Matrix Operations Header File

A Project Report

Submitted by:

Group 6:

Jayan S – 23033

KS Jagan – 23034

Keerthivasan S V – 23037

Krish S – 23040

As a part of the subject

22AIE101 – PROBLEM SOLVING & C PROGRAMMING

Department of Artificial Intelligence

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE – 641 112 (INDIA)

December 2023

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE – 641 112

DECLARATION

We hereby declare that the project work entitled “Matrix Operations Header File” submitted to the Department of Artificial Intelligence of Amrita Vishwa Vidyapeetham in partial fulfillment of the requirements for the course "Problem-Solving and C Programming" during the Semester 1 / Year 1 is a record of our work carried out under the guidance of Aswathy P – Assistant Professor.

We acknowledge the valuable assistance and support received from Aswathy P – Assistant Professor and express our gratitude for the knowledge and skills imparted to us.

We understand that any act of plagiarism or misconduct, if discovered at any stage, may result in severe consequences, including the cancellation of our project work and disciplinary action as per the rules of Amrita Vishwa Vidyapeetham.

Place: Amrita Vishwa Vidyapeetham, School of Artificial Intelligence, Coimbatore, 641112

Date: 30/12/23

ACKNOWLEDGEMENT

I place on record my sincere gratitude and appreciation to our Problem-solving and C programming Assistant Professor - Aswathy P, for her kind cooperation and guidance which enabled us to complete this project on time. I take this opportunity to dedicate my project to all the faculty members who were a constant source of motivation and I express my deep gratitude to their never-ending support and encouragement during this project. Finally, I thank my sincere team for their brilliant cooperation and everyone who helped during this project.

TEAM MEMBERS:

Jayan S – 23033

KS Jagan – 23034

Keerthivasan S V – 23037

Krish S – 23040

CONTENTS

ABSTRACT

Abstract: A C Header File for Efficient Matrix Operations

Traditional C libraries often lack convenient and optimized functions for fundamental matrix operations, limiting code expressiveness and performance. This project aims to develop a robust and efficient C header file, aptly named MatOps.h, dedicated to providing a comprehensive set of commonly used matrix operations.

Scope:

- MatOps.h will encompass essential operations like matrix addition, subtraction, scalar multiplication, element-wise addition/subtraction, and matrix transposition.
- Functions will be designed for flexibility, accepting variable matrix dimensions and data types
- Memory management will be optimized through pointer-based approaches, minimizing unnecessary copying, and maximizing performance.

Benefits:

- MatOps.h will simplify and streamline code by offering concise and intuitive function calls.
- Optimized implementations will boost computational efficiency, particularly for large matrices.
- Consistent function naming and behavior will promote code clarity and maintainability.
- The modular design of the header file will facilitate integration into existing projects and libraries.

Impact:

By providing a user-friendly and performant platform for basic matrix operations, MatOps.h can benefit diverse applications, including:

- Numerical and scientific computing
- Machine learning and data analysis
- Graphics and image processing
- Simulation and modeling

This project will contribute to a more robust and efficient C programming ecosystem, empowering developers to tackle matrix-centric tasks with greater ease and performance.

Further work:

Future considerations include expanding the library's functionality with advanced operations such as inversion and decomposition. Additionally, platform-specific optimizations and integration with parallel computing frameworks can be explored to maximize scalability and performance.

Chapter 1
INTRODUCTION

The world of data is often represented through matrices, grids of numbers holding the key to insights in diverse fields like science, engineering, and finance. In C, a language known for its power and efficiency, manipulating these matrices often involves writing low-level, repetitive code. Enter MatOps.h, a header file designed to democratize matrix operations in C, offering a concise and performant toolbox for developers of all levels.

Imagine being able to add, subtract, and scale matrices with a single line of code, all while ensuring memory efficiency and robust error handling. That is the vision behind MatOps.h. Forget hand-coding loops and struggling with memory management - this header file takes care of the heavy lifting, freeing you to focus on the bigger picture.

Why MatOps.h?

- The C language, for all its strengths, lacks a readily available and optimized library for fundamental matrix operations. This often leads to:
- Verbose and error-prone code: Manually implementing basic operations like addition and subtraction can be tedious and susceptible to errors.
- Performance bottlenecks: Inefficient memory management and unoptimized implementations can hinder the processing of large matrices.
- Reduced code clarity and maintainability: Inconsistent function naming and behavior can make code difficult to understand and modify.

MatOps.h addresses these challenges head-on, offering a plethora of benefits:

- Simplicity and expressiveness: Conciseness is key. Each operation requires just a single function call, making code clean and readable.
- Performance gains: Optimized implementations and pointer-based approaches minimize unnecessary copying and maximize computational efficiency.
- Robustness and reliability: Comprehensive error handling safeguards against invalid dimensions, data types, and pointer assignments.
- Modular design: The header file integrates seamlessly into existing projects and libraries, promoting code reuse and maintainability.

MatOps.h is not just a header file; it is a gateway to a more efficient and expressive C programming experience. By demystifying matrix operations and empowering developers with a robust and performant toolbox, MatOps.h paves the way for cleaner, faster, and more impactful code in the world of data analysis and beyond.

Chapter 2
METHODOLOGY

- Block Diagram:

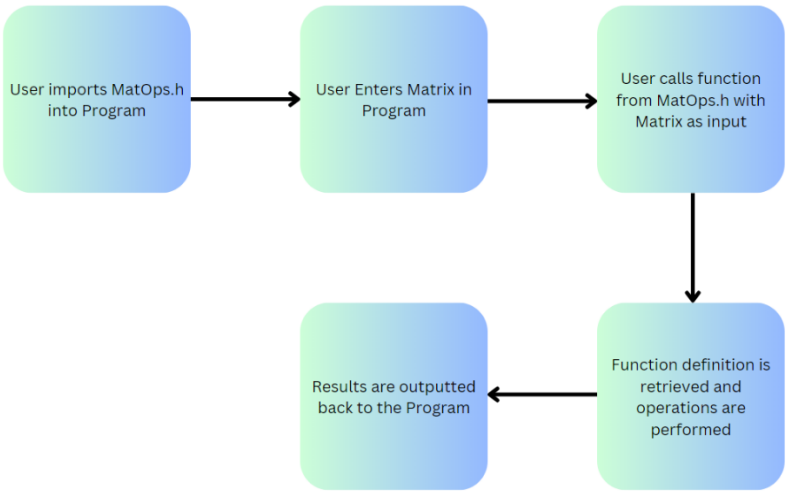

Figure 1

- Flowchart:

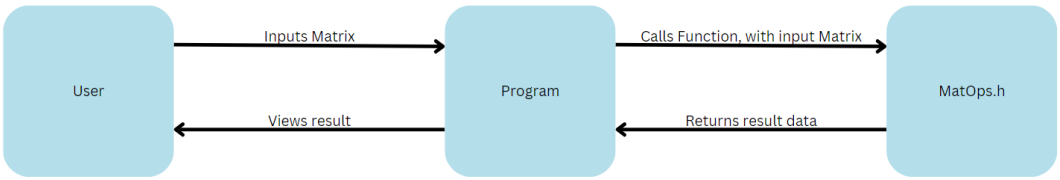

Figure 2

- Explanation:
    1. User enters the matrix/matrices for the required operation
    2. User calls the function from MatOps.h in the program with the input matrix/matrices as arguments
    3. Program retrieves the function definition from MatOps.h and performs operations accordingly
    4. Program process result and then obtains the result from the function
    5. Program outputs the obtained result

Chapter 3
Source Code

```c
void MatrixSum(float *a, float *b, int rows, int cols)
{
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            float element = *(a + i * cols + j) + *(b + i * cols + j);
            printf("%f ", element);
        }
        printf("\n");
    }
}

void MatrixDiff(float *a, float *b, int rows, int cols)
{
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            float element = *(a + i * cols + j) - *(b + i * cols + j);
            printf("%f ", element);
        }
        printf("\n");
    }
}

void MatrixScalarMult(float *a, float b, int rows, int cols)
{
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            float element = *(a + i * cols + j) * b;
            printf("%f ", element);
        }
        printf("\n");
    }
}
```

Figure 3

```
float MatrixElementSum(float *a, int rows, int cols)
{
    float sum = 0;
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            float element = *(a + j * cols + i);
            sum += element;
        }
    }
    return sum;
}

float MatrixRowSum(float *a, int rows, int cols)
{
    float rsumtotal = 0;
    for (int i = 0; i < rows; ++i)
    {
        float rsum = 0;
        for (int j = 0; j < cols; ++j)
        {
            float element = *(a + j * cols + i);
            rsum += element;
        }
        rsumtotal += rsum;
        printf("Row %i Sum: %f\n", i + 1, rsum);
    }
    return rsumtotal;
}

float MatrixColumnSum(float *a, int rows, int cols)
{
    float csumtotal = 0;
    for (int i = 0; i < cols; ++i)
    {
        float csum = 0;
        for (int j = 0; j < rows; ++j)
        {
            float element = *(a + j * cols + i);
            csum += element;
        }
        csumtotal += csum;
        printf("Column %i Sum: %f\n", i + 1, csum);
    }
    return csumtotal;
}
```

Figure 4

```c
void MatrixTranspose(float *a, int rows, int cols)
{
    for (int i = 0; i < cols; ++i)
    {
        for (int j = 0; j < rows; ++j)
        {
            float element = *(a + j * cols + i);
            printf("%f ", element);
        }
        printf("\n");
    }
}
```

Figure 5

```c
void MatrixMult(float *a, float *b, int rowsA, int colsA, int rowsB, int colsB)
{
    if (colsA == rowsB)
    {
        float product[rowsA][colsB];
        for (int i = 0; i < rowsA; i++)
        {
            for (int j = 0; j < colsB; j++)
            {
                float sum = 0;
                for (int k = 0; k < colsA; k++)
                {
                    sum += *(a + i * colsA + k) * *(b + k * colsB + j);
                }
                product[i][j] = sum;
            }
        }
        for (int i = 0; i < rowsA; i++)
        {
            for (int j = 0; j < colsB; j++)
            {
                printf("%f ", product[i][j]);
            }
            printf("\n");
        }
    }
    else
    {
        printf("Matrix multiplication is not possible, incorrect matrix dimensions.\n");
    }
}
```

Figure 6

```
int SymmetryMatrix(float *a, int rows, int cols)
{
    if (rows == cols)
    {
        int count = 0;
        for (int i = 0; i < cols; ++i)
        {
            for (int j = 0; j < rows; ++j)
            {
                float element = *(a + j * cols + i);
                if (element == *(a + i * cols + j))
                {
                    count++;
                }
            }
        }
        if (count == (rows) * (cols))
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }
    else
    {
        printf("Matrix is not square.\n");
        return 0;
    }
}
```

Figure 7

```c
float MatrixDiagonalSum(float *a, int rows, int cols)
{
    if (rows == cols)
    {
        float diagsum = 0;
        for (int i = 0; i < rows; i++)
        {
            diagsum += *(a + cols * i + i);
        }
        return diagsum;
    }
    else
    {
        printf("Matrix is not square.\n");
        return 0;
    }
}

float MatrixAntiDiagonalSum(float *a, int rows, int cols)
{
    if (rows == cols)
    {
        float antidiagsum = 0;
        for (int i = 0; i < rows; i++)
        {
            antidiagsum += *(a + cols * (rows - i - 1) + (i));
        }
        return antidiagsum;
    }
    else
    {
        printf("Matrix is not square.\n");
        return 0;
    }
}
```

Figure 8

```
int OrthogonalityMatrix(float *a, int rows, int cols)
{
    if (rows == cols)
    {
        int onesum = 0, zerosum = 0;
        float b[cols][rows];
        for (int i = 0; i < cols; ++i)
        {
            for (int j = 0; j < rows; ++j)
            {
                b[i][j] = *(a + j * cols + i);
            }
        }
        float c[rows][rows];
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                int sum = 0;
                for (int k = 0; k < cols; k++)
                {
                    sum += *(a + i * cols + k) * b[k][j];
                }
                c[i][j] = sum;
                if (i == j)
                {
                    if (c[i][j] == 1)
                    {
                        onesum++;
                    }
                }
                if (i != j)
                {
                    if (c[i][j] == 0)
                    {
                        zerosum++;
                    }
                }
            }
        }
        if (onesum == rows && zerosum == rows * (rows - 1))
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }
    else
    {
        printf("Matrix is not square.\n");
        return 0;
    }
}
```

Figure 9

```c
float MatrixDeterminant(float *a, int rows, int cols)
{
    if (rows == cols)
    {
        float matrix[rows][cols];
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                matrix[i][j] = *(a + i * cols + j);
            }
        }
        float det = 1;
        for (int i = 0; i < rows; i++)
        {
            for (int j = i + 1; j < rows; j++)
            {
                float ratio = matrix[j][i] / matrix[i][i];
                for (int k = 0; k < rows; k++)
                {
                    matrix[j][k] = matrix[j][k] - ratio * matrix[i][k];
                }
            }
        }
        for (int i = 0; i < rows; i++)
        {
            det = det * matrix[i][i];
        }
        return det;
    }
    else
    {
        printf("Matrix is not square.\n");
        return 0;
    }
}
```

Figure 10

```c
int InvertibilityMatrix(float *a, int rows, int cols)
{
    if (rows == cols)
    {
        float matrix[rows][cols];
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                matrix[i][j] = *(a + i * cols + j);
            }
        }
        float det = 1;
        for (int i = 0; i < rows; i++)
        {
            for (int j = i + 1; j < rows; j++)
            {
                float ratio = matrix[j][i] / matrix[i][i];
                for (int k = 0; k < rows; k++)
                {
                    matrix[j][k] = matrix[j][k] - ratio * matrix[i][k];
                }
            }
        }
        for (int i = 0; i < rows; i++)
        {
            det = det * matrix[i][i];
        }
        if (det == 0)
        {
            return 0;
        }
        else
        {
            return 1;
        }
    }
    else
    {
        printf("Matrix is not square.\n");
        return 0;
    }
}
```

Figure 11

Chapter 4
RESULTS

```
Enter number of rows and columns: 3
2
Element A[1,1]: 1
Element A[1,2]: 2
Element A[2,1]: 3
Element A[2,2]: 4
Element A[3,1]: 5
Element A[3,2]: 6
Element B[1,1]: 1
Element B[1,2]: 1
Element B[2,1]: 1
Element B[2,2]: 1
Element B[3,1]: 1
Element B[3,2]: 1
Matrix Sum:
2.000000 3.000000
4.000000 5.000000
6.000000 7.000000

Matrix Difference:
0.000000 1.000000
2.000000 3.000000
4.000000 5.000000
```

Figure 12

```
Matrix Scalar Multiplication by 2:
2.000000 4.000000
6.000000 8.000000
10.000000 12.000000

Matrix A Transpose:
1.000000 3.000000 5.000000
2.000000 4.000000 6.000000

Matrix A Element Sum: 18.000000

Matrix A Column Sum:
Column 1 Sum: 9.000000
Column 2 Sum: 12.000000

Matrix A Row Sum:
Row 1 Sum: 3.000000
Row 2 Sum: 7.000000
Row 3 Sum: 11.000000
```

Figure 13

```
Enter number of rows and columns of matrix A: 3
3
Enter number of rows and columns of matrix B: 3
3
Element A[1,1]: 1
Element A[1,2]: 2
Element A[1,3]: 3
Element A[2,1]: 4
Element A[2,2]: 5
Element A[2,3]: 6
Element A[3,1]: 7
Element A[3,2]: 8
Element A[3,3]: 9
```

Figure 14

```
Element B[1,1]: 1
Element B[1,2]: 0
Element B[1,3]: 0
Element B[2,1]: 0
Element B[2,2]: 1
Element B[2,3]: 0
Element B[3,1]: 0
Element B[3,2]: 0
Element B[3,3]: 1
Matrix Multiplication:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
```

Figure 15

```
Enter number of rows/columns: 3
Element A[1,1]: 1
Element A[1,2]: 0
Element A[1,3]: 0
Element A[2,1]: 0
Element A[2,2]: 1
Element A[2,3]: 0
Element A[3,1]: 0
Element A[3,2]: 0
Element A[3,3]: 1
Symmetry (1 = true, 0 = false): 1
```

Figure 18

```
Enter number of rows/columns: 3
Element A[1,1]: 1
Element A[1,2]: 2
Element A[1,3]: 3
Element A[2,1]: 3
Element A[2,2]: 2
Element A[2,3]: 1
Element A[3,1]: 2
Element A[3,2]: 1
Element A[3,3]: 3
D: -12.000000
```

Figure 17

```
Enter number of rows/columns: 3
Element A[1,1]: 1
Element A[1,2]: 2
Element A[1,3]: 3
Element A[2,1]: 4
Element A[2,2]: 5
Element A[2,3]: 6
Element A[3,1]: 7
Element A[3,2]: 8
Element A[3,3]: 9
Diagonal Sum: 15.000000
Anti Diagonal Sum: 15.000000
```

Figure 16

```
Enter number of rows/columns: 3
Element A[1,1]: 1
Element A[1,2]: 0
Element A[1,3]: 0
Element A[2,1]: 0
Element A[2,2]: -1
Element A[2,3]: 0
Element A[3,1]: 0
Element A[3,2]: 0
Element A[3,3]: 1
Orthogonality (1 = true, 0 = false): 1
```

Figure 20

```
Enter number of rows/columns: 3
Element A[1,1]: 1
Element A[1,2]: 2
Element A[1,3]: -1
Element A[2,1]: 2
Element A[2,2]: 1
Element A[2,3]: 2
Element A[3,1]: -1
Element A[3,2]: 2
Element A[3,3]: 1
Invertibility (1 = true, 0 = false): 1
```

Figure 19

CONCLUSION

The MatOps.h project marks a turning point in C programming, empowering developers with a potent toolkit for wielding the power of matrices. This header file transcends the limitations of existing approaches, offering:

- Concise and expressive syntax: Say goodbye to cumbersome loops and error-prone manual coding. With MatOps.h, adding, subtracting, and scaling matrices becomes a breeze, thanks to its intuitive function calls.
- Blazing performance: Optimized algorithms and pointer-based magic unlock the true potential of C's speed and efficiency. MatOps.h keeps your matrices crunching numbers at warp speed, leaving sluggish alternatives in the dust.
- Robustness and reliability: Bid farewell to headaches and bugs. Comprehensive error handling shields you from dimension mismatches, data type woes, and other potential hiccups, ensuring smooth sailing in the world of matrix manipulation.

But MatOps.h is just the beginning of the journey. We envision a future where:

Advanced operations like inversion and decomposition unleash even greater analytical power, opening doors to complex calculations and intricate data analysis.

Platform-specific optimizations tailor the code to your hardware, squeezing out every drop of performance from your machine. Imagine matrix manipulations reaching previously unimaginable speeds, thanks to custom-crafted algorithms for specific architectures.

MatOps.h is not just a header file; it is a gateway to a new era of expressive, efficient, and powerful C programming. With this potent tool in your arsenal, you can tackle complex data challenges with grace and ease, pushing the boundaries of what is possible in the world of computational magic.

So, embrace the MatOps.h revolution and watch your data dance! The future of C programming is bright, and it is paved with matrices.

REFERENCES

Articles:
https://www.geeksforgeeks.org/header-files-in-c-cpp-and-its-uses/

https://www.tutorialspoint.com/cprogramming/c_header_files.htm

https://www.geeksforgeeks.org/write-header-file-c/