicable, provide documentation or training for users to understand and effectively use the program.

**10. Maintenance and Updates:**

- Continue to maintain and update the program based on user feedback and potential future enhancements.

**Testing:**

**Unit Testing:**

- **Perform unit tests for individual functions:** Test each matrix operation function (addition, subtraction, etc.) with a variety of test cases. Ensure that each function produces the expected results for valid inputs. Test functions with boundary cases and edge cases to validate their behaviour.

Integration Testing

- **Test the interaction between different components of the program:** Check that the menu system correctly routes user choices to the corresponding functions. Verify that data is passed correctly between functions.

**User Interface Testing:**

Test the user interface for usability and clarity :Verify that the menu system is intuitive and guides users effectively. Check that prompts and error messages are clear and informative. Ensure that the output is well-formatted and easy to understand.

**Error Handling Testing**

- Validate error handling mechanisms: Test the program's response to invalid inputs, such as non-numeric values or incompatible matrix dimensions. Confirm that error messages are displayed when necessary, guiding the user on how to correct the issue.

**Functional Testing:**

- Conduct functional tests for each matrix operation: Test matrix addition and subtraction with matrices of various sizes and values. Verify that scalar multiplication works as expected. Test vector multiplication with different vectors and matrices. Ensure matrix transposition produces the correct results. Validate determinant calculation for square matrices. Test matrix inversion for invertible square matrices.

**Performance Testing:**

- Assess the program's performance, especially when working with large matrices: Measure execution time for various operations with matrices of different sizes. Identify potential bottlenecks or areas for optimization.

**Edge Case Testing:**

- Test extreme or edge cases to ensure the program behaves correctly: Test with matrices of very large sizes to evaluate memory management and performance. Check the program's behaviour with matrices containing very small or very large values .Test determinant and inversion calculations with matrices that have special properties, such as having a determinant of 0.

**Usability Testing:**

- If possible, involve real users (such as students, educators, or professionals) to test the program's usability and gather feedback.

**Regression Testing:**

- After making changes or enhancements to the program, perform regression testing to ensure that existing functionality remains unaffected.

Documentation Testing

- Review user documentation to ensure that it accurately describes how to use the program and its features.

**Final Testing:**

- Before deployment, conduct a final round of testing to ensure that all issues have been resolved and that the program is ready for release.

**Appendices:**

**Appendix1. Source Code:**

- Include the complete source code of the Matrix Calculator Program as an appendix. This allows readers to examine the actual code and understand how the program is implemented.

**Appendix2. User Documentation:**

- Provide a user manual or documentation as an appendix. This can include instructions on how to use the program, explanations of menu options, and examples of input and output.

**Appendix3. Test Cases and Results:**

- Include a section with detailed test cases used during the testing phase. Describe the inputs, expected outcomes, and actual results for each test case.

**Appendix4. Flowcharts and Diagrams:**

- If you've created flowcharts, diagrams, or system architecture illustrations during the design phase, include them in the appendices to visually represent the program's structure.

**Appendix5. Mathematical Algorithms:**

- If your program includes complex mathematical algorithms, provide detailed explanations and mathematical formulas in an appendix. This can help readers understand the underlying calculations.

**Appendix6. Error Messages:**

- Include a list of error messages and explanations for users. This can be helpful for users encountering issues with the program.

**Appendix7. Sample Input and Output:**

- Provide examples of sample input data and the corresponding output from the program. This helps users understand how to use the program effectively.

**Appendix8. File Formats:**

- If your program supports file handling, describe the file formats used and provide examples of file structures in an appendix.

**Appendix9. References and Citations:**

- Include a list of references, citations, and sources used during the project's development. This is particularly important if you referred to textbooks, academic papers, or online resources.

**Appendix10. Glossary of Terms:**

- Define technical terms or acronyms used in the project that may not be familiar to all readers.

**Appendix11. Acknowledgments:**

- If you want to acknowledge individuals or organizations that provided support, guidance, or resources during the project, you can include an acknowledgments section in the appendices.

**Appendix12. Survey Questionnaires:**

- If your project involves survey data, include copies of the questionnaires or surveys used to collect data from participants.

**Appendix13. Ethics and Permissions:**

- If your project involved human subjects or data that required ethical considerations or permissions, include relevant documents and approvals.

**Code Snippet:**

#include <stdio.h>

#include <stdlib.h>

// Function to read a matrix from the user

void readMatrix(int rows, int cols, float matrix[][cols]) {

    printf("Enter the elements of the matrix (%d x %d):\n", rows, cols);

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < cols; j++) {

            scanf("%f", &matrix[i][j]);

        }

    }

}

// Function to print a matrix

void printMatrix(int rows, int cols, float matrix[][cols]) {

    printf("Matrix:\n");

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < cols; j++) {

            printf("%.2f\t", matrix[i][j]);

        }

        printf("\n");

    }

}

// Function to add two matrices

void addMatrices(int rows, int cols, float matrix1[][cols], float matrix2[][cols], float result[][cols]) {

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < cols; j++) {

            result[i][j] = matrix1[i][j] + matrix2[i][j];

        }

    }

}

// Function to subtract two matrices

void subtractMatrices(int rows, int cols, float matrix1[][cols], float matrix2[][cols], float result[][cols]) {

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < cols; j++) {

            result[i][j] = matrix1[i][j] - matrix2[i][j];

        }

    }

}

// Function for scalar multiplication of a matrix

void scalarMultiply(int rows, int cols, float matrix[][cols], float scalar, float result[][cols]) {

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < cols; j++) {

            result[i][j] = matrix[i][j] \* scalar;

        }

    }

}

// Function for vector multiplication of a matrix

void vectorMultiply(int rows, int cols, float matrix[][cols], float vector[], float result[]) {

    for (int i = 0; i < rows; i++) {

        result[i] = 0;

        for (int j = 0; j < cols; j++) {

            result[i] += matrix[i][j] \* vector[j];

        }

    }

}

// Function for matrix transposition

void transposeMatrix(int rows, int cols, float matrix[][cols], float result[][rows]) {

    for (int i = 0; i < cols; i++) {

        for (int j = 0; j < rows; j++) {

            result[i][j] = matrix[j][i];

        }

    }

}

// Function to calculate the determinant of a square matrix

float determinant(int n, float matrix[][n]) {

    if (n == 1) {

        return matrix[0][0];

    } else if (n == 2) {

        return (matrix[0][0] \* matrix[1][1]) - (matrix[0][1] \* matrix[1][0]);

    } else {

        float det = 0;

        for (int i = 0; i < n; i++) {

            float temp[n - 1][n - 1];

            for (int j = 1; j < n; j++) {

                for (int k = 0; k < i; k++) {

                    temp[j - 1][k] = matrix[j][k];

                }

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

                    temp[j - 1][k - 1] = matrix[j][k];

                }

            }

            det += (i % 2 == 0 ? 1 : -1) \* matrix[0][i] \* determinant(n - 1, temp);

        }

        return det;

    }

}

// Function to find the cofactor of a matrix

void getCofactor(int n, float matrix[][n], float temp[][n], int p, int q) {

    int i = 0, j = 0;

    for (int row = 0; row < n; row++) {

        for (int col = 0; col < n; col++) {

            if (row != p && col != q) {

                temp[i][j++] = matrix[row][col];

                if (j == n - 1) {

                    j = 0;

                    i++;

                }

            }

        }

    }

}

// Function to calculate the adjugate of a matrix

void adjugate(int n, float matrix[][n], float adj[][n]) {

    if (n == 1) {

        adj[0][0] = 1;

        return;

    }

    int sign = 1;

    float temp[n][n];

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            getCofactor(n, matrix, temp, i, j);

            sign = ((i + j) % 2 == 0) ? 1 : -1;

            adj[j][i] = (sign) \* (determinant(n - 1, temp));

        }

    }

}

// Function to calculate the inverse of a matrix

void inverse(int n, float matrix[][n], float inverse[][n]) {

    float det = determinant(n, matrix);

    if (det == 0) {

        printf("The matrix is not invertible (singular).\n");

        return;

    }

    float adj[n][n];

    adjugate(n, matrix, adj);

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            inverse[i][j] = adj[i][j] / det;

        }

    }

}

int main() {

    int choice;

    int rows, cols;

    printf("Matrix Calculator Menu:\n");

    printf("1. Addition\n");

    printf("2. Subtraction\n");

    printf("3. Scalar Multiplication\n");

    printf("4. Vector Multiplication\n");

    printf("5. Matrix Transposition\n");

    printf("6. Determinant Calculation\n");

    printf("7. Matrix Inversion\n");

    printf("Enter your choice: ");

    scanf("%d", &choice);

    if(choice<8){

    printf("Enter the number of rows: ");

    scanf("%d", &rows);

    printf("Enter the number of columns: ");

    scanf("%d", &cols);

    }

    float matrix1[rows][cols],matrix2[rows][cols],result[rows][cols];

    float scalar, inverseMatrix[rows][cols];

    float vector[cols], vectorResult[rows]; // Declare vector and vectorResult arrays

    float transposed[cols][rows];

    switch (choice) {

        case 1:

            readMatrix(rows, cols, matrix1);

            readMatrix(rows, cols, matrix2);

            addMatrices(rows, cols, matrix1, matrix2, result);

            printMatrix(rows, cols, result);

            break;

        case 2:

            readMatrix(rows, cols, matrix1);

            readMatrix(rows, cols, matrix2);

            subtractMatrices(rows, cols, matrix1, matrix2, result);

            printMatrix(rows, cols, result);

            break;

        case 3:

            printf("Enter the scalar value: ");

            scanf("%f", &scalar);

            readMatrix(rows, cols, matrix1);

            scalarMultiply(rows, cols, matrix1, scalar, result);

            printMatrix(rows, cols, result);

            break;

        case 4:

            printf("Enter the vector of size %d:\n", cols);

            for (int i = 0; i < cols; i++) {

                scanf("%f", &vector[i]);

            }

            readMatrix(rows, cols, matrix1);

            vectorMultiply(rows, cols, matrix1, vector, vectorResult);

            printf("Resulting vector:\n");

            for (int i = 0; i < rows; i++) {

                printf("%.2f\t", vectorResult[i]);

            }

            printf("\n");

            transposeMatrix(rows, cols, matrix1, transposed);

            printf("Transposed Matrix:\n");

            printMatrix(cols, rows, transposed);

        break;

            break;

        case 5:

            readMatrix(rows, cols, matrix1);

            transposeMatrix(rows, cols, matrix1, transposed);

            printf("Transposed Matrix:\n");

            printMatrix(cols, rows, transposed);

            break;

        case 6:

            if (rows != cols) {

                printf("Determinant can only be calculated for square matrices.\n");

            } else {

                readMatrix(rows, cols, matrix1);

                float det = determinant(rows, matrix1);

                printf("Determinant: %.2f\n", det);

            }

            break;

        case 7:

            if (rows != cols) {

                printf("Matrix inversion is only possible for square matrices.\n");

            } else {

                readMatrix(rows, cols, matrix1);

                inverse(rows, matrix1, inverseMatrix);

                printf("Inverse Matrix:\n");

                printMatrix(rows, cols, inverseMatrix);

            }

            break;

        default:

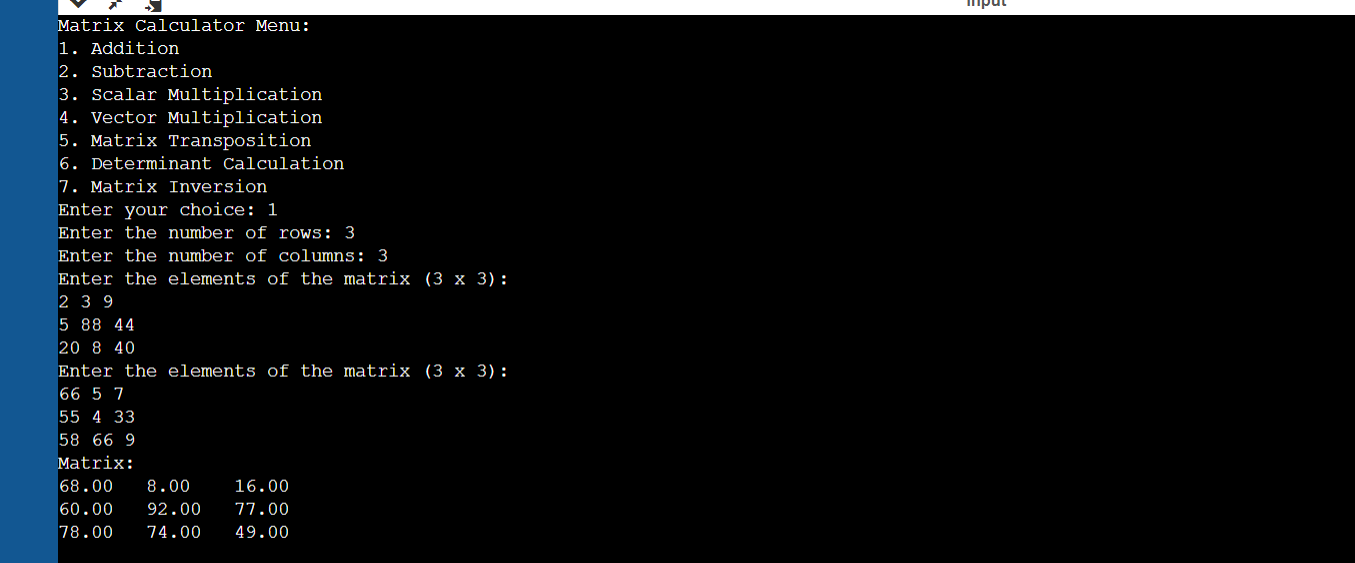
            printf("Invalid choice\n");

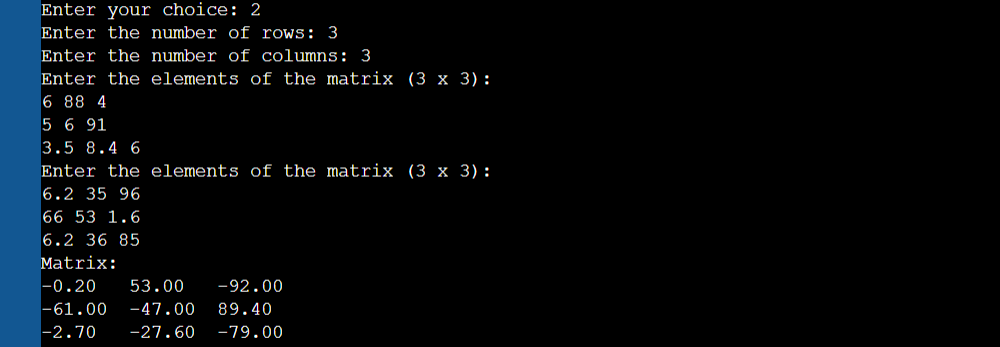
    }

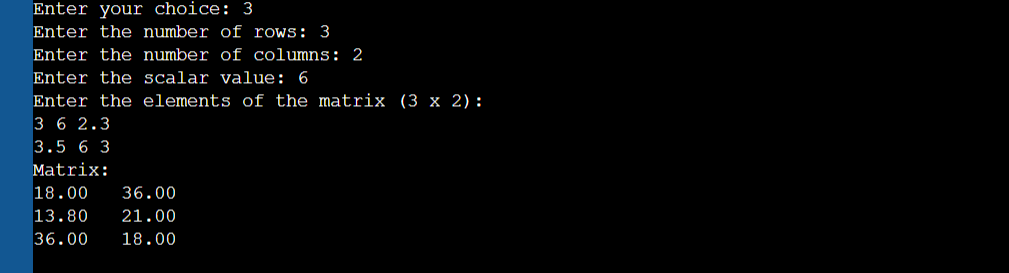
    return 0;

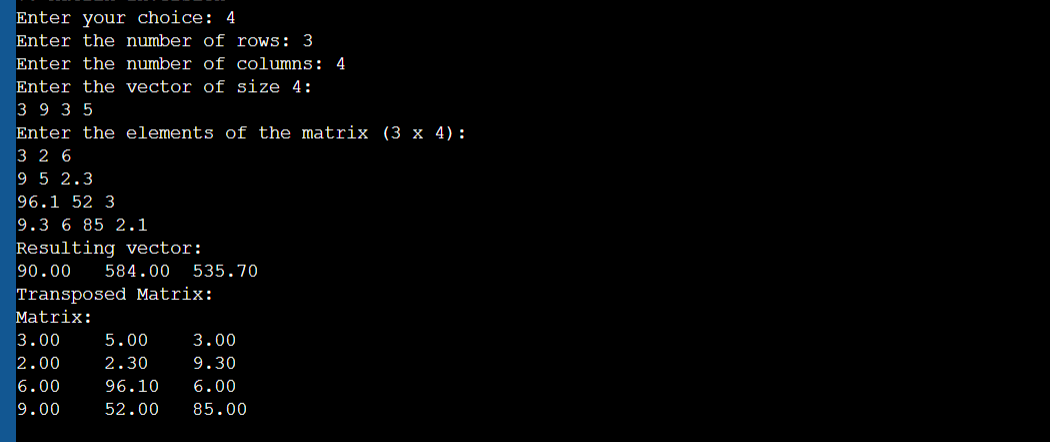
}

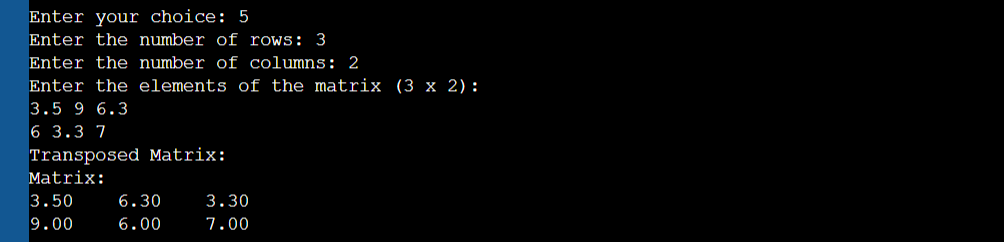
**Output:**

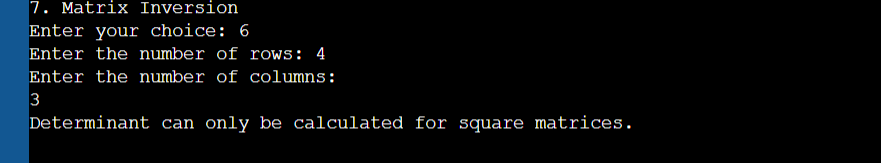


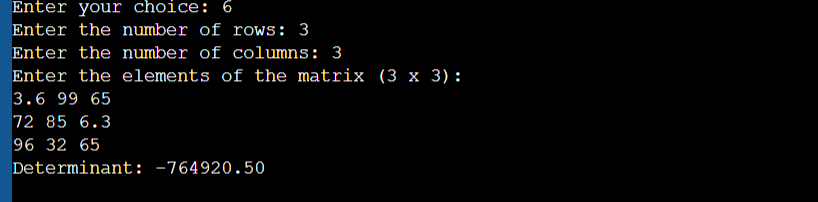


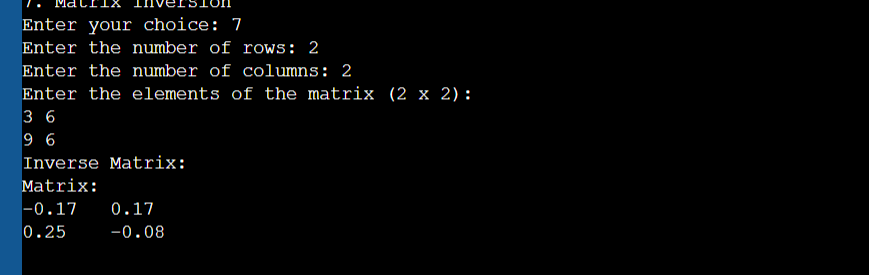












**Conclusion:**

* Matrix operations play a crucial role in various domains, and the Matrix Calculator Program provides a valuable tool to simplify these calculations.
* The program's development is driven by the need for efficient and user-friendly matrix operations.
* It serves as a resource for students, professionals, and educators working with matrices in their academic and practical endeavors.

This background information provides context for the Matrix Calculator Program, explaining its significance, target audience, features, design principles, and potential for future improvements. The program's development is driven by the aim to simplify matrix calculations and make them more accessible to a wide range of users.