UNIVERSITY PARTNER

UNIVERSITY OF WOLVERHAMPTON

HERALD COLLEGE KATHMANDU

6CS005 High Performance Computing

Portfolio 1 : Task2

Student Name: Pratik Dhimal

University ID: 2407779

Group: L6CG1

Tutor: Yamu Poudel

Submission Date: 20th December, 2025

Table of Contents

1. Introduction

The aim of this coursework was to develop a high-performance C program that is reliable and uses the OpenMP library to perform addition, subtraction, element-wise multiplication/division, transposition and matrix multiplication among other matrix operations. Increasing the matrix dimension causes single threaded matrix processing to become highly inefficient and it does not exploit the multi-core capabilities of modern CPUs.

To deal with this, the project employs a multi-threaded technique, which has spread the load between multiple threads. This study describes the transition of a brittle implementation of a solver to a stable, high-performance solver. The completed application is capable of working with corrupted data without crashing, reading multiple matrices in an input file and operating them two at a time and the ability to control the memory dynamically. This method concentration is on the vital concepts of memory safety, synchronisation, and parallel running in the field of high-performance computing.

2. Execution

2.1. Input Validation

Endptr checks and strtod() function are now utilized in the program, as in **Figure 1**. This will ensure that every letter in a token is a valid part of a number. The software identifies an error and removes the memory already in use and proceeds to the next pair of matrices when it comes across a non-numeric token. This will ensure integrity of output and prevent the system to process garbage data.

2.2. Memory Management and Safety

This part is concerned with the safe creation, deallocation and access of matrices in memory, and the reading and writing of files. As shown in **Figure 2** and **Figure 3**, the **allocate_matrix** function is a dynamic means of constructing a 2D array of doubles and detects allocation failures and the **deallocate_matrix** is used to free all memory allocated to avoid leakage. In managing files, load matrix uses the read token to read out a complete matrix of the input file with the dimensions and contents being valid integers and read token reads numeric values in a file and ignores commas, spaces and new lines. Similarly, as shown in **Figure 4, display_matrix** to file is a form of representation that numbers extensive numbers and elegantly receives NaNs when writing a matrix to a file. A combination of these processes ensures uniform file I/O and high memory usage within the application.

2.3.OpenMP Parallelisation and Thread Capping

As shown in **Figure 5,** the program uses OpenMP to parallelize computationally intensive tasks across multiple threads, allowing for the simultaneous execution of loops for matrix arithmetic, multiplication, and transposition. Thread capping is used to prevent the creation of more threads than the relevant matrix dimension in order to reduce overhead and ensure efficient resource use. By **dynamically adjusting** the number of threads in accordance with matrix size, the approach optimizes performance without unnecessary context switching or memory contention, striking a balance between speed and system stability. This architecture scales effectively with matrix dimensions while ensuring safe and predictable parallel execution.

2.4.Matrix Calculations

2.4.1. Addition

The **matrix addition** process involves the addition of two matrices of same dimensions element by element. The function calculates the sum of the matching elements through the iteration of the row and column using simultaneous threads, after memory allocation of the ultimate matrix. To avoid corrupt operations, addition is not performed when the matrices of the input are of dissimilar shapes. As shwon in **Figure 6** This technique involves the OpenMP threading that can be used to integrate secure memory handling with efficient computation.

2.4.2. Subtraction

Element-wise Matrix subtraction refers to the operation of subtraction between two matrices of equal size, similar to addition. The difference between the respective elements of the input matrices is calculated to obtain the elements of the output matrix, and a new result matrix is allotted. Whereas dimension checks prevent any not valid practices, parallelisation is employed to accelerate computation. This will ensure that accuracy and performance is maintained This is shwon well in **Figure 6.**

2.4.3. Element wise multiplication

Element-wise multiplication is the computation of the product of similar elements in two matrices of the same size. The process multiplies every element of the former matrix by the element of the latter one after assigning a new matrix to contain the outcomes. As shown in **Figure 6,** OpenMP parallel threads are used to improve performance and shape validation used to maintain stability in the program to ensure that only compatible matrices are multiplied.

2.4.4. Element wise division

The **Division** of corresponding elements of two matrices with the same dimensions is determined by element-wise division. Each element is divided into a result matrix, with a check to assign NaN in the event that the division is by zero, as shown in **Figure 6**. Performance can be increased by processing multiple rows at once thanks to parallelization, which also makes sure that invalid operations—like division by zero or mismatched shapes—are handled safely.

2.4.5. Transpose

As shown in **Figure 6, Transpose** operation transforms rows into columns and vice versa. Each element is copied to its transposed position and a new matrix is assigned with the dimensions reversed. To increase processing speed, OpenMP parallelisation is used row-wise. This operation is essential for later operations like matrix multiplication and is always valid regardless of the original matrix structure.

2.4.6. Dot Product (Multiplication*)*

When the number of columns in the first matrix is equal to the number of rows in the second, **matrix multiplication** calculates the dot product of the two matrices. Each element is calculated as the sum of products of related elements in the row of the first matrix and the column of the second, and a result matrix with the proper output dimensions is assigned. While shape validation guarantees that only compatible matrices are multiplied, preserving correctness, parallelisation is used across rows to speed up computation. This whole porcess is shown in Figure **7**.

2.5.Handling Edge Cases and Division

Decoupling the operations was the last logic improvement to make sure that a failure in one would not halt the process as a whole. Mismatched dimensions for addition could completely halt the processing of that matrix pair in the base code.The updated program independently verifies each operation's requirements. Additionally, a particular check for "division by zero" was added for element-wise division. When a divisor is zero, the program specifically gives that cell a NaN (Not a Number) value. This keeps the program from crashing with a floating-point exception and guarantees that the output complies with standards for scientific computing.

3. Conclusion

This project effectively uses OpenMP-based multithreading and dynamic memory allocation to create a scalable and reliable matrix operations program in C. The software performs a wide range of matrix operations, validates dimensional compatibility, and correctly parses matrices of any size from an input file while gracefully handling invalid cases. Correctness and performance are guaranteed by careful file handling, effective memory management, and thread capping. All things considered, the solution shows how to effectively use parallel computing techniques to optimize numerical computations while preserving design clarity, dependability, and extensibility.

4. Appendix

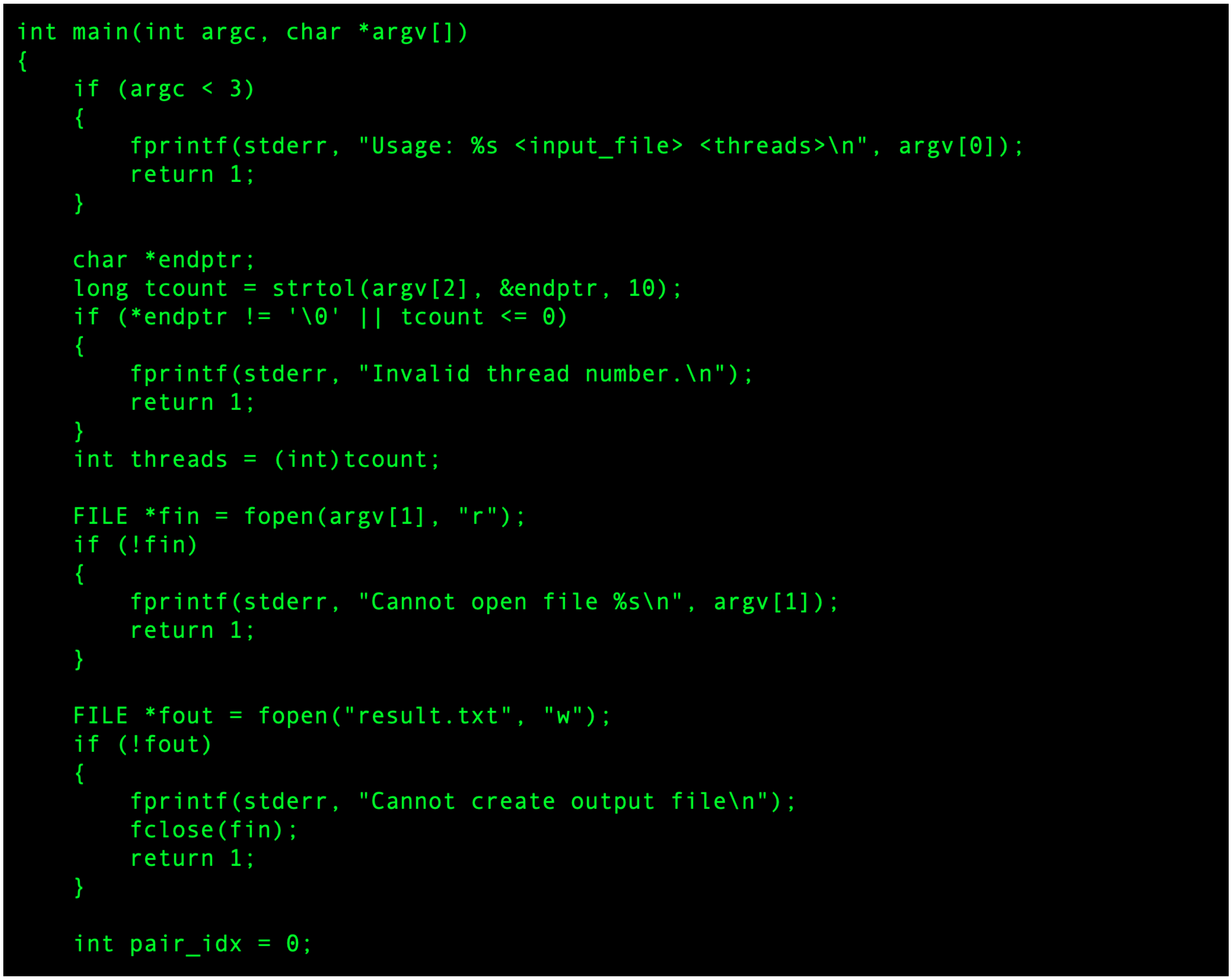
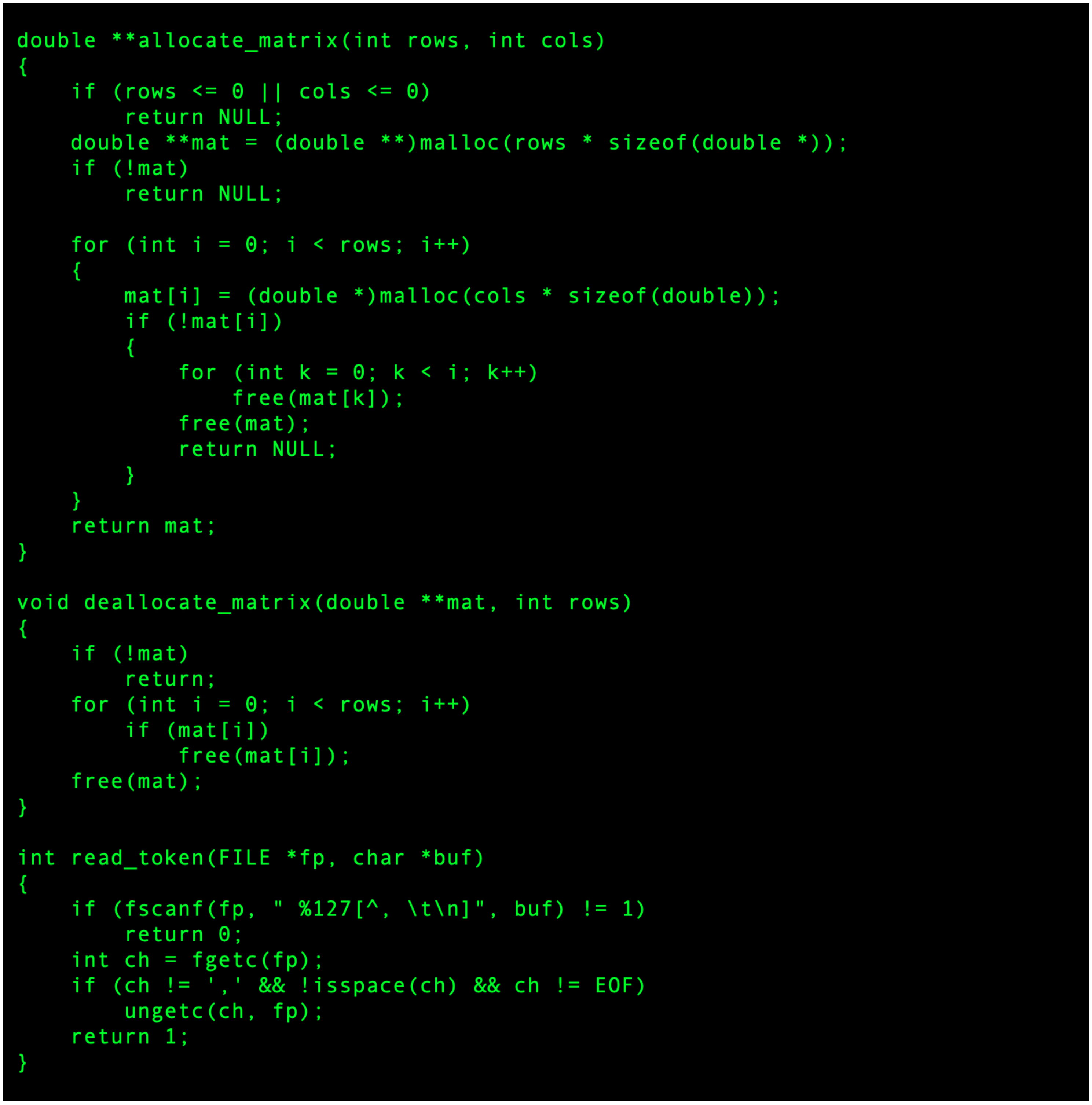
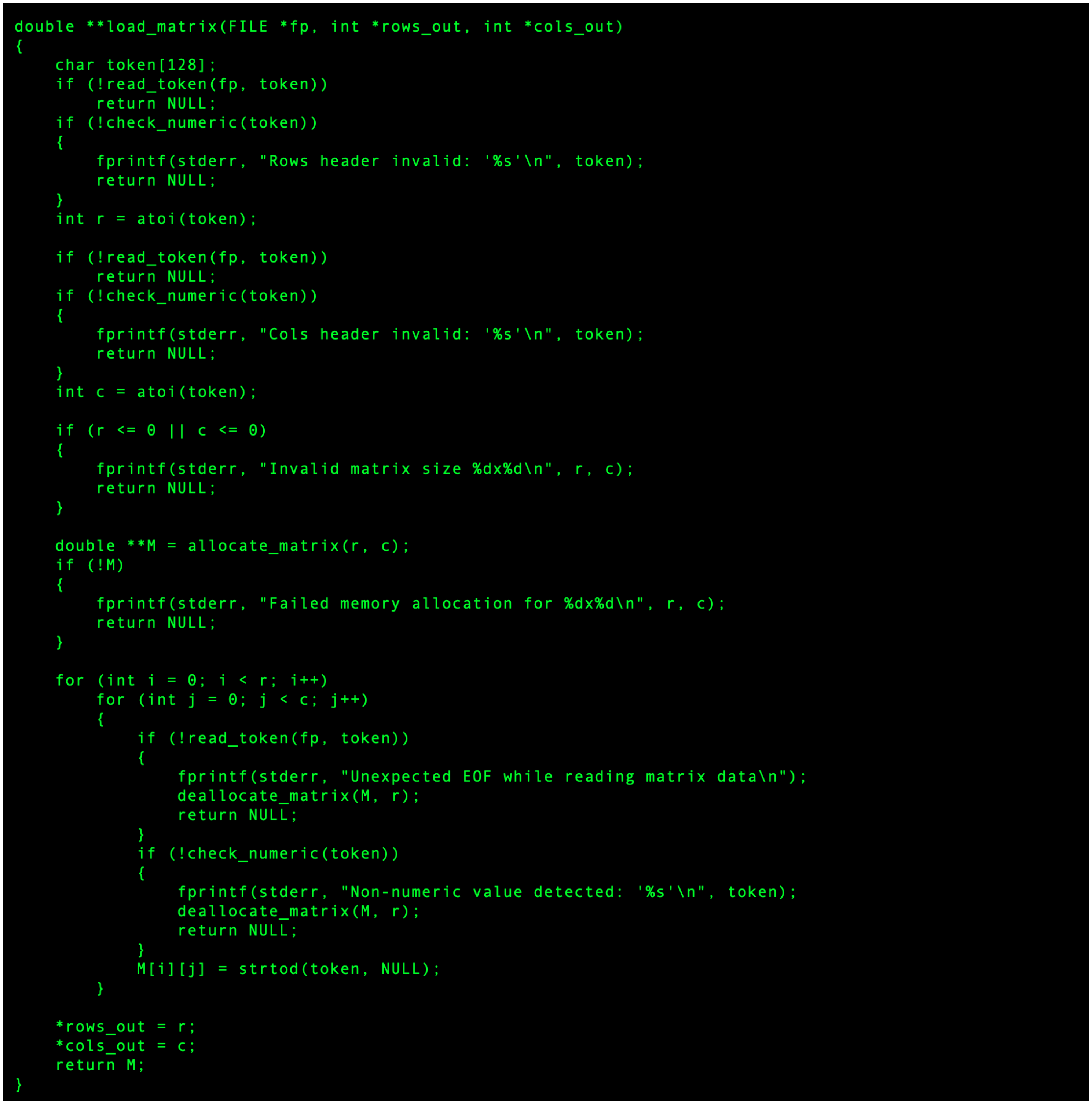

```
int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s <input_file> <threads>\n", argv[0]);
        return 1;
    }

    char *endptr;
    long tcount = strtol(argv[2], &endptr, 10);
    if (*endptr != '\0' || tcount <= 0)
    {
        fprintf(stderr, "Invalid thread number.\n");
        return 1;
    }
    int threads = (int)tcount;

    FILE *fin = fopen(argv[1], "r");
    if (!fin)
    {
        fprintf(stderr, "Cannot open file %s\n", argv[1]);
        return 1;
    }

    FILE *fout = fopen("result.txt", "w");
    if (!fout)
    {
        fprintf(stderr, "Cannot create output file\n");
        fclose(fin);
        return 1;
    }

    int pair_idx = 0;
```

Figure 1 Input validiation and file handeling



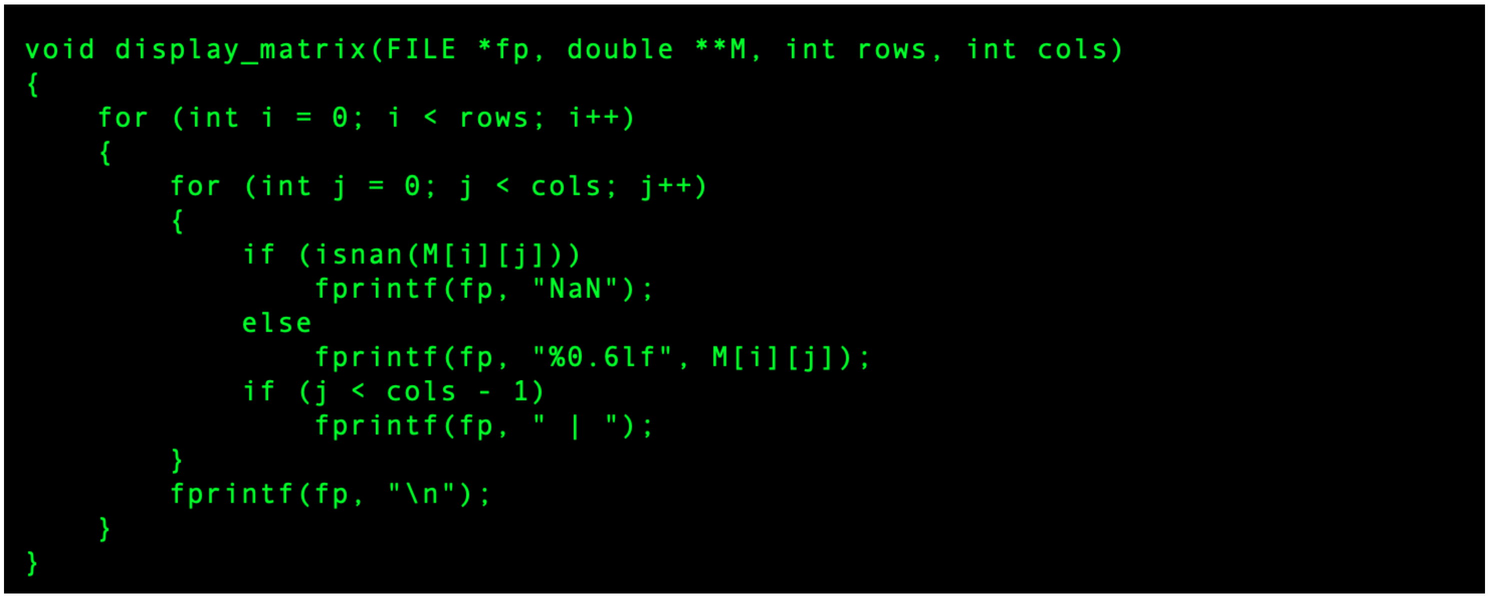
```c
double **allocate_matrix(int rows, int cols)
{
    if (rows <= 0 || cols <= 0)
        return NULL;
    double **mat = (double **)malloc(rows * sizeof(double *));
    if (!mat)
        return NULL;

    for (int i = 0; i < rows; i++)
    {
        mat[i] = (double *)malloc(cols * sizeof(double));
        if (!mat[i])
        {
            for (int k = 0; k < i; k++)
                free(mat[k]);
            free(mat);
            return NULL;
        }
    }
    return mat;
}

void deallocate_matrix(double **mat, int rows)
{
    if (!mat)
        return;
    for (int i = 0; i < rows; i++)
        if (mat[i])
            free(mat[i]);
    free(mat);
}

int read_token(FILE *fp, char *buf)
{
    if (fscanf(fp, " %127[^, \t\n]", buf) != 1)
        return 0;
    int ch = fgetc(fp);
    if (ch != ',' && !isspace(ch) && ch != EOF)
        ungetc(ch, fp);
    return 1;
}
```

Figure 2 Memory allocation



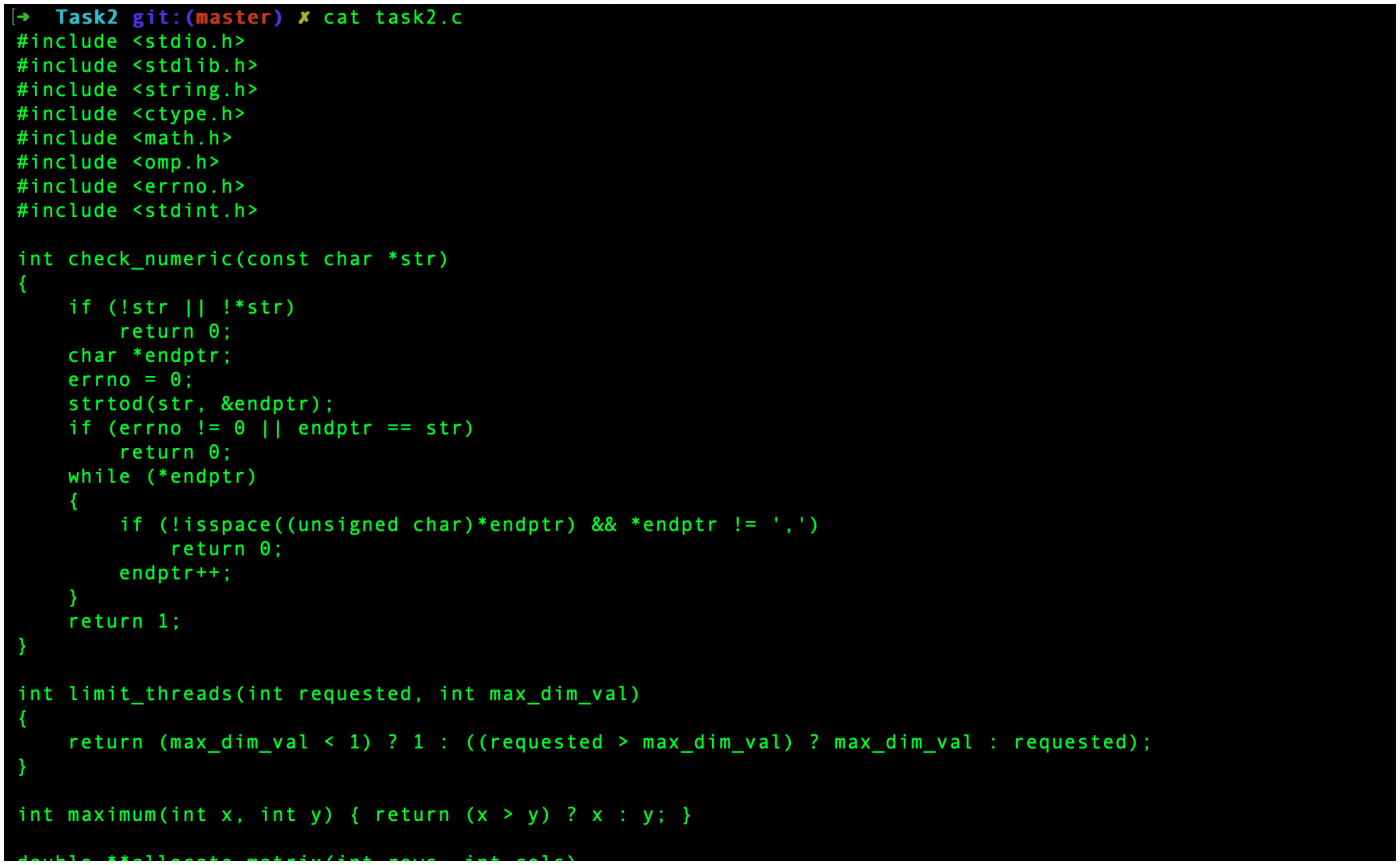
```c
double **load_matrix(FILE *fp, int *rows_out, int *cols_out)
{
    char token[128];
    if (!read_token(fp, token))
        return NULL;
    if (!check_numeric(token))
    {
        fprintf(stderr, "Rows header invalid: '%s'\n", token);
        return NULL;
    }
    int r = atoi(token);

    if (!read_token(fp, token))
        return NULL;
    if (!check_numeric(token))
    {
        fprintf(stderr, "Cols header invalid: '%s'\n", token);
        return NULL;
    }
    int c = atoi(token);

    if (r <= 0 || c <= 0)
    {
        fprintf(stderr, "Invalid matrix size %dx%d\n", r, c);
        return NULL;
    }

    double **M = allocate_matrix(r, c);
    if (!M)
    {
        fprintf(stderr, "Failed memory allocation for %dx%d\n", r, c);
        return NULL;
    }

    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
        {
            if (!read_token(fp, token))
            {
                fprintf(stderr, "Unexpected EOF while reading matrix data\n");
                deallocate_matrix(M, r);
                return NULL;
            }
            if (!check_numeric(token))
            {
                fprintf(stderr, "Non-numeric value detected: '%s'\n", token);
                deallocate_matrix(M, r);
                return NULL;
            }
            M[i][j] = strtod(token, NULL);
        }

    *rows_out = r;
    *cols_out = c;
    return M;
}
```

Figure 3 Load matrix to memory



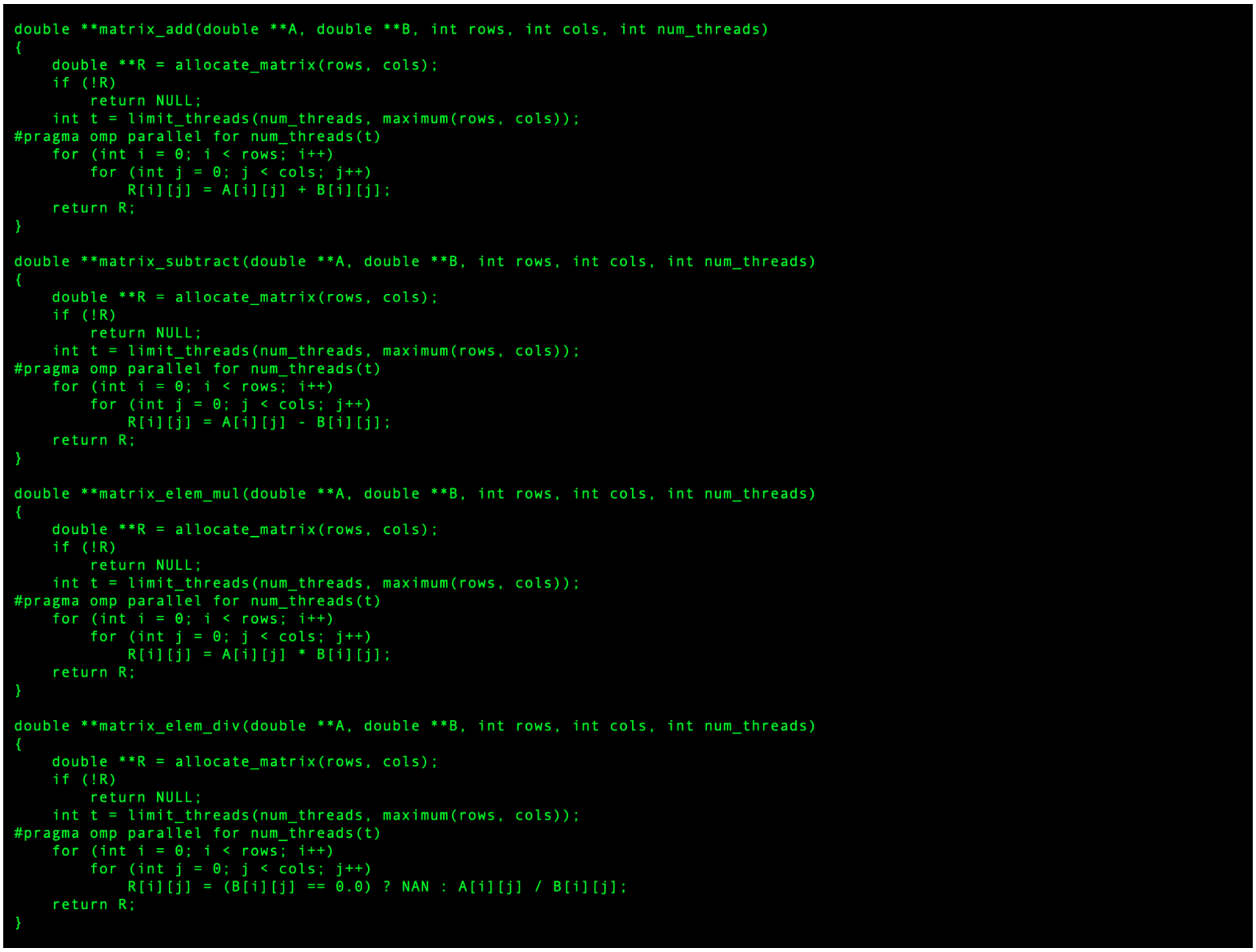
```c
void display_matrix(FILE *fp, double **M, int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            if (isnan(M[i][j]))
                fprintf(fp, "NaN");
            else
                fprintf(fp, "%0.6lf", M[i][j]);
            if (j < cols - 1)
                fprintf(fp, " | ");
        }
        fprintf(fp, "\n");
    }
}
```

Figure 4 Display Matric



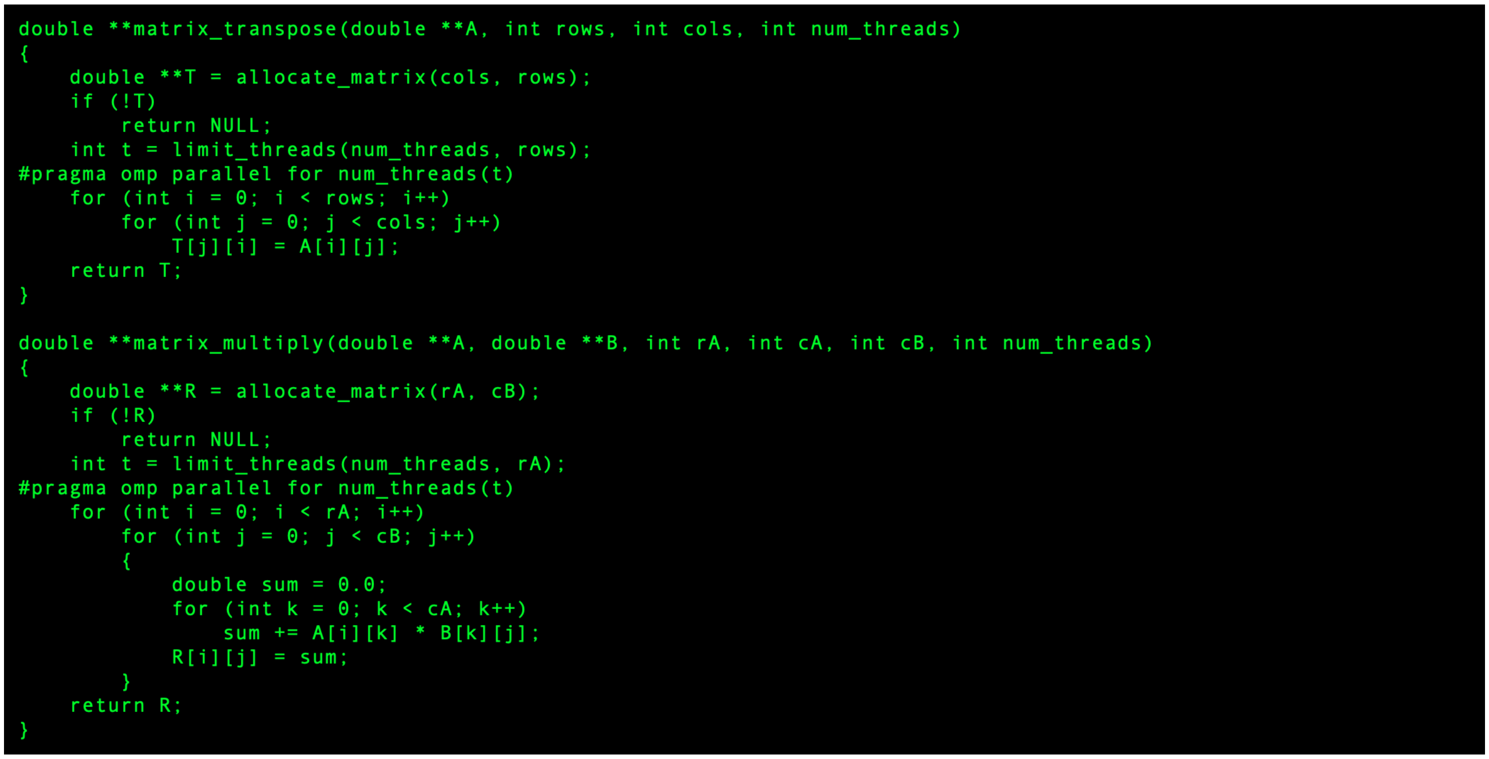
```
[→  Task2 git:(master) ✗ cat task2.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <omp.h>
#include <errno.h>
#include <stdint.h>

int check_numeric(const char *str)
{
    if (!str || !*str)
        return 0;
    char *endptr;
    errno = 0;
    strtod(str, &endptr);
    if (errno != 0 || endptr == str)
        return 0;
    while (*endptr)
    {
        if (!isspace((unsigned char)*endptr) && *endptr != ',')
            return 0;
        endptr++;
    }
    return 1;
}

int limit_threads(int requested, int max_dim_val)
{
    return (max_dim_val < 1) ? 1 : ((requested > max_dim_val) ? max_dim_val : requested);
}

int maximum(int x, int y) { return (x > y) ? x : y; }

double **allocate_matrix(int rows, int cols)
```

Figure 5 Utility Functions



```
double **matrix_add(double **A, double **B, int rows, int cols, int num_threads)
{
    double **R = allocate_matrix(rows, cols);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, maximum(rows, cols));
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            R[i][j] = A[i][j] + B[i][j];
    return R;
}

double **matrix_subtract(double **A, double **B, int rows, int cols, int num_threads)
{
    double **R = allocate_matrix(rows, cols);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, maximum(rows, cols));
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            R[i][j] = A[i][j] - B[i][j];
    return R;
}

double **matrix_elem_mul(double **A, double **B, int rows, int cols, int num_threads)
{
    double **R = allocate_matrix(rows, cols);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, maximum(rows, cols));
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            R[i][j] = A[i][j] * B[i][j];
    return R;
}

double **matrix_elem_div(double **A, double **B, int rows, int cols, int num_threads)
{
    double **R = allocate_matrix(rows, cols);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, maximum(rows, cols));
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            R[i][j] = (B[i][j] == 0.0) ? NAN : A[i][j] / B[i][j];
    return R;
}
```

Figure 6 Matrix calculation 1



```c
double **matrix_transpose(double **A, int rows, int cols, int num_threads)
{
    double **T = allocate_matrix(cols, rows);
    if (!T)
        return NULL;
    int t = limit_threads(num_threads, rows);
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            T[j][i] = A[i][j];
    return T;
}

double **matrix_multiply(double **A, double **B, int rA, int cA, int cB, int num_threads)
{
    double **R = allocate_matrix(rA, cB);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, rA);
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rA; i++)
        for (int j = 0; j < cB; j++)
        {
            double sum = 0.0;
            for (int k = 0; k < cA; k++)
                sum += A[i][k] * B[k][j];
            R[i][j] = sum;
        }
    return R;
}
```

Figure 7 Matrix Calculation 2