High-Level Documentation: Parallel and Distributed Numerical Computation System

Project Overview

This project is a Parallel and Distributed Numerical Computation System designed to provide solutions to various mathematical problems using parallel programming techniques. The system allows users to solve linear equations, evaluate polynomials, compute numerical integration, estimate the value of Pi using Monte Carlo methods, and solve non-linear equations using Newton-Raphson's method. The program leverages multithreading for parallel computation, ensuring improved performance and scalability. Users can execute multiple algorithms concurrently, and computation results are logged for future reference.

Introduction and Contextualization

Context

This system is designed to address the challenge of performing large-scale numerical computations efficiently. In fields such as engineering, physics, and data science, solving mathematical problems such as systems of linear equations, polynomial evaluations, and numerical integration is a crucial part of research and real-world applications. However, as the problem size increases, the time and resources required to solve these problems grow exponentially. This is where parallel computing becomes essential, as it allows computations to be distributed across multiple processors, dramatically reducing computation time.

For instance, in engineering, the Gaussian elimination method is widely used in circuit simulation and structural analysis to solve systems of linear equations. Similarly, Monte Carlo simulations, which involve random sampling, are extensively used in data science and finance to model complex probabilistic systems, such as pricing options in financial markets or risk assessment. Numerical integration techniques like the Trapezoidal rule are also employed in physics to solve differential equations and compute areas under curves, where analytical solutions are either difficult or impossible to obtain.

By leveraging multithreading and asynchronous operations, this system can perform computational tasks in parallel, reducing the time required to process large data sets or complex problems. This is especially useful for students and researchers who need to solve such problems frequently as part of their academic work or research projects. The system thus bridges the gap between conceptual understanding and practical implementation by providing an efficient tool for solving computational problems.

Objective

The primary objective of this system is to provide a high-performance computing solution for solving various mathematical problems that are commonly encountered in research and academic settings. By employing parallel computing techniques, the system is designed to address the challenge of scaling computational workloads, making it an ideal tool for handling large and complex datasets.

The system integrates parallelism in multiple algorithms, demonstrating how task-based parallelism can be applied to numerical computations. It is intended to help students and researchers by reducing the execution time for problems that would otherwise take hours or even days to compute sequentially. With this system, users can experience firsthand the advantages of parallel computing and better understand how it can be used to accelerate problem-solving in domains like physics, engineering, and data science.

Project Solution

This system is designed to provide a robust, multi-threaded computational tool for solving a variety of complex mathematical problems, which is particularly useful for students and researchers dealing with computational tasks in areas like engineering, physics, and applied mathematics. The system leverages parallel and asynchronous processing to handle intensive numerical tasks efficiently, such as solving systems of linear equations using Gaussian elimination, evaluating polynomials using Horner's method, performing numerical integration with the trapezoidal rule, and estimating the value of Pi through Monte Carlo simulations. Additionally, the system can find the roots of non-linear equations using the Newton-Raphson method.

One of the key innovations of the system is its ability to execute multiple algorithms concurrently, which demonstrates its applicability to high-performance computing tasks where speed and efficiency are critical. The project integrates concurrency using threads, mutexes for thread safety, and asynchronous programming to ensure that the system can scale and handle large problem sizes effectively. Furthermore, it includes input validation mechanisms to prevent user errors, making the system robust and user-friendly.

This system addresses a real-world problem of making complex mathematical computations accessible and efficient, even for non-experts who need to solve such problems as part of their academic or research work. It is designed to be flexible, allowing users to run different algorithms with customized inputs, and outputs are logged to files for tracking and further analysis. For students, this system provides an invaluable tool that simplifies solving typical mathematical challenges in coursework or research, such as solving equations, performing integrations, or estimating constants like Pi, thereby saving time and enhancing accuracy. It also allows users to experiment with algorithmic performance through the concurrent execution of tasks, making it a powerful solution in computational mathematics for education and research.

This code is a multi-threaded, parallel computing application designed to solve a variety of mathematical problems, including Gaussian Elimination, polynomial evaluation, numerical integration, Monte Carlo simulation for Pi estimation, and the Newton-Raphson method for finding roots of non-linear equations. It leverages modern C++ features like std::thread, std::mutex, and std::async to ensure efficient parallel execution of tasks.

By using multi-threading, the code distributes the workload across multiple threads, which can be executed simultaneously, improving performance on multi-core processors. For example, when evaluating polynomials at multiple points or calculating Pi using Monte Carlo simulations, the tasks are split into smaller units of work, each assigned to different threads. This reduces the overall time required for computations compared to running them sequentially.

The code also includes robust error handling and input validation to ensure that users provide valid inputs, enhancing the usability of the system. A log system records the input parameters, results, and the execution time for each task into a file, providing a detailed history of previous computations. This feature is particularly useful in a student or research setting where reproducibility and performance tracking are important.

Furthermore, the system allows users to run multiple algorithms concurrently, showcasing its flexibility in handling diverse mathematical tasks in parallel. Overall, the code is an example of how parallel programming can be applied to computational mathematics, making it a valuable tool for students studying algorithms, numerical methods, or parallel computing in engineering, physics, or data science.

Key Features

1. Solve Linear Equations using Gaussian Elimination

- Implements the Gaussian Elimination algorithm to solve systems of linear equations.
- Uses forward elimination followed by back-substitution to compute the solution vector.
- Parallelism: Single-threaded computation (serial execution of Gaussian elimination).
- User Input: Matrix coefficients and result vector.
- Logged Outputs: Solution vector and execution time.

2. Polynomial Evaluation using Horner's Method

- Evaluates polynomials at given points using Horner's Method, a numerically stable approach for evaluating polynomials.
- Parallelism: Each evaluation point is computed independently using std::async, enabling parallel evaluations.
- User Input: Polynomial coefficients and evaluation points.
- Logged Outputs: Evaluated polynomial results and execution time.

3. Numerical Integration using the Trapezoidal Rule

- Performs numerical integration over a specified range using the Trapezoidal Rule.
- Parallelism: The integration is divided into sub-intervals and computed in parallel across multiple threads.
- User Input: Integration limits and the number of sub-intervals.
- Logged Outputs: Result of the integration and execution time.

4. Monte Carlo Pi Estimation

- Uses a Monte Carlo simulation to estimate the value of Pi by generating random points in a unit square and calculating the ratio of points that fall inside the unit circle.
- Parallelism: Points are distributed across multiple threads for concurrent calculation.
- User Input: Total number of points to simulate.
- Logged Outputs: Estimated value of Pi and execution time.

5. Newton-Raphson Root Finding

- Solves non-linear equations using the Newton-Raphson method to iteratively approximate the roots of a given function.
- Parallelism: Each initial guess is processed in parallel using separate threads to find roots for different starting points.
- User Input: Initial guesses for root finding.
- Logged Outputs: Computed roots and execution time.

6. Concurrent Execution of Multiple Algorithms

- Allows users to run multiple algorithms concurrently in parallel threads.
- User Input: Choice of algorithms and their respective inputs.
- Logged Outputs: Results of all concurrent tasks and execution time.

7. Computation History Log

- All computations are logged to a file, allowing users to view the history of previous tasks, their inputs, results, and execution times.

Software Architecture

Modules

Main Program Loop: Manages user input and orchestrates the flow of tasks.

Task-specific Functions:

- Gaussian Elimination for linear equations.
- Horner’s Method for polynomial evaluation.
- Trapezoidal Rule for numerical integration.
- Monte Carlo Simulation for Pi estimation.
- Newton-Raphson Method for non-linear root finding.

Parallelism Layer:

- Implements parallel computation using threads and std::async.
- Supports concurrent execution of multiple algorithms.

Logging Module:

- Captures task names, inputs, results, and execution times in a log file.

Menu System:

- Provides a user-friendly interface for selecting algorithms and entering inputs.

Technologies Used

- C++ Standard Library:
- Multithreading: std::thread, std::async, std::mutex, std::lock_guard for managing concurrency.
- I/O Management: std::ofstream and std::ifstream for logging results and reading history.
- Error Handling: Input validation and mutex-protected logging to ensure thread safety.
- Colourful Console Output: ANSI escape codes are used for colored and styled console output to enhance user experience.

Design Considerations

Parallelism and Scalability

- **Task Parallelism:** The project emphasizes task-based parallelism. Each computational task (such as polynomial evaluation or numerical integration) can be divided across multiple threads or asynchronous tasks.

- **Concurrency in Algorithms:** The system supports running multiple algorithms concurrently, optimizing for multi-core processors.

User Input Validation

- Robust input validation is implemented to ensure the system handles erroneous or invalid inputs gracefully, prompting users to re-enter data where necessary.

Performance Logging

- Each computation task is timed, and the execution time is recorded and presented to the user, along with the results. This allows for performance evaluation and debugging.

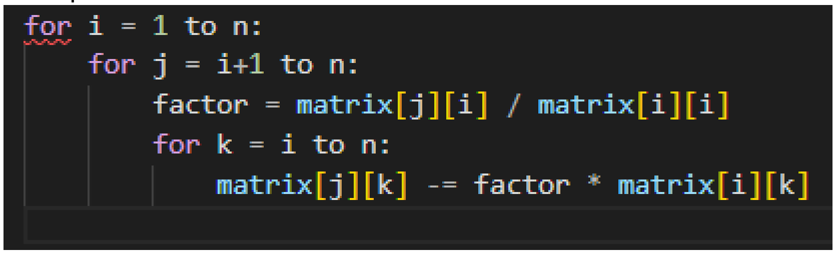
User Interaction

- Main Menu: Users can choose which mathematical algorithm they want to execute.
- Input Collection: Depending on the algorithm chosen, users are prompted to enter necessary inputs (e.g., matrix coefficients for Gaussian elimination, polynomial coefficients for Horner's method, or bounds for numerical integration).
- Execution and Results: The program executes the chosen algorithm in parallel, displays the results to the user, and logs the computation history.
- Computation History: Users can view a detailed log of previous computations and their results.

Algorithms

Gaussian Elimination

Mathematical Background: Gaussian elimination is a method used to solve systems of linear equations. It involves converting a matrix into an upper triangular form through a series of row operations, after which back substitution is used to find the solutions. The method works for any system that can be represented in matrix form and is particularly useful when dealing with multiple variables.



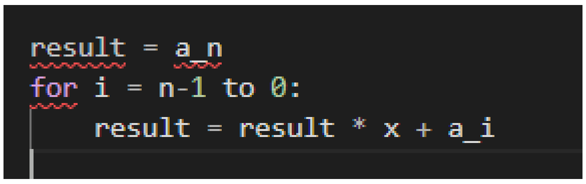
```
for i = 1 to n:
    for j = i+1 to n:
        factor = matrix[j][i] / matrix[i][i]
        for k = i to n:
            matrix[j][k] -= factor * matrix[i][k]
```

Use Cases: Gaussian elimination is widely used in circuit analysis to solve Kirchhoff's current law equations and in structural engineering for analyzing forces on structures like bridges or trusses. The method can also be used to solve heat distribution problems in thermodynamics, where the system of linear equations represents temperatures at different points in a medium.

Horner's Method for Polynomial Evaluation

Mathematical Background: Horner’s method is an efficient algorithm for evaluating polynomials of the form:

$$P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

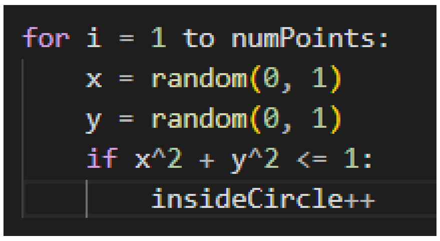


```
result = a_n
for i = n-1 to 0:
    result = result * x + a_i
```

Use Cases: This method is particularly useful in computer graphics for evaluating Bézier curves or for root finding in algebraic equations, making it a practical choice in fields like robotics, signal processing, and data interpolation.

Monte Carlo Simulation for Pi Estimation

Mathematical Background: The Monte Carlo method is a statistical technique that uses random sampling to estimate numerical values. In this system, it is used to estimate the value of Pi by randomly generating points inside a square and counting how many fall within the inscribed circle.

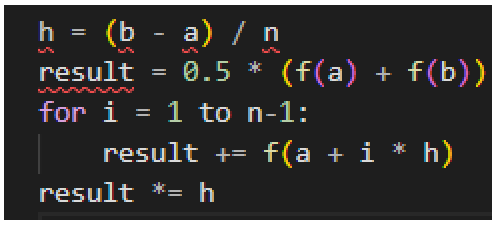


```
for i = 1 to numPoints:
    x = random(0, 1)
    y = random(0, 1)
    if x^2 + y^2 <= 1:
        insideCircle++
```

Use Cases: Monte Carlo simulations are used in financial modeling to evaluate risks and in physics to simulate particle behavior in complex systems. For example, they are used in quantum mechanics and nuclear physics to simulate random events and interactions over time.

Numerical Integration (Trapezoidal Rule)

Mathematical Background: The Trapezoidal rule is a method of numerical integration that approximates the area under a curve by dividing it into a series of trapezoids. It is useful when the function being integrated cannot be solved analytically.



```
h = (b - a) / n
result = 0.5 * (f(a) + f(b))
for i = 1 to n-1:
    result += f(a + i * h)
result *= h
```

Use Cases: This method is widely used in physics to solve integrals that arise in motion equations, as well as in finance to compute the area under a stock price graph over time.

Parallelism

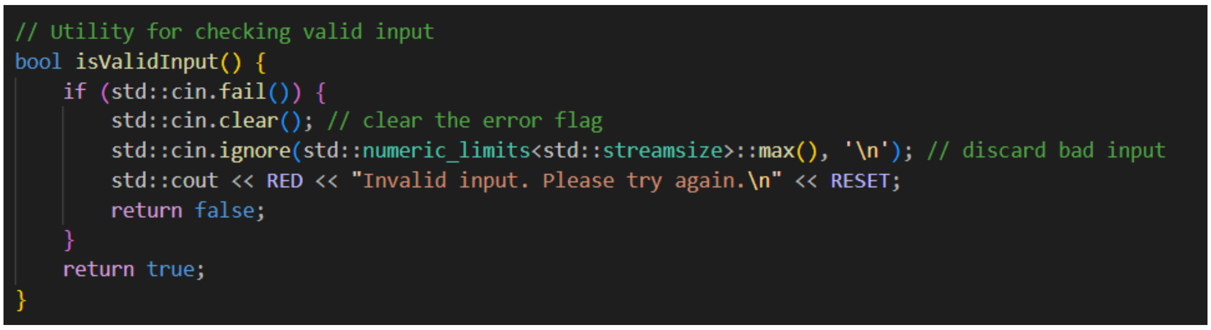
Parallelism Benefits: In each of the algorithms above, parallelism provides significant improvements in execution time. By breaking the task into smaller sub-tasks that can be computed concurrently, the system reduces the overall time required to complete the operation. For example:

In Gaussian elimination, parallelism can be used to perform row operations concurrently.

In the Monte Carlo simulation, the task of randomly generating points and checking their position relative to the circle can be split across multiple threads, making the estimation faster as more points are evaluated simultaneously.

Function Documentation

Is-Valid-Input



```cpp
// Utility for checking valid input
bool isValidInput() {
    if (std::cin.fail()) {
        std::cin.clear(); // clear the error flag
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // discard bad input
        std::cout << RED << "Invalid input. Please try again.\n" << RESET;
        return false;
    }
    return true;
}
```

Purpose:

- ➢ This function checks whether the input provided by the user is valid or not.
- ➢ It clears the input buffer and prompts the user to enter valid data if an invalid input is detected.

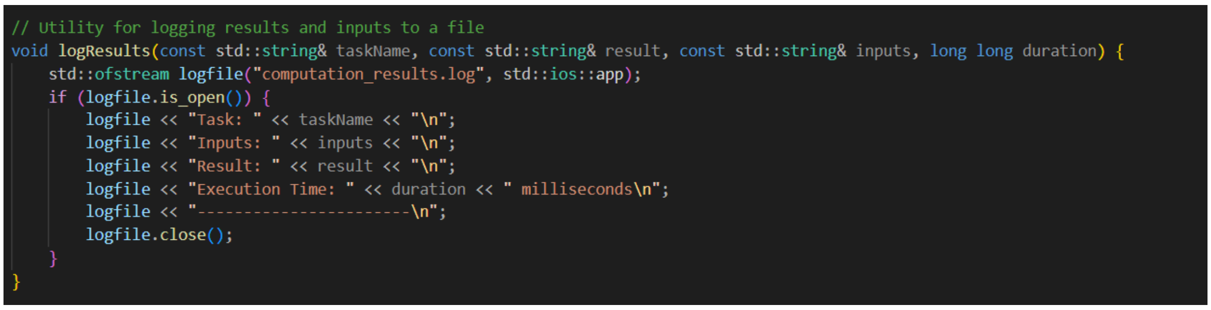
Input:

- ➢ This function does not take any parameters.

Output:

- ➢ Returns true if the input is valid, otherwise returns false after resetting the input stream.

Log Results



```cpp
// Utility for logging results and inputs to a file
void logResults(const std::string& taskName, const std::string& result, const std::string& inputs, long long duration) {
    std::ofstream logfile("computation_results.log", std::ios::app);
    if (logfile.is_open()) {
        logfile << "Task: " << taskName << "\n";
        logfile << "Inputs: " << inputs << "\n";
        logfile << "Result: " << result << "\n";
        logfile << "Execution Time: " << duration << " milliseconds\n";
        logfile << "----------------------\n";
        logfile.close();
    }
}
```

Purpose:

- This function logs the results of computations to a file named computation_results.log. It includes the task name, inputs, results, and the time taken to perform the task.

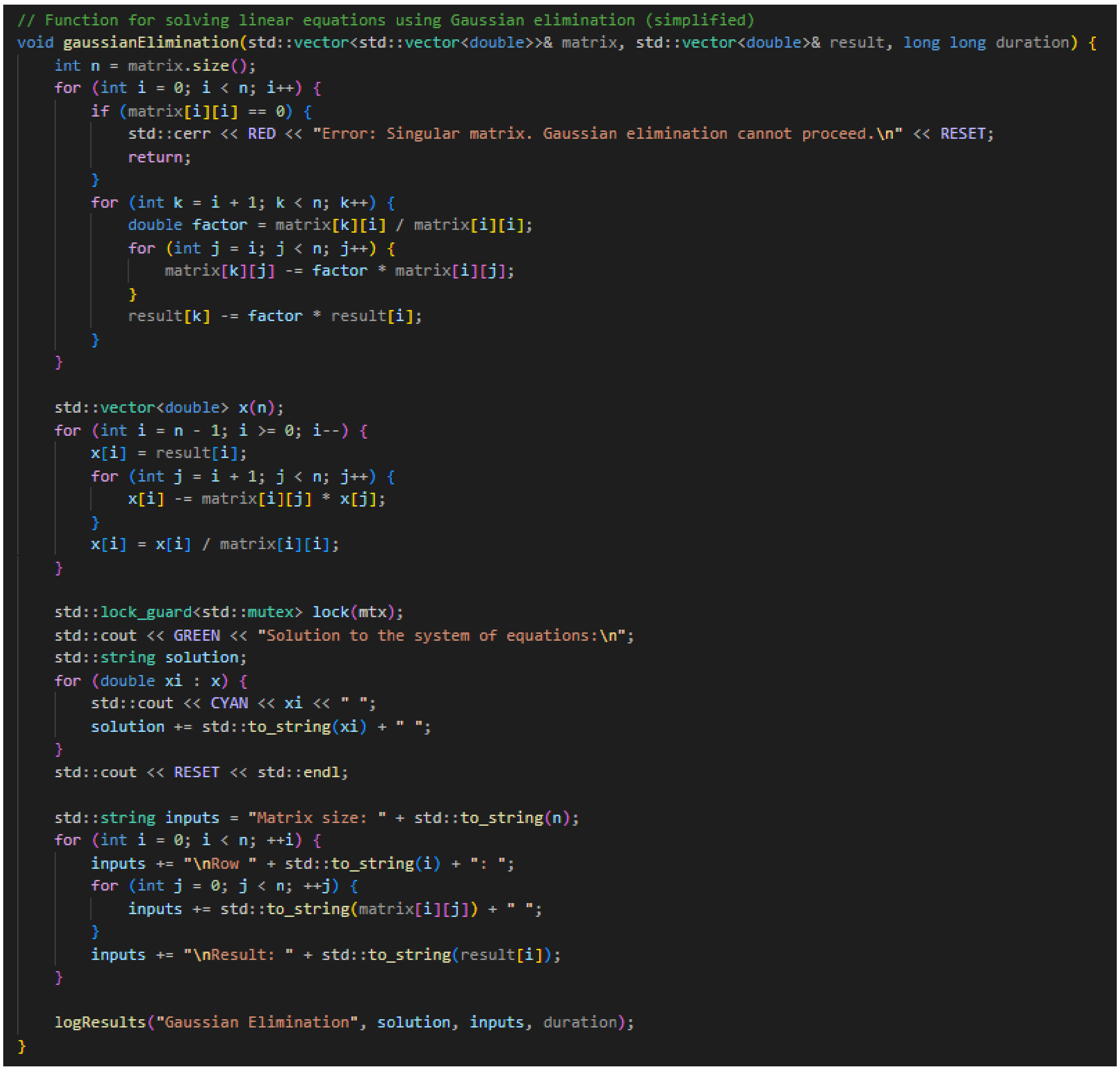
Input:

- taskName: The name of the task (e.g., Gaussian Elimination).
- result: The result of the computation.
- inputs: The inputs provided by the user.
- duration: The time taken to complete the task in milliseconds.

Output:

- None. Writes data to a log file.

Gaussian Elimination



```cpp
// Function for solving linear equations using Gaussian elimination (simplified)
void gaussianElimination(std::vector<std::vector<double>>& matrix, std::vector<double>& result, long long duration) {
    int n = matrix.size();
    for (int i = 0; i < n; i++) {
        if (matrix[i][i] == 0) {
            std::cerr << RED << "Error: Singular matrix. Gaussian elimination cannot proceed.\n" << RESET;
            return;
        }
        for (int k = i + 1; k < n; k++) {
            double factor = matrix[k][i] / matrix[i][i];
            for (int j = i; j < n; j++) {
                matrix[k][j] -= factor * matrix[i][j];
            }
            result[k] -= factor * result[i];
        }
    }

    std::vector<double> x(n);
    for (int i = n - 1; i >= 0; i--) {
        x[i] = result[i];
        for (int j = i + 1; j < n; j++) {
            x[i] -= matrix[i][j] * x[j];
        }
        x[i] = x[i] / matrix[i][i];
    }

    std::lock_guard<std::mutex> lock(mtx);
    std::cout << GREEN << "Solution to the system of equations:\n";
    std::string solution;
    for (double xi : x) {
        std::cout << CYAN << xi << " ";
        solution += std::to_string(xi) + " ";
    }
    std::cout << RESET << std::endl;

    std::string inputs = "Matrix size: " + std::to_string(n);
    for (int i = 0; i < n; ++i) {
        inputs += "\nRow " + std::to_string(i) + ": ";
        for (int j = 0; j < n; ++j) {
            inputs += std::to_string(matrix[i][j]) + " ";
        }
        inputs += "\nResult: " + std::to_string(result[i]);
    }

    logResults("Gaussian Elimination", solution, inputs, duration);
}
```

Purpose:

- Solves a system of linear equations using the **Gaussian Elimination** method.
- This function performs forward elimination and back-substitution to compute the solution vector.

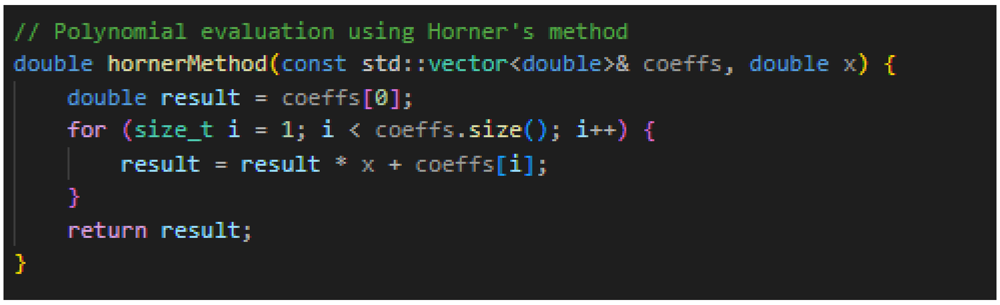
Input:

- matrix: A 2D vector containing the coefficients of the system of linear equations.
- result: A vector containing the result values (right-hand side of the equations).
- duration: The time taken to complete the Gaussian Elimination task (passed for logging purposes).

Output:

- Outputs the solution to the system of equations on the console.
- The result is also logged into a file.

Horner Method



```cpp
// Polynomial evaluation using Horner's method
double hornerMethod(const std::vector<double>& coeffs, double x) {
    double result = coeffs[0];
    for (size_t i = 1; i < coeffs.size(); i++) {
        result = result * x + coeffs[i];
    }
    return result;
}
```

Purpose:

- Evaluates a polynomial at a given point using Horner's Method, which minimizes the number of multiplications required.

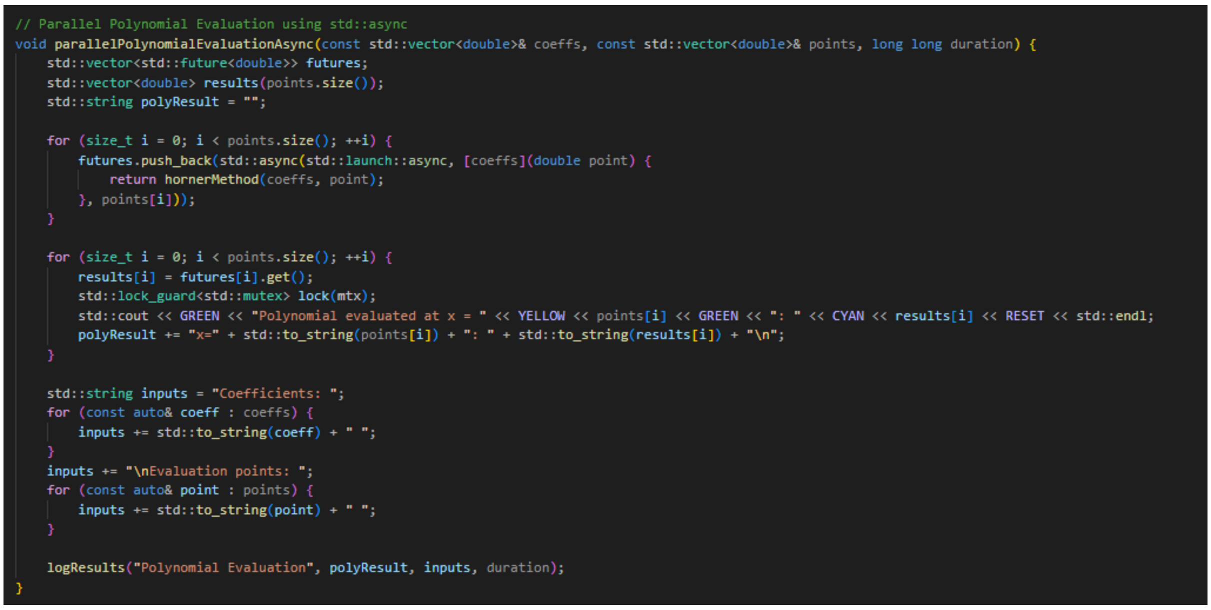
Input:

- coeffs: A vector of polynomial coefficients.
- x: The value at which the polynomial should be evaluated.

Output:

- Returns the computed value of the polynomial at x.

Parallel Polynomial Evaluation Async



```cpp
// Parallel Polynomial Evaluation using std::async
void parallelPolynomialEvaluationAsync(const std::vector<double>& coeffs, const std::vector<double>& points, long long duration) {
    std::vector<std::future<double>> futures;
    std::vector<double> results(points.size());
    std::string polyResult = "";

    for (size_t i = 0; i < points.size(); ++i) {
        futures.push_back(std::async(std::launch::async, [coeffs](double point) {
            return hornerMethod(coeffs, point);
        }, points[i]));
    }

    for (size_t i = 0; i < points.size(); ++i) {
        results[i] = futures[i].get();
        std::lock_guard<std::mutex> lock(mtx);
        std::cout << GREEN << "Polynomial evaluated at x = " << YELLOW << points[i] << GREEN << ": " << CYAN << results[i] << RESET << std::endl;
        polyResult += "x=" + std::to_string(points[i]) + ": " + std::to_string(results[i]) + "\n";
    }

    std::string inputs = "Coefficients: ";
    for (const auto& coeff : coeffs) {
        inputs += std::to_string(coeff) + " ";
    }
    inputs += "\nEvaluation points: ";
    for (const auto& point : points) {
        inputs += std::to_string(point) + " ";
    }

    logResults("Polynomial Evaluation", polyResult, inputs, duration);
}
```

Purpose:

- This function evaluates a polynomial at multiple points concurrently using std::async for parallel execution.

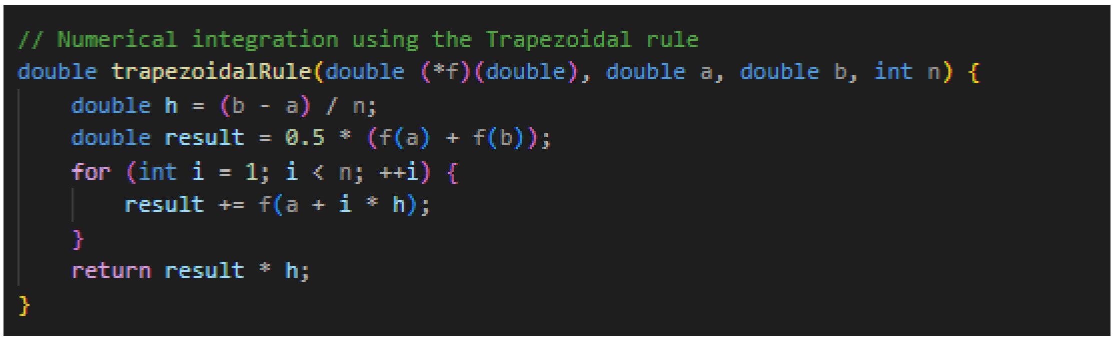
Input:

- coeffs: A vector of polynomial coefficients.
- points: A vector of points at which the polynomial will be evaluated.
- duration: Time taken for the task (used in logging).

Output:

- Displays the evaluated results for each point on the console.
- Logs the results.

Trapezoidal Rule



```cpp
// Numerical integration using the Trapezoidal rule
double trapezoidalRule(double (*f)(double), double a, double b, int n) {
    double h = (b - a) / n;
    double result = 0.5 * (f(a) + f(b));
    for (int i = 1; i < n; ++i) {
        result += f(a + i * h);
    }
    return result * h;
}
```

Purpose:

- Implements the Trapezoidal Rule for numerical integration.

Input:

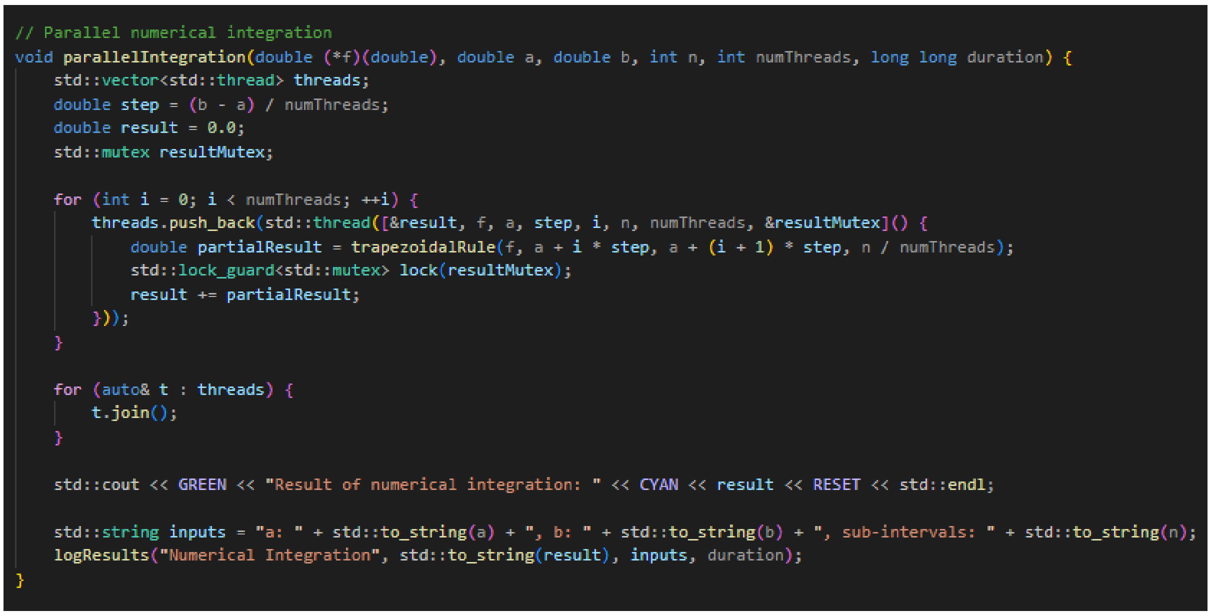
- f: The function to integrate.

- a: Lower bound of integration.
- b: Upper bound of integration.
- n: Number of sub-intervals.

Output:

- Returns the computed value of the integral

Parallel Integration



```cpp
// Parallel numerical integration
void parallelIntegration(double (*f)(double), double a, double b, int n, int numThreads, long long duration) {
    std::vector<std::thread> threads;
    double step = (b - a) / numThreads;
    double result = 0.0;
    std::mutex resultMutex;

    for (int i = 0; i < numThreads; ++i) {
        threads.push_back(std::thread([&result, f, a, step, i, n, numThreads, &resultMutex]() {
            double partialResult = trapezoidalRule(f, a + i * step, a + (i + 1) * step, n / numThreads);
            std::lock_guard<std::mutex> lock(resultMutex);
            result += partialResult;
        }));
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << GREEN << "Result of numerical integration: " << CYAN << result << RESET << std::endl;

    std::string inputs = "a: " + std::to_string(a) + ", b: " + std::to_string(b) + ", sub-intervals: " + std::to_string(n);
    logResults("Numerical Integration", std::to_string(result), inputs, duration);
}
```

Purpose:

- Implements the Trapezoidal Rule for numerical integration.

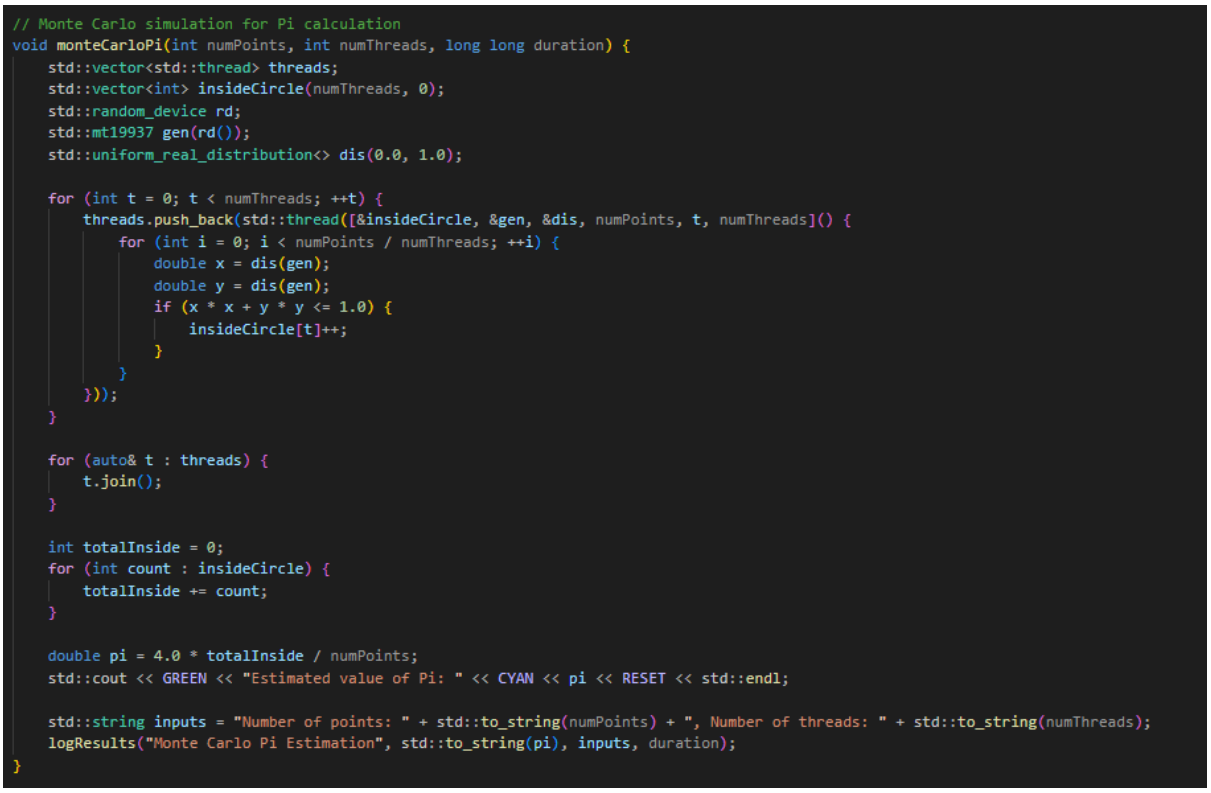
Input:

- The function to integrate.
- Lower bound of integration.
- Upper bound of integration.
- Number of sub-intervals.

Output:

- Returns the computed value of the integral

Montecarlo



```cpp
// Monte Carlo simulation for Pi calculation
void monteCarloPi(int numPoints, int numThreads, long long duration) {
    std::vector<std::thread> threads;
    std::vector<int> insideCircle(numThreads, 0);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);

    for (int t = 0; t < numThreads; ++t) {
        threads.push_back(std::thread([&insideCircle, &gen, &dis, numPoints, t, numThreads]() {
            for (int i = 0; i < numPoints / numThreads; ++i) {
                double x = dis(gen);
                double y = dis(gen);
                if (x * x + y * y <= 1.0) {
                    insideCircle[t]++;
                }
            }
        }));
    }

    for (auto& t : threads) {
        t.join();
    }

    int totalInside = 0;
    for (int count : insideCircle) {
        totalInside += count;
    }

    double pi = 4.0 * totalInside / numPoints;
    std::cout << GREEN << "Estimated value of Pi: " << CYAN << pi << RESET << std::endl;

    std::string inputs = "Number of points: " + std::to_string(numPoints) + ", Number of threads: " + std::to_string(numThreads);
    logResults("Monte Carlo Pi Estimation", std::to_string(pi), inputs, duration);
}
```

Purpose:

- ➢ Estimates the value of Pi using a Monte Carlo simulation. It generates random points inside a unit square and calculates how many fall within the unit circle.

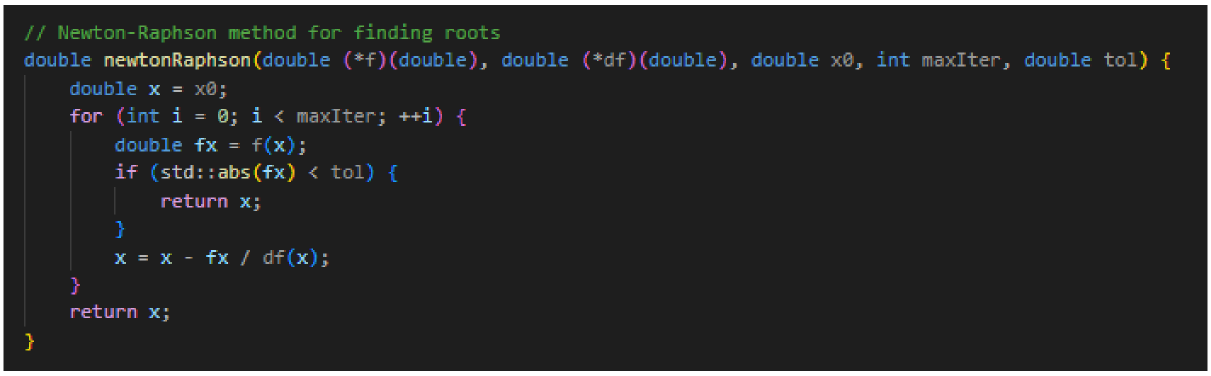
Input:

- ➢ numPoints: Total number of points to simulate.
- ➢ numThreads: Number of threads to distribute the work.
- ➢ duration: Time for the task (for logging).

Output:

- ➢ Displays the estimated value of Pi and logs the result.

Newton Raphson



```cpp
// Newton-Raphson method for finding roots
double newtonRaphson(double (*f)(double), double (*df)(double), double x0, int maxIter, double tol) {
    double x = x0;
    for (int i = 0; i < maxIter; ++i) {
        double fx = f(x);
        if (std::abs(fx) < tol) {
            return x;
        }
        x = x - fx / df(x);
    }
    return x;
}
```

Purpose:

- ➢ Finds the root of a non-linear equation using the Newton-Raphson method.

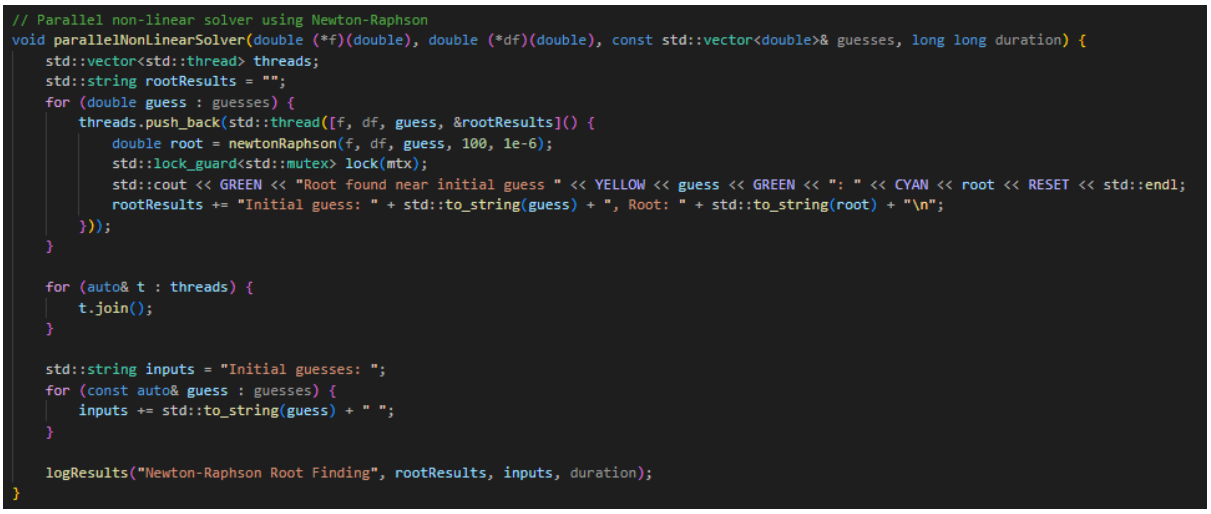
Input:

- ➢ The function whose root is to be found.
- ➢ The derivative of the function f.
- ➢ The initial guess for the root.
- ➢ Maximum number of iterations.
- ➢ Tolerance level for root approximation.

Output:

- ➢ Returns the computed root of the function

Parallel Non-Linear Solver



```cpp
// Parallel non-linear solver using Newton-Raphson
void parallelNonLinearSolver(double (*f)(double), double (*df)(double), const std::vector<double>& guesses, long long duration) {
    std::vector<std::thread> threads;
    std::string rootResults = "";
    for (double guess : guesses) {
        threads.push_back(std::thread([f, df, guess, &rootResults]() {
            double root = newtonRaphson(f, df, guess, 100, 1e-6);
            std::lock_guard<std::mutex> lock(mtx);
            std::cout << GREEN << "Root found near initial guess " << YELLOW << guess << GREEN << ": " << CYAN << root << RESET << std::endl;
            rootResults += "Initial guess: " + std::to_string(guess) + ", Root: " + std::to_string(root) + "\n";
        }));
    }

    for (auto& t : threads) {
        t.join();
    }

    std::string inputs = "Initial guesses: ";
    for (const auto& guess : guesses) {
        inputs += std::to_string(guess) + " ";
    }

    logResults("Newton-Raphson Root Finding", rootResults, inputs, duration);
}
```

Purpose:

- ➢ Solves non-linear equations using the Newton-Raphson method for multiple initial guesses in parallel.

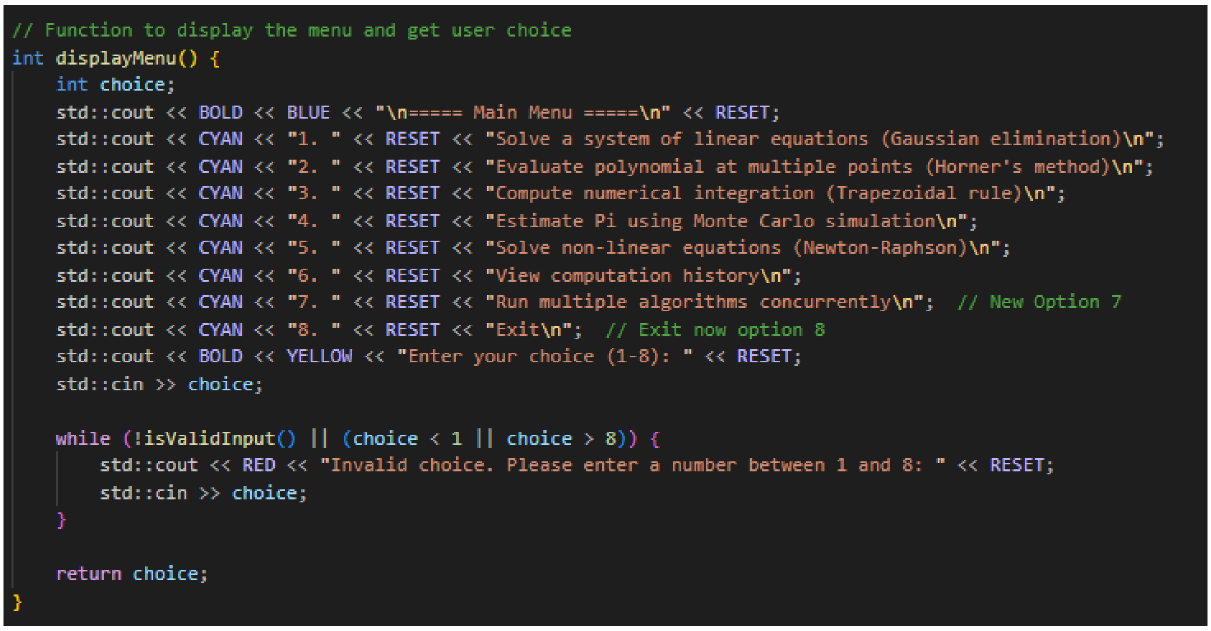
Input:

- ➢ The function whose root is to be found.
- ➢ The derivative of the function f.
- ➢ guesses: A vector of initial guesses.
- ➢ duration: Time for the task (for logging).

Output:

- ➢ Displays the computed root for each guess and logs the results.

Display Menu



```cpp
// Function to display the menu and get user choice
int displayMenu() {
    int choice;
    std::cout << BOLD << BLUE << "\n===== Main Menu =====\n" << RESET;
    std::cout << CYAN << "1. " << RESET << "Solve a system of linear equations (Gaussian elimination)\n";
    std::cout << CYAN << "2. " << RESET << "Evaluate polynomial at multiple points (Horner's method)\n";
    std::cout << CYAN << "3. " << RESET << "Compute numerical integration (Trapezoidal rule)\n";
    std::cout << CYAN << "4. " << RESET << "Estimate Pi using Monte Carlo simulation\n";
    std::cout << CYAN << "5. " << RESET << "Solve non-linear equations (Newton-Raphson)\n";
    std::cout << CYAN << "6. " << RESET << "View computation history\n";
    std::cout << CYAN << "7. " << RESET << "Run multiple algorithms concurrently\n";  // New Option 7
    std::cout << CYAN << "8. " << RESET << "Exit\n";  // Exit now option 8
    std::cout << BOLD << YELLOW << "Enter your choice (1-8): " << RESET;
    std::cin >> choice;

    while (!isValidInput() || (choice < 1 || choice > 8)) {
        std::cout << RED << "Invalid choice. Please enter a number between 1 and 8: " << RESET;
        std::cin >> choice;
    }

    return choice;
}
```

Purpose:

- Displays the main menu and prompts the user to select a task.

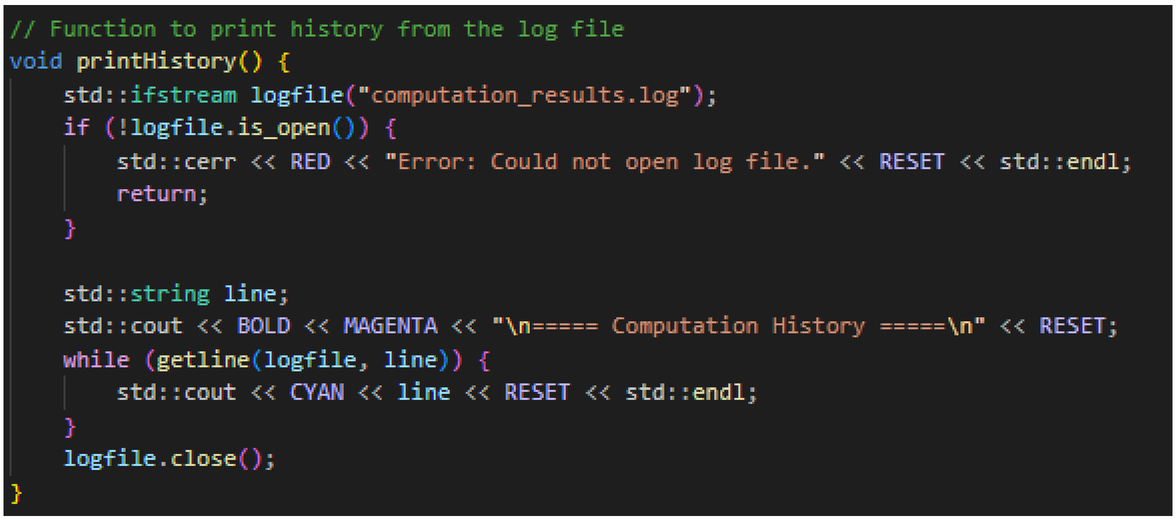
Input:

- None (except user input).

Output:

- Returns the user's choice.

Print History



```cpp
// Function to print history from the log file
void printHistory() {
    std::ifstream logfile("computation_results.log");
    if (!logfile.is_open()) {
        std::cerr << RED << "Error: Could not open log file." << RESET << std::endl;
        return;
    }

    std::string line;
    std::cout << BOLD << MAGENTA << "\n===== Computation History =====\n" << RESET;
    while (getline(logfile, line)) {
        std::cout << CYAN << line << RESET << std::endl;
    }
    logfile.close();
}
```

Purpose:

- Prints the computation history stored in computation_results.log.

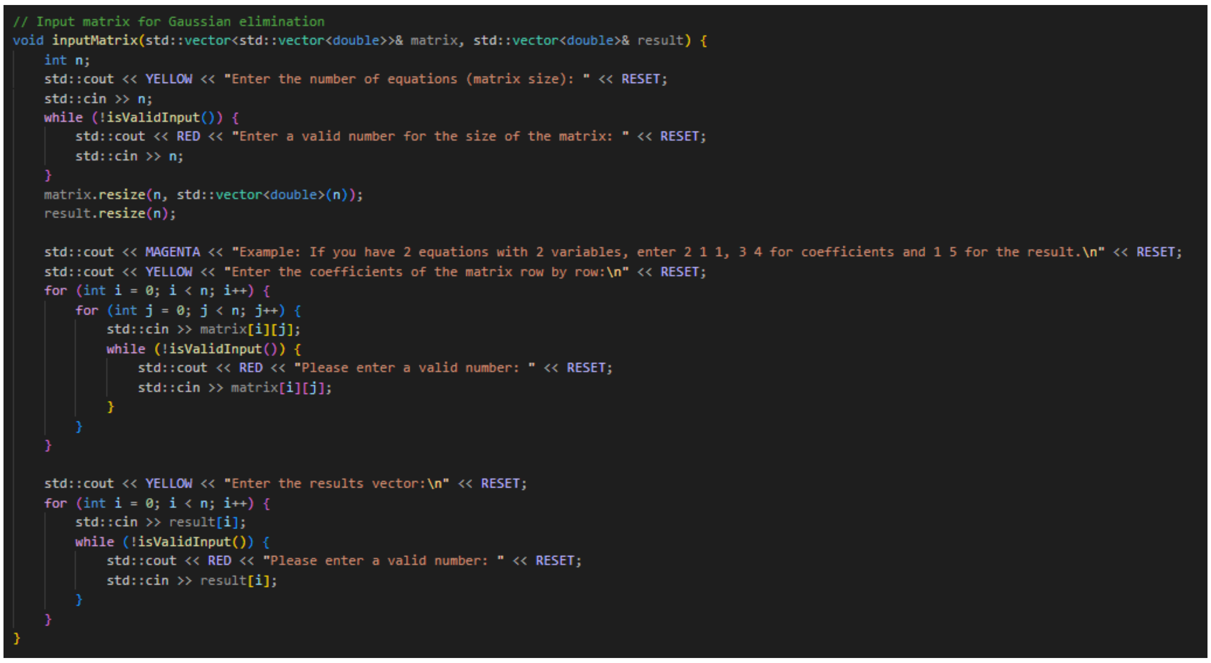
Input:

- ➢ None

Output:

- ➢ Displays the computation history on the console.

Input Matrix



```cpp
// Input matrix for Gaussian elimination
void inputMatrix(std::vector<std::vector<double>>& matrix, std::vector<double>& result) {
    int n;
    std::cout << YELLOW << "Enter the number of equations (matrix size): " << RESET;
    std::cin >> n;
    while (!isValidInput()) {
        std::cout << RED << "Enter a valid number for the size of the matrix: " << RESET;
        std::cin >> n;
    }
    matrix.resize(n, std::vector<double>(n));
    result.resize(n);

    std::cout << MAGENTA << "Example: If you have 2 equations with 2 variables, enter 2 1 1, 3 4 for coefficients and 1 5 for the result.\n" << RESET;
    std::cout << YELLOW << "Enter the coefficients of the matrix row by row:\n" << RESET;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            std::cin >> matrix[i][j];
            while (!isValidInput()) {
                std::cout << RED << "Please enter a valid number: " << RESET;
                std::cin >> matrix[i][j];
            }
        }
    }

    std::cout << YELLOW << "Enter the results vector:\n" << RESET;
    for (int i = 0; i < n; i++) {
        std::cin >> result[i];
        while (!isValidInput()) {
            std::cout << RED << "Please enter a valid number: " << RESET;
            std::cin >> result[i];
        }
    }
}
```

Purpose:

- ➢ This function takes input from the user to form a matrix and a result vector, which are used in Gaussian Elimination.

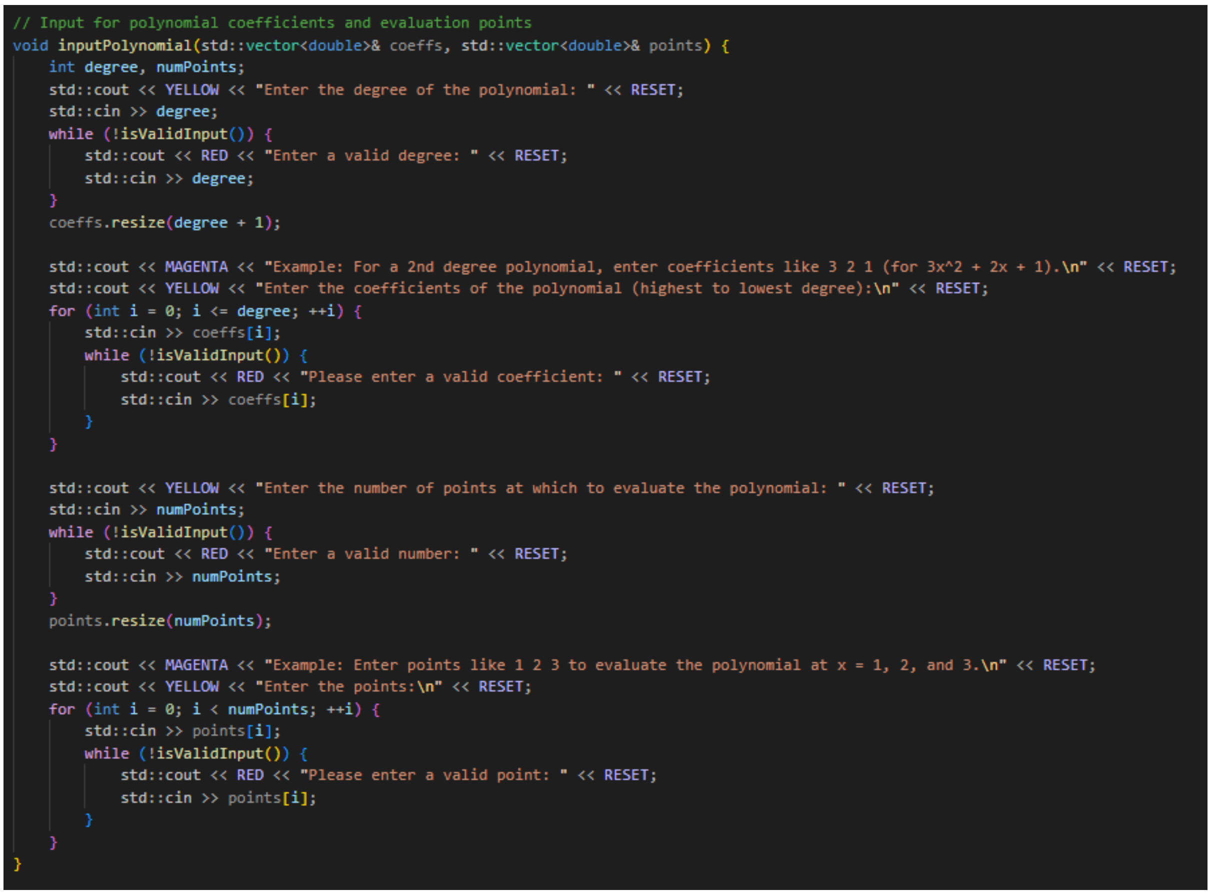
Input:

- ➢ matrix: A reference to a 2D vector that will store the coefficients of the linear system.
- ➢ result: A reference to a vector that will store the result of the equations.

Output:

- ➢ Populates the matrix and result with user input.

Input Polynomial



```cpp
// Input for polynomial coefficients and evaluation points
void inputPolynomial(std::vector<double>& coeffs, std::vector<double>& points) {
    int degree, numPoints;
    std::cout << YELLOW << "Enter the degree of the polynomial: " << RESET;
    std::cin >> degree;
    while (!isValidInput()) {
        std::cout << RED << "Enter a valid degree: " << RESET;
        std::cin >> degree;
    }
    coeffs.resize(degree + 1);

    std::cout << MAGENTA << "Example: For a 2nd degree polynomial, enter coefficients like 3 2 1 (for 3x^2 + 2x + 1).\n" << RESET;
    std::cout << YELLOW << "Enter the coefficients of the polynomial (highest to lowest degree):\n" << RESET;
    for (int i = 0; i <= degree; ++i) {
        std::cin >> coeffs[i];
        while (!isValidInput()) {
            std::cout << RED << "Please enter a valid coefficient: " << RESET;
            std::cin >> coeffs[i];
        }
    }

    std::cout << YELLOW << "Enter the number of points at which to evaluate the polynomial: " << RESET;
    std::cin >> numPoints;
    while (!isValidInput()) {
        std::cout << RED << "Enter a valid number: " << RESET;
        std::cin >> numPoints;
    }
    points.resize(numPoints);

    std::cout << MAGENTA << "Example: Enter points like 1 2 3 to evaluate the polynomial at x = 1, 2, and 3.\n" << RESET;
    std::cout << YELLOW << "Enter the points:\n" << RESET;
    for (int i = 0; i < numPoints; ++i) {
        std::cin >> points[i];
        while (!isValidInput()) {
            std::cout << RED << "Please enter a valid point: " << RESET;
            std::cin >> points[i];
        }
    }
}
```

Purpose:

- This function collects input for polynomial evaluation, including the polynomial's coefficients and the points at which the polynomial will be evaluated.

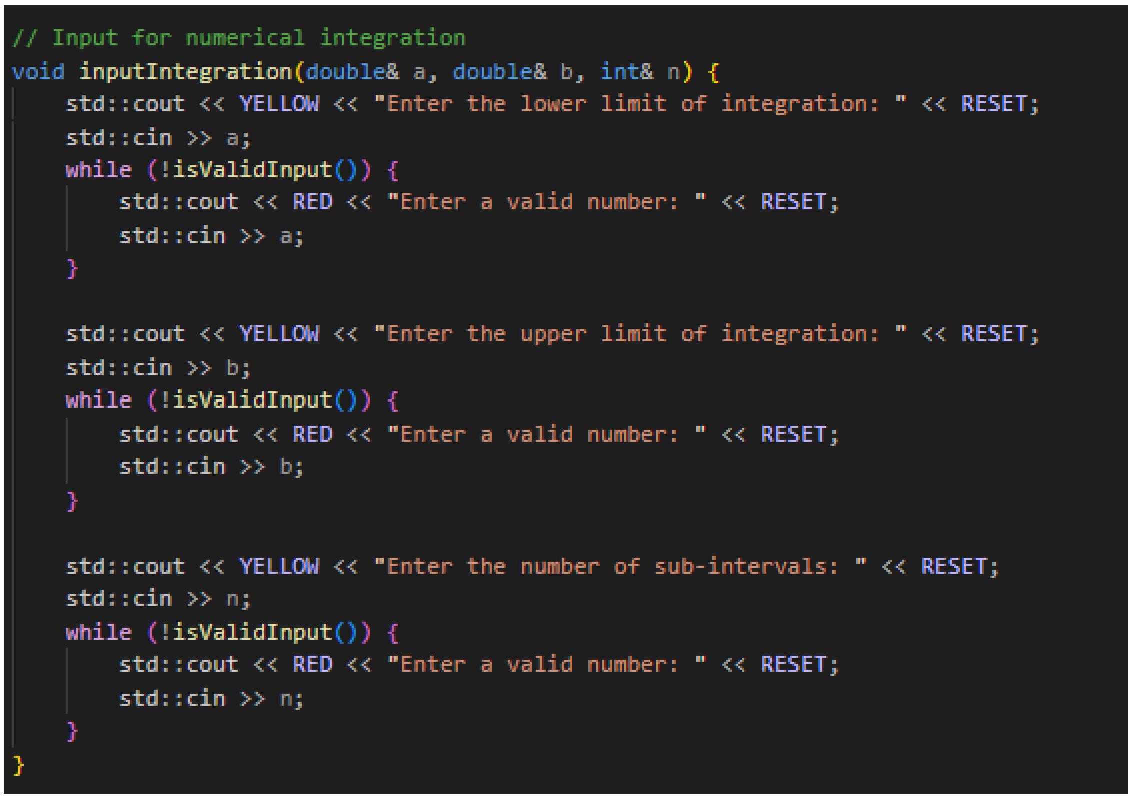
Input:

- coeffs: A reference to a vector that will store the polynomial coefficients.
- points: A reference to a vector that will store the points where the polynomial will be evaluated.

Output:

- Populates the coeffs and points vectors with user input

Input Integration



```cpp
// Input for numerical integration
void inputIntegration(double& a, double& b, int& n) {
    std::cout << YELLOW << "Enter the lower limit of integration: " << RESET;
    std::cin >> a;
    while (!isValidInput()) {
        std::cout << RED << "Enter a valid number: " << RESET;
        std::cin >> a;
    }

    std::cout << YELLOW << "Enter the upper limit of integration: " << RESET;
    std::cin >> b;
    while (!isValidInput()) {
        std::cout << RED << "Enter a valid number: " << RESET;
        std::cin >> b;
    }

    std::cout << YELLOW << "Enter the number of sub-intervals: " << RESET;
    std::cin >> n;
    while (!isValidInput()) {
        std::cout << RED << "Enter a valid number: " << RESET;
        std::cin >> n;
    }
}
```

Purpose:

- This function collects input for numerical integration, including the integration limits (a, b) and the number of sub-intervals (n).

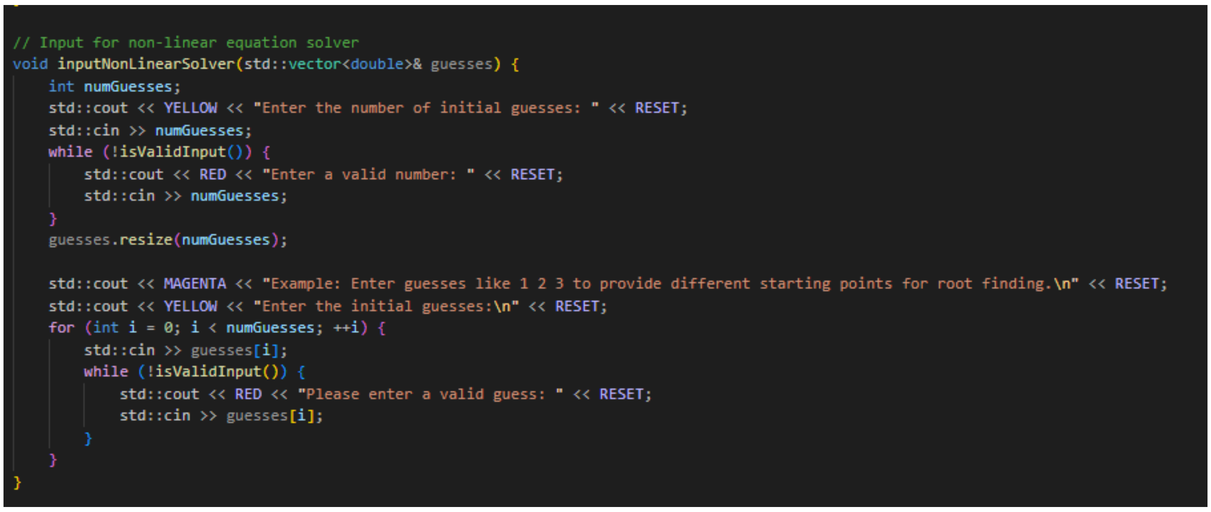
Input:

- A reference to a double that stores the lower limit of integration.
- A reference to a double that stores the upper limit of integration.
- A reference to an integer that stores the number of sub-intervals.

Output:

- Populates the values of a, b, and n with user input.

Input Non-Linear Solver



```cpp
// Input for non-linear equation solver
void inputNonLinearSolver(std::vector<double>& guesses) {
    int numGuesses;
    std::cout << YELLOW << "Enter the number of initial guesses: " << RESET;
    std::cin >> numGuesses;
    while (!isValidInput()) {
        std::cout << RED << "Enter a valid number: " << RESET;
        std::cin >> numGuesses;
    }
    guesses.resize(numGuesses);

    std::cout << MAGENTA << "Example: Enter guesses like 1 2 3 to provide different starting points for root finding.\n" << RESET;
    std::cout << YELLOW << "Enter the initial guesses:\n" << RESET;
    for (int i = 0; i < numGuesses; ++i) {
        std::cin >> guesses[i];
        while (!isValidInput()) {
            std::cout << RED << "Please enter a valid guess: " << RESET;
            std::cin >> guesses[i];
        }
    }
}
```

Purpose:

- ➢ This function collects initial guesses from the user for the Newton-Raphson root-finding algorithm.

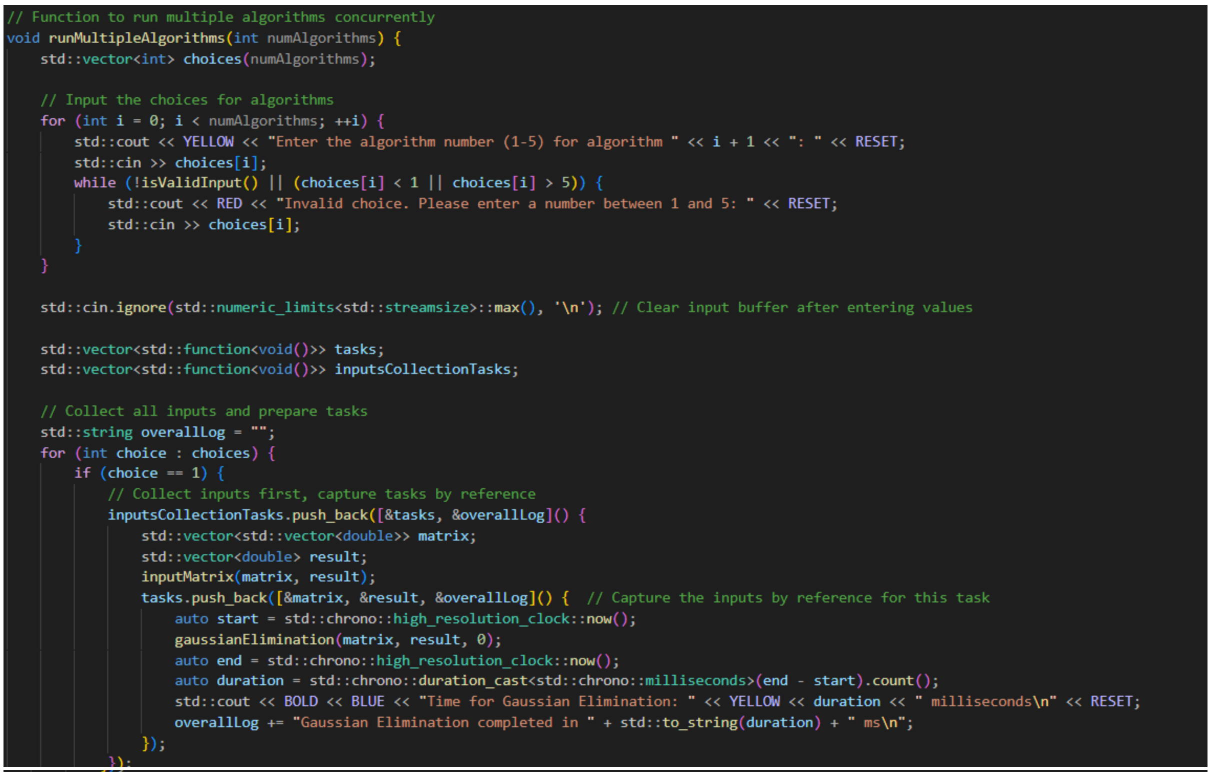
Input:

- ➢ guesses: A reference to a vector that will store the initial guesses.

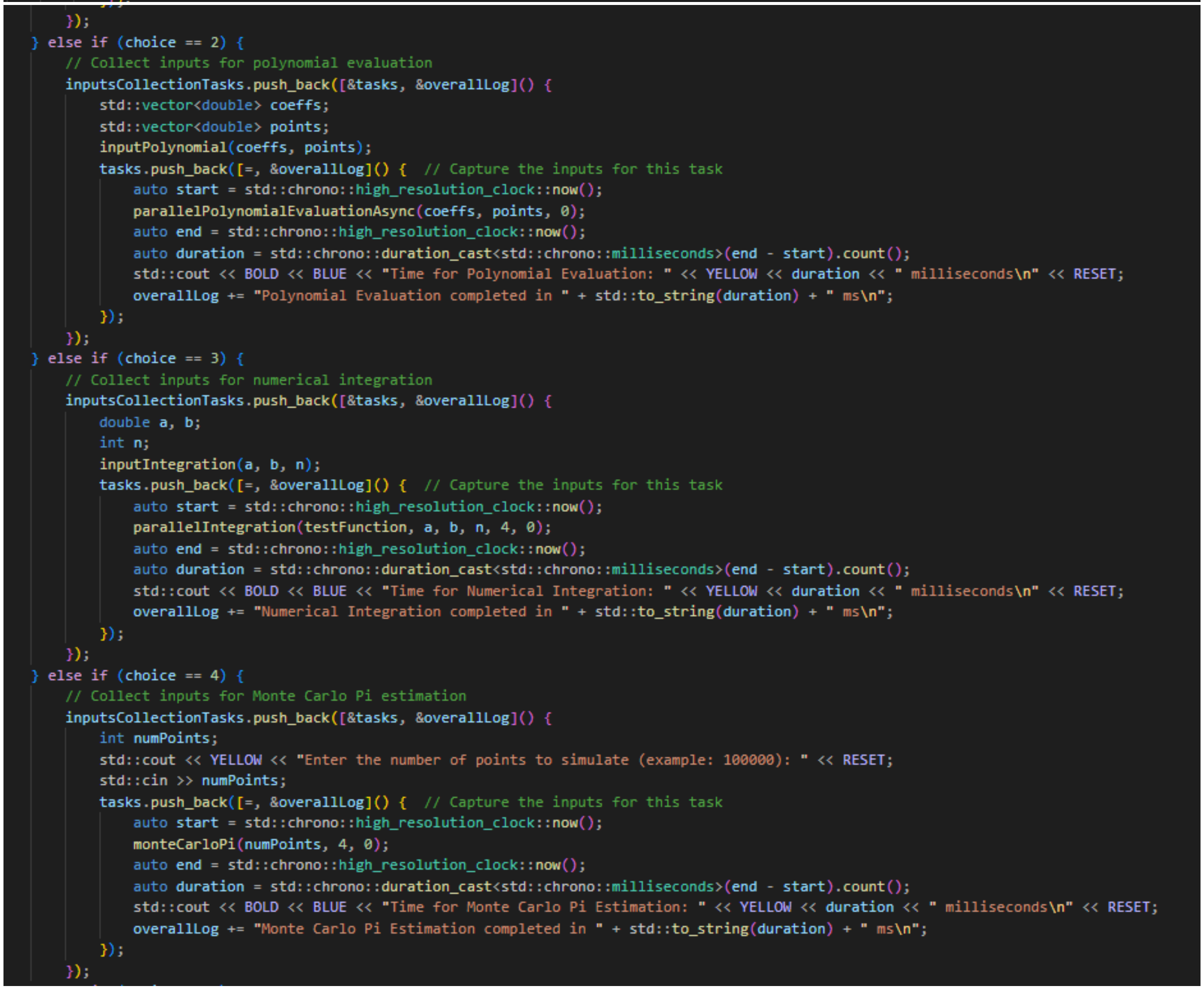
Output:

- ➢ Populates the guesses vector with user input.

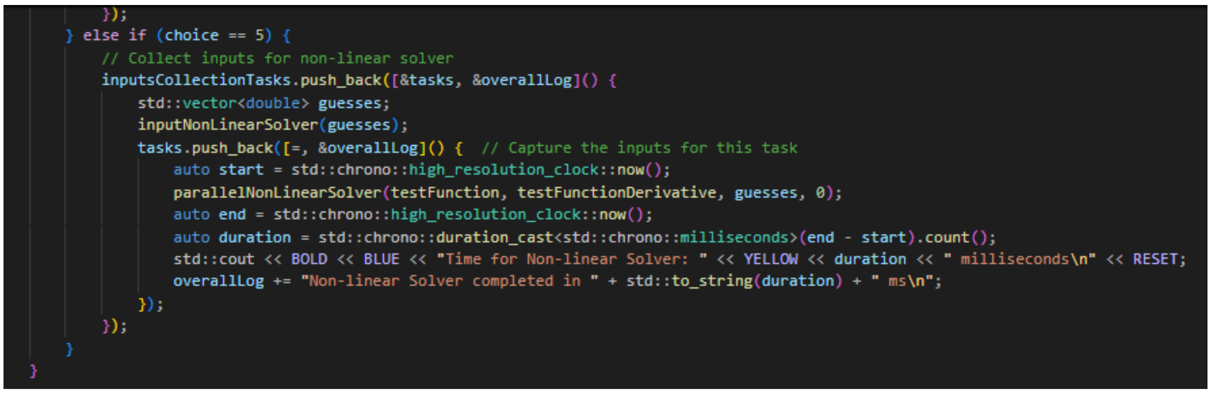
Run Multiple Algorithms



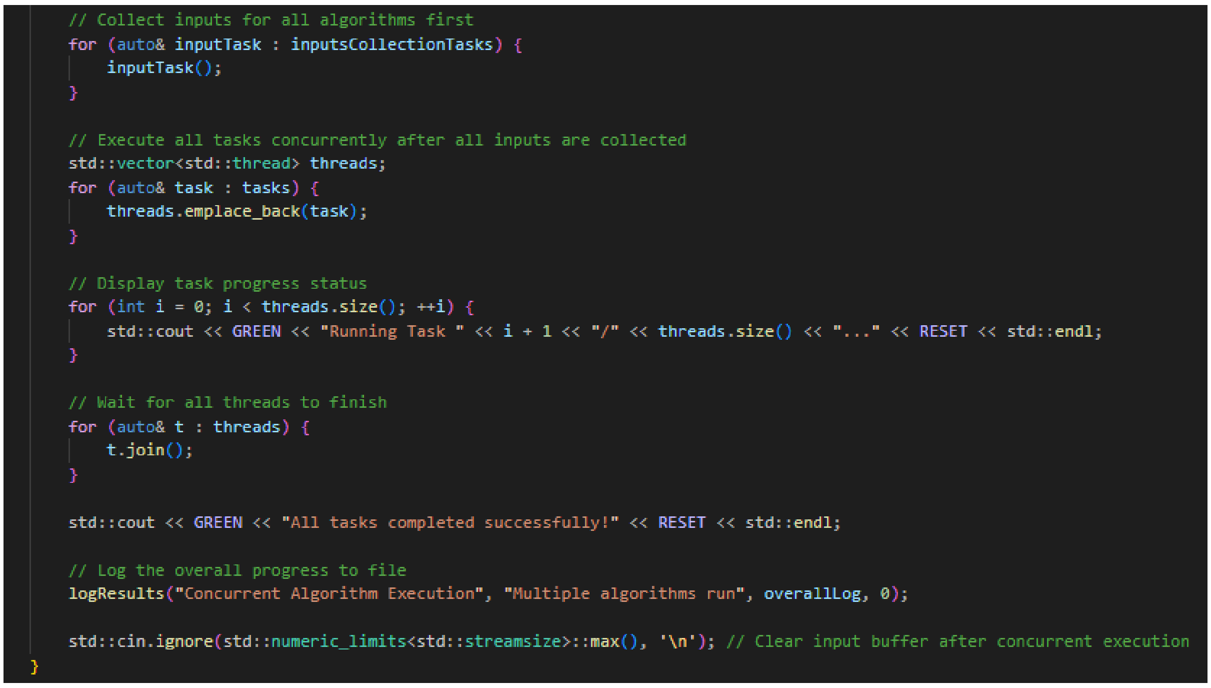
```cpp
// Function to run multiple algorithms concurrently
void runMultipleAlgorithms(int numAlgorithms) {
    std::vector<int> choices(numAlgorithms);

    // Input the choices for algorithms
    for (int i = 0; i < numAlgorithms; ++i) {
        std::cout << YELLOW << "Enter the algorithm number (1-5) for algorithm " << i + 1 << ": " << RESET;
        std::cin >> choices[i];
        while (!isValidInput() || (choices[i] < 1 || choices[i] > 5)) {
            std::cout << RED << "Invalid choice. Please enter a number between 1 and 5: " << RESET;
            std::cin >> choices[i];
        }
    }

    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Clear input buffer after entering values

    std::vector<std::function<void()>> tasks;
    std::vector<std::function<void()>> inputsCollectionTasks;

    // Collect all inputs and prepare tasks
    std::string overallLog = "";
    for (int choice : choices) {
        if (choice == 1) {
            // Collect inputs first, capture tasks by reference
            inputsCollectionTasks.push_back([&tasks, &overallLog]() {
                std::vector<std::vector<double>> matrix;
                std::vector<double> result;
                inputMatrix(matrix, result);
                tasks.push_back([&matrix, &result, &overallLog]() {  // Capture the inputs by reference for this task
                    auto start = std::chrono::high_resolution_clock::now();
                    gaussianElimination(matrix, result, 0);
                    auto end = std::chrono::high_resolution_clock::now();
                    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
                    std::cout << BOLD << BLUE << "Time for Gaussian Elimination: " << YELLOW << duration << " milliseconds\n" << RESET;
                    overallLog += "Gaussian Elimination completed in " + std::to_string(duration) + " ms\n";
                });
            });
```



```cpp
            });
        } else if (choice == 2) {
            // Collect inputs for polynomial evaluation
            inputsCollectionTasks.push_back([&tasks, &overallLog]() {
                std::vector<double> coeffs;
                std::vector<double> points;
                inputPolynomial(coeffs, points);
                tasks.push_back([=, &overallLog]() {  // Capture the inputs for this task
                    auto start = std::chrono::high_resolution_clock::now();
                    parallelPolynomialEvaluationAsync(coeffs, points, 0);
                    auto end = std::chrono::high_resolution_clock::now();
                    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
                    std::cout << BOLD << BLUE << "Time for Polynomial Evaluation: " << YELLOW << duration << " milliseconds\n" << RESET;
                    overallLog += "Polynomial Evaluation completed in " + std::to_string(duration) + " ms\n";
                });
            });
        } else if (choice == 3) {
            // Collect inputs for numerical integration
            inputsCollectionTasks.push_back([&tasks, &overallLog]() {
                double a, b;
                int n;
                inputIntegration(a, b, n);
                tasks.push_back([=, &overallLog]() {  // Capture the inputs for this task
                    auto start = std::chrono::high_resolution_clock::now();
                    parallelIntegration(testFunction, a, b, n, 4, 0);
                    auto end = std::chrono::high_resolution_clock::now();
                    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
                    std::cout << BOLD << BLUE << "Time for Numerical Integration: " << YELLOW << duration << " milliseconds\n" << RESET;
                    overallLog += "Numerical Integration completed in " + std::to_string(duration) + " ms\n";
                });
            });
        } else if (choice == 4) {
            // Collect inputs for Monte Carlo Pi estimation
            inputsCollectionTasks.push_back([&tasks, &overallLog]() {
                int numPoints;
                std::cout << YELLOW << "Enter the number of points to simulate (example: 100000): " << RESET;
                std::cin >> numPoints;
                tasks.push_back([=, &overallLog]() {  // Capture the inputs for this task
                    auto start = std::chrono::high_resolution_clock::now();
                    monteCarloPi(numPoints, 4, 0);
                    auto end = std::chrono::high_resolution_clock::now();
                    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
                    std::cout << BOLD << BLUE << "Time for Monte Carlo Pi Estimation: " << YELLOW << duration << " milliseconds\n" << RESET;
                    overallLog += "Monte Carlo Pi Estimation completed in " + std::to_string(duration) + " ms\n";
                });
            });
```



```cpp
        });
    } else if (choice == 5) {
        // Collect inputs for non-linear solver
        inputsCollectionTasks.push_back([&tasks, &overallLog]() {
            std::vector<double> guesses;
            inputNonLinearSolver(guesses);
            tasks.push_back([=, &overallLog]() {  // Capture the inputs for this task
                auto start = std::chrono::high_resolution_clock::now();
                parallelNonLinearSolver(testFunction, testFunctionDerivative, guesses, 0);
                auto end = std::chrono::high_resolution_clock::now();
                auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
                std::cout << BOLD << BLUE << "Time for Non-linear Solver: " << YELLOW << duration << " milliseconds\n" << RESET;
                overallLog += "Non-linear Solver completed in " + std::to_string(duration) + " ms\n";
            });
        });
    }
}
```



```cpp
    // Collect inputs for all algorithms first
    for (auto& inputTask : inputsCollectionTasks) {
        inputTask();
    }

    // Execute all tasks concurrently after all inputs are collected
    std::vector<std::thread> threads;
    for (auto& task : tasks) {
        threads.emplace_back(task);
    }

    // Display task progress status
    for (int i = 0; i < threads.size(); ++i) {
        std::cout << GREEN << "Running Task " << i + 1 << "/" << threads.size() << "..." << RESET << std::endl;
    }

    // Wait for all threads to finish
    for (auto& t : threads) {
        t.join();
    }

    std::cout << GREEN << "All tasks completed successfully!" << RESET << std::endl;

    // Log the overall progress to file
    logResults("Concurrent Algorithm Execution", "Multiple algorithms run", overallLog, 0);

    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Clear input buffer after concurrent execution
}
```

Purpose:

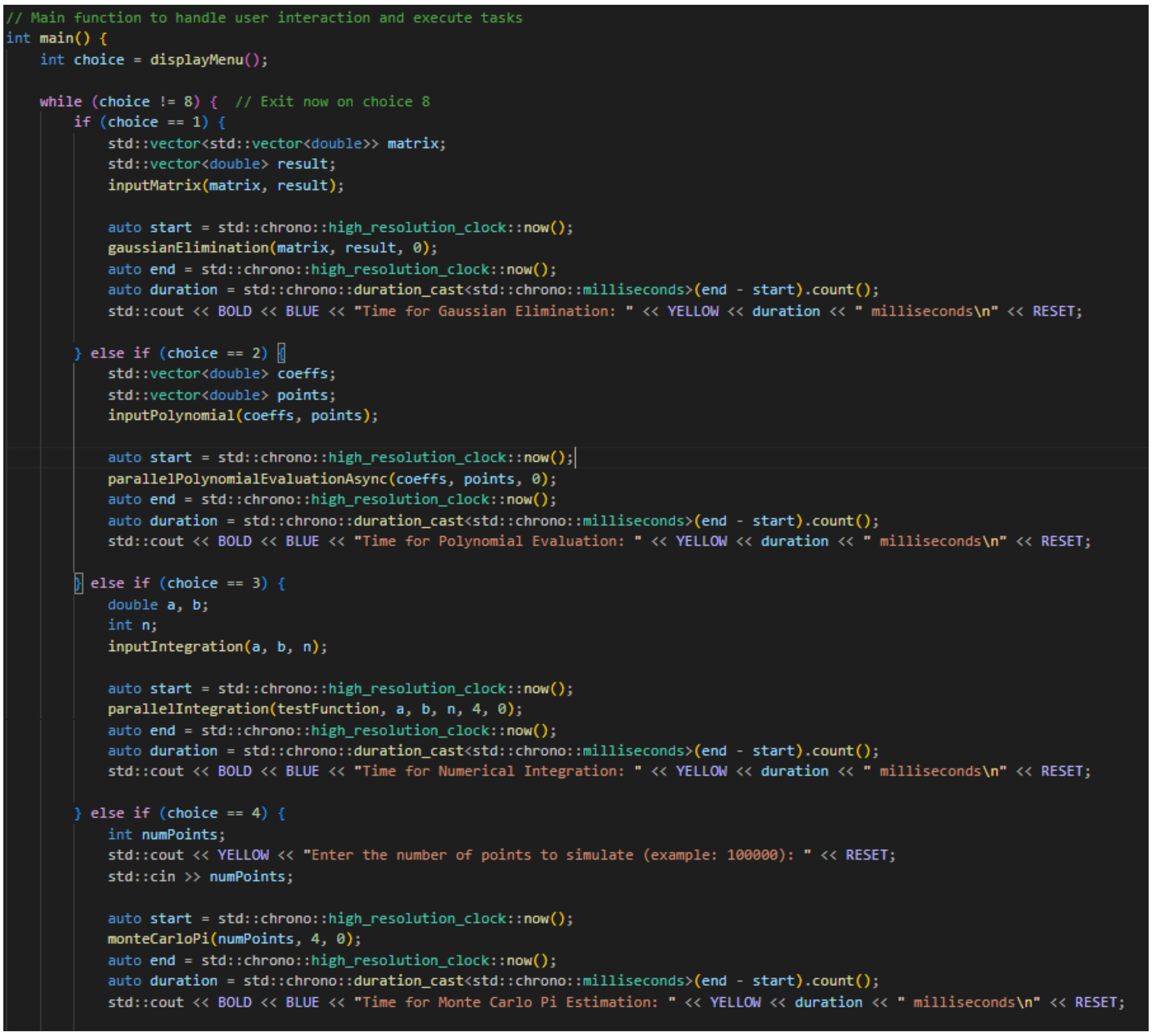
- This function allows the user to run multiple algorithms concurrently by specifying the algorithm choices and collecting necessary inputs for each task.

Input:

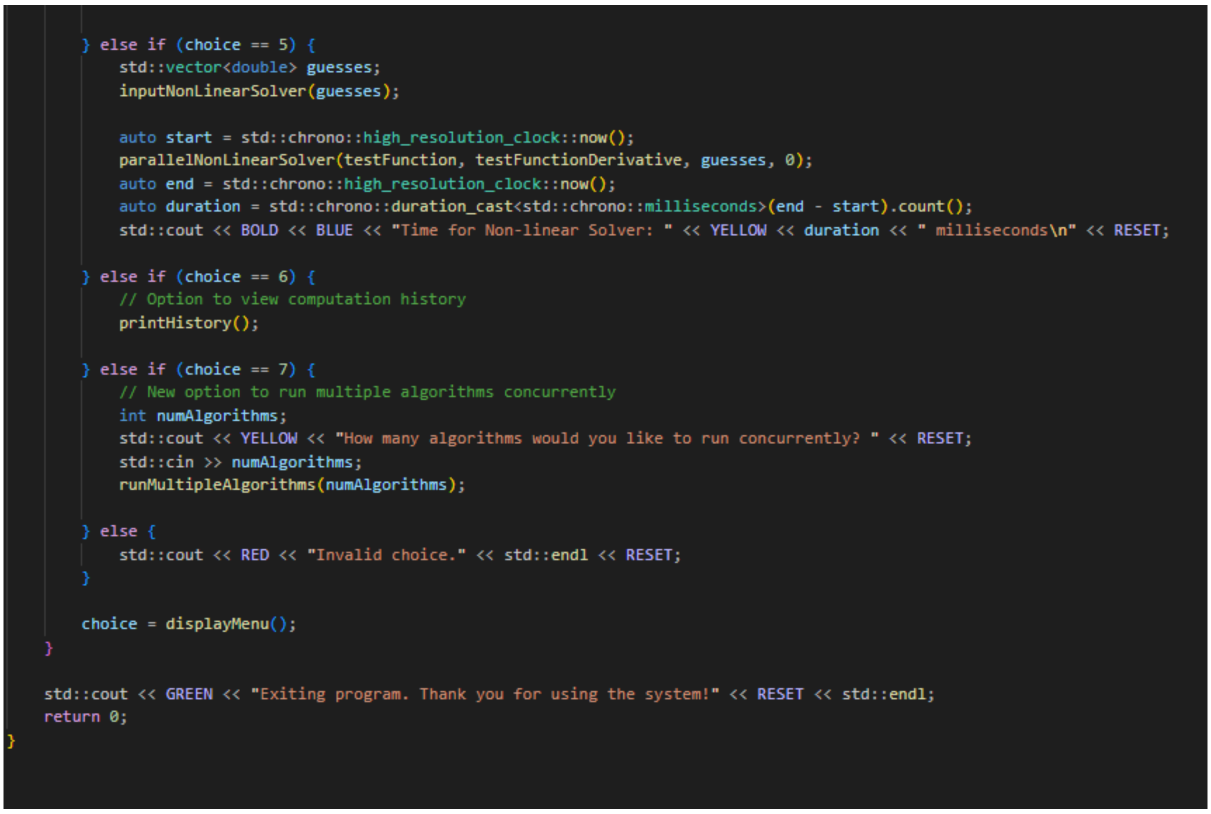
- numAlgorithms: The number of algorithms to run concurrently.

Output:

- Executes the specified algorithms concurrently and logs the overall result.



```cpp
// Main function to handle user interaction and execute tasks
int main() {
    int choice = displayMenu();

    while (choice != 8) {  // Exit now on choice 8
        if (choice == 1) {
            std::vector<std::vector<double>> matrix;
            std::vector<double> result;
            inputMatrix(matrix, result);

            auto start = std::chrono::high_resolution_clock::now();
            gaussianElimination(matrix, result, 0);
            auto end = std::chrono::high_resolution_clock::now();
            auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
            std::cout << BOLD << BLUE << "Time for Gaussian Elimination: " << YELLOW << duration << " milliseconds\n" << RESET;

        } else if (choice == 2) {
            std::vector<double> coeffs;
            std::vector<double> points;
            inputPolynomial(coeffs, points);

            auto start = std::chrono::high_resolution_clock::now();
            parallelPolynomialEvaluationAsync(coeffs, points, 0);
            auto end = std::chrono::high_resolution_clock::now();
            auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
            std::cout << BOLD << BLUE << "Time for Polynomial Evaluation: " << YELLOW << duration << " milliseconds\n" << RESET;

        } else if (choice == 3) {
            double a, b;
            int n;
            inputIntegration(a, b, n);

            auto start = std::chrono::high_resolution_clock::now();
            parallelIntegration(testFunction, a, b, n, 4, 0);
            auto end = std::chrono::high_resolution_clock::now();
            auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
            std::cout << BOLD << BLUE << "Time for Numerical Integration: " << YELLOW << duration << " milliseconds\n" << RESET;

        } else if (choice == 4) {
            int numPoints;
            std::cout << YELLOW << "Enter the number of points to simulate (example: 100000): " << RESET;
            std::cin >> numPoints;

            auto start = std::chrono::high_resolution_clock::now();
            monteCarloPi(numPoints, 4, 0);
            auto end = std::chrono::high_resolution_clock::now();
            auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
            std::cout << BOLD << BLUE << "Time for Monte Carlo Pi Estimation: " << YELLOW << duration << " milliseconds\n" << RESET;
```



```cpp
        } else if (choice == 5) {
            std::vector<double> guesses;
            inputNonLinearSolver(guesses);

            auto start = std::chrono::high_resolution_clock::now();
            parallelNonLinearSolver(testFunction, testFunctionDerivative, guesses, 0);
            auto end = std::chrono::high_resolution_clock::now();
            auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
            std::cout << BOLD << BLUE << "Time for Non-linear Solver: " << YELLOW << duration << " milliseconds\n" << RESET;

        } else if (choice == 6) {
            // Option to view computation history
            printHistory();

        } else if (choice == 7) {
            // New option to run multiple algorithms concurrently
            int numAlgorithms;
            std::cout << YELLOW << "How many algorithms would you like to run concurrently? " << RESET;
            std::cin >> numAlgorithms;
            runMultipleAlgorithms(numAlgorithms);

        } else {
            std::cout << RED << "Invalid choice." << std::endl << RESET;
        }

        choice = displayMenu();
    }

    std::cout << GREEN << "Exiting program. Thank you for using the system!" << RESET << std::endl;
    return 0;
}
```

Purpose:

- The main function serves as the entry point of the program and provides a loop for interacting with the user. It presents a menu, processes user input, executes the selected mathematical task, and finally exits the program upon user request.

Input:

- None directly, but indirectly handles user input through the menu and calls to other functions that prompt the user.

Output:

- Executes user-selected tasks (Gaussian Elimination, Polynomial Evaluation, Numerical Integration, etc.), displays results, logs computation history, and exits the program

Brief Explanation of the code

The provided C++ code implements a Parallel and Distributed Numerical Computation System, designed to solve a variety of mathematical problems efficiently using parallel computing techniques. The program offers functionality for solving linear equations using Gaussian Elimination, evaluating polynomials through Horner’s Method, performing numerical integration using the Trapezoidal Rule, estimating the value of Pi via Monte Carlo simulation, and solving non-linear equations using the Newton-Raphson method. Each of these tasks can be executed in parallel, utilizing threads to speed up computation and ensure scalability.

The program's design is centred around a user-friendly menu interface, where users can select from different computational tasks. Input validation is handled carefully using the isValidInput() function, which ensures that erroneous user inputs are managed gracefully. Each task logs its results, including execution time, to a log file, providing an audit trail for computations.

Parallelism is a key aspect of this system, particularly in the functions for polynomial evaluation, numerical integration, and Monte Carlo simulation. These computations are distributed across multiple threads or executed asynchronously using std::async and std::thread constructs, ensuring that large-scale problems can be processed efficiently on multi-core processors. Additionally, the program allows users to run multiple algorithms concurrently, further demonstrating its capacity to handle parallel workloads.

Overall, this code reflects an advanced understanding of parallel programming concepts in C++, employing multithreading, mutex locks for thread safety, and the use of various mathematical algorithms, making it a powerful tool for computational problem-solving.

High-Level Architectural Overview

1. Input Layer

This layer handles user inputs through:

- ***Input Functions:*** Functions like inputMatrix(), inputPolynomial(), inputIntegration(), and inputNonLinearSolver() gather the necessary data for each computational task. The user provides inputs such as matrix values, polynomial coefficients, and initial guesses.
- ***Validation:*** The function isValidInput() is used in all input functions to ensure that the user provides valid data, and prompts them to re-enter if necessary.

2. Core Computation Layer

This is the core layer of the architecture where the numerical computations take place. Each function implements a specific mathematical operation:

- ***gaussianElimination():*** Solves systems of linear equations using the Gaussian elimination method.
- ***parallelPolynomialEvaluationAsync():*** Evaluates polynomials at multiple points using Horner's method in parallel.
- ***parallelIntegration():*** Calculates the numerical integration using the Trapezoidal rule with multithreading.
- ***monteCarloPi():*** Estimates the value of Pi using a Monte Carlo simulation with multithreading.
- ***parallelNonLinearSolver():*** Solves non-linear equations using the Newton-Raphson method, with each initial guess processed in parallel.

3. Parallelism Layer

This layer manages the parallel execution of computational tasks, distributing workload across threads:

- ***Multithreading:*** The program leverages std::thread and std::async to execute different parts of the calculations in parallel (e.g., parallelPolynomialEvaluationAsync(), parallelIntegration(), monteCarloPi()).
- ***Concurrency Management:*** Mutexes (std::mutex) and locks (std::lock_guard) are used to ensure thread safety when multiple threads write to shared resources (like logs).

4. Logging and Reporting Layer

This layer is responsible for recording computation results:

- ***logResults():*** Logs the results of each computational task, along with its inputs and execution time, into a log file named computation_results.log.
- ***printHistory():*** Allows users to view a history of all previous computations from the log file.

5. User Interface Layer

The user interacts with the program through a terminal-based interface:

- ***Menu System:*** The function displayMenu() provides a menu with different mathematical tasks. Based on the user's selection, it triggers the appropriate functions.
- ***Main Function:*** The main() function handles the overall program flow, including calling input functions, executing computational tasks, and managing the interaction between layers.

6. Task Management Layer

This layer controls task execution:

- ***runMultipleAlgorithms():*** Manages the execution of multiple computational tasks concurrently, allowing the user to run different algorithms in parallel.

Architectural Diagram Description

Input Layer:

- Functions: inputMatrix(), inputPolynomial(), inputIntegration(), inputNonLinearSolver()
- Connected to: Core Computation Layer

Core Computation Layer:

- Functions: gaussianElimination(), parallelPolynomialEvaluationAsync(), parallelIntegration(), monteCarloPi(), parallelNonLinearSolver()
- Connected to: Parallelism Layer, Logging and Reporting Layer

Parallelism Layer:

- Mechanisms: std::thread, std::async, std::mutex
- Handles: Threading, concurrency management

Logging and Reporting Layer:

- Functions: logResults(), printHistory()
- Connected to: Core Computation Layer

User Interface Layer:

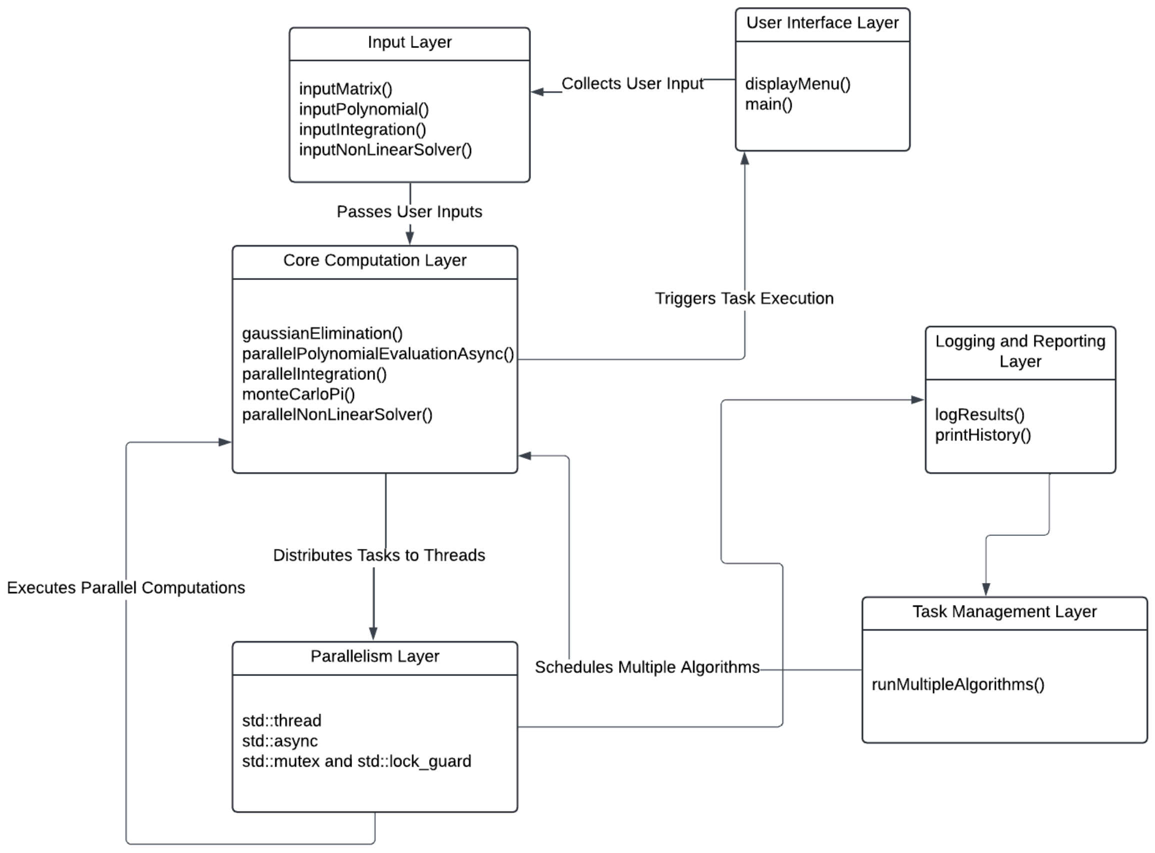
- Functions: displayMenu(), main()
- Connected to: Core Computation Layer

Task Management Layer:

- ***Function:*** runMultipleAlgorithms()
- ***Connected to:*** Core Computation Layer, Parallelism Layer

Architectural Flow

- User selects an option from the menu (handled by displayMenu() and main()).
- Input functions gather necessary data, validate it, and pass it to the core computation functions.
- Core computation functions perform the desired mathematical tasks (Gaussian elimination, polynomial evaluation, integration, etc.).
- Parallelism is utilized where possible to speed up computations using threads or asynchronous operations.
- Logging stores the results of each computation, along with the time taken, in a log file.
- If selected, the user can view past computations from the log file.
- Concurrent execution of multiple algorithms is managed by runMultipleAlgorithms().



User Experience

The system provides a straightforward and intuitive user experience, primarily through a terminal-based interface that guides users through each step of the process. Upon launching the program, users are presented with a menu of options, clearly labeled to indicate the various mathematical algorithms that can be executed. The interface includes choices like solving a system of linear equations using Gaussian Elimination, evaluating polynomials at multiple points, performing numerical integration with the Trapezoidal Rule, estimating Pi using Monte Carlo simulations, or solving non-linear equations using the Newton-Raphson method.

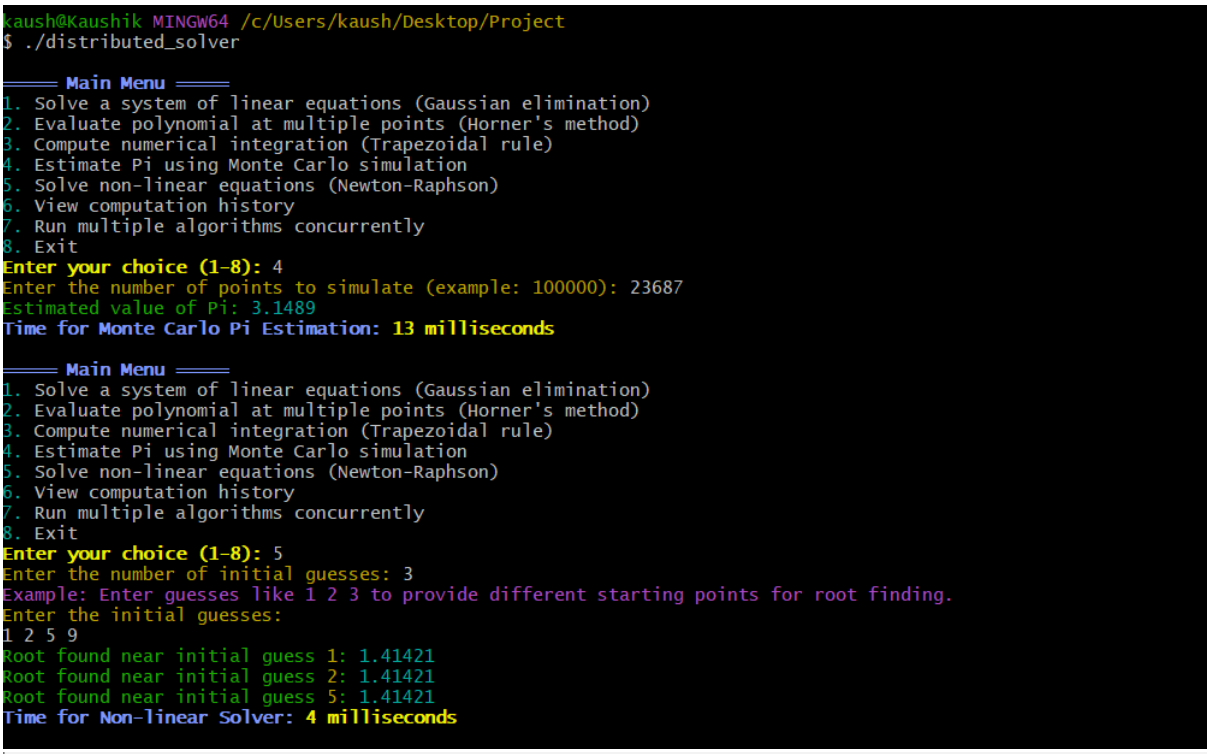
User Interaction and Flow

- ***Starting the System:*** When the program is run, the user is greeted by a numbered main menu, allowing them to select which algorithm or task they want to execute. Users input their choice by typing the corresponding number (e.g., entering 1 to select Gaussian Elimination). This is followed by a prompt to enter the necessary input data, such as matrix coefficients for Gaussian Elimination or polynomial coefficients for polynomial evaluation.
- ***Data Input:*** Once a task is selected, the system guides the user through input prompts to provide the required parameters. For example, in Gaussian Elimination, users are asked to input the matrix size, followed by the values for each row of the matrix and the corresponding results vector. To ensure robust input handling, the system validates the inputs and prompts the user to re-enter values if invalid data is detected.
- ***Executing the Algorithm***: After inputting the required data, the system immediately processes the task using parallel computing where applicable. During execution, the system efficiently leverages multithreading to distribute computation across multiple CPU cores, thereby enhancing performance. Users can monitor the progress directly in the terminal as results are printed in real-time.
- ***Viewing the Output:*** Once the computation is complete, the solution or result is displayed in the terminal. For example, in the case of Gaussian Elimination, the solution to the system of equations is presented, while for Monte Carlo simulations, an estimated value of Pi is outputted. Additionally, the system logs all results into a log file (computation_results.log), which users can later view to track all their previous computations.
- ***Log File and History:*** The log file records the details of each task performed, including the inputs, results, and the execution time. Users can select the option from the main menu to view this computation history, providing them with an easy way to reference past calculations. This is particularly useful for students or researchers who need to keep track of iterative computations or analyse performance over time.
- ***Running Multiple Algorithms:*** A new feature allows users to run multiple algorithms concurrently. Users can specify how many tasks they want to execute simultaneously, and the system will manage the parallel execution of these tasks, displaying progress and results for each. This feature is ideal for users who want to compare different algorithms or solve multiple problems at once.

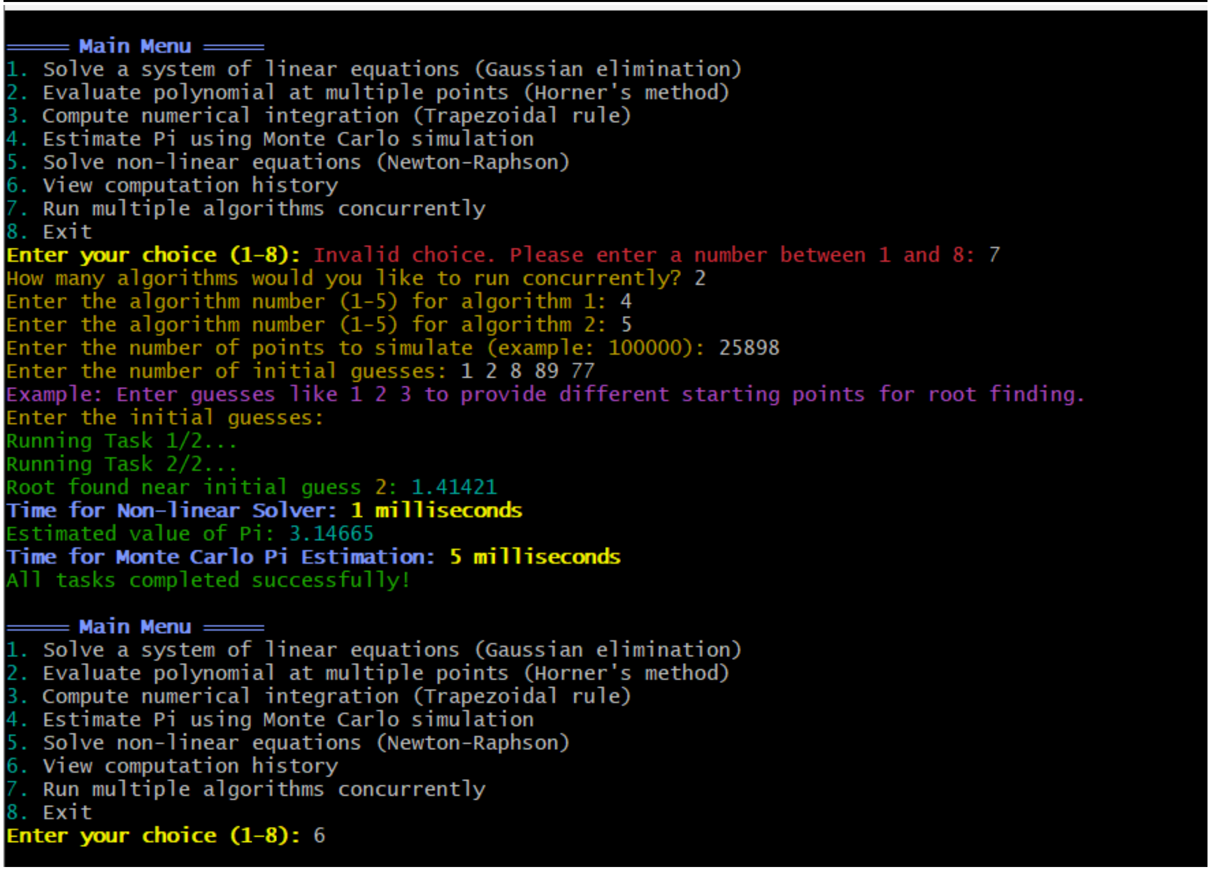
Example of User Flow for Gaussian Elimination:

- Menu: The user launches the program and selects option 1 for Gaussian Elimination.
- Input: The system prompts the user to input the size of the matrix and the coefficients for each row, along with the results vector.
- Computation: Once the inputs are validated, the system runs the Gaussian Elimination algorithm and displays the solution directly in the terminal.
- Output: The results are printed in the terminal, showing the solution for each variable. Additionally, the execution time is displayed, and the result is logged into the log file for future reference.

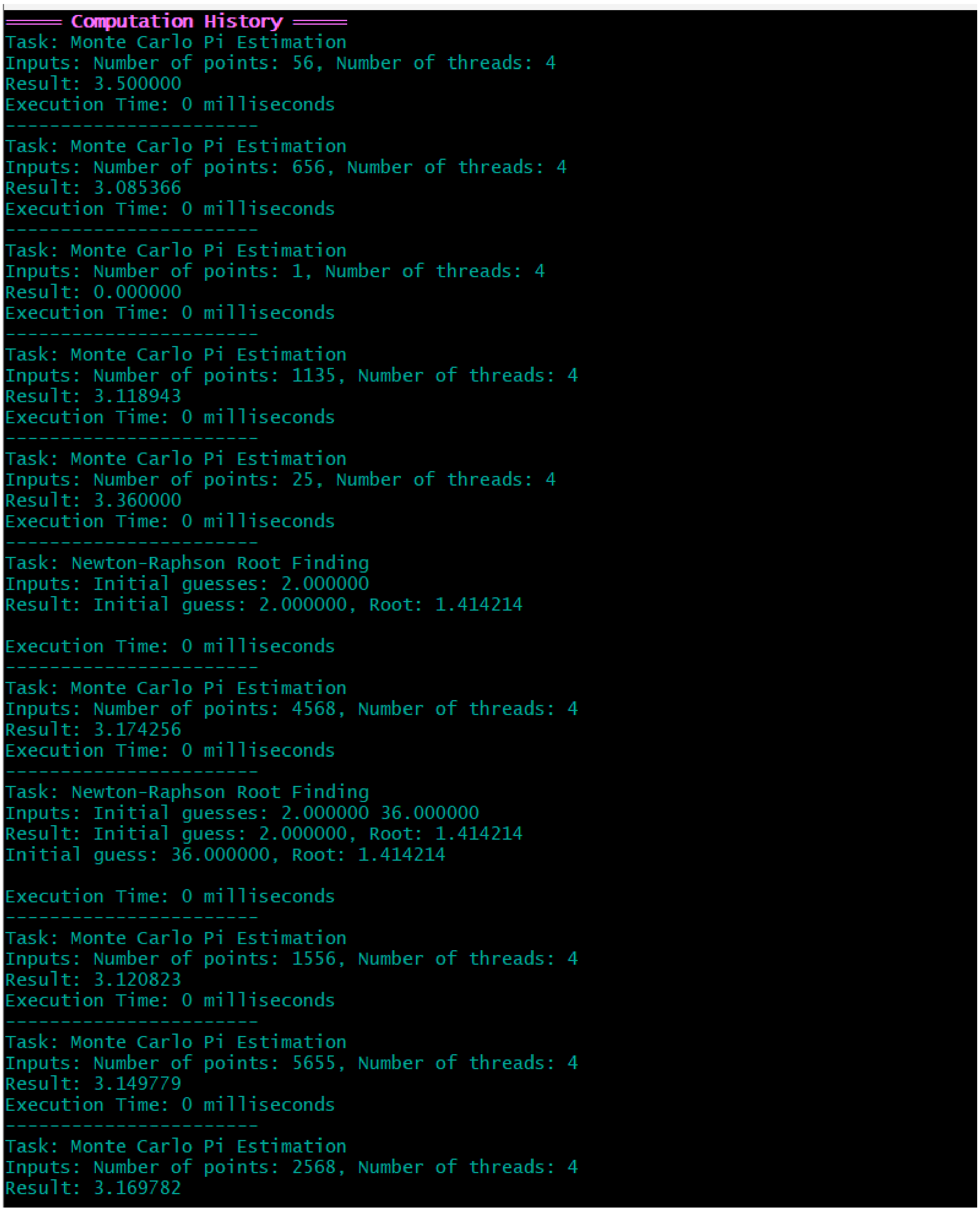
OUTPUT SCREENSHOT



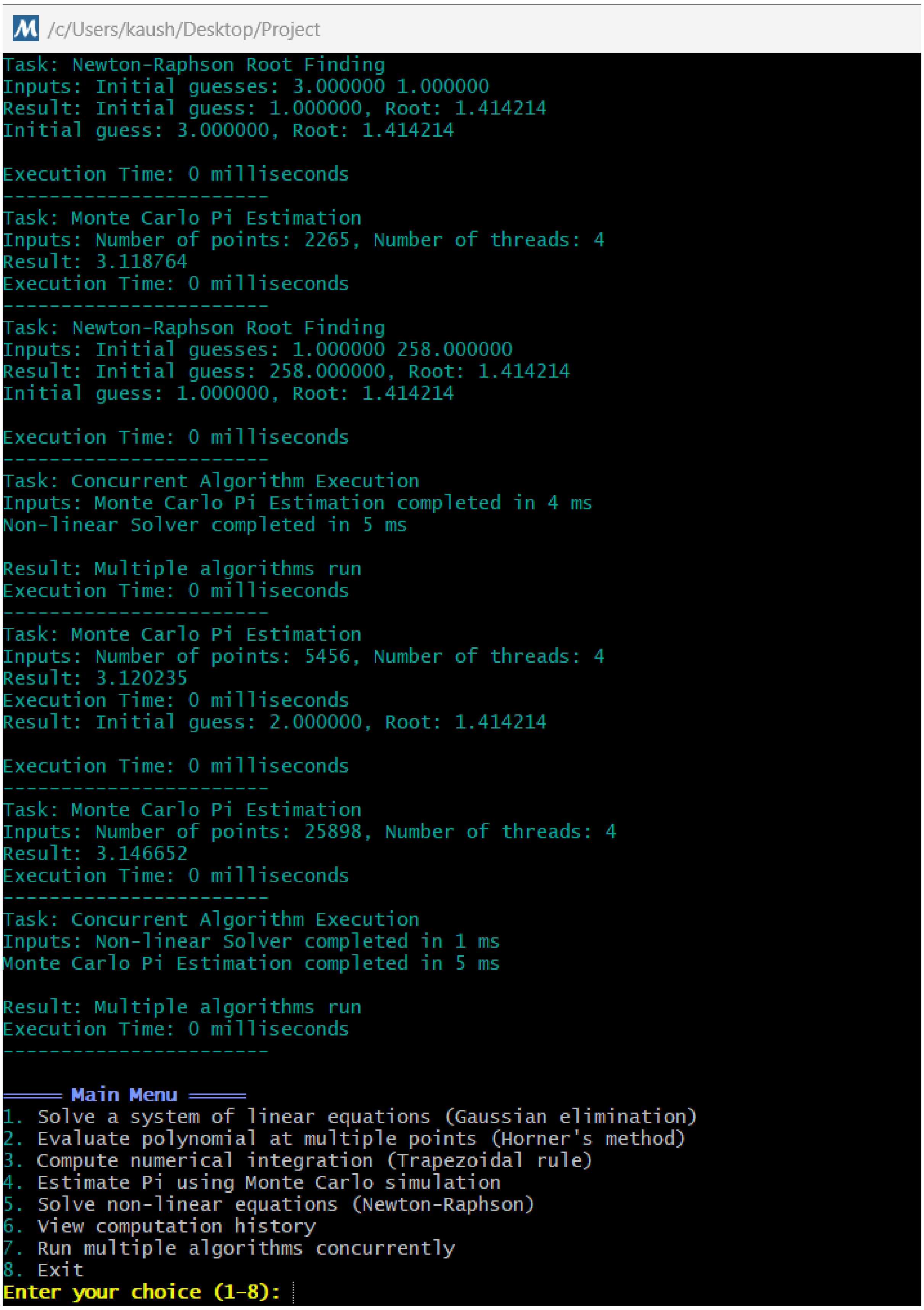
```
kaush@Kaushik MINGW64 /c/Users/kaush/Desktop/Project
$ ./distributed_solver

===== Main Menu =====
1. Solve a system of linear equations (Gaussian elimination)
2. Evaluate polynomial at multiple points (Horner's method)
3. Compute numerical integration (Trapezoidal rule)
4. Estimate Pi using Monte Carlo simulation
5. Solve non-linear equations (Newton-Raphson)
6. View computation history
7. Run multiple algorithms concurrently
8. Exit
Enter your choice (1-8): 4
Enter the number of points to simulate (example: 100000): 23687
Estimated value of Pi: 3.1489
Time for Monte Carlo Pi Estimation: 13 milliseconds

===== Main Menu =====
1. Solve a system of linear equations (Gaussian elimination)
2. Evaluate polynomial at multiple points (Horner's method)
3. Compute numerical integration (Trapezoidal rule)
4. Estimate Pi using Monte Carlo simulation
5. Solve non-linear equations (Newton-Raphson)
6. View computation history
7. Run multiple algorithms concurrently
8. Exit
Enter your choice (1-8): 5
Enter the number of initial guesses: 3
Example: Enter guesses like 1 2 3 to provide different starting points for root finding.
Enter the initial guesses:
1 2 5 9
Root found near initial guess 1: 1.41421
Root found near initial guess 2: 1.41421
Root found near initial guess 5: 1.41421
Time for Non-linear Solver: 4 milliseconds
```



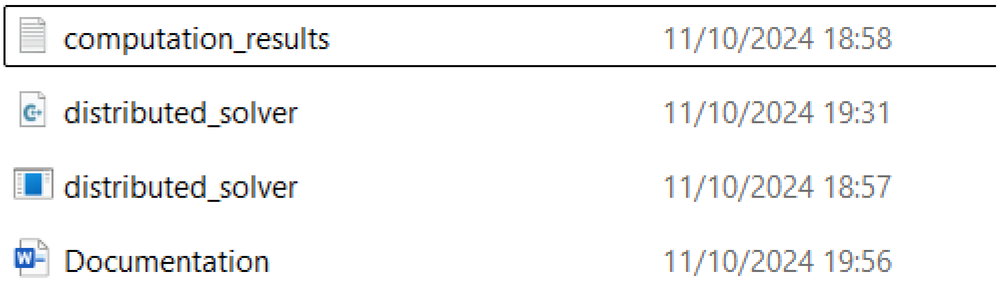
```
===== Main Menu =====
1. Solve a system of linear equations (Gaussian elimination)
2. Evaluate polynomial at multiple points (Horner's method)
3. Compute numerical integration (Trapezoidal rule)
4. Estimate Pi using Monte Carlo simulation
5. Solve non-linear equations (Newton-Raphson)
6. View computation history
7. Run multiple algorithms concurrently
8. Exit
Enter your choice (1-8): Invalid choice. Please enter a number between 1 and 8: 7
How many algorithms would you like to run concurrently? 2
Enter the algorithm number (1-5) for algorithm 1: 4
Enter the algorithm number (1-5) for algorithm 2: 5
Enter the number of points to simulate (example: 100000): 25898
Enter the number of initial guesses: 1 2 8 89 77
Example: Enter guesses like 1 2 3 to provide different starting points for root finding.
Enter the initial guesses:
Running Task 1/2...
Running Task 2/2...
Root found near initial guess 2: 1.41421
Time for Non-linear Solver: 1 milliseconds
Estimated value of Pi: 3.14665
Time for Monte Carlo Pi Estimation: 5 milliseconds
All tasks completed successfully!

===== Main Menu =====
1. Solve a system of linear equations (Gaussian elimination)
2. Evaluate polynomial at multiple points (Horner's method)
3. Compute numerical integration (Trapezoidal rule)
4. Estimate Pi using Monte Carlo simulation
5. Solve non-linear equations (Newton-Raphson)
6. View computation history
7. Run multiple algorithms concurrently
8. Exit
Enter your choice (1-8): 6
```



```
===== Computation History =====
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 56, Number of threads: 4
Result: 3.500000
Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 656, Number of threads: 4
Result: 3.085366
Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 1, Number of threads: 4
Result: 0.000000
Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 1135, Number of threads: 4
Result: 3.118943
Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 25, Number of threads: 4
Result: 3.360000
Execution Time: 0 milliseconds
-----------------------
Task: Newton-Raphson Root Finding
Inputs: Initial guesses: 2.000000
Result: Initial guess: 2.000000, Root: 1.414214

Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 4568, Number of threads: 4
Result: 3.174256
Execution Time: 0 milliseconds
-----------------------
Task: Newton-Raphson Root Finding
Inputs: Initial guesses: 2.000000 36.000000
Result: Initial guess: 2.000000, Root: 1.414214
Initial guess: 36.000000, Root: 1.414214

Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 1556, Number of threads: 4
Result: 3.120823
Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 5655, Number of threads: 4
Result: 3.149779
Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 2568, Number of threads: 4
Result: 3.169782
```



```
/c/Users/kaush/Desktop/Project
Task: Newton-Raphson Root Finding
Inputs: Initial guesses: 3.000000 1.000000
Result: Initial guess: 1.000000, Root: 1.414214
Initial guess: 3.000000, Root: 1.414214

Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 2265, Number of threads: 4
Result: 3.118764
Execution Time: 0 milliseconds
-----------------------
Task: Newton-Raphson Root Finding
Inputs: Initial guesses: 1.000000 258.000000
Result: Initial guess: 258.000000, Root: 1.414214
Initial guess: 1.000000, Root: 1.414214

Execution Time: 0 milliseconds
-----------------------
Task: Concurrent Algorithm Execution
Inputs: Monte Carlo Pi Estimation completed in 4 ms
Non-linear Solver completed in 5 ms

Result: Multiple algorithms run
Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 5456, Number of threads: 4
Result: 3.120235
Execution Time: 0 milliseconds
Result: Initial guess: 2.000000, Root: 1.414214

Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 25898, Number of threads: 4
Result: 3.146652
Execution Time: 0 milliseconds
-----------------------
Task: Concurrent Algorithm Execution
Inputs: Non-linear Solver completed in 1 ms
Monte Carlo Pi Estimation completed in 5 ms

Result: Multiple algorithms run
Execution Time: 0 milliseconds
-----------------------

===== Main Menu =====
1. Solve a system of linear equations (Gaussian elimination)
2. Evaluate polynomial at multiple points (Horner's method)
3. Compute numerical integration (Trapezoidal rule)
4. Estimate Pi using Monte Carlo simulation
5. Solve non-linear equations (Newton-Raphson)
6. View computation history
7. Run multiple algorithms concurrently
8. Exit
Enter your choice (1-8):
```



computation_results	11/10/2024 18:58
distributed_solver	11/10/2024 19:31
distributed_solver	11/10/2024 18:57
Documentation	11/10/2024 19:56

```
Task: Newton-Raphson Root Finding
Inputs: Initial guesses: 1.000000 255.000000
Result: Initial guess: 255.000000, Root: 1.414214
Initial guess: 1.000000, Root: 1.414214

Execution Time: 0 milliseconds
-----------------------
Task: Concurrent Algorithm Execution
Inputs: Monte Carlo Pi Estimation completed in 4 ms
Non-linear Solver completed in 5 ms

Result: Multiple algorithms run
Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 23687, Number of threads: 4
Result: 3.148900
Execution Time: 0 milliseconds
-----------------------
Task: Newton-Raphson Root Finding
Inputs: Initial guesses: 1.000000 2.000000 5.000000
Result: Initial guess: 1.000000, Root: 1.414214
Initial guess: 2.000000, Root: 1.414214
Initial guess: 5.000000, Root: 1.414214

Execution Time: 0 milliseconds
-----------------------
Task: Newton-Raphson Root Finding
Inputs: Initial guesses: 2.000000
Result: Initial guess: 2.000000, Root: 1.414214

Execution Time: 0 milliseconds
-----------------------
Task: Monte Carlo Pi Estimation
Inputs: Number of points: 25898, Number of threads: 4
Result: 3.146652
Execution Time: 0 milliseconds
-----------------------
Task: Concurrent Algorithm Execution
Inputs: Non-linear Solver completed in 1 ms
Monte Carlo Pi Estimation completed in 5 ms

Result: Multiple algorithms run
Execution Time: 0 milliseconds
```

Future Enhancements

As the system continues to evolve, several potential enhancements could further improve its functionality, usability, and performance. Below are key areas for future work, including the integration of a graphical user interface (GUI), distributed computing, and extensions into advanced computational methods.

Graphical User Interface (GUI) Integration

Currently, the system is terminal-based, which requires users to interact through text commands. To improve usability and make the system more accessible, integrating a GUI would provide a more intuitive experience, especially for users who are not familiar with command-line interfaces. A GUI can offer better visualization of results, allow for easier input, and provide interactive control over the computation process.

Frameworks and Technologies for GUI Integration:

- Qt: One of the most popular cross-platform C++ libraries for building GUIs, Qt offers powerful tools for creating windows, buttons, input fields, and even data visualization features such as charts and graphs. Qt would allow the system to present the computational results, such as matrices, graphs of functions, and Monte Carlo results, in an intuitive visual format.
- ImGui: ImGui is an immediate-mode graphical user interface library that is lightweight and suitable for applications needing minimal but efficient UIs. It can provide the system with easy-to-implement controls and visual output windows for students and researchers who need a quick interaction without a complex interface.

Advantages of GUI Integration:

- User-Friendly: A GUI would make the system easier to use, especially for non-programmers, by eliminating the need to manually enter commands.
- Visualization: Users could visualize the steps of matrix operations, graph the polynomial evaluation over a range of inputs, or display the convergence behaviour of the Monte Carlo simulation in real-time.
- Data Export: A GUI can easily support features such as exporting results to CSV or other formats, allowing users to save their work and results for further analysis.

Distributed Computing Integration

For larger computational tasks that exceed the capabilities of a single machine, distributed computing can be implemented to allow the system to scale across multiple machines in a cluster. This would enable the system to handle much larger datasets and perform more intensive computations in less time by leveraging the power of multiple computers working together.

Frameworks and Technologies for Distributed Computing:

- MPI (Message Passing Interface): MPI is the standard framework for distributed computing and would allow this system to run computations across multiple nodes in a cluster. MPI works by distributing data across nodes and coordinating computation using message passing, making it suitable for distributed Gaussian elimination, Monte Carlo simulations, and other parallelizable algorithms.

- OpenMP with MPI: Although OpenMP is traditionally used for shared-memory parallelism, combining it with MPI enables the system to benefit from hybrid parallelism. OpenMP can manage parallel threads on each node, while MPI distributes the workload across different machines. This hybrid model would ensure that the system makes full use of both multi-core processors and multi-node clusters.

Advantages of Distributed Computing:

- Scalability: Distributed computing allows the system to scale beyond the limits of a single machine by distributing workloads across many nodes. For example, a large matrix for Gaussian elimination can be split into smaller sub-matrices processed concurrently on different machines.
- Reduced Execution Time: By harnessing the power of multiple nodes, complex tasks such as Monte Carlo simulations with billions of iterations can be completed in a fraction of the time compared to using a single machine.
- High-Performance Computing (HPC): Integration with clusters or cloud-based HPC solutions would make the system suitable for solving industry-scale problems, opening the door to use in research institutions or companies.

Extensions to Other Algorithms

The system can be extended to incorporate additional numerical algorithms and methods, making it even more versatile and useful in different domains.

Potential Algorithm Extensions:

- ***LU Decomposition:*** Adding support for LU decomposition would enhance the system's ability to solve linear systems more efficiently, especially for cases where multiple systems need to be solved with the same coefficient matrix. This would be particularly useful in real-time simulation applications in engineering and economics.
- ***Fast Fourier Transform (FFT):*** FFT is widely used in signal processing, data compression, and image analysis. By integrating FFT into the system, it could be applied to problems in audio signal processing, compression algorithms, and image recognition.
- ***Sparse Matrix Solvers:*** For extremely large systems of equations where the matrices are sparse (e.g., in network graphs or computational fluid dynamics simulations), adding sparse matrix solvers like Conjugate Gradient or GMRES would make the system applicable to fields like machine learning, graph theory, and fluid simulations.

Machine Learning Integration

- Incorporating machine learning (ML) frameworks into the system would take its capabilities to a new level by allowing it to handle predictive modeling and optimization tasks that require advanced learning algorithms. This extension would make the system highly useful for data-driven research and analysis, particularly in fields like finance, genomics, and artificial intelligence.

Potential Machine Learning Extensions:

- Neural Networks: Integrating libraries such as TensorFlow could enable the system to perform supervised and unsupervised learning, providing users the ability to train models on large datasets for tasks like classification or regression.
- Optimization Algorithms: Adding optimization algorithms like Gradient Descent and Stochastic Gradient Descent (SGD) could be highly useful for solving optimization problems that frequently appear in machine learning models. This could apply to problems in logistics, portfolio optimization, or game theory.

Advantages of Machine Learning Integration:

- Data Analysis: Users can apply machine learning techniques to the results of mathematical computations, such as using polynomial evaluation results to train models for regression analysis.
- Advanced Problem Solving: Machine learning can enhance the system's ability to solve more complex real-world problems that involve pattern recognition, predictive modelling, and decision-making under uncertainty.

Cloud Integration

- Another future enhancement could involve integrating the system with cloud platforms like AWS, Google Cloud, or Microsoft Azure, allowing it to scale elastically. This would allow users to run computations on demand, paying only for the resources they use.

Benefits of Cloud Integration:

- On-Demand Scalability: Users can allocate as many cloud resources (CPUs/GPUs) as needed for a particular task, ensuring fast results without investing in expensive hardware.
- Collaboration: Cloud integration would make it easier for teams to collaborate, share results, and work together on large-scale computations, making the system ideal for group projects or research teams.

Conclusion

- This project provides an efficient, scalable platform for solving a wide range of mathematical problems using parallel computing techniques. It allows users to experiment with multiple algorithms concurrently, making it suitable for both educational and research purposes. The modular architecture and clear logging make it easy to extend and optimize further for more advanced parallel and distributed systems.