Document & Analysis for Circuit Simulator

Introduction

This document provides a comprehensive design framework for the `CSimulator` class and its associated components. The `CSimulator` class is designed to simulate the mass balance of a separation circuit composed of multiple units, evaluate the performance of the circuit, and ensure that the circuit operates correctly under specified constraints.

1. Overview of `CSimulator`

1.1 Purpose

The purpose of the `CSimulator` class is to simulate the mass balance of a circuit that separates valuable material (gerardium) from waste. It calculates the mass flows through each unit in the circuit and determines the overall performance of the circuit.

1.2 Scope

The scope of the `CSimulator` includes:

- Parsing the circuit vector into a structured representation.
- Simulating the mass balance for the entire circuit.
- Evaluating the performance of the circuit based on the mass balance.
- Ensuring the circuit reaches a steady-state solution.

2. Class Design

2.1 UML Class Diagram

```
+------------------+        +----------------+
|   CSimulator     |        |     CUnit      |
+------------------+        +----------------+
| - circuit_vector: std::vector<int>       |
| - units: std::vector<CUnit>              |
| - feed_rate: double                      |
| - tolerance: double                      |
| - max_iterations: int                    |
+------------------+        +----------------+
| + CSimulator(circuit_vector, feed_rate,  |
|   tolerance, max_iterations)             |
| + EvaluateCircuit(): double              |
| + SimulateUnit(unit: CUnit): bool        |
| + SolveSteadyState(): bool               |
| + ParseCircuitVector(): void             |
+------------------+        +----------------+
                          | + CUnit(conc_num: int,        |
                          |   inter_num: int, tails_num: int)|
                          | + CalculateFlows(feed_rate: double, |
                          |   kC: double, kI: double): void|
+-----------------+        +----------------+
|     Attributes   |        |  Attributes   |
|  + concNum: int           |  + concNum: int           |
|  + interNum: int          |  + interNum: int          |
|  + tailsNum: int          |  + tailsNum: int          |
|  + feedGerardium: double  |  + feedGerardium: double  |
|  + feedWaste: double      |  + feedWaste: double      |
|  + newFeedGerardium: double |  + newFeedGerardium: double |
|  + newFeedWaste: double   |  + newFeedWaste: double   |
|  + differenceG: double    |  + differenceG: double    |
|  + differenceW: double    |  + differenceW: double    |
|  + concPtr: CUnit*        |  + concPtr: CUnit*        |
|  + interPtr: CUnit*       |  + interPtr: CUnit*       |
|  + tailsPtr: CUnit*       |  + tailsPtr: CUnit*       |
|  + mark: bool             |  + mark: bool             |
+-----------------+          +-----------------+
```

2.2 `CSimulator` Class

2.2.1 Attributes

- **circuit_vector:** A vector of integers representing the configuration of the circuit.
- **units:** A vector of `CUnit` objects, each representing a unit in the circuit.
- **feed_rate:** A double representing the feed rate of the input stream.
- **tolerance:** A double specifying the tolerance for convergence in the simulation.
- **max_iterations:** An integer representing the maximum number of iterations allowed for achieving convergence.

2.2.2 Methods

Constructor:
The constructor initializes the `CSimulator` with a given circuit vector, feed rate, tolerance, and maximum number of iterations. It also calls a method to parse the circuit vector into a structured format.

EvaluateCircuit:
This method evaluates the entire circuit and returns a performance score. It checks if the circuit reaches a steady-state solution within the specified maximum iterations. If convergence is not achieved, it returns a very negative value to indicate failure.

SimulateUnit:
This method calculates the mass balance for a single unit in the circuit. It uses the feed rate and rate constants to compute the flow rates of the high-grade concentrate, intermediate stream, and tailings stream.

SolveSteadyState:
This method iteratively solves for the steady-state mass balance of the entire circuit. It repeatedly updates the feed rates of all units and checks for convergence within the specified tolerance. If all units' feed rates stabilize within the tolerance, it returns true; otherwise, it continues until the maximum iterations are reached.

ParseCircuitVector:
This method parses the circuit vector into a structured representation by creating `CUnit` objects for each unit in the circuit. It sets the connections between units based on the vector.

2.3 `CUnit` Class

2.3.1 Attributes

- **concNum:** An integer representing the index of the unit for the high-grade concentrate stream.
- **interNum:** An integer representing the index of the unit for the intermediate stream.
- **tailsNum:** An integer representing the index of the unit for the tailings stream.
- **feedGerardium:** A double representing the feed rate of gerardium to the unit.
- **feedWaste:** A double representing the feed rate of waste to the unit.
- **newFeedGerardium:** A double representing the new feed rate of gerardium after processing.
- **newFeedWaste:** A double representing the new feed rate of waste after processing.
- **differenceG:** A double representing the relative change in feed rate of gerardium.
- **differenceW:** A double representing the relative change in feed rate of waste.
- **concPtr:** A pointer to the `CUnit` receiving the concentrate stream.
- **interPtr:** A pointer to the `CUnit` receiving the intermediate stream.
- **tailsPtr:** A pointer to the `CUnit` receiving the tailings stream.
- **mark:** A boolean indicating whether the unit has been marked for traversal.

2.3.2 Methods

Constructor:
The constructor initializes a `CUnit` with the indices for its concentrate, intermediate, and tailings streams. It also initializes the feed rates and other attributes.

CalculateFlows:
This method calculates the mass flows for the unit based on the feed rate and given rate constants for the high-grade and intermediate streams. It computes the residence time and uses it to determine the flow rates for the concentrate, intermediate, and tailings streams.

3. Function Definitions

3.1 `CSimulator` Methods

Constructor:

- The constructor initializes the `CSimulator` with the provided circuit vector, feed rate, tolerance, and maximum iterations.
- It calls the `ParseCircuitVector` method to convert the circuit vector into a structured format, creating a collection of `CUnit` objects representing the units in the circuit.

EvaluateCircuit:

- This method first calls `SolveSteadyState` to iteratively solve for the steady-state mass balance.
- If the steady-state solution is not achieved, it returns a very negative value to indicate failure.
- If convergence is achieved, it calculates the overall performance score by summing the contributions from all units based on their concentrate and tailings rates.

SimulateUnit:

- This method calculates the mass flows for a single unit.
- It uses the feed rate and rate constants to compute the recovery rates for the high-grade concentrate and intermediate streams.
- It then determines the flow rates for the concentrate, intermediate, and tailings streams based on these recovery rates.

SolveSteadyState:

- This method iteratively updates the feed rates of all units and checks for convergence within the specified tolerance.
- It stores the old feed rates and compares them with the new feed rates to check for convergence.
- If all units' feed rates stabilize within the tolerance, it returns true; otherwise, it continues until the maximum number of iterations is reached.

ParseCircuitVector:

- This method converts the circuit vector into a structured representation by creating `CUnit` objects for each unit.
- It sets the connections between units based on the vector, ensuring each unit is correctly linked to its concentrate, intermediate, and tailings streams.

3.2 `CUnit` Methods

Constructor:

- The constructor initializes a `CUnit` with the indices for its concentrate, intermediate, and tailings streams.
- It sets the initial feed rates and other attributes.

CalculateFlows:

- This method calculates the mass flows for the unit based on the feed rate and given rate constants.
- It computes the residence time and uses it to determine the flow rates for the concentrate, intermediate, and tailings streams.
- The calculated flow rates are then assigned to the respective attributes of the `CUnit`.

4. Validation and Testing

4.1 Unit Tests

- **CSimulator Constructor:**
    - Test initialization with valid and invalid circuit vectors to ensure proper setup and error handling.
- **EvaluateCircuit:**
    - Validate performance evaluation by comparing the calculated performance score with expected values for known valid circuits.
- **SimulateUnit:**
    - Verify mass flow calculations by checking the output flow rates for various input feed rates and rate constants.
- **SolveSteadyState:**
    - Check convergence behavior for different circuit configurations and feed rates, ensuring the method stops iterating once convergence is achieved.
- **ParseCircuitVector:**
    - Test the parsing of circuit vectors to ensure that the resulting unit configurations match the expected structure.

4.2 Integration Tests

- **Complete Circuit Simulation:**
    - Validate end-to-end simulation for sample circuits, ensuring correct performance score and convergence for the entire circuit.
- **Boundary Conditions:**
    - Test simulator behavior with edge cases such as zero feed rate, maximum allowable iterations, and extreme values for tolerance.

5. Performance Considerations

- **Efficiency:**
    - Optimize the iterative solver to achieve faster convergence by improving the algorithm and reducing computational overhead.
- **Scalability:**
    - Ensure the simulator can handle larger circuits with more units without significant performance degradation, potentially through parallel processing.
- **Parallelization:**
    - Consider parallelizing unit simulations to leverage multi-core processors, which can improve performance and reduce simulation time.

6. Usage

- **User Guide:**
  - Provide detailed instructions for setting up the environment, running the simulator, and interpreting the output. Include examples and usage scenarios to help users understand how to use the tool effectively.
  - Ensure that the user guide includes steps to compile and run the simulator, required dependencies, and configuration options.
  - Example:

    ### Setup

    1. Clone the repository.
    2. Navigate to the project directory.
    3. Install required dependencies:

    ```
    sudo apt-get install g++
    sudo apt-get install cmake
    ```

    ### Build the project

    ```
    mkdir build
    cd build
    cmake ..
    make
    ```

    ### Running the Simulator

    To run the simulator, use the following command:

    ```
    ./simulator <input_file>
    ```

    Replace `<input_file>` with the path to your input configuration file.

    ### Example

    ```
    ./simulator config.txt
    ```

    ### Interpreting the Output

    The simulator will output the performance score and the detailed mass flow rates for each unit in the circuit.

- **Code Comments:**
  - Ensure all methods and key sections of the code are well-documented with comments explaining their functionality, parameters, and return values.
  - Example:

```
// Constructor for CSimulator
// Initializes the simulator with the given circuit vector, feed rate, tolerance,
and max iterations.
CSimulator::CSimulator(std::vector<int>& circuit_vector, double feed_rate, double
tolerance, int max_iterations) {
    // Initialization code
}
```

- **Examples:**
  - Include example circuit vectors and expected outputs to help users understand the simulator's behavior and validate its performance.
  - Example:

    ## Example Circuit Vector

    ```
    0, 1, 3, 3, 2, 2, 0, 4, 1, 1, 1, 0, 5
    ```

    ## Expected Output

    ```
    Performance Score: 110.25
    Unit 0: Concentrate Rate: 1.5, Intermediate Rate: 0.3, Tailings Rate: 8.2
    ...
    ```

By following this detailed design framework, the `CSimulator` class can be developed to provide robust and efficient simulation of circuit configurations, meeting the requirements of the project and delivering accurate and reliable performance evaluations.