Gerardium Rush - Ilmenite

1.0

Generated by Doxygen 1.11.0

1 Class Index 1

1.1 Class List 1

2 File Index 3

2.1 File List 3

3 Class Documentation 5

3.1 Algorithm_Parameters Struct Reference 5

3.1.1 Detailed Description 5

3.1.2 Member Data Documentation 5

3.1.2.1 crossover_rate 5

3.1.2.2 elitism_rate 6

3.1.2.3 initial_pop 6

3.1.2.4 max_iterations 6

3.1.2.5 mutation_rate 6

3.2 Calculate_constants Struct Reference 6

3.2.1 Detailed Description 7

3.2.2 Member Data Documentation 7

3.2.2.1 k_concentrate_gerardium 7

3.2.2.2 k_concentrate_waste 7

3.2.2.3 k_inter_gerardium 7

3.2.2.4 k_inter_waste 7

3.2.2.5 phi 7

3.2.2.6 rho 8

3.2.2.7 V 8

3.3 Circuit Struct Reference 8

3.3.1 Detailed Description 9

3.3.2 Constructor & Destructor Documentation 9

3.3.2.1 Circuit() 9

3.3.3 Member Function Documentation 9

3.3.3.1 check_feed_value() 9

3.3.3.2 Check_Validity() 10

3.3.3.3 check_values() 12

3.3.3.4 concentrate_tailing_check() 13

3.3.3.5 end_of_vector_check() 14

3.3.3.6 is_reachable() 15

3.3.3.7 mark_units() 16

3.3.3.8 max_value_check() 17

3.3.3.9 same_unit_dest_check() 18

3.3.3.10 self_recycle_check() 19

3.3.3.11 tail_percentage_to_concentrate_outlet_check() 20

3.3.4 Member Data Documentation 20

3.3.4.1 units 20

3.4 Circuit_Parameters Struct Reference . . . 21
3.4.1 Detailed Description . . . 21
3.4.2 Member Data Documentation . . . 21
3.4.2.1 max_iterations . . . 21
3.4.2.2 tolerance . . . 21
3.5 CUnit Class Reference . . . 21
3.5.1 Detailed Description . . . 22
3.5.2 Constructor & Destructor Documentation . . . 22
3.5.2.1 CUnit() . . . 22
3.5.3 Member Data Documentation . . . 22
3.5.3.1 conc_num . . . 22
3.5.3.2 inter_num . . . 23
3.5.3.3 mark . . . 23
3.5.3.4 new_flow_G . . . 23
3.5.3.5 new_flow_W . . . 23
3.5.3.6 old_flow_G . . . 23
3.5.3.7 old_flow_W . . . 23
3.5.3.8 tails_num . . . 24
3.6 Economic_parameters Struct Reference . . . 24
3.6.1 Detailed Description . . . 24
3.6.2 Member Data Documentation . . . 24
3.6.2.1 penalty . . . 24
3.6.2.2 price . . . 24
3.7 Initial_flow Struct Reference . . . 25
3.7.1 Detailed Description . . . 25
3.7.2 Member Data Documentation . . . 25
3.7.2.1 init_Fg . . . 25
3.7.2.2 init_Fw . . . 25
3.8 Recovery Struct Reference . . . 25
3.8.1 Detailed Description . . . 26
3.8.2 Member Data Documentation . . . 26
3.8.2.1 concentrate_gerardium . . . 26
3.8.2.2 concentrate_waste . . . 26
3.8.2.3 inter_gerardium . . . 26
3.8.2.4 inter_waste . . . 26

4 File Documentation **27**
4.1 include/CCircuit.h File Reference . . . 27
4.1.1 Function Documentation . . . 28
4.1.1.1 Check_Validity() . . . 28
4.2 CCircuit.h . . . 29
4.3 include/CSimulator.h File Reference . . . 31

4.3.1 Function Documentation . . . 33
4.3.1.1 calculate_flow_rate() . . . 33
4.3.1.2 calculate_recovery() . . . 34
4.3.1.3 calculate_residence_time() . . . 35
4.3.1.4 Evaluate_Circuit() . . . 36
4.3.1.5 get_performance() . . . 39
4.3.1.6 vector_to_units() . . . 40
4.4 CSimulator.h . . . 41
4.5 include/CUnit.h File Reference . . . 43
4.5.1 Detailed Description . . . 43
4.6 CUnit.h . . . 43
4.7 include/Genetic_Algorithm.h File Reference . . . 44
4.7.1 Detailed Description . . . 46
4.7.2 Macro Definition Documentation . . . 46
4.7.2.1 DEFAULT_ALGORITHM_PARAMETERS . . . 46
4.7.3 Function Documentation . . . 46
4.7.3.1 all_true() . . . 46
4.7.3.2 crossover() . . . 46
4.7.3.3 find_max_double() . . . 47
4.7.3.4 genetic_algorithm() . . . 48
4.7.3.5 initialize_population() . . . 51
4.7.3.6 mutate_vector() . . . 52
4.7.3.7 NonUniform_Mutation() . . . 53
4.7.3.8 optimize() . . . 54
4.7.3.9 select_index() . . . 56
4.8 Genetic_Algorithm.h . . . 56
4.9 src/CCircuit.cpp File Reference . . . 57
4.9.1 Function Documentation . . . 58
4.9.1.1 Check_Validity() . . . 58
4.9.2 Variable Documentation . . . 59
4.9.2.1 conc_outlet_reached . . . 59
4.9.2.2 inter_outlet_reached . . . 60
4.9.2.3 tails_outlet_reached . . . 60
4.10 CCircuit.cpp . . . 60
4.11 src/CSimulator.cpp File Reference . . . 64
4.11.1 Function Documentation . . . 65
4.11.1.1 calculate_flow_rate() . . . 65
4.11.1.2 calculate_recovery() . . . 66
4.11.1.3 calculate_residence_time() . . . 67
4.11.1.4 Evaluate_Circuit() . . . 68
4.11.1.5 get_performance() . . . 70
4.11.1.6 vector_to_units() . . . 71

4.12 CSimulator.cpp 72
4.13 src/Genetic_Algorithm.cpp File Reference 75
4.13.1 Macro Definition Documentation 76
4.13.1.1 PBSTR 76
4.13.1.2 PBWIDTH 76
4.13.2 Function Documentation 76
4.13.2.1 crossover() 76
4.13.2.2 find_max_double() 77
4.13.2.3 generate_random_number() 78
4.13.2.4 genetic_algorithm() 79
4.13.2.5 initialize_population() 81
4.13.2.6 mutate_vector() 82
4.13.2.7 NonUniform_Mutation() 83
4.13.2.8 optimize() 84
4.13.2.9 printProgress() 87
4.13.2.10 regenerate_population() 87
4.13.2.11 select() 88
4.13.2.12 select_index() 89
4.13.3 Variable Documentation 90
4.13.3.1 number_of_units 90
4.14 Genetic_Algorithm.cpp 90
4.15 src/main.cpp File Reference 95
4.15.1 Function Documentation 95
4.15.1.1 main() 95
4.16 main.cpp 96

Index **99**

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Algorithm_Parameters
Parameters for the genetic algorithm 5
Calculate_constants
Constants used in the calculation of the circuit 6
Circuit
Represents a circuit of units 8
Circuit_Parameters
Parameters for the circuit simulator 21
CUnit
Represents a unit in the circuit 21
Economic_parameters
Economic parameters for evaluating the circuit performance 24
Initial_flow
Initial flow rates for the simulation 25
Recovery
Recovery rates of the materials 25

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

include/CCircuit.h 27
include/CSimulator.h 31
include/CUnit.h
Header for the unit class 43
include/Genetic_Algorithm.h
Header for the genetic algorithm and related functions 44
src/CCircuit.cpp 57
src/CSimulator.cpp 64
src/Genetic_Algorithm.cpp 75
src/main.cpp 95

Chapter 3

Class Documentation

3.1 Algorithm_Parameters Struct Reference

Parameters for the genetic algorithm.

```
#include <Genetic_Algorithm.h>
```

Public Attributes

- int max_iterations

  Maximum number of iterations.
- double crossover_rate

  Population crossover rate.
- double mutation_rate

  Population mutation rate.
- double elitism_rate

  Population elitism rate.
- double initial_pop

  Initial population size.

3.1.1 Detailed Description

Parameters for the genetic algorithm.

Definition at line 17 of file Genetic_Algorithm.h.

3.1.2 Member Data Documentation

3.1.2.1 crossover_rate

```
double Algorithm_Parameters::crossover_rate
```

Population crossover rate.

Definition at line 19 of file Genetic_Algorithm.h.

3.1.2.2 elitism_rate

```
double Algorithm_Parameters::elitism_rate
```

Population elitism rate.

Definition at line 21 of file Genetic_Algorithm.h.

3.1.2.3 initial_pop

```
double Algorithm_Parameters::initial_pop
```

Initial population size.

Definition at line 22 of file Genetic_Algorithm.h.

3.1.2.4 max_iterations

```
int Algorithm_Parameters::max_iterations
```

Maximum number of iterations.

Definition at line 18 of file Genetic_Algorithm.h.

3.1.2.5 mutation_rate

```
double Algorithm_Parameters::mutation_rate
```

Population mutation rate.

Definition at line 20 of file Genetic_Algorithm.h.

The documentation for this struct was generated from the following file:

- include/Genetic_Algorithm.h

3.2 Calculate_constants Struct Reference

Constants used in the calculation of the circuit.

```
#include <CSimulator.h>
```

Public Attributes

- double rho = 3000.0
- double phi = 0.1
- double V = 10.0
- double k_concentrate_gerardium = 0.004
- double k_inter_gerardium = 0.001
- double k_concentrate_waste = 0.0002
- double k_inter_waste = 0.0003

3.2.1 Detailed Description

Constants used in the calculation of the circuit.

Definition at line 48 of file CSimulator.h.

3.2.2 Member Data Documentation

3.2.2.1 k_concentrate_gerardium

```
double Calculate_constants::k_concentrate_gerardium = 0.004
```

Rate constant for concentrate gerardium

Definition at line 52 of file CSimulator.h.

3.2.2.2 k_concentrate_waste

```
double Calculate_constants::k_concentrate_waste = 0.0002
```

Rate constant for concentrate waste

Definition at line 54 of file CSimulator.h.

3.2.2.3 k_inter_gerardium

```
double Calculate_constants::k_inter_gerardium = 0.001
```

Rate constant for intermediate gerardium

Definition at line 53 of file CSimulator.h.

3.2.2.4 k_inter_waste

```
double Calculate_constants::k_inter_waste = 0.0003
```

Rate constant for intermediate waste

Definition at line 55 of file CSimulator.h.

3.2.2.5 phi

```
double Calculate_constants::phi = 0.1
```

Porosity

Definition at line 50 of file CSimulator.h.

3.2.2.6 rho

```
double Calculate_constants::rho = 3000.0
```

Density

Definition at line 49 of file CSimulator.h.

3.2.2.7 V

```
double Calculate_constants::V = 10.0
```

Volume

Definition at line 51 of file CSimulator.h.

The documentation for this struct was generated from the following file:

- include/CSimulator.h

3.3 Circuit Struct Reference

Represents a circuit of units.

```
#include <CCircuit.h>
```

Public Member Functions

- Circuit (int num_units)

  Constructs a Circuit with a given number of units.
- bool Check_Validity (int vector_size, int *circuit_vector)

  Checks the validity of a given circuit vector.
- bool is_reachable ()

  Checks if all units are reachable.
- bool concentrate_tailing_check ()

  Checks if the concentrate and tailing outlets are properly defined.
- bool self_recycle_check ()

  Checks for self-recycling units.
- bool same_unit_dest_check ()

  Checks if the same unit has different destinations.
- bool check_values (int vector_size, int *circuit_vector)

  Checks if the values in the circuit vector are valid.
- bool end_of_vector_check ()

  Checks if the end of the vector is reached correctly.
- bool max_value_check (int vector_size, int *circuit_vector)

  Checks if the maximum values in the circuit vector are valid.
- bool tail_percentage_to_concentrate_outlet_check ()

  Checks if the tail percentage to the concentrate outlet is within limits.
- bool check_feed_value (int *circuit_vector)

  Checks the feed value in the circuit vector.

Public Attributes

- std::vector< CUnit > units

Private Member Functions

- void mark_units (int unit_num)

  Marks units for reachability check.

3.3.1 Detailed Description

Represents a circuit of units.

Definition at line 26 of file CCircuit.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Circuit()

```
Circuit::Circuit (
            int num_units)
```

Constructs a Circuit with a given number of units.

Construct a new Circuit object with the specified number of units.

Parameters

num_units	The number of units in the circuit.

Definition at line 30 of file CCircuit.cpp.

```
00030                         {
00031     // Initialize the circuit with a specified number of units.
00032     this->units.resize(num_units);
00033 }
```

3.3.3 Member Function Documentation

3.3.3.1 check_feed_value()

```
bool Circuit::check_feed_value (
            int * circuit_vector)
```

Checks the feed value in the circuit vector.

Check if the feed value is within the valid range (0 to units.size() - 1).

Parameters

circuit_vector	The circuit vector.

Returns

True if the feed value is valid, false otherwise.

Parameters

circuit_vector	The input vector representing the circuit configuration.

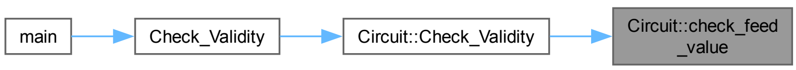
Returns

true if the feed value is within the valid range, false otherwise.

Definition at line 373 of file CCircuit.cpp.

```
00373                                                   {
00374     // Check if the feed value is within the valid range (0 to units.size() - 1)
00375
00376     int max = this->units.size();  // Get the number of units in the circuit
00377     if (circuit_vector[0] < 0 || circuit_vector[0] >= max) {  // Check if the first value is within
      the range
00378         return false;  // If not within the range, return false
00379     }
00380     return true;  // If within the range, return true
00381 }
```

Here is the caller graph for this function:



3.3.3.2 Check_Validity()

```
bool Circuit::Check_Validity (
              int vector_size,
              int * circuit_vector)
```

Checks the validity of a given circuit vector.

Check the validity of the circuit based on given criteria.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The circuit vector.

Returns

True if the circuit vector is valid, false otherwise.

Parameters

vector_size	The size of the input vector.
circuit_vector	The input vector representing the circuit configuration.

Returns

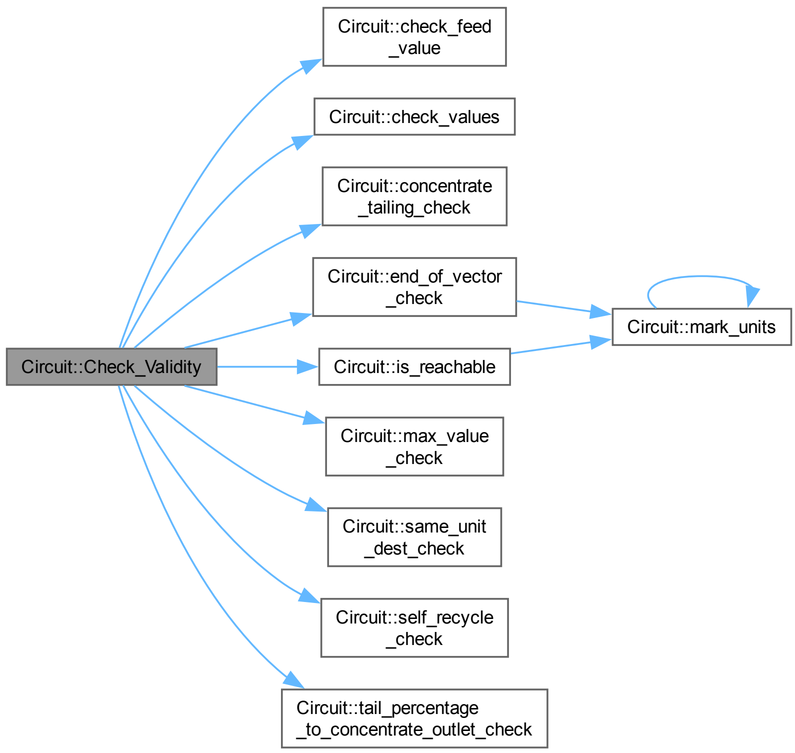
true if the circuit is valid, false otherwise.

Definition at line 42 of file CCircuit.cpp.

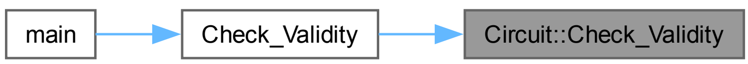
```
                                                                {
    // Calculate the size of the vector in terms of number of elements.
    int unit_size = 0;

    // Determine the number of units and resize the units vector accordingly.
    if ((vector_size - 1) % 3 == 0) {
        this->units.resize(vector_size / 3);
        unit_size = vector_size / 3;
    } else {
        this->units.resize(vector_size / 3 + 1);
        unit_size = vector_size / 3 + 1;
    }

    // Assign the values from the circuit_vector to the respective units.
    for (int i = 0; i < unit_size; ++i) {
        this->units[i].conc_num = circuit_vector[3 * i + 1];
        if (vector_size > 3 * i + 2)
            this->units[i].inter_num = circuit_vector[3 * i + 2];
        if (vector_size > 3 * i + 3)
            this->units[i].tails_num = circuit_vector[3 * i + 3];
        this->units[i].mark = false;
    }

    // Check if all required values are present in the circuit vector.
    if (!this->check_values(vector_size, circuit_vector)) {
        return false;
    }

    // Check if all units are reachable from the feed.
    if (!this->is_reachable()) {
        return false;
    }

    // Check if the concentrate and tailing streams are valid.
    if (!this->concentrate_tailing_check()) {
        return false;
    }

    // Check for self-recycles.
    if (!this->self_recycle_check()) {
        return false;
    }

    // Check if all product streams of a unit do not point to the same unit.
    if (!this->same_unit_dest_check()) {
        return false;
    }

    // Check if all units have a path to all outlets.
    if (!this->end_of_vector_check()) {
        return false;
    }

    // Check if the maximum value constraints are met.
    if (!this->max_value_check(vector_size, circuit_vector)) {
        return false;
    }

    // Check if the tailings percentage to the concentrate outlet is greater than 50%.
    if (!this->tail_percentage_to_concentrate_outlet_check()) {
        return false;
    }

    // Check if the feed value is within the valid range.
    if (!this->check_feed_value(circuit_vector)) {
        return false;
    }

    return true;
```

```
00111 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.3.3 check_values()

```
bool Circuit::check_values (
            int vector_size,
            int * circuit_vector)
```

Checks if the values in the circuit vector are valid.

Check if all required values are present in the circuit vector.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The circuit vector.

Returns

True if the values are valid, false otherwise.

Parameters

vector_size	The size of the input vector.
circuit_vector	The input vector representing the circuit configuration.

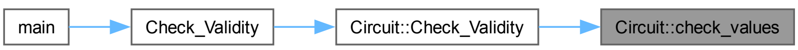
Returns

true if all required values are present, false otherwise.

Definition at line 246 of file CCircuit.cpp.

```
00246                                                                    {
00247     // Find the max in the vector
00248     int max = 0;
00249     for (int i = 0; i < vector_size; i++){
00250         if (circuit_vector[i] > max){
00251             max = circuit_vector[i];
00252         }
00253     }
00254
00255     // Check if the 0-max values appear in the vector
00256     for (int i = 0; i < max; i++){
00257         bool found = false;
00258         for (int j = 0; j < vector_size; j++){
00259             if (circuit_vector[j] == i){
00260                 found = true;
00261                 break;
00262             }
00263         }
00264         if (!found){
00265             return false;
00266         }
00267     }
00268
00269     return true;
00270 }
```

Here is the caller graph for this function:



3.3.3.4 concentrate_tailing_check()

```
bool Circuit::concentrate_tailing_check ()
```

Checks if the concentrate and tailing outlets are properly defined.

Check if the concentrate and tailing streams are valid.

Returns

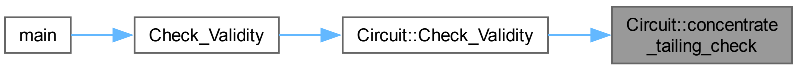
True if the outlets are properly defined, false otherwise.

true if the concentrate and tailing streams are valid, false otherwise.

Definition at line 277 of file CCircuit.cpp.

```
00277                                           {
00278     int num_units = units.size(); // Get the number of units
00279     bool has_concentrate = false; // Flag to check if at least one unit outputs to concentrate
00280     bool has_tailings = false;    // Flag to check if at least one unit outputs to tailings
00281
00282     for (const auto& unit : units) {
00283         // Ensure intermediate or tailings streams do not output to concentrate
00284         if (unit.inter_num == num_units || unit.tails_num == num_units) {
00285             return false;
00286         }
00287         // Ensure concentrate or intermediate streams do not output to tailings
00288         if (unit.conc_num == num_units + 1 || unit.inter_num == num_units + 1) {
00289             return false;
00290         }
00291         // Check if any unit's concentrate stream outputs to concentrate
00292         if (unit.conc_num == num_units) {
00293             has_concentrate = true;
00294         }
00295         // Check if any unit's tailings stream outputs to tailings
00296         if (unit.tails_num == num_units + 1) {
00297             has_tailings = true;
00298         }
00299     }
00300
00301     // Ensure at least one unit outputs to concentrate and tailings
00302     if (!has_concentrate || !has_tailings) {
00303         return false;
00304     }
00305
00306     return true; // All checks passed
00307 }
```

Here is the caller graph for this function:



3.3.3.5 end_of_vector_check()

```
bool Circuit::end_of_vector_check ()
```

Checks if the end of the vector is reached correctly.

Check if all units have a path to all outlets.

Returns

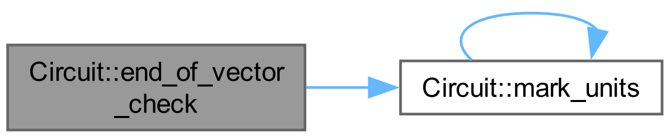
True if the end of the vector is reached correctly, false otherwise.

true if all units have a path to all outlets, false otherwise.

Definition at line 181 of file CCircuit.cpp.

```
                                    {
    // Reset the mark status for all units.
    for (auto& unit : this->units) {
        unit.mark = false;
    }

    // Mark units starting from the feed.
    mark_units(0);

    // Check if any unit has an invalid outlet path.
    for (auto& unit : this->units) {
        if (unit.conc_num == -1 ||
            unit.inter_num == -1 ||
            unit.tails_num == -1) {
            return false;
        }
    }

    return true;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

3.3.3.6 is_reachable()

`bool Circuit::is_reachable ()`

Checks if all units are reachable.

Check if all units are reachable from the feed.

Returns

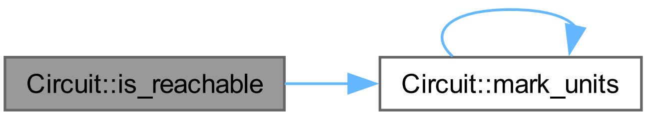
True if all units are reachable, false otherwise.

true if all units are reachable, false otherwise.

Definition at line 156 of file CCircuit.cpp.

```
00156                                {
00157
00158     // Reset the mark status for all units.
00159     for (auto& unit : this->units) {
00160         unit.mark = false;
00161     }
00162
00163     // Mark units starting from the feed.
00164     mark_units(0);
00165
00166     // Check if all units are marked as reachable.
00167     for (const auto& unit : this->units) {
00168         if (!unit.mark) {
00169             return false;
00170         }
00171     }
00172
00173     return true;
00174 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

3.3.3.7 mark_units()

```
void Circuit::mark_units (
            int unit_num)  [private]
```

Marks units for reachability check.

Mark units as reachable starting from a specific unit.

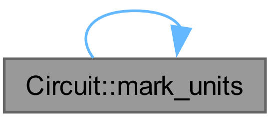
Parameters

unit_num	The unit number to start marking from.
unit_num	The starting unit number.

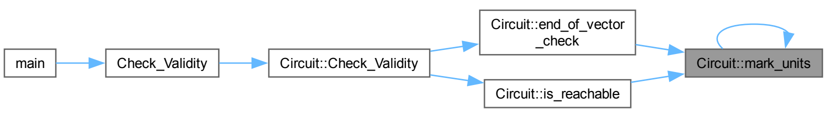
Definition at line 121 of file CCircuit.cpp.

```
00121                                        {
00122     // If the unit is already marked, return.
00123     if (this->units[unit_num].mark)
00124         return;
00125 
00126     // Mark the current unit.
00127     this->units[unit_num].mark = true;
00128 
00129     // Recursively mark units reachable from conc_num.
00130     if (this->units[unit_num].conc_num < this->units.size()) {
00131         mark_units(this->units[unit_num].conc_num);
00132     } else {
00133         conc_outlet_reached = true;
00134     }
00135 
00136     // Recursively mark units reachable from inter_num.
00137     if (this->units[unit_num].inter_num < this->units.size()) {
00138         mark_units(this->units[unit_num].inter_num);
00139     } else {
00140         inter_outlet_reached = true;
00141     }
00142 
00143     // Recursively mark units reachable from tails_num.
00144     if (this->units[unit_num].tails_num < this->units.size()) {
00145         mark_units(this->units[unit_num].tails_num);
00146     } else {
00147         tails_outlet_reached = true;
00148     }
00149 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.3.8 max_value_check()

```
bool Circuit::max_value_check (
            int vector_size,
            int * circuit_vector)
```

Checks if the maximum values in the circuit vector are valid.

Check if the maximum value constraints are met.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The circuit vector.

Returns

True if the maximum values are valid, false otherwise.

Parameters

vector_size	The size of the input vector.
circuit_vector	The input vector representing the circuit configuration.

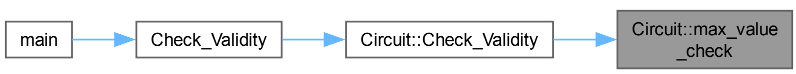
Returns

true if the maximum value constraints are met, false otherwise.

Definition at line 316 of file CCircuit.cpp.

```
00316                                                                  {
00317     // Find the max in the vector
00318     int cnt = this->units.size();
00319 
00320     // Check some value exceeds the max
00321     for (const auto& unit : units) {
00322         if (unit.conc_num > cnt || unit.inter_num > cnt - 1 || unit.tails_num > cnt + 1) {
00323             return false;
00324         }
00325     }
00326 
00327     return true;
00328 }
```

Here is the caller graph for this function:



3.3.3.9 same_unit_dest_check()

`bool Circuit::same_unit_dest_check ()`

Checks if the same unit has different destinations.

Check if all product streams of a unit do not point to the same unit.

Returns

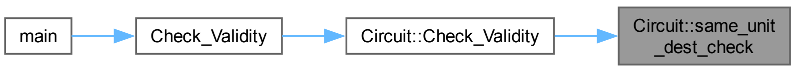
True if no unit has different destinations, false otherwise.

true if the streams do not all point to the same unit, false otherwise.

Definition at line 226 of file CCircuit.cpp.

```
00226                                   {
00227     // Ensure the destinations for the three product streams of each unit are not all the same unit.
00228     for (const auto& unit : this->units) {
00229         // Check if all three product streams point to the same unit.
00230         if (unit.conc_num == unit.inter_num &&
00231             unit.inter_num == unit.tails_num) {
00232             return false;  // If they all point to the same unit, return false indicating invalid circuit.
00233         }
00234     }
00235 
00236     return true;  // If no such condition is found, return true indicating valid circuit.
00237 }
```

Here is the caller graph for this function:



3.3.3.10 self_recycle_check()

`bool Circuit::self_recycle_check ()`

Checks for self-recycling units.

Check for self-recycles in the circuit.

Returns

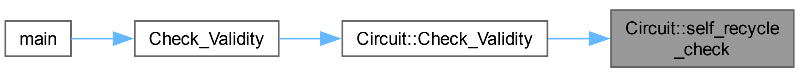
True if no self-recycling units are found, false otherwise.

true if no self-recycles exist, false otherwise.

Definition at line 207 of file CCircuit.cpp.

```
00207                                 {
00208     // Ensure no self-recycle exists.
00209     for (size_t i = 0; i < this->units.size(); ++i) {
00210         const auto& unit = this->units[i];
00211         // Check if any product stream points to the unit itself.
00212         if (unit.conc_num == i ||
00213             unit.inter_num == i ||
00214             unit.tails_num == i) {
00215             return false;
00216         }
00217     }
00218     return true;
00219 }
```

Here is the caller graph for this function:



3.3.3.11 tail_percentage_to_concentrate_outlet_check()

`bool Circuit::tail_percentage_to_concentrate_outlet_check ()`

Checks if the tail percentage to the concentrate outlet is within limits.

Check if the tailings percentage to the concentrate outlet is greater than 50%.

Returns

True if the tail percentage is within limits, false otherwise.

true if the tailings percentage is within the limit, false otherwise.

Definition at line 335 of file CCircuit.cpp.

```
00335                                                                   {
00336     // Check if the tailings percentage is greater than 50%
00337     int max = this->units.size();
00338     int unit_conc = 0;
00339
00340     for (const auto& unit : units) {
00341         if (unit.conc_num == max) {
00342             int cnt_tails = 0;
00343             int cnt = 0;
00344
00345             for (const auto& unit1 : units) {
00346                 if (unit1.tails_num == unit_conc) {
00347                     cnt_tails++;
00348                     cnt++;
00349                 }
00350                 if (unit1.conc_num == unit_conc) {
00351                     cnt++;
00352                 }
00353                 if (unit1.inter_num == unit_conc) {
00354                     cnt++;
00355                 }
00356             }
00357
00358             if (cnt_tails > cnt * 0.5) {
00359                 return false;
00360             }
00361         }
00362         unit_conc++;
00363     }
00364     return true;
00365 }
```

Here is the caller graph for this function:

3.3.4 Member Data Documentation

3.3.4.1 units

`std::vector<CUnit> Circuit::units`

Vector of units in the circuit.

Definition at line 112 of file CCircuit.h.

The documentation for this struct was generated from the following files:

- include/CCircuit.h
- src/CCircuit.cpp

3.4 Circuit_Parameters Struct Reference

Parameters for the circuit simulator.

```
#include <CSimulator.h>
```

Public Attributes

- double tolerance = 0.1
- int max_iterations = 1000

3.4.1 Detailed Description

Parameters for the circuit simulator.

Definition at line 20 of file CSimulator.h.

3.4.2 Member Data Documentation

3.4.2.1 max_iterations

```
int Circuit_Parameters::max_iterations = 1000
```

Maximum number of iterations

Definition at line 22 of file CSimulator.h.

3.4.2.2 tolerance

```
double Circuit_Parameters::tolerance = 0.1
```

Tolerance for the simulation convergence

Definition at line 21 of file CSimulator.h.

The documentation for this struct was generated from the following file:

- include/CSimulator.h

3.5 CUnit Class Reference

Represents a unit in the circuit.

```
#include <CUnit.h>
```

Public Member Functions

- CUnit ()

    Default constructor for CUnit.

Public Attributes

- int conc_num

    Index of the unit to which this unit's concentrate stream is connected.
- int inter_num

    Index of the unit to which this unit's intermediate stream is connected.
- int tails_num

    Index of the unit to which this unit's tailings stream is connected.
- bool mark

    A Boolean that is changed to true if the unit has been seen.
- double old_flow_G

    Total old input flow of gerardium.
- double old_flow_W

    Total old input flow of waste.
- double new_flow_G

    Total new input flow of gerardium.
- double new_flow_W

    Total new input flow of waste.

3.5.1 Detailed Description

Represents a unit in the circuit.

Definition at line 14 of file CUnit.h.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 CUnit()

```
CUnit::CUnit ()  [inline]
```

Default constructor for CUnit.

Initializes the unit with default values.

Definition at line 62 of file CUnit.h.

```
00062 : conc_num(-1), inter_num(-1), tails_num(-1), mark(false), old_flow_G(0.0), old_flow_W(0.0),
      new_flow_G(0.0), new_flow_W(0.0) {}
```

3.5.3 Member Data Documentation

3.5.3.1 conc_num

```
int CUnit::conc_num
```

Index of the unit to which this unit's concentrate stream is connected.

Definition at line 20 of file CUnit.h.

3.5.3.2 inter_num

```
int CUnit::inter_num
```

Index of the unit to which this unit's intermediate stream is connected.

Definition at line 25 of file CUnit.h.

3.5.3.3 mark

```
bool CUnit::mark
```

A Boolean that is changed to true if the unit has been seen.

Definition at line 35 of file CUnit.h.

3.5.3.4 new_flow_G

```
double CUnit::new_flow_G
```

Total new input flow of gerardium.

Definition at line 50 of file CUnit.h.

3.5.3.5 new_flow_W

```
double CUnit::new_flow_W
```

Total new input flow of waste.

Definition at line 55 of file CUnit.h.

3.5.3.6 old_flow_G

```
double CUnit::old_flow_G
```

Total old input flow of gerardium.

Definition at line 40 of file CUnit.h.

3.5.3.7 old_flow_W

```
double CUnit::old_flow_W
```

Total old input flow of waste.

Definition at line 45 of file CUnit.h.

3.5.3.8 tails_num

```
int CUnit::tails_num
```

Index of the unit to which this unit's tailings stream is connected.

Definition at line 30 of file CUnit.h.

The documentation for this class was generated from the following file:

- include/CUnit.h

3.6 Economic_parameters Struct Reference

Economic parameters for evaluating the circuit performance.

```
#include <CSimulator.h>
```

Public Attributes

- double price = 100.0
- double penalty = -750

3.6.1 Detailed Description

Economic parameters for evaluating the circuit performance.

Definition at line 30 of file CSimulator.h.

3.6.2 Member Data Documentation

3.6.2.1 penalty

```
double Economic_parameters::penalty = -750
```

Penalty for failing to meet requirements

Definition at line 32 of file CSimulator.h.

3.6.2.2 price

```
double Economic_parameters::price = 100.0
```

Price of the concentrate

Definition at line 31 of file CSimulator.h.

The documentation for this struct was generated from the following file:

- include/CSimulator.h

3.7 Initial_flow Struct Reference

Initial flow rates for the simulation.

```
#include <CSimulator.h>
```

Public Attributes

- double init_Fg = 10
- double init_Fw = 90

3.7.1 Detailed Description

Initial flow rates for the simulation.

Definition at line 39 of file CSimulator.h.

3.7.2 Member Data Documentation

3.7.2.1 init_Fg

```
double Initial_flow::init_Fg = 10
```

Initial flow rate of gerardium

Definition at line 40 of file CSimulator.h.

3.7.2.2 init_Fw

```
double Initial_flow::init_Fw = 90
```

Initial flow rate of waste

Definition at line 41 of file CSimulator.h.

The documentation for this struct was generated from the following file:

- include/CSimulator.h

3.8 Recovery Struct Reference

Recovery rates of the materials.

```
#include <CSimulator.h>
```

Public Attributes

- double concentrate_gerardium
- double concentrate_waste
- double inter_gerardium
- double inter_waste

3.8.1 Detailed Description

Recovery rates of the materials.

Definition at line 62 of file CSimulator.h.

3.8.2 Member Data Documentation

3.8.2.1 concentrate_gerardium

```
double Recovery::concentrate_gerardium
```

Recovery rate of concentrate gerardium

Definition at line 63 of file CSimulator.h.

3.8.2.2 concentrate_waste

```
double Recovery::concentrate_waste
```

Recovery rate of concentrate waste

Definition at line 64 of file CSimulator.h.

3.8.2.3 inter_gerardium

```
double Recovery::inter_gerardium
```

Recovery rate of intermediate gerardium

Definition at line 65 of file CSimulator.h.

3.8.2.4 inter_waste

```
double Recovery::inter_waste
```

Recovery rate of intermediate waste

Definition at line 66 of file CSimulator.h.

The documentation for this struct was generated from the following file:

- include/CSimulator.h

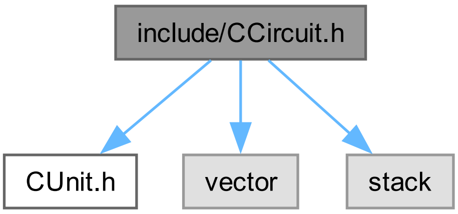
Chapter 4

File Documentation

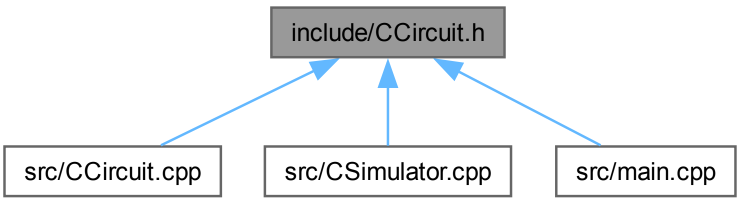
4.1 include/CCircuit.h File Reference

```
#include "CUnit.h"
#include <vector>
#include <stack>
```

Include dependency graph for CCircuit.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct Circuit

  Represents a circuit of units.

Functions

- bool Check_Validity (int vector_size, int ∗circuit_vector)

  Checks the validity of a given circuit vector.

4.1.1 Function Documentation

4.1.1.1 Check_Validity()

```
bool Check_Validity (
            int vector_size,
            int * circuit_vector)
```

Checks the validity of a given circuit vector.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The circuit vector.

Returns

True if the circuit vector is valid, false otherwise.

Checks the validity of a given circuit vector.

This function creates an instance of the Circuit class and uses it to validate the given circuit configuration.

Parameters

vector_size	The size of the input vector.
circuit_vector	The input vector representing the circuit configuration.

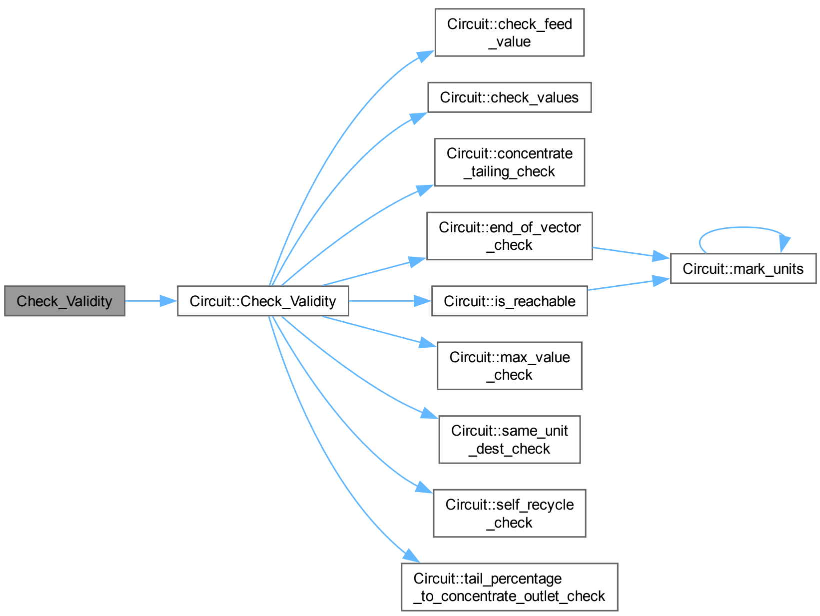
Returns

true if the circuit is valid, false otherwise.

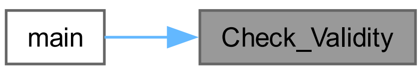
Definition at line 20 of file CCircuit.cpp.

```
00020                                                         {
00021     Circuit circuitInstance(1);
00022     return circuitInstance.Check_Validity(vector_size, circuit_vector);
00023 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.2 CCircuit.h

Go to the documentation of this file.

```
/**
 * @file Circuit.h
 * @brief Header for the circuit struct.
 *
 * This header defines the circuit struct and its associated functions.
 */
```

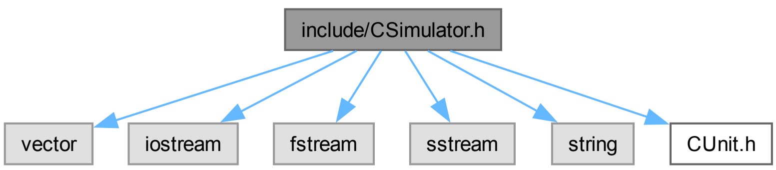
```

#pragma once

#include "CUnit.h"
#include <vector>
#include <stack>

/**
 * @brief Checks the validity of a given circuit vector.
 *
 * @param vector_size The size of the circuit vector.
 * @param circuit_vector The circuit vector.
 * @return True if the circuit vector is valid, false otherwise.
 */
bool Check_Validity(int vector_size, int *circuit_vector);

/**
 * @brief Represents a circuit of units.
 */
struct Circuit
{
    /**
     * @brief Constructs a Circuit with a given number of units.
     *
     * @param num_units The number of units in the circuit.
     */
    Circuit(int num_units);

    /**
     * @brief Checks the validity of a given circuit vector.
     *
     * @param vector_size The size of the circuit vector.
     * @param circuit_vector The circuit vector.
     * @return True if the circuit vector is valid, false otherwise.
     */
    bool Check_Validity(int vector_size, int *circuit_vector);

    /**
     * @brief Checks if all units are reachable.
     *
     * @return True if all units are reachable, false otherwise.
     */
    bool is_reachable();

    /**
     * @brief Checks if the concentrate and tailing outlets are properly defined.
     *
     * @return True if the outlets are properly defined, false otherwise.
     */
    bool concentrate_tailing_check();

    /**
     * @brief Checks for self-recycling units.
     *
     * @return True if no self-recycling units are found, false otherwise.
     */
    bool self_recycle_check();

    /**
     * @brief Checks if the same unit has different destinations.
     *
     * @return True if no unit has different destinations, false otherwise.
     */
    bool same_unit_dest_check();

    /**
     * @brief Checks if the values in the circuit vector are valid.
     *
     * @param vector_size The size of the circuit vector.
     * @param circuit_vector The circuit vector.
     * @return True if the values are valid, false otherwise.
     */
    bool check_values(int vector_size, int *circuit_vector);

    /**
     * @brief Checks if the end of the vector is reached correctly.
     *
     * @return True if the end of the vector is reached correctly, false otherwise.
     */
    bool end_of_vector_check();

    /**
     * @brief Checks if the maximum values in the circuit vector are valid.
     *
     * @param vector_size The size of the circuit vector.
     * @param circuit_vector The circuit vector.
     * @return True if the maximum values are valid, false otherwise.
```

```
     */
    bool max_value_check(int vector_size, int *circuit_vector);

    /**
     * @brief Checks if the tail percentage to the concentrate outlet is within limits.
     *
     * @return True if the tail percentage is within limits, false otherwise.
     */
    bool tail_percentage_to_concentrate_outlet_check();

    /**
     * @brief Checks the feed value in the circuit vector.
     *
     * @param circuit_vector The circuit vector.
     * @return True if the feed value is valid, false otherwise.
     */
    bool check_feed_value(int *circuit_vector);

    std::vector<CUnit> units; /**< Vector of units in the circuit. */

private:
    /**
     * @brief Marks units for reachability check.
     *
     * @param unit_num The unit number to start marking from.
     */
    void mark_units(int unit_num);
};
```

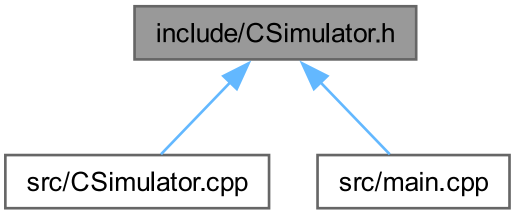
4.3 include/CSimulator.h File Reference

```
#include <vector>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include "CUnit.h"
```

Include dependency graph for CSimulator.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct Circuit_Parameters

  Parameters for the circuit simulator.
- struct Economic_parameters

  Economic parameters for evaluating the circuit performance.
- struct Initial_flow

  Initial flow rates for the simulation.
- struct Calculate_constants

  Constants used in the calculation of the circuit.
- struct Recovery

  Recovery rates of the materials.

Functions

- double Evaluate_Circuit (int vector_size, int ∗circuit_vector)

  Evaluates the circuit performance.
- double calculate_residence_time (const Calculate_constants &constants, double Fg, double Fw)

  Calculates the residence time.
- struct Recovery calculate_recovery (const Calculate_constants &constants, double tau)

  Calculates the recovery rates.
- std::vector< double > calculate_flow_rate (const Calculate_constants &constants, Recovery &recovery, double init_Fg, double init_Fw)

  Calculates the flow rates.
- double get_performance (double cg, double cw, const Economic_parameters &eco)

  Gets the performance of the circuit.
- std::vector< CUnit > vector_to_units (int ∗vector, int size, const Initial_flow &init_flow)

  Converts a vector to a vector of CUnit objects.

4.3.1 Function Documentation

4.3.1.1 calculate_flow_rate()

```
std::vector< double > calculate_flow_rate (
            const Calculate_constants & constants,
            Recovery & recovery,
            double init_Fg,
            double init_Fw)
```

Calculates the flow rates.

Parameters

constants	The constants used in the calculation.
recovery	The recovery rates.
init_Fg	Initial flow rate of gerardium.
init_Fw	Initial flow rate of waste.

Returns

A vector containing the flow rates.

Calculates the flow rates.

Parameters

constants	The constants used for calculation.
recovery	The recovery of materials in the unit.
init_Fg	Initial flow rate of gerardium.
init_Fw	Initial flow rate of waste.

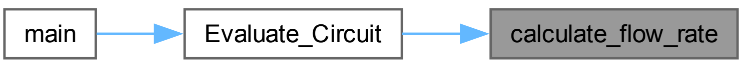
Returns

The calculated flow rates.

Definition at line 189 of file CSimulator.cpp.

```
{
  double cg = init_Fg * recovery.concentrate_gerardium;
  double cw = init_Fw * recovery.concentrate_waste;
  double ig = init_Fg * recovery.inter_gerardium;
  double iw = init_Fw * recovery.inter_waste;
  double tg = init_Fg - cg - ig;
  double tw = init_Fw - cw - iw;

  return {cg, cw, ig, iw, tg, tw};
};
```

Here is the caller graph for this function:



4.3.1.2 calculate_recovery()

```
struct Recovery calculate_recovery (
            const Calculate_constants & constants,
            double tau)
```

Calculates the recovery rates.

Parameters

constants	The constants used in the calculation.
tau	The residence time.

Returns

The recovery rates.

Calculates the recovery rates.

Parameters

constants	The constants used for calculation.
tau	The residence time of materials in the unit.

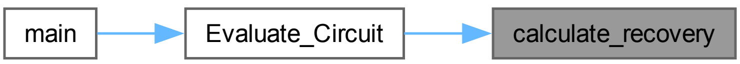
Returns

The calculated recovery.

Definition at line 169 of file CSimulator.cpp.

```
00170 {
00171   double rcg = (constants.k_concentrate_gerardium * tau) / (1 + (constants.k_concentrate_gerardium +
      constants.k_inter_gerardium) * tau);
00172   double rcw = (constants.k_concentrate_waste * tau) / (1 + (constants.k_concentrate_waste +
      constants.k_inter_waste) * tau);
00173   double rig = (constants.k_inter_gerardium * tau) / (1 + (constants.k_concentrate_gerardium +
      constants.k_inter_gerardium) * tau);
00174   double riw = (constants.k_inter_waste * tau) / (1 + (constants.k_concentrate_waste +
      constants.k_inter_waste) * tau);
00175
00176   struct Recovery recoveries = {rcg, rcw, rig, riw};
00177   return recoveries;
00178 };
```

Here is the caller graph for this function:



4.3.1.3 calculate_residence_time()

```
double calculate_residence_time (
            const Calculate_constants & constants,
            double Fg,
            double Fw)
```

Calculates the residence time.

Parameters

constants	The constants used in the calculation.
Fg	Flow rate of gerardium.
Fw	Flow rate of waste.

Returns

The residence time.

Calculates the residence time.

Parameters

constants	The constants used for calculation.
Fg	Flow rate of gerardium.
Fw	Flow rate of waste.

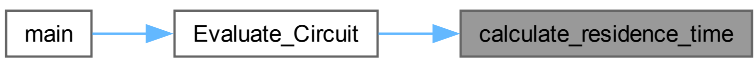
Returns

The calculated residence time.

Definition at line 153 of file CSimulator.cpp.

```
{
  double total_mass_flow_rate = Fw + Fg;
  total_mass_flow_rate = std::max(total_mass_flow_rate, 1e-10);
  double volume_flow_rate = total_mass_flow_rate / constants.rho;
  double tau = constants.phi * constants.V / volume_flow_rate;
  return tau;
};
```

Here is the caller graph for this function:



4.3.1.4 Evaluate_Circuit()

```
double Evaluate_Circuit (
            int vector_size,
            int * circuit_vector)
```

Evaluates the circuit performance.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The circuit vector.

Returns

The performance value.

Evaluates the circuit performance.

This function simulates the operation of a circuit and calculates its performance. Judge if the circuit has converged (using the difference between the old and new flow rates), if not, set the performance to 90 $*$ -750.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The array representing the circuit configuration.

Returns

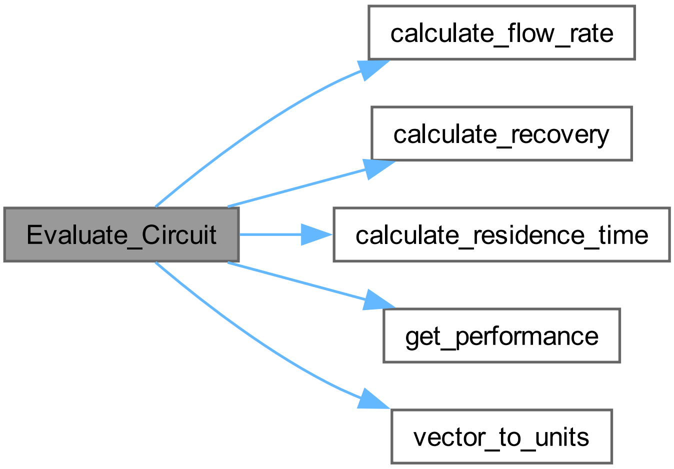
The performance value of the circuit.

Definition at line 20 of file CSimulator.cpp.

```
{
  struct Circuit_Parameters default_circuit_parameters;
  struct Calculate_constants constants;
  struct Initial_flow init_flow;
  struct Economic_parameters eco;

  double Performance = 0.0;
  double Recovery = 0.0;
  double Grade = 0.0;

  // Calculate the number of units in the circuit
  int length = (vector_size - 1) / 3;
  std::vector<CUnit> units = vector_to_units(circuit_vector, length, init_flow);
  int i;
  for (i = 0; i < default_circuit_parameters.max_iterations; i++)
  {
    double concentrate_gerardium = 0.0;
    double concentrate_waste = 0.0;

    // Update the flow rates for the units
    #pragma omp parallel for reduction(+:concentrate_gerardium, concentrate_waste)
    for (int j = 0; j < length; j++)
    {
      double tau = calculate_residence_time(constants, units[j].old_flow_W, units[j].old_flow_G);
      struct Recovery recovery = calculate_recovery(constants, tau);
      std::vector<double> all_flow_rate = calculate_flow_rate(constants, recovery,
units[j].old_flow_G, units[j].old_flow_W);

      if (units[j].conc_num < length)
      {
        #pragma omp atomic
        units[units[j].conc_num].new_flow_G += all_flow_rate[0];
        #pragma omp atomic
        units[units[j].conc_num].new_flow_W += all_flow_rate[1];
      }
      else if (units[j].conc_num == length) // If the unit points to the concentrate stream
      {
        concentrate_gerardium += all_flow_rate[0];
        concentrate_waste += all_flow_rate[1];
      }

      if (units[j].inter_num < length)
      {
        #pragma omp atomic
        units[units[j].inter_num].new_flow_G += all_flow_rate[2];
        #pragma omp atomic
        units[units[j].inter_num].new_flow_W += all_flow_rate[3];
      }

      if (units[j].tails_num < length)
      {
        #pragma omp atomic
        units[units[j].tails_num].new_flow_G += all_flow_rate[4];
        #pragma omp atomic
        units[units[j].tails_num].new_flow_W += all_flow_rate[5];
      }
```

```
    }

    // Add initial flow to the start unit
    int start = circuit_vector[0];
    units[start].new_flow_G += init_flow.init_Fg;
    units[start].new_flow_W += init_flow.init_Fw;

    // Judge if the circuit has converged
    bool converge = true;
    for (int j = 0; j < length; j++)
    {
      // Calculate the relative difference between the old and new flow rates
      double diff_fg = std::abs(units[j].new_flow_G - units[j].old_flow_G) / units[j].old_flow_G;
      double diff_fw = std::abs(units[j].new_flow_W - units[j].old_flow_W) / units[j].old_flow_W;

      if ((diff_fg > 1e-6 || diff_fw > 1e-6))
      {
        converge = false;
        break;
      }
    }

    if (converge)
    {
      // std::cout « i « std::endl;
      // std::cout « concentrate_gerardium « " " « concentrate_waste « std::endl;
      // You can remove the comments above to see the number of iterations and the final concentrate
values
      Performance = get_performance(concentrate_gerardium, concentrate_waste, eco);
      break;
    }
    else
    {
      for (int j = 0; j < length; j++)
      {
        units[j].old_flow_G = units[j].new_flow_G;
        units[j].old_flow_W = units[j].new_flow_W;
        units[j].new_flow_G = 0.0;
        units[j].new_flow_W = 0.0;
      }
    }
    // Calculate the recovery and grade of the circuit
    Recovery = concentrate_gerardium / init_flow.init_Fg;
    Grade = concentrate_gerardium / (concentrate_gerardium + concentrate_waste);
  }

  // If the circuit does not converge, set the performance to 90 * -750
  if (i == 1000)
  {
    Performance = init_flow.init_Fw * eco.penalty;
  }

  try
  {
    // Write the performance, recovery, and grade to a file
    std::ofstream outFile("./output/performance.dat");

    outFile « Performance « "\n";
    outFile « Recovery « "\n";
    outFile « Grade « "\n";

    outFile.close();
  }
  catch (const std::exception &e)
  {
  }

  return Performance;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.3.1.5 get_performance()

```
double get_performance (
            double cg,
            double cw,
            const Economic_parameters & eco)
```

Gets the performance of the circuit.

Parameters

cg	Concentrate gerardium.
cw	Concentrate waste.
eco	The economic parameters.

Returns

The performance value.

Gets the performance of the circuit.

Parameters

cg	Concentrate grade of gerardium.
cw	Concentrate grade of waste.
eco	Economic parameters.

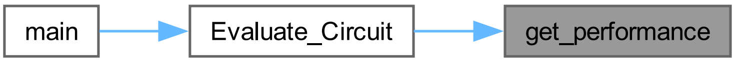
Returns

The calculated performance.

Definition at line 209 of file CSimulator.cpp.

```
00210 {
00211   double result = cg * eco.price + cw * eco.penalty;
00212   return result;
00213 };
```

Here is the caller graph for this function:



4.3.1.6 vector_to_units()

```
std::vector< CUnit > vector_to_units (
            int * vector,
            int n,
            const Initial_flow & init_flow)
```

Converts a vector to a vector of CUnit objects.

Parameters

vector	The input vector.
size	The size of the input vector.
init_flow	The initial flow rates.

Returns

A vector of CUnit objects.

Converts a vector to a vector of CUnit objects.

Parameters

vector	The vector to be converted.
n	The size of the vector.
init_flow	The initial flow rates.

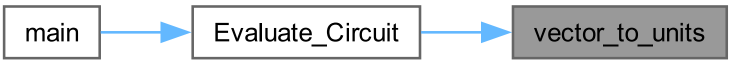
Returns

The vector of units.

Definition at line 223 of file CSimulator.cpp.

```
00224 {
00225   std::vector<CUnit> units(n);
00226   for (int i = 0; i < n; i++)
00227   {
00228     units[i].old_flow_G = init_flow.init_Fg;
00229     units[i].old_flow_W = init_flow.init_Fw;
00230     units[i].new_flow_G = 0;
00231     units[i].new_flow_W = 0;
00232     units[i].conc_num = vector[3 * i + 1];
00233     units[i].inter_num = vector[3 * i + 2];
00234     units[i].tails_num = vector[3 * i + 3];
00235   }
00236   return units;
00237 };
```

Here is the caller graph for this function:



4.4 CSimulator.h

Go to the documentation of this file.

```
/** @file Circuit_Simulator.h
 *  @brief Header file for the circuit simulator.
 *
 *  This header file defines the function and structures that will be used to evaluate the circuit.
 */

#include <vector>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include "CUnit.h"

#pragma once

/**
 * @struct Circuit_Parameters
 * @brief Parameters for the circuit simulator.
 */
struct Circuit_Parameters{
    double tolerance = 0.1;          /**< Tolerance for the simulation convergence */
    int max_iterations = 1000;       /**< Maximum number of iterations */
    // other parameters for your circuit simulator
};

/**
 * @struct Economic_parameters
 * @brief Economic parameters for evaluating the circuit performance.
 */
struct Economic_parameters{
    double price = 100.0;            /**< Price of the concentrate */
    double penalty = -750;           /**< Penalty for failing to meet requirements */
};

/**
 * @struct Initial_flow
 * @brief Initial flow rates for the simulation.
 */
```

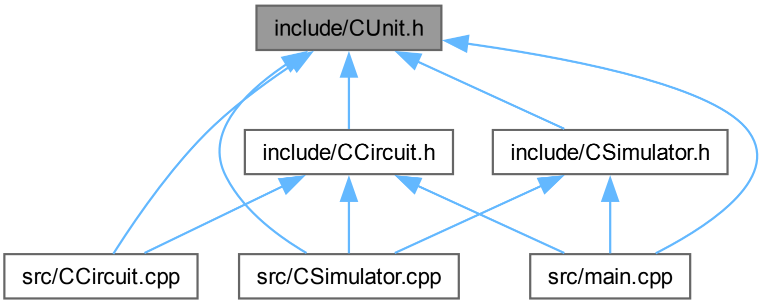
```cpp
struct Initial_flow{
    double init_Fg = 10;             /**< Initial flow rate of gerardium */
    double init_Fw = 90;             /**< Initial flow rate of waste */
};

/**
 * @struct Calculate_constants
 * @brief Constants used in the calculation of the circuit.
 */
struct Calculate_constants{
    double rho = 3000.0;             /**< Density */
    double phi = 0.1;                /**< Porosity */
    double V = 10.0;                 /**< Volume */
    double k_concentrate_gerardium = 0.004; /**< Rate constant for concentrate gerardium */
    double k_inter_gerardium = 0.001;       /**< Rate constant for intermediate gerardium */
    double k_concentrate_waste = 0.0002;    /**< Rate constant for concentrate waste */
    double k_inter_waste = 0.0003;          /**< Rate constant for intermediate waste */
};

/**
 * @struct Recovery
 * @brief Recovery rates of the materials.
 */
struct Recovery{
    double concentrate_gerardium;    /**< Recovery rate of concentrate gerardium */
    double concentrate_waste;        /**< Recovery rate of concentrate waste */
    double inter_gerardium;          /**< Recovery rate of intermediate gerardium */
    double inter_waste;              /**< Recovery rate of intermediate waste */
};

/**
 * @brief Evaluates the circuit performance.
 *
 * @param vector_size The size of the circuit vector.
 * @param circuit_vector The circuit vector.
 * @return The performance value.
 */
double Evaluate_Circuit(int vector_size, int *circuit_vector);

/**
 * @brief Calculates the residence time.
 *
 * @param constants The constants used in the calculation.
 * @param Fg Flow rate of gerardium.
 * @param Fw Flow rate of waste.
 * @return The residence time.
 */
double calculate_residence_time(const Calculate_constants& constants, double Fg, double Fw);

/**
 * @brief Calculates the recovery rates.
 *
 * @param constants The constants used in the calculation.
 * @param tau The residence time.
 * @return The recovery rates.
 */
struct Recovery calculate_recovery(const Calculate_constants& constants, double tau);

/**
 * @brief Calculates the flow rates.
 *
 * @param constants The constants used in the calculation.
 * @param recovery The recovery rates.
 * @param init_Fg Initial flow rate of gerardium.
 * @param init_Fw Initial flow rate of waste.
 * @return A vector containing the flow rates.
 */
std::vector<double> calculate_flow_rate(const Calculate_constants& constants, Recovery& recovery, double init_Fg, double init_Fw);

/**
 * @brief Gets the performance of the circuit.
 *
 * @param cg Concentrate gerardium.
 * @param cw Concentrate waste.
 * @param eco The economic parameters.
 * @return The performance value.
 */
double get_performance(double cg, double cw, const Economic_parameters &eco);

/**
 * @brief Converts a vector to a vector of CUnit objects.
 *
 * @param vector The input vector.
 * @param size The size of the input vector.
 * @param init_flow The initial flow rates.
 * @return A vector of CUnit objects.
```

```
 */
std::vector<CUnit> vector_to_units(int* vector, int size, const Initial_flow &init_flow);
```

4.5 include/CUnit.h File Reference

Header for the unit class.

This graph shows which files directly or indirectly include this file:



Classes

- class CUnit

    Represents a unit in the circuit.

4.5.1 Detailed Description

Header for the unit class.

This header defines the CUnit class, which represents a unit in the circuit.

Definition in file CUnit.h.

4.6 CUnit.h

Go to the documentation of this file.

```
/**
 * @file CUnit.h
 * @brief Header for the unit class.
 *
 * This header defines the CUnit class, which represents a unit in the circuit.
 */

#pragma once

/**
 * @class CUnit
 * @brief Represents a unit in the circuit.
 */
```

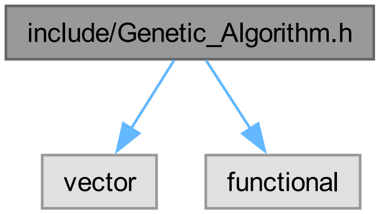
```
class CUnit
{
public:
  /**
   * @brief Index of the unit to which this unit's concentrate stream is connected.
   */
  int conc_num;

  /**
   * @brief Index of the unit to which this unit's intermediate stream is connected.
   */
  int inter_num;

  /**
   * @brief Index of the unit to which this unit's tailings stream is connected.
   */
  int tails_num;

  /**
   * @brief A Boolean that is changed to true if the unit has been seen.
   */
  bool mark;

  /**
   * @brief Total old input flow of gerardium.
   */
  double old_flow_G;

  /**
   * @brief Total old input flow of waste.
   */
  double old_flow_W;

  /**
   * @brief Total new input flow of gerardium.
   */
  double new_flow_G;

  /**
   * @brief Total new input flow of waste.
   */
  double new_flow_W;

  /**
   * @brief Default constructor for CUnit.
   *
   * Initializes the unit with default values.
   */
  CUnit() : conc_num(-1), inter_num(-1), tails_num(-1), mark(false), old_flow_G(0.0), old_flow_W(0.0),
new_flow_G(0.0), new_flow_W(0.0) {}

  /*
   * ...other member functions and variables of CUnit
   */
};
```

4.7 include/Genetic_Algorithm.h File Reference

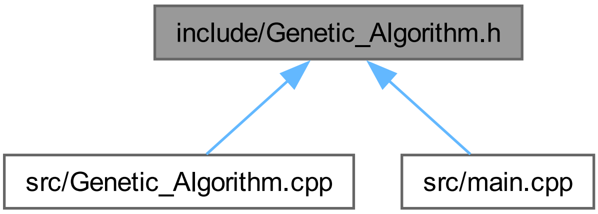
Header for the genetic algorithm and related functions.

```
#include <vector>
#include <functional>
```

Include dependency graph for Genetic_Algorithm.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct Algorithm_Parameters

  Parameters for the genetic algorithm.

Macros

- #define DEFAULT_ALGORITHM_PARAMETERS Algorithm_Parameters{1000, 0.8, 0.1, 0.1, 100}

  Default parameters for the genetic algorithm.

Functions

- bool all_true (int vector_size, int ∗vector)

  Checks if all elements in the vector are true.
- double genetic_algorithm (std::vector< std::vector< int > > &population, double(&func)(int, int ∗), std::function< bool(int, int ∗)> validity, const Algorithm_Parameters &parameters)

  Performs a genetic algorithm optimization.
- int optimize (int vector_size, int ∗vector, double(&func)(int, int ∗), std::function< bool(int, int ∗)> validity, struct Algorithm_Parameters parameters=DEFAULT_ALGORITHM_PARAMETERS)

Optimizes a vector using the genetic algorithm.

- double find_max_double (const double ∗array, int size)
- std::vector< std::vector< int > > initialize_population (int population_size, int vector_size, const int ∗initial_vector, std::function< bool(int, int ∗)> validity, double elitism_rate)
- void NonUniform_Mutation (std::vector< int > &individual, double mutation_rate, int max_value, int currentGeneration, int maxGenerations)
- void mutate_vector (std::vector< int > &vector, double mutation_rate, int max_unit)
- void crossover (std::vector< int > &parent1, std::vector< int > &parent2, double crossover_rate, int max_value)
- int select_index (const std::vector< double > &cumulative_fitness)

4.7.1 Detailed Description

Header for the genetic algorithm and related functions.

This header defines the genetic algorithm and its related functions and structures.

Definition in file Genetic_Algorithm.h.

4.7.2 Macro Definition Documentation

4.7.2.1 DEFAULT_ALGORITHM_PARAMETERS

```
#define DEFAULT_ALGORITHM_PARAMETERS Algorithm_Parameters{1000, 0.8, 0.1, 0.1, 100}
```

Default parameters for the genetic algorithm.

Definition at line 30 of file Genetic_Algorithm.h.

4.7.3 Function Documentation

4.7.3.1 all_true()

```
bool all_true (
            int vector_size,
            int * vector)
```

Checks if all elements in the vector are true.

Parameters

vector_size	Size of the vector.
vector	Pointer to the vector.

Returns

True if all elements are true, false otherwise.

4.7.3.2 crossover()

```
void crossover (
            std::vector< int > & parent1,
            std::vector< int > & parent2,
            double crossover_rate,
            int max_value)
```

Performs a single-point crossover between two parent vectors.

Parameters

parent1	The first parent vector.
parent2	The second parent vector.
crossover_rate	The probability of performing a crossover.
max_value	The maximum value for any gene in the vectors.

Definition at line 215 of file Genetic_Algorithm.cpp.

```
00215
      {
00216
00217     if (generate_random_number(0.0, 1.0) < crossover_rate) {
00218         std::uniform_int_distribution<> point_dist(1, parent1.size() - 2);
00219         int crossover_point = static_cast<int>(generate_random_number(1, parent1.size() - 2));
00220
00221         for (int i = crossover_point; i < parent1.size(); ++i) {
00222             std::swap(parent1[i], parent2[i]);
00223         }
00224     }
00225 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

4.7.3.3 find_max_double()

```
double find_max_double (
            const double * array,
            int size)
```

Finds the maximum value in a double array.

Parameters

array	Pointer to the first element of the double array.
size	Size of the array.

Returns

The maximum value found in the array.

Definition at line 48 of file Genetic_Algorithm.cpp.

```
00048                                                                  {
00049     double max_value = array[0];
00050     for (int i = 1; i < size; ++i) {
00051         if (array[i] > max_value) {
00052             max_value = array[i];
00053         }
00054     }
00055     return max_value;
00056 }
```

Here is the caller graph for this function:

4.7.3.4 genetic_algorithm()

```
double genetic_algorithm (
            std::vector< std::vector< int > > & population,
            double(&)(int, int *) func,
            std::function< bool(int, int *)> validity,
            const Algorithm_Parameters & parameters)
```

Performs a genetic algorithm optimization.

Parameters

population	The population of solutions.
func	The objective function.
validity	The validity function.
parameters	The parameters for the genetic algorithm.

Returns

The best performance value found.

Conducts the entire genetic algorithm process, managing the population through multiple generations and applying genetic operations like selection, crossover, and mutation to evolve solutions.

Parameters

population	The initial population of solutions.
func	A function pointer to the fitness evaluation function.
validity	A function to check the validity of individual solutions.
parameters	Struct containing parameters for the genetic algorithm.

Returns

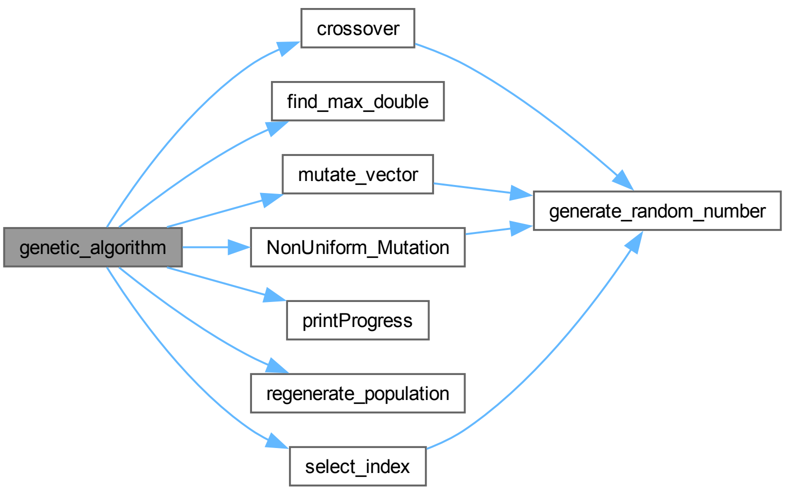
The maximum fitness achieved by the best solution in the population.

Definition at line 272 of file Genetic_Algorithm.cpp.

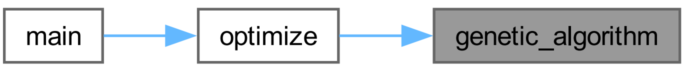
```
                                                                                  {
    int population_size = population.size();
    int vector_size = population[0].size();
    std::vector<double> fitness(population_size);
    double max_fitness = std::numeric_limits<double>::lowest();
    int fitness_unchanged_count = 0;

    int elitism_count = static_cast<int>(population.size() * parameters.elitism_rate);

    for (int generation = 0; generation < parameters.max_iterations; ++generation) {
        // Evaluate fitness for each vector in the population
        #pragma omp parallel for
        for (int i = 0; i < population_size; ++i) {
            if (validity(vector_size, population[i].data())) {
                fitness[i] = func(vector_size, population[i].data());
            } else {
                fitness[i] = std::numeric_limits<double>::lowest();
            }
        }

        // Sort the population based on fitness
        std::vector<int> idx(population_size);
        std::iota(idx.begin(), idx.end(), 0);
        std::sort(idx.begin(), idx.end(), [&](int i1, int i2) { return fitness[i1] > fitness[i2]; });

        printProgress((double)generation / (parameters.max_iterations - 1), fitness[idx[0]]);

        // Implement elitism, save the best individuals
        std::vector<std::vector<int» new_population;
        for (int i = 0; i < elitism_count; ++i) {
            new_population.push_back(population[idx[i]]);
        }

        // Create a cumulative fitness sum for roulette wheel selection
        std::vector<double> cumulative_fitness(population_size);
        std::partial_sum(fitness.begin(), fitness.end(), cumulative_fitness.begin());

        // #pragma omp parallel for
        for (int i = elitism_count; i < population.size(); i+=2) {
            std::vector<int> parent1 = population[select_index(cumulative_fitness)];
            std::vector<int> parent2 = population[select_index(cumulative_fitness)];

            crossover(parent1, parent2, parameters.crossover_rate, number_of_units);
            crossover(parent1, parent2, parameters.crossover_rate, number_of_units);

            // if (vector_size > 100) {
            //     while (!validity(vector_size, parent1.data()) && !validity(vector_size, parent2.data())) {
            //         crossover(parent1, parent2, parameters.crossover_rate, number_of_units);
            //     }
            // }

            NonUniform_Mutation(parent1, parameters.mutation_rate, number_of_units, generation, parameters.max_iterations);
            NonUniform_Mutation(parent2, parameters.mutation_rate, number_of_units, generation, parameters.max_iterations);
            double mutator = 0.0;

            if (fitness_unchanged_count > (parameters.max_iterations * 0.1)) {
                mutator = parameters.mutation_rate + (fitness_unchanged_count * 0.001);
                mutator = mutator < 0.5 ? mutator : 0.5;
            } else {

                mutator = parameters.mutation_rate;
            }
            mutate_vector(parent1, mutator, number_of_units);
            mutate_vector(parent2, mutator, number_of_units);

            new_population.push_back(parent1);
            if (new_population.size() < population_size) {
                new_population.push_back(parent2);
            }
        }

        double temp_fitness = find_max_double(fitness.data(), fitness.size());
        if (temp_fitness - max_fitness < 0.1) {
            fitness_unchanged_count++;
        }
        else{
            fitness_unchanged_count = 0;
```

```
        }
        if (temp_fitness > max_fitness) {
          fs::path dir("./output");
          if (!fs::exists(dir)) {
              fs::create_directories(dir);
          }
          std::ofstream vector_file("./output/vector.dat");
          if (vector_file.is_open()) {
              for (int i = 0; i < vector_size; i++) {
                  vector_file « population[0][i] « " ";
              }
              vector_file.close();
          }

        }
        population = new_population;
        if (fitness_unchanged_count > 50) {
            regenerate_population(population, vector_size, number_of_units);
            fitness_unchanged_count = 0;
        }
        #pragma omp barrier
        max_fitness = *std::max_element(fitness.begin(), fitness.end());
    }
    std::cout « std::endl;
    return max_fitness;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.7.3.5 initialize_population()

```
std::vector< std::vector< int > > initialize_population (
            int population_size,
            int vector_size,
            const int * initial_vector,
            std::function< bool(int, int *)> validity,
            double elitism_rate)
```

Initializes a population for the genetic algorithm.

Parameters

population_size	The size of the population to initialize.
vector_size	The size of each individual in the population.
initial_vector	Initial values for the first individual in the population.
validity	A function that checks the validity of an individual.
elitism_rate	The rate of elitism to apply during evolution.

Returns

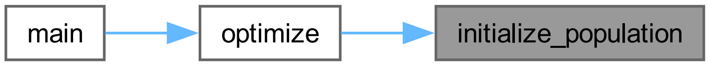
A vector of vectors containing the initialized population.

Definition at line 83 of file Genetic_Algorithm.cpp.

```
00083 {
00084     std::vector<std::vector<int>> population;
00085     population.reserve(population_size);  // Reserve space to avoid reallocations
00086
00087     std::vector<int> start_vector(initial_vector, initial_vector + vector_size);
00088     population.push_back(start_vector);
00089
00090     number_of_units = *std::max_element(initial_vector, initial_vector + vector_size) + 1;
00091
00092     #pragma omp parallel
00093     {
00094         std::random_device rd;
00095         std::mt19937 gen(rd() + omp_get_thread_num());  // Unique seed for each thread
00096         std::uniform_int_distribution<> distr(0, number_of_units - 1);
00097
00098         #pragma omp for
00099         for (int i = 1; i < population_size; ++i) {
00100             std::vector<int> individual(vector_size);
00101             for (int j = 0; j < vector_size; ++j) {
00102                 individual[j] = distr(gen);
00103
00104                 bool valid = true;
00105                 if (j == 0) {
00106                     if (individual[j] == number_of_units - 2 || individual[j] == number_of_units - 3)
      {
00107                         valid = false;
00108                     }
00109                 } else if ((j - 1) / 3 == individual[j]) {
00110                     valid = false;
00111                 }
00112
00113                 if (!valid) {
00114                     j--;
00115                 }
00116             }
00117
00118             #pragma omp critical
00119             {
00120                 if (validity(vector_size, individual.data()) || population.size() < population_size *
      0.8) {
00121                     population.push_back(individual);
00122                 } else {
00123                     --i;  // Retry this iteration
00124                 }
00125             }
00126         }
```

```
    }

    return population;
}
```

Here is the caller graph for this function:



4.7.3.6 mutate_vector()

```
void mutate_vector (
            std::vector< int > & vector,
            double mutation_rate,
            int max_unit)
```

Mutates a given vector with a specified mutation rate. Each element in the vector has a chance to be changed based on the mutation rate.

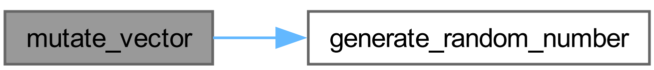
Parameters

vector	The vector to mutate.
mutation_rate	The probability of mutating each element of the vector.
max_unit	The maximum value any element in the vector can take.

Definition at line 197 of file Genetic_Algorithm.cpp.

```
                                                                         {

    for (int& value : vector) {
        if (generate_random_number(0.0, 1.0) < mutation_rate) {
            value = (value + static_cast<int>(generate_random_number(0, max_unit))) % (max_unit + 1);
        }
    }
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.7.3.7 NonUniform_Mutation()

```
void NonUniform_Mutation (
            std::vector< int > & individual,
            double mutation_rate,
            int max_value,
            int currentGeneration,
            int maxGenerations)
```

Applies a non-uniform mutation to an individual in the population. Mutation depends on the current generation, allowing for finer mutations as the number of generations increases.

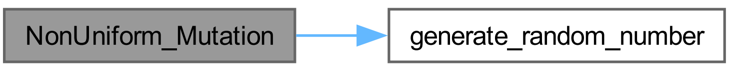
Parameters

individual	A reference to the individual (vector of ints) to mutate.
mutation_rate	The mutation rate to apply.
max_value	The maximum value for any gene in the individual.
currentGeneration	The current generation number in the genetic algorithm.
maxGenerations	The maximum number of generations expected to run.

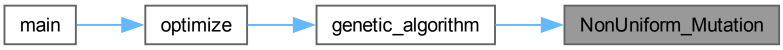
Definition at line 171 of file Genetic_Algorithm.cpp.

```
00171
     {
00172
00173     for (int& gene : individual) {
00174         if (generate_random_number(0.0, 1.0) < mutation_rate) {
00175             double delta = (generate_random_number(0.0, 1.0) < 0.5) ? gene : max_value - gene;
00176             double b = 5;
00177             double r = generate_random_number(0.0, 1.0);
00178
00179             double change = delta * (1 - pow(r, pow((1 - double(currentGeneration) / maxGenerations),
     b)));
00180             gene = (generate_random_number(0.0, 1.0) < 0.5) ? gene - static_cast<int>(change) : gene +
     static_cast<int>(change);
00181
00182             if (gene < 0) gene = 0;
00183             if (gene > max_value) gene = max_value;
00184         }
00185     }
00186 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.7.3.8 optimize()

```
int optimize (
            int vector_size,
            int * vector,
            double(&)(int, int *) func,
            std::function< bool(int, int *)> validity,
            struct Algorithm_Parameters parameters)
```

Optimizes a vector using the genetic algorithm.

Parameters

vector_size	Size of the vector.
vector	Pointer to the vector.
func	The objective function.
validity	The validity function.
parameters	The parameters for the genetic algorithm.

Returns

The index of the best solution found.

Optimizes a vector using genetic algorithm principles. Initializes a population, runs the genetic algorithm, and stores the best solution back into the original vector.

Parameters

vector_size	The size of the vector to be optimized.
vector	The vector containing initial values, modified in-place to store the best solution found.
func	A function pointer to the fitness evaluation function.
validity	A function to check the validity of individual solutions.
parameters	Struct containing parameters for the genetic algorithm.

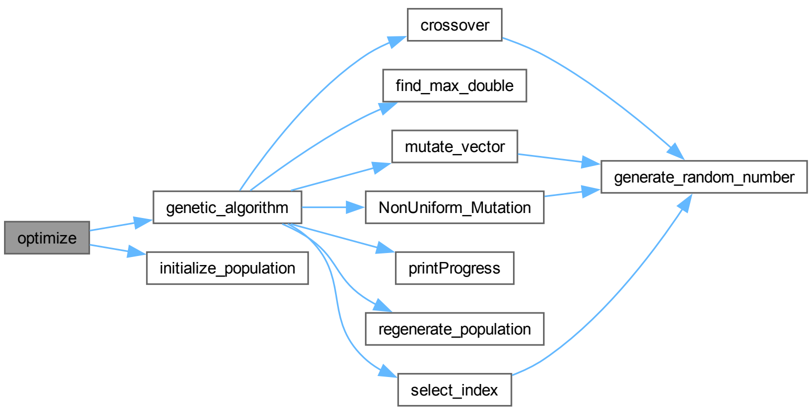
Returns

Returns 0 on successful execution and optimization, -1 if file operation fails.

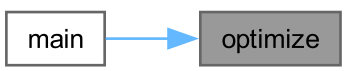
Definition at line 390 of file Genetic_Algorithm.cpp.

```
                                                            {
    // print the number of threads
    std::cout « "Number of threads: " « omp_get_max_threads() « std::endl;

    int unit_num = (vector_size - 1) / 3;

    for (int i = 0; i <= unit_num+1; ++i){
        vector[i] = i;
    }
    for (int i = unit_num+2; i < vector_size; ++i){
        vector[i] = 0;
    }

    std::vector<std::vector<int» population = initialize_population(parameters.initial_pop,
vector_size, vector, validity, parameters.elitism_rate);
    double max_fitness = genetic_algorithm(population, func, validity, parameters);
    std::copy(population[0].begin(), population[0].end(), vector);

    std::ofstream vector_file("./output/vector.dat");
    if (vector_file.is_open()) {
        for (int i = 0; i < vector_size; i++) {
            vector_file « vector[i] « " ";
        }
        vector_file.close();
    } else {
        return -1;
    }

    return 0;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.7.3.9 select_index()

```
int select_index (
            const std::vector< double > & cumulative_fitness)
```

Selects an index for roulette wheel selection based on cumulative fitness scores.

Parameters

cumulative_fitness	A vector of cumulative fitness scores.

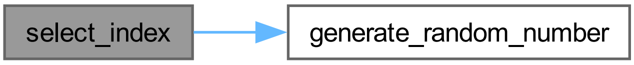
Returns

The selected index based on the random choice in the cumulative distribution.

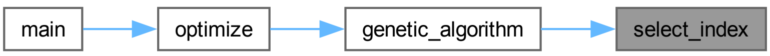
Definition at line 234 of file Genetic_Algorithm.cpp.

```
                                                        {
    double rnd = generate_random_number(0.0, cumulative_fitness.back());
    return std::lower_bound(cumulative_fitness.begin(), cumulative_fitness.end(), rnd) -
cumulative_fitness.begin();
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.8 Genetic_Algorithm.h

Go to the documentation of this file.

```
/**
 * @file Genetic_Algorithm.h
 * @brief Header for the genetic algorithm and related functions.
 *
 * This header defines the genetic algorithm and its related functions and structures.
 */

#pragma once

#include <vector>
```

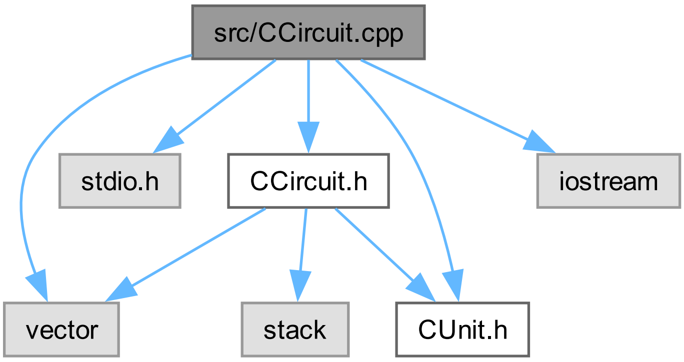
```
#include <functional>  // Include this for std::function

/**
 * @struct Algorithm_Parameters
 * @brief Parameters for the genetic algorithm.
 */
struct Algorithm_Parameters {
    int max_iterations;  ///< Maximum number of iterations.
    double crossover_rate;  ///< Population crossover rate.
    double mutation_rate;   ///< Population mutation rate.
    double elitism_rate;    ///< Population elitism rate.
    double initial_pop;     ///< Initial population size.
    // other parameters for your algorithm
};

/**
 * @def DEFAULT_ALGORITHM_PARAMETERS
 * @brief Default parameters for the genetic algorithm.
 */
#define DEFAULT_ALGORITHM_PARAMETERS Algorithm_Parameters{1000, 0.8, 0.1, 0.1, 100}

/**
 * @brief Checks if all elements in the vector are true.
 *
 * @param vector_size Size of the vector.
 * @param vector Pointer to the vector.
 * @return True if all elements are true, false otherwise.
 */
bool all_true(int vector_size, int *vector);

/**
 * @brief Performs a genetic algorithm optimization.
 *
 * @param population The population of solutions.
 * @param func The objective function.
 * @param validity The validity function.
 * @param parameters The parameters for the genetic algorithm.
 * @return The best performance value found.
 */
double genetic_algorithm(std::vector<std::vector<int» &population,
                         double (&func)(int, int *),
                         std::function<bool(int, int *)> validity,
                         const Algorithm_Parameters &parameters);

/**
 * @brief Optimizes a vector using the genetic algorithm.
 *
 * @param vector_size Size of the vector.
 * @param vector Pointer to the vector.
 * @param func The objective function.
 * @param validity The validity function.
 * @param parameters The parameters for the genetic algorithm.
 * @return The index of the best solution found.
 */
int optimize(int vector_size, int *vector,
             double (&func)(int, int *),
             std::function<bool(int, int *)> validity,
             struct Algorithm_Parameters parameters = DEFAULT_ALGORITHM_PARAMETERS);

double find_max_double(const double *array, int size);

std::vector<std::vector<int» initialize_population(int population_size, int vector_size, const int* initial_vector, std::function<bool(int, int*)> validity, double elitism_rate);

void NonUniform_Mutation(std::vector<int>& individual, double mutation_rate, int max_value, int currentGeneration, int maxGenerations);

void mutate_vector(std::vector<int>& vector, double mutation_rate, int max_unit);

void crossover(std::vector<int>& parent1, std::vector<int>& parent2, double crossover_rate, int max_value);

int select_index(const std::vector<double>& cumulative_fitness);

```

4.9 src/CCircuit.cpp File Reference

```
#include <vector>
#include <stdio.h>
#include <CUnit.h>
```

```
#include <CCircuit.h>
#include <iostream>
```

Include dependency graph for CCircuit.cpp:



Functions

- bool Check_Validity (int vector_size, int ∗circuit_vector)

  Check the validity of the circuit based on given criteria.

Variables

- bool conc_outlet_reached
- bool inter_outlet_reached
- bool tails_outlet_reached

4.9.1 Function Documentation

4.9.1.1 Check_Validity()

```
bool Check_Validity (
            int vector_size,
            int * circuit_vector)
```

Check the validity of the circuit based on given criteria.

Checks the validity of a given circuit vector.

This function creates an instance of the Circuit class and uses it to validate the given circuit configuration.

Parameters

vector_size	The size of the input vector.
circuit_vector	The input vector representing the circuit configuration.

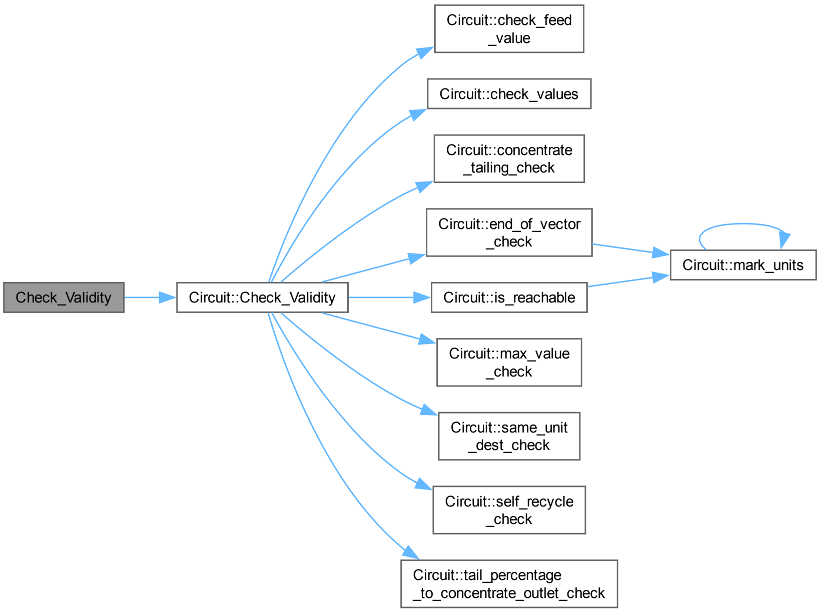
Returns

true if the circuit is valid, false otherwise.

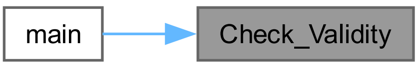
Definition at line 20 of file CCircuit.cpp.

```
00020                                                         {
00021     Circuit circuitInstance(1);
00022     return circuitInstance.Check_Validity(vector_size, circuit_vector);
00023 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.9.2 Variable Documentation

4.9.2.1 conc_outlet_reached

```
bool conc_outlet_reached
```

Definition at line 114 of file CCircuit.cpp.

4.9.2.2 inter_outlet_reached

```
bool inter_outlet_reached
```

Definition at line 114 of file CCircuit.cpp.

4.9.2.3 tails_outlet_reached

```
bool tails_outlet_reached
```

Definition at line 114 of file CCircuit.cpp.

4.10 CCircuit.cpp

Go to the documentation of this file.

```
00001
00002 #include <vector>
00003 #include <stdio.h>
00004 #include <CUnit.h>
00005 #include <CCircuit.h>
00006 #include <iostream>
00007
00008 using namespace std;
00009
00010 /**
00011  * @brief Check the validity of the circuit based on given criteria.
00012  *
00013  * This function creates an instance of the Circuit class and uses it to
00014  * validate the given circuit configuration.
00015  *
00016  * @param vector_size The size of the input vector.
00017  * @param circuit_vector The input vector representing the circuit configuration.
00018  * @return true if the circuit is valid, false otherwise.
00019  */
00020 bool Check_Validity(int vector_size, int *circuit_vector){
00021     Circuit circuitInstance(1);
00022     return circuitInstance.Check_Validity(vector_size, circuit_vector);
00023 }
00024
00025 /**
00026  * @brief Construct a new Circuit object with the specified number of units.
00027  *
00028  * @param num_units The number of units in the circuit.
00029  */
00030 Circuit::Circuit(int num_units) {
00031     // Initialize the circuit with a specified number of units.
00032     this->units.resize(num_units);
00033 }
00034
00035 /**
00036  * @brief Check the validity of the circuit based on given criteria.
00037  *
00038  * @param vector_size The size of the input vector.
00039  * @param circuit_vector The input vector representing the circuit configuration.
00040  * @return true if the circuit is valid, false otherwise.
00041  */
00042 bool Circuit::Check_Validity(int vector_size, int *circuit_vector) {
00043     // Calculate the size of the vector in terms of number of elements.
00044     int unit_size = 0;
00045
00046     // Determine the number of units and resize the units vector accordingly.
00047     if ((vector_size - 1) % 3 == 0) {
00048         this->units.resize(vector_size / 3);
00049         unit_size = vector_size / 3;
00050     } else {
00051         this->units.resize(vector_size / 3 + 1);
00052         unit_size = vector_size / 3 + 1;
00053     }
00054
00055     // Assign the values from the circuit_vector to the respective units.
00056     for (int i = 0; i < unit_size; ++i) {
00057         this->units[i].conc_num = circuit_vector[3 * i + 1];
00058         if (vector_size > 3 * i + 2)
```

```
            this->units[i].inter_num = circuit_vector[3 * i + 2];
        if (vector_size > 3 * i + 3)
            this->units[i].tails_num = circuit_vector[3 * i + 3];
        this->units[i].mark = false;
    }

    // Check if all required values are present in the circuit vector.
    if (!this->check_values(vector_size, circuit_vector)) {
        return false;
    }

    // Check if all units are reachable from the feed.
    if (!this->is_reachable()) {
        return false;
    }

    // Check if the concentrate and tailing streams are valid.
    if (!this->concentrate_tailing_check()) {
        return false;
    }

    // Check for self-recycles.
    if (!this->self_recycle_check()) {
        return false;
    }

    // Check if all product streams of a unit do not point to the same unit.
    if (!this->same_unit_dest_check()) {
        return false;
    }

    // Check if all units have a path to all outlets.
    if (!this->end_of_vector_check()) {
        return false;
    }

    // Check if the maximum value constraints are met.
    if (!this->max_value_check(vector_size, circuit_vector)) {
        return false;
    }

    // Check if the tailings percentage to the concentrate outlet is greater than 50%.
    if (!this->tail_percentage_to_concentrate_outlet_check()) {
        return false;
    }

    // Check if the feed value is within the valid range.
    if (!this->check_feed_value(circuit_vector)) {
        return false;
    }

    return true;
}

// Variables to track if each outlet has been reached.
bool conc_outlet_reached, inter_outlet_reached, tails_outlet_reached;

/**
 * @brief Mark units as reachable starting from a specific unit.
 *
 * @param unit_num The starting unit number.
 */
void Circuit::mark_units(int unit_num) {
    // If the unit is already marked, return.
    if (this->units[unit_num].mark)
        return;

    // Mark the current unit.
    this->units[unit_num].mark = true;

    // Recursively mark units reachable from conc_num.
    if (this->units[unit_num].conc_num < this->units.size()) {
        mark_units(this->units[unit_num].conc_num);
    } else {
        conc_outlet_reached = true;
    }

    // Recursively mark units reachable from inter_num.
    if (this->units[unit_num].inter_num < this->units.size()) {
        mark_units(this->units[unit_num].inter_num);
    } else {
        inter_outlet_reached = true;
    }

    // Recursively mark units reachable from tails_num.
    if (this->units[unit_num].tails_num < this->units.size()) {
        mark_units(this->units[unit_num].tails_num);
```

```
    } else {
        tails_outlet_reached = true;
    }
}

/**
 * @brief Check if all units are reachable from the feed.
 *
 * @return true if all units are reachable, false otherwise.
 */
bool Circuit::is_reachable() {

    // Reset the mark status for all units.
    for (auto& unit : this->units) {
        unit.mark = false;
    }

    // Mark units starting from the feed.
    mark_units(0);

    // Check if all units are marked as reachable.
    for (const auto& unit : this->units) {
        if (!unit.mark) {
            return false;
        }
    }

    return true;
}

/**
 * @brief Check if all units have a path to all outlets.
 *
 * @return true if all units have a path to all outlets, false otherwise.
 */
bool Circuit::end_of_vector_check() {
    // Reset the mark status for all units.
    for (auto& unit : this->units) {
        unit.mark = false;
    }

    // Mark units starting from the feed.
    mark_units(0);

    // Check if any unit has an invalid outlet path.
    for (auto& unit : this->units) {
        if (unit.conc_num == -1 ||
            unit.inter_num == -1 ||
            unit.tails_num == -1) {
            return false;
        }
    }

    return true;
}

/**
 * @brief Check for self-recycles in the circuit.
 *
 * @return true if no self-recycles exist, false otherwise.
 */
bool Circuit::self_recycle_check() {
    // Ensure no self-recycle exists.
    for (size_t i = 0; i < this->units.size(); ++i) {
        const auto& unit = this->units[i];
        // Check if any product stream points to the unit itself.
        if (unit.conc_num == i ||
            unit.inter_num == i ||
            unit.tails_num == i) {
            return false;
        }
    }
    return true;
}

/**
 * @brief Check if all product streams of a unit do not point to the same unit.
 *
 * @return true if the streams do not all point to the same unit, false otherwise.
 */
bool Circuit::same_unit_dest_check() {
    // Ensure the destinations for the three product streams of each unit are not all the same unit.
    for (const auto& unit : this->units) {
        // Check if all three product streams point to the same unit.
        if (unit.conc_num == unit.inter_num &&
            unit.inter_num == unit.tails_num) {
            return false;  // If they all point to the same unit, return false indicating invalid
```

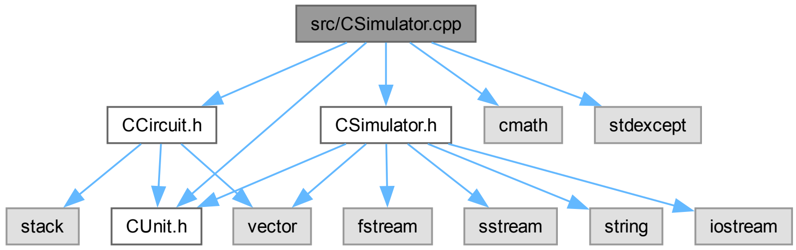
```
circuit.
        }
    }

    return true;  // If no such condition is found, return true indicating valid circuit.
}

/**
 * @brief Check if all required values are present in the circuit vector.
 *
 * @param vector_size The size of the input vector.
 * @param circuit_vector The input vector representing the circuit configuration.
 * @return true if all required values are present, false otherwise.
 */
bool Circuit::check_values(int vector_size, int *circuit_vector){
    // Find the max in the vector
    int max = 0;
    for (int i = 0; i < vector_size; i++){
        if (circuit_vector[i] > max){
            max = circuit_vector[i];
        }
    }

    // Check if the 0-max values appear in the vector
    for (int i = 0; i < max; i++){
        bool found = false;
        for (int j = 0; j < vector_size; j++){
            if (circuit_vector[j] == i){
                found = true;
                break;
            }
        }
        if (!found){
            return false;
        }
    }

    return true;
}

/**
 * @brief Check if the concentrate and tailing streams are valid.
 *
 * @return true if the concentrate and tailing streams are valid, false otherwise.
 */
bool Circuit::concentrate_tailing_check() {
    int num_units = units.size(); // Get the number of units
    bool has_concentrate = false; // Flag to check if at least one unit outputs to concentrate
    bool has_tailings = false;    // Flag to check if at least one unit outputs to tailings

    for (const auto& unit : units) {
        // Ensure intermediate or tailings streams do not output to concentrate
        if (unit.inter_num == num_units || unit.tails_num == num_units) {
            return false;
        }
        // Ensure concentrate or intermediate streams do not output to tailings
        if (unit.conc_num == num_units + 1 || unit.inter_num == num_units + 1) {
            return false;
        }
        // Check if any unit's concentrate stream outputs to concentrate
        if (unit.conc_num == num_units) {
            has_concentrate = true;
        }
        // Check if any unit's tailings stream outputs to tailings
        if (unit.tails_num == num_units + 1) {
            has_tailings = true;
        }
    }

    // Ensure at least one unit outputs to concentrate and tailings
    if (!has_concentrate || !has_tailings) {
        return false;
    }

    return true; // All checks passed
}

/**
 * @brief Check if the maximum value constraints are met.
 *
 * @param vector_size The size of the input vector.
 * @param circuit_vector The input vector representing the circuit configuration.
 * @return true if the maximum value constraints are met, false otherwise.
 */
bool Circuit::max_value_check(int vector_size, int *circuit_vector){
    // Find the max in the vector
    int cnt = this->units.size();
```

```cpp

    // Check some value exceeds the max
    for (const auto& unit : units) {
        if (unit.conc_num > cnt || unit.inter_num > cnt - 1 || unit.tails_num > cnt + 1) {
            return false;
        }
    }

    return true;
}

/**
 * @brief Check if the tailings percentage to the concentrate outlet is greater than 50%.
 *
 * @return true if the tailings percentage is within the limit, false otherwise.
 */
bool Circuit::tail_percentage_to_concentrate_outlet_check() {
    // Check if the tailings percentage is greater than 50%
    int max = this->units.size();
    int unit_conc = 0;

    for (const auto& unit : units) {
        if (unit.conc_num == max) {
            int cnt_tails = 0;
            int cnt = 0;

            for (const auto& unit1 : units) {
                if (unit1.tails_num == unit_conc) {
                    cnt_tails++;
                    cnt++;
                }
                if (unit1.conc_num == unit_conc) {
                    cnt++;
                }
                if (unit1.inter_num == unit_conc) {
                    cnt++;
                }
            }

            if (cnt_tails > cnt * 0.5) {
                return false;
            }
        }
        unit_conc++;
    }
    return true;
}

/**
 * @brief Check if the feed value is within the valid range (0 to units.size() - 1).
 *
 * @param circuit_vector The input vector representing the circuit configuration.
 * @return true if the feed value is within the valid range, false otherwise.
 */
bool Circuit::check_feed_value(int *circuit_vector) {
    // Check if the feed value is within the valid range (0 to units.size() - 1)

    int max = this->units.size();  // Get the number of units in the circuit
    if (circuit_vector[0] < 0 || circuit_vector[0] >= max) {  // Check if the first value is within the range
        return false;  // If not within the range, return false
    }
    return true;  // If within the range, return true
}
```

4.11 src/CSimulator.cpp File Reference

```cpp
#include "CUnit.h"
#include "CCircuit.h"
#include "CSimulator.h"
#include <cmath>
#include <stdexcept>
```

Include dependency graph for CSimulator.cpp:



Functions

- double Evaluate_Circuit (int vector_size, int ∗circuit_vector)

  Evaluates the performance of a circuit based on a given vector.
- double calculate_residence_time (const Calculate_constants &constants, double Fg, double Fw)

  Calculates the residence time of materials in a unit.
- struct Recovery calculate_recovery (const Calculate_constants &constants, double tau)

  Calculates the recovery of materials in a unit.
- std::vector< double > calculate_flow_rate (const Calculate_constants &constants, Recovery &recovery, double init_Fg, double init_Fw)

  Calculates the flow rates of materials in a unit.
- double get_performance (double cg, double cw, const Economic_parameters &eco)

  Calculates the performance of a circuit.
- std::vector< CUnit > vector_to_units (int ∗vector, int n, const Initial_flow &init_flow)

  Converts a vector to a vector of units.

4.11.1 Function Documentation

4.11.1.1 calculate_flow_rate()

```
std::vector< double > calculate_flow_rate (
            const Calculate_constants & constants,
            Recovery & recovery,
            double init_Fg,
            double init_Fw)
```

Calculates the flow rates of materials in a unit.

Calculates the flow rates.

Parameters

constants	The constants used for calculation.
recovery	The recovery of materials in the unit.
init_Fg	Initial flow rate of gerardium.
init_Fw	Initial flow rate of waste.

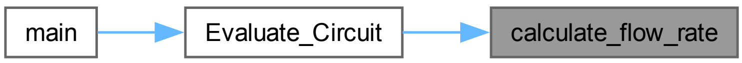
Returns

The calculated flow rates.

Definition at line 189 of file CSimulator.cpp.

```
{
  double cg = init_Fg * recovery.concentrate_gerardium;
  double cw = init_Fw * recovery.concentrate_waste;
  double ig = init_Fg * recovery.inter_gerardium;
  double iw = init_Fw * recovery.inter_waste;
  double tg = init_Fg - cg - ig;
  double tw = init_Fw - cw - iw;

  return {cg, cw, ig, iw, tg, tw};
};
```

Here is the caller graph for this function:



4.11.1.2 calculate_recovery()

```
struct Recovery calculate_recovery (
            const Calculate_constants & constants,
            double tau)
```

Calculates the recovery of materials in a unit.

Calculates the recovery rates.

Parameters

constants	The constants used for calculation.
tau	The residence time of materials in the unit.

Returns

The calculated recovery.

Definition at line 169 of file CSimulator.cpp.

```
{
  double rcg = (constants.k_concentrate_gerardium * tau) / (1 + (constants.k_concentrate_gerardium +
  constants.k_inter_gerardium) * tau);
  double rcw = (constants.k_concentrate_waste * tau) / (1 + (constants.k_concentrate_waste +
  constants.k_inter_waste) * tau);
  double rig = (constants.k_inter_gerardium * tau) / (1 + (constants.k_concentrate_gerardium +
  constants.k_inter_gerardium) * tau);
  double riw = (constants.k_inter_waste * tau) / (1 + (constants.k_concentrate_waste +
  constants.k_inter_waste) * tau);

  struct Recovery recoveries = {rcg, rcw, rig, riw};
  return recoveries;
};
```

Here is the caller graph for this function:

4.11.1.3 calculate_residence_time()

```
double calculate_residence_time (
            const Calculate_constants & constants,
            double Fg,
            double Fw)
```

Calculates the residence time of materials in a unit.

Calculates the residence time.

Parameters

constants	The constants used for calculation.
Fg	Flow rate of gerardium.
Fw	Flow rate of waste.

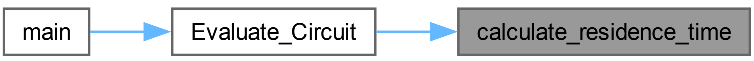
Returns

The calculated residence time.

Definition at line 153 of file CSimulator.cpp.

```
00154 {
00155   double total_mass_flow_rate = Fw + Fg;
00156   total_mass_flow_rate = std::max(total_mass_flow_rate, 1e-10);
00157   double volume_flow_rate = total_mass_flow_rate / constants.rho;
00158   double tau = constants.phi * constants.V / volume_flow_rate;
00159   return tau;
00160 };
```

Here is the caller graph for this function:



4.11.1.4 Evaluate_Circuit()

```
double Evaluate_Circuit (
            int vector_size,
            int * circuit_vector)
```

Evaluates the performance of a circuit based on a given vector.

Evaluates the circuit performance.

This function simulates the operation of a circuit and calculates its performance. Judge if the circuit has converged (using the difference between the old and new flow rates), if not, set the performance to 90 ∗ -750.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The array representing the circuit configuration.

Returns

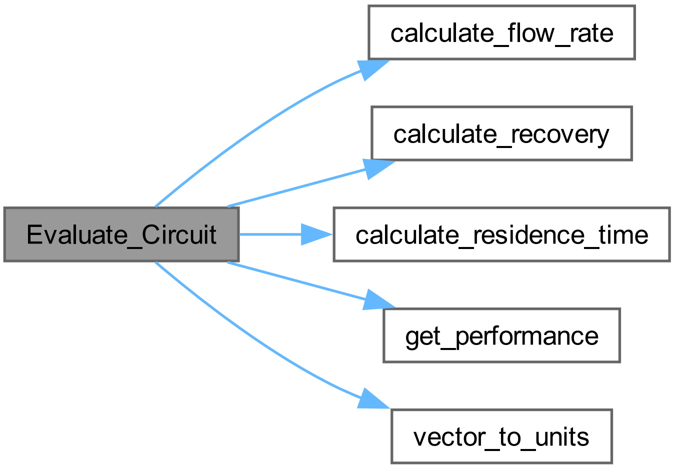
The performance value of the circuit.

Definition at line 20 of file CSimulator.cpp.

```
{
  struct Circuit_Parameters default_circuit_parameters;
  struct Calculate_constants constants;
  struct Initial_flow init_flow;
  struct Economic_parameters eco;

  double Performance = 0.0;
  double Recovery = 0.0;
  double Grade = 0.0;

  // Calculate the number of units in the circuit
  int length = (vector_size - 1) / 3;
  std::vector<CUnit> units = vector_to_units(circuit_vector, length, init_flow);
  int i;
  for (i = 0; i < default_circuit_parameters.max_iterations; i++)
  {
    double concentrate_gerardium = 0.0;
    double concentrate_waste = 0.0;

    // Update the flow rates for the units
    #pragma omp parallel for reduction(+:concentrate_gerardium, concentrate_waste)
    for (int j = 0; j < length; j++)
    {
      double tau = calculate_residence_time(constants, units[j].old_flow_W, units[j].old_flow_G);
      struct Recovery recovery = calculate_recovery(constants, tau);
      std::vector<double> all_flow_rate = calculate_flow_rate(constants, recovery,
   units[j].old_flow_G, units[j].old_flow_W);

      if (units[j].conc_num < length)
      {
        #pragma omp atomic
        units[units[j].conc_num].new_flow_G += all_flow_rate[0];
        #pragma omp atomic
        units[units[j].conc_num].new_flow_W += all_flow_rate[1];
      }
      else if (units[j].conc_num == length) // If the unit points to the concentrate stream
      {
        concentrate_gerardium += all_flow_rate[0];
        concentrate_waste += all_flow_rate[1];
      }

      if (units[j].inter_num < length)
      {
        #pragma omp atomic
        units[units[j].inter_num].new_flow_G += all_flow_rate[2];
        #pragma omp atomic
        units[units[j].inter_num].new_flow_W += all_flow_rate[3];
      }

      if (units[j].tails_num < length)
```

```
          {
            #pragma omp atomic
            units[units[j].tails_num].new_flow_G += all_flow_rate[4];
            #pragma omp atomic
            units[units[j].tails_num].new_flow_W += all_flow_rate[5];
          }
        }

        // Add initial flow to the start unit
        int start = circuit_vector[0];
        units[start].new_flow_G += init_flow.init_Fg;
        units[start].new_flow_W += init_flow.init_Fw;

        // Judge if the circuit has converged
        bool converge = true;
        for (int j = 0; j < length; j++)
        {
          // Calculate the relative difference between the old and new flow rates
          double diff_fg = std::abs(units[j].new_flow_G - units[j].old_flow_G) / units[j].old_flow_G;
          double diff_fw = std::abs(units[j].new_flow_W - units[j].old_flow_W) / units[j].old_flow_W;

          if ((diff_fg > 1e-6 || diff_fw > 1e-6))
          {
            converge = false;
            break;
          }
        }

        if (converge)
        {
          // std::cout « i « std::endl;
          // std::cout « concentrate_gerardium « " " « concentrate_waste « std::endl;
          // You can remove the comments above to see the number of iterations and the final concentrate values
          Performance = get_performance(concentrate_gerardium, concentrate_waste, eco);
          break;
        }
        else
        {
          for (int j = 0; j < length; j++)
          {
            units[j].old_flow_G = units[j].new_flow_G;
            units[j].old_flow_W = units[j].new_flow_W;
            units[j].new_flow_G = 0.0;
            units[j].new_flow_W = 0.0;
          }
        }
        // Calculate the recovery and grade of the circuit
        Recovery = concentrate_gerardium / init_flow.init_Fg;
        Grade = concentrate_gerardium / (concentrate_gerardium + concentrate_waste);
      }

      // If the circuit does not converge, set the performance to 90 * -750
      if (i == 1000)
      {
        Performance = init_flow.init_Fw * eco.penalty;
      }

      try
      {
        // Write the performance, recovery, and grade to a file
        std::ofstream outFile("./output/performance.dat");

        outFile « Performance « "\n";
        outFile « Recovery « "\n";
        outFile « Grade « "\n";

        outFile.close();
      }
      catch (const std::exception &e)
      {
      }

      return Performance;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.11.1.5 get_performance()

```
double get_performance (
            double cg,
            double cw,
            const Economic_parameters & eco)
```

Calculates the performance of a circuit.

Gets the performance of the circuit.

Parameters

cg	Concentrate grade of gerardium.
cw	Concentrate grade of waste.
eco	Economic parameters.

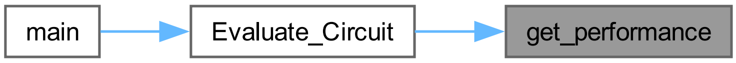
Returns

The calculated performance.

Definition at line 209 of file CSimulator.cpp.

```
{
  double result = cg * eco.price + cw * eco.penalty;
  return result;
};
```

Here is the caller graph for this function:



4.11.1.6 vector_to_units()

```
std::vector< CUnit > vector_to_units (
            int * vector,
            int n,
            const Initial_flow & init_flow)
```

Converts a vector to a vector of units.

Converts a vector to a vector of CUnit objects.

Parameters

vector	The vector to be converted.
n	The size of the vector.
init_flow	The initial flow rates.

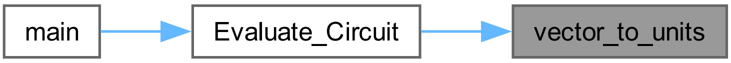
Returns

The vector of units.

Definition at line 223 of file CSimulator.cpp.

```
{
  std::vector<CUnit> units(n);
  for (int i = 0; i < n; i++)
  {
    units[i].old_flow_G = init_flow.init_Fg;
    units[i].old_flow_W = init_flow.init_Fw;
    units[i].new_flow_G = 0;
    units[i].new_flow_W = 0;
    units[i].conc_num = vector[3 * i + 1];
    units[i].inter_num = vector[3 * i + 2];
    units[i].tails_num = vector[3 * i + 3];
  }
  return units;
};
```

Here is the caller graph for this function:



4.12 CSimulator.cpp

Go to the documentation of this file.

```
#include "CUnit.h"
#include "CCircuit.h"
#include "CSimulator.h"

#include <cmath>
#include <stdexcept>

/**
 * @brief Evaluates the performance of a circuit based on a given vector.
 *
 * This function simulates the operation of a circuit and calculates its performance.
 * Judge if the circuit has converged (using the difference between the old and new flow rates),
 * if not, set the performance to 90 * -750.
 *
 * @param vector_size The size of the circuit vector.
 * @param circuit_vector The array representing the circuit configuration.
 * @return The performance value of the circuit.
 */

double Evaluate_Circuit(int vector_size, int *circuit_vector)
{
  struct Circuit_Parameters default_circuit_parameters;
  struct Calculate_constants constants;
  struct Initial_flow init_flow;
  struct Economic_parameters eco;

  double Performance = 0.0;
  double Recovery = 0.0;
  double Grade = 0.0;

  // Calculate the number of units in the circuit
  int length = (vector_size - 1) / 3;
  std::vector<CUnit> units = vector_to_units(circuit_vector, length, init_flow);
  int i;
  for (i = 0; i < default_circuit_parameters.max_iterations; i++)
  {
    double concentrate_gerardium = 0.0;
    double concentrate_waste = 0.0;

    // Update the flow rates for the units
    #pragma omp parallel for reduction(+:concentrate_gerardium, concentrate_waste)
    for (int j = 0; j < length; j++)
    {
      double tau = calculate_residence_time(constants, units[j].old_flow_W, units[j].old_flow_G);
      struct Recovery recovery = calculate_recovery(constants, tau);
      std::vector<double> all_flow_rate = calculate_flow_rate(constants, recovery, units[j].old_flow_G, units[j].old_flow_W);

      if (units[j].conc_num < length)
      {
        #pragma omp atomic
        units[units[j].conc_num].new_flow_G += all_flow_rate[0];
        #pragma omp atomic
        units[units[j].conc_num].new_flow_W += all_flow_rate[1];
      }
      else if (units[j].conc_num == length) // If the unit points to the concentrate stream
      {
        concentrate_gerardium += all_flow_rate[0];
        concentrate_waste += all_flow_rate[1];
      }

```

```
      if (units[j].inter_num < length)
      {
        #pragma omp atomic
        units[units[j].inter_num].new_flow_G += all_flow_rate[2];
        #pragma omp atomic
        units[units[j].inter_num].new_flow_W += all_flow_rate[3];
      }

      if (units[j].tails_num < length)
      {
        #pragma omp atomic
        units[units[j].tails_num].new_flow_G += all_flow_rate[4];
        #pragma omp atomic
        units[units[j].tails_num].new_flow_W += all_flow_rate[5];
      }
    }

    // Add initial flow to the start unit
    int start = circuit_vector[0];
    units[start].new_flow_G += init_flow.init_Fg;
    units[start].new_flow_W += init_flow.init_Fw;

    // Judge if the circuit has converged
    bool converge = true;
    for (int j = 0; j < length; j++)
    {
      // Calculate the relative difference between the old and new flow rates
      double diff_fg = std::abs(units[j].new_flow_G - units[j].old_flow_G) / units[j].old_flow_G;
      double diff_fw = std::abs(units[j].new_flow_W - units[j].old_flow_W) / units[j].old_flow_W;

      if ((diff_fg > 1e-6 || diff_fw > 1e-6))
      {
        converge = false;
        break;
      }
    }

    if (converge)
    {
      // std::cout « i « std::endl;
      // std::cout « concentrate_gerardium « " " « concentrate_waste « std::endl;
      // You can remove the comments above to see the number of iterations and the final concentrate values
      Performance = get_performance(concentrate_gerardium, concentrate_waste, eco);
      break;
    }
    else
    {
      for (int j = 0; j < length; j++)
      {
        units[j].old_flow_G = units[j].new_flow_G;
        units[j].old_flow_W = units[j].new_flow_W;
        units[j].new_flow_G = 0.0;
        units[j].new_flow_W = 0.0;
      }
    }
    // Calculate the recovery and grade of the circuit
    Recovery = concentrate_gerardium / init_flow.init_Fg;
    Grade = concentrate_gerardium / (concentrate_gerardium + concentrate_waste);
  }

  // If the circuit does not converge, set the performance to 90 * -750
  if (i == 1000)
  {
    Performance = init_flow.init_Fw * eco.penalty;
  }

  try
  {
    // Write the performance, recovery, and grade to a file
    std::ofstream outFile("./output/performance.dat");

    outFile « Performance « "\n";
    outFile « Recovery « "\n";
    outFile « Grade « "\n";

    outFile.close();
  }
  catch (const std::exception &e)
  {
  }

  return Performance;
}

/**
 * @brief Calculates the residence time of materials in a unit.
```

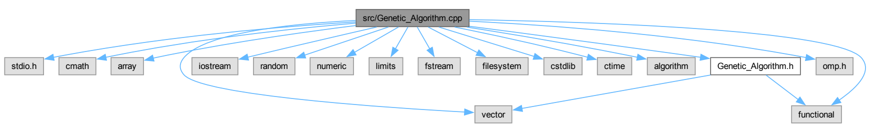
```
 *
 * @param constants The constants used for calculation.
 * @param Fg Flow rate of gerardium.
 * @param Fw Flow rate of waste.
 * @return The calculated residence time.
 */
double calculate_residence_time(const Calculate_constants &constants, double Fg, double Fw)
{
  double total_mass_flow_rate = Fw + Fg;
  total_mass_flow_rate = std::max(total_mass_flow_rate, 1e-10);
  double volume_flow_rate = total_mass_flow_rate / constants.rho;
  double tau = constants.phi * constants.V / volume_flow_rate;
  return tau;
};

/**
 * @brief Calculates the recovery of materials in a unit.
 *
 * @param constants The constants used for calculation.
 * @param tau The residence time of materials in the unit.
 * @return The calculated recovery.
 */
struct Recovery calculate_recovery(const Calculate_constants &constants, double tau)
{
  double rcg = (constants.k_concentrate_gerardium * tau) / (1 + (constants.k_concentrate_gerardium + constants.k_inter_gerardium) * tau);
  double rcw = (constants.k_concentrate_waste * tau) / (1 + (constants.k_concentrate_waste + constants.k_inter_waste) * tau);
  double rig = (constants.k_inter_gerardium * tau) / (1 + (constants.k_concentrate_gerardium + constants.k_inter_gerardium) * tau);
  double riw = (constants.k_inter_waste * tau) / (1 + (constants.k_concentrate_waste + constants.k_inter_waste) * tau);

  struct Recovery recoveries = {rcg, rcw, rig, riw};
  return recoveries;
};

/**
 * @brief Calculates the flow rates of materials in a unit.
 *
 * @param constants The constants used for calculation.
 * @param recovery The recovery of materials in the unit.
 * @param init_Fg Initial flow rate of gerardium.
 * @param init_Fw Initial flow rate of waste.
 * @return The calculated flow rates.
 */
std::vector<double> calculate_flow_rate(const Calculate_constants &constants, Recovery &recovery, double init_Fg, double init_Fw)
{
  double cg = init_Fg * recovery.concentrate_gerardium;
  double cw = init_Fw * recovery.concentrate_waste;
  double ig = init_Fg * recovery.inter_gerardium;
  double iw = init_Fw * recovery.inter_waste;
  double tg = init_Fg - cg - ig;
  double tw = init_Fw - cw - iw;

  return {cg, cw, ig, iw, tg, tw};
};

/**
 * @brief Calculates the performance of a circuit.
 *
 * @param cg Concentrate grade of gerardium.
 * @param cw Concentrate grade of waste.
 * @param eco Economic parameters.
 * @return The calculated performance.
 */
double get_performance(double cg, double cw, const Economic_parameters &eco)
{
  double result = cg * eco.price + cw * eco.penalty;
  return result;
};

/**
 * @brief Converts a vector to a vector of units.
 *
 * @param vector The vector to be converted.
 * @param n The size of the vector.
 * @param init_flow The initial flow rates.
 * @return The vector of units.
 */
std::vector<CUnit> vector_to_units(int *vector, int n, const Initial_flow &init_flow)
{
  std::vector<CUnit> units(n);
  for (int i = 0; i < n; i++)
  {
    units[i].old_flow_G = init_flow.init_Fg;
```

```
    units[i].old_flow_W = init_flow.init_Fw;
    units[i].new_flow_G = 0;
    units[i].new_flow_W = 0;
    units[i].conc_num = vector[3 * i + 1];
    units[i].inter_num = vector[3 * i + 2];
    units[i].tails_num = vector[3 * i + 3];
  }
  return units;
};
```

4.13 src/Genetic_Algorithm.cpp File Reference

```
#include <stdio.h>
#include <cmath>
#include <array>
#include <vector>
#include <iostream>
#include <random>
#include <numeric>
#include <limits>
#include <fstream>
#include <filesystem>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <functional>
#include <omp.h>
#include "Genetic_Algorithm.h"
```

Include dependency graph for Genetic_Algorithm.cpp:



Macros

- #define PBSTR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
- #define PBWIDTH 60

Functions

- void printProgress (double percentage, double performance)
- double find_max_double (const double *array, int size)
- double generate_random_number (double min, double max)
- std::vector< std::vector< int > > initialize_population (int population_size, int vector_size, const int *initial_vector, std::function< bool(int, int *)> validity, double elitism_rate)
- std::vector< std::vector< int > > select (const std::vector< std::vector< int > > &population, const std::vector< double > &fitness)
- void NonUniform_Mutation (std::vector< int > &individual, double mutation_rate, int max_value, int currentGeneration, int maxGenerations)
- void mutate_vector (std::vector< int > &vector, double mutation_rate, int max_unit)

- void crossover (std::vector< int > &parent1, std::vector< int > &parent2, double crossover_rate, int max_value)
- int select_index (const std::vector< double > &cumulative_fitness)
- void regenerate_population (std::vector< std::vector< int > > &population, int vector_size, int number_of_units)
- double genetic_algorithm (std::vector< std::vector< int > > &population, double(&func)(int, int *), std::function< bool(int, int *)> validity, const Algorithm_Parameters &parameters)

  Performs a genetic algorithm optimization.
- int optimize (int vector_size, int *vector, double(&func)(int, int *), std::function< bool(int, int *)> validity, struct Algorithm_Parameters parameters)

  Optimizes a vector using the genetic algorithm.

Variables

- int number_of_units = 1

4.13.1 Macro Definition Documentation

4.13.1.1 PBSTR

```
#define PBSTR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
```

Definition at line 22 of file Genetic_Algorithm.cpp.

4.13.1.2 PBWIDTH

```
#define PBWIDTH 60
```

Definition at line 23 of file Genetic_Algorithm.cpp.

4.13.2 Function Documentation

4.13.2.1 crossover()

```
void crossover (
            std::vector< int > & parent1,
            std::vector< int > & parent2,
            double crossover_rate,
            int max_value)
```

Performs a single-point crossover between two parent vectors.

Parameters

parent1	The first parent vector.
parent2	The second parent vector.
crossover_rate	The probability of performing a crossover.
max_value	The maximum value for any gene in the vectors.

Definition at line 215 of file Genetic_Algorithm.cpp.

```
00215
      {
00216
00217     if (generate_random_number(0.0, 1.0) < crossover_rate) {
00218         std::uniform_int_distribution<> point_dist(1, parent1.size() - 2);
00219         int crossover_point = static_cast<int>(generate_random_number(1, parent1.size() - 2));
00220
00221         for (int i = crossover_point; i < parent1.size(); ++i) {
00222             std::swap(parent1[i], parent2[i]);
00223         }
00224     }
00225 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

4.13.2.2 find_max_double()

```
double find_max_double (
            const double * array,
            int size)
```

Finds the maximum value in a double array.

Parameters

array	Pointer to the first element of the double array.
size	Size of the array.

Returns

The maximum value found in the array.

Definition at line 48 of file Genetic_Algorithm.cpp.

```
00048                                                        {
00049     double max_value = array[0];
00050     for (int i = 1; i < size; ++i) {
00051         if (array[i] > max_value) {
00052             max_value = array[i];
00053         }
00054     }
00055     return max_value;
00056 }
```

Here is the caller graph for this function:

4.13.2.3 generate_random_number()

```
double generate_random_number (
            double min,
            double max)
```

Generates a random number within a specified range.

Parameters

min	The lower bound of the range.
max	The upper bound of the range.

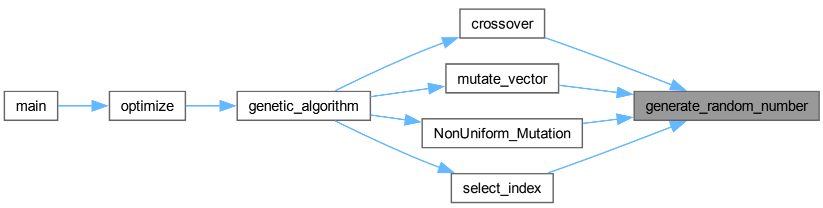
Returns

A randomly generated number within the specified range.

Definition at line 66 of file Genetic_Algorithm.cpp.

```
00066                                                        {
00067     static thread_local std::mt19937 generator(omp_get_thread_num());
00068     std::uniform_real_distribution<double> distribution(min, max);
00069     return distribution(generator);
00070 }
```

Here is the caller graph for this function:



4.13.2.4 genetic_algorithm()

```
double genetic_algorithm (
            std::vector< std::vector< int > > & population,
            double(&)(int, int *) func,
            std::function< bool(int, int *)> validity,
            const Algorithm_Parameters & parameters)
```

Performs a genetic algorithm optimization.

Conducts the entire genetic algorithm process, managing the population through multiple generations and applying genetic operations like selection, crossover, and mutation to evolve solutions.

Parameters

population	The initial population of solutions.
func	A function pointer to the fitness evaluation function.
validity	A function to check the validity of individual solutions.
parameters	Struct containing parameters for the genetic algorithm.

Returns

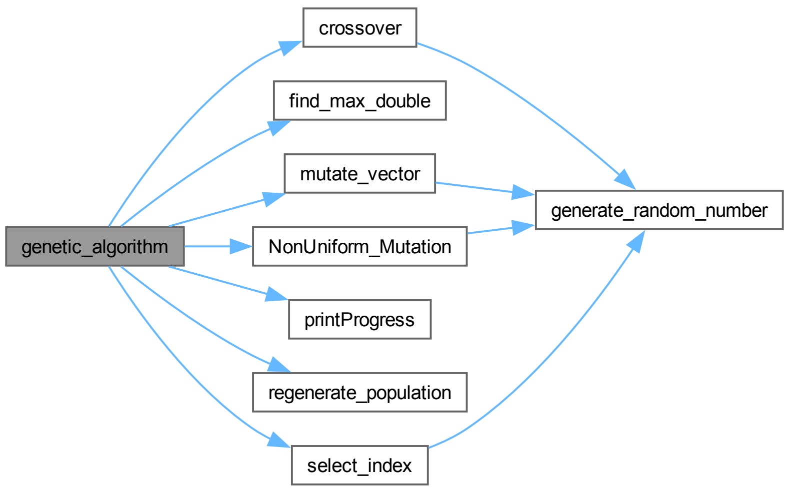
The maximum fitness achieved by the best solution in the population.

Definition at line 272 of file Genetic_Algorithm.cpp.

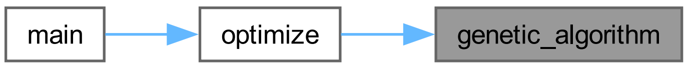
```
                                                                    {
    int population_size = population.size();
    int vector_size = population[0].size();
    std::vector<double> fitness(population_size);
    double max_fitness = std::numeric_limits<double>::lowest();
    int fitness_unchanged_count = 0;

    int elitism_count = static_cast<int>(population.size() * parameters.elitism_rate);

    for (int generation = 0; generation < parameters.max_iterations; ++generation) {
        // Evaluate fitness for each vector in the population
        #pragma omp parallel for
        for (int i = 0; i < population_size; ++i) {
            if (validity(vector_size, population[i].data())) {
                fitness[i] = func(vector_size, population[i].data());
            } else {
                fitness[i] = std::numeric_limits<double>::lowest();
            }
        }

        // Sort the population based on fitness
        std::vector<int> idx(population_size);
        std::iota(idx.begin(), idx.end(), 0);
        std::sort(idx.begin(), idx.end(), [&](int i1, int i2) { return fitness[i1] > fitness[i2]; });

        printProgress((double)generation / (parameters.max_iterations - 1), fitness[idx[0]]);

        // Implement elitism, save the best individuals
        std::vector<std::vector<int» new_population;
        for (int i = 0; i < elitism_count; ++i) {
            new_population.push_back(population[idx[i]]);
        }

        // Create a cumulative fitness sum for roulette wheel selection
        std::vector<double> cumulative_fitness(population_size);
        std::partial_sum(fitness.begin(), fitness.end(), cumulative_fitness.begin());

        // #pragma omp parallel for
        for (int i = elitism_count; i < population.size(); i+=2) {
            std::vector<int> parent1 = population[select_index(cumulative_fitness)];
            std::vector<int> parent2 = population[select_index(cumulative_fitness)];

            crossover(parent1, parent2, parameters.crossover_rate, number_of_units);
            crossover(parent1, parent2, parameters.crossover_rate, number_of_units);

            // if (vector_size > 100) {
```

```
            //      while (!validity(vector_size, parent1.data()) && !validity(vector_size, parent2.data())) {
            //          crossover(parent1, parent2, parameters.crossover_rate, number_of_units);
            //      }
            // }

            NonUniform_Mutation(parent1, parameters.mutation_rate, number_of_units, generation, parameters.max_iterations);
            NonUniform_Mutation(parent2, parameters.mutation_rate, number_of_units, generation, parameters.max_iterations);
            double mutator = 0.0;

            if (fitness_unchanged_count > (parameters.max_iterations * 0.1)) {
                mutator = parameters.mutation_rate + (fitness_unchanged_count * 0.001);
                mutator = mutator < 0.5 ? mutator : 0.5;
            } else {

                mutator = parameters.mutation_rate;
            }
            mutate_vector(parent1, mutator, number_of_units);
            mutate_vector(parent2, mutator, number_of_units);

            new_population.push_back(parent1);
            if (new_population.size() < population_size) {
                new_population.push_back(parent2);
            }
        }

        double temp_fitness = find_max_double(fitness.data(), fitness.size());
        if (temp_fitness - max_fitness < 0.1) {
            fitness_unchanged_count++;
        }
        else{
            fitness_unchanged_count = 0;
        }
        if (temp_fitness > max_fitness) {
          fs::path dir("./output");
          if (!fs::exists(dir)) {
              fs::create_directories(dir);
          }
          std::ofstream vector_file("./output/vector.dat");
          if (vector_file.is_open()) {
              for (int i = 0; i < vector_size; i++) {
                  vector_file « population[0][i] « " ";
              }
              vector_file.close();
          }

        }
        population = new_population;
        if (fitness_unchanged_count > 50) {
            regenerate_population(population, vector_size, number_of_units);
            fitness_unchanged_count = 0;
        }
        #pragma omp barrier
        max_fitness = *std::max_element(fitness.begin(), fitness.end());
    }
    std::cout « std::endl;
    return max_fitness;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.13.2.5 initialize_population()

```
std::vector< std::vector< int > > initialize_population (
             int population_size,
             int vector_size,
             const int * initial_vector,
             std::function< bool(int, int *)> validity,
             double elitism_rate)
```

Initializes a population for the genetic algorithm.

Parameters

population_size	The size of the population to initialize.
vector_size	The size of each individual in the population.
initial_vector	Initial values for the first individual in the population.
validity	A function that checks the validity of an individual.
elitism_rate	The rate of elitism to apply during evolution.

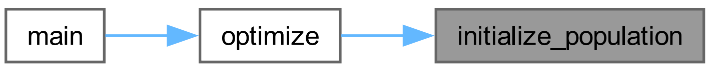
Returns

A vector of vectors containing the initialized population.

Definition at line 83 of file Genetic_Algorithm.cpp.

```
00083 {
00084     std::vector<std::vector<int>> population;
00085     population.reserve(population_size);  // Reserve space to avoid reallocations
00086 
00087     std::vector<int> start_vector(initial_vector, initial_vector + vector_size);
00088     population.push_back(start_vector);
00089 
00090     number_of_units = *std::max_element(initial_vector, initial_vector + vector_size) + 1;
00091 
00092     #pragma omp parallel
00093     {
00094         std::random_device rd;
00095         std::mt19937 gen(rd() + omp_get_thread_num());  // Unique seed for each thread
00096         std::uniform_int_distribution<> distr(0, number_of_units - 1);
00097 
00098         #pragma omp for
00099         for (int i = 1; i < population_size; ++i) {
00100             std::vector<int> individual(vector_size);
00101             for (int j = 0; j < vector_size; ++j) {
00102                 individual[j] = distr(gen);
00103 
00104                 bool valid = true;
00105                 if (j == 0) {
00106                     if (individual[j] == number_of_units - 2 || individual[j] == number_of_units - 3) {
00107                         valid = false;
00108                     }
00109                 } else if ((j - 1) / 3 == individual[j]) {
00110                     valid = false;
00111                 }
00112 
00113                 if (!valid) {
00114                     j--;
00115                 }
00116             }
00117 
00118             #pragma omp critical
00119             {
00120                 if (validity(vector_size, individual.data()) || population.size() < population_size * 0.8) {
00121                     population.push_back(individual);
00122                 } else {
00123                     --i;  // Retry this iteration
00124                 }
00125             }
00126         }
00127     }
00128 
00129     return population;
00130 }
```

Here is the caller graph for this function:



4.13.2.6 mutate_vector()

```
void mutate_vector (
            std::vector< int > & vector,
```

```
double mutation_rate,
int max_unit)
```

Mutates a given vector with a specified mutation rate. Each element in the vector has a chance to be changed based on the mutation rate.

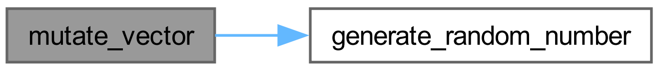
Parameters

vector	The vector to mutate.
mutation_rate	The probability of mutating each element of the vector.
max_unit	The maximum value any element in the vector can take.

Definition at line 197 of file Genetic_Algorithm.cpp.

```
{

    for (int& value : vector) {
        if (generate_random_number(0.0, 1.0) < mutation_rate) {
            value = (value + static_cast<int>(generate_random_number(0, max_unit))) % (max_unit + 1);
        }
    }
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.13.2.7 NonUniform_Mutation()

```
void NonUniform_Mutation (
            std::vector< int > & individual,
            double mutation_rate,
            int max_value,
            int currentGeneration,
            int maxGenerations)
```

Applies a non-uniform mutation to an individual in the population. Mutation depends on the current generation, allowing for finer mutations as the number of generations increases.

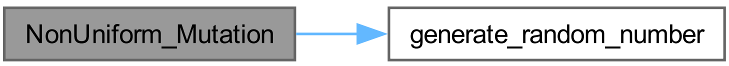
Parameters

individual	A reference to the individual (vector of ints) to mutate.
mutation_rate	The mutation rate to apply.
max_value	The maximum value for any gene in the individual.
currentGeneration	The current generation number in the genetic algorithm.
maxGenerations	The maximum number of generations expected to run.

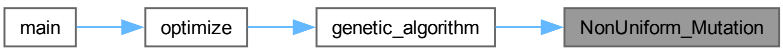
Definition at line 171 of file Genetic_Algorithm.cpp.

```
00171
        {
00172
00173     for (int& gene : individual) {
00174         if (generate_random_number(0.0, 1.0) < mutation_rate) {
00175             double delta = (generate_random_number(0.0, 1.0) < 0.5) ? gene : max_value - gene;
00176             double b = 5;
00177             double r = generate_random_number(0.0, 1.0);
00178
00179             double change = delta * (1 - pow(r, pow((1 - double(currentGeneration) / maxGenerations),
        b)));
00180             gene = (generate_random_number(0.0, 1.0) < 0.5) ? gene - static_cast<int>(change) : gene +
        static_cast<int>(change);
00181
00182             if (gene < 0) gene = 0;
00183             if (gene > max_value) gene = max_value;
00184         }
00185     }
00186 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.13.2.8 optimize()

```
int optimize (
            int vector_size,
            int * vector,
            double(&)(int, int *) func,
            std::function< bool(int, int *)> validity,
            struct Algorithm_Parameters parameters)
```

Optimizes a vector using the genetic algorithm.

Optimizes a vector using genetic algorithm principles. Initializes a population, runs the genetic algorithm, and stores the best solution back into the original vector.

Parameters

vector_size	The size of the vector to be optimized.
vector	The vector containing initial values, modified in-place to store the best solution found.
func	A function pointer to the fitness evaluation function.
validity	A function to check the validity of individual solutions.
parameters	Struct containing parameters for the genetic algorithm.

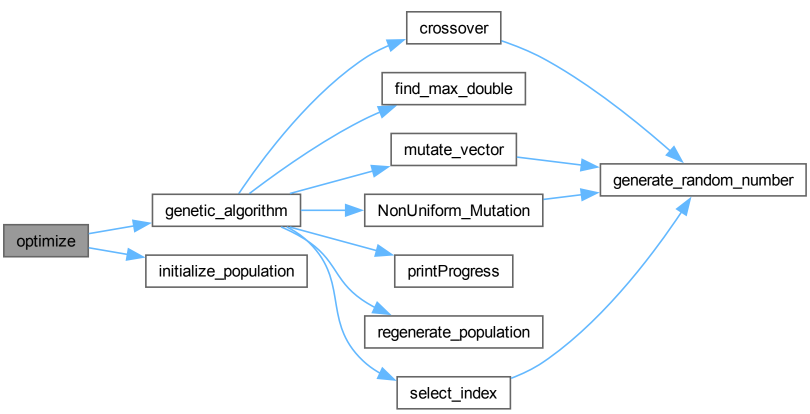
Returns

Returns 0 on successful execution and optimization, -1 if file operation fails.

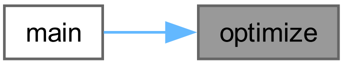
Definition at line 390 of file Genetic_Algorithm.cpp.

```
                                                             {
    // print the number of threads
    std::cout « "Number of threads: " « omp_get_max_threads() « std::endl;

    int unit_num = (vector_size - 1) / 3;

    for (int i = 0; i <= unit_num+1; ++i){
        vector[i] = i;
    }
    for (int i = unit_num+2; i < vector_size; ++i){
        vector[i] = 0;
    }

    std::vector<std::vector<int» population = initialize_population(parameters.initial_pop,
vector_size, vector, validity, parameters.elitism_rate);
    double max_fitness = genetic_algorithm(population, func, validity, parameters);
    std::copy(population[0].begin(), population[0].end(), vector);

    std::ofstream vector_file("./output/vector.dat");
    if (vector_file.is_open()) {
        for (int i = 0; i < vector_size; i++) {
            vector_file « vector[i] « " ";
        }
        vector_file.close();
    } else {
        return -1;
    }

    return 0;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.13.2.9 printProgress()

```
void printProgress (
            double percentage,
            double performance)
```

Prints the current progress of an operation to the console.

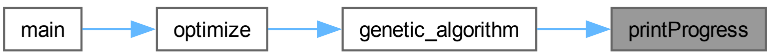
Parameters

percentage	The completion percentage of the operation.
performance	A floating point value indicating current performance metrics.

Definition at line 32 of file Genetic_Algorithm.cpp.

```
00032                                                          {
00033     int val = (int)(percentage * 100);
00034     int lpad = (int)(percentage * PBWIDTH);
00035     int rpad = PBWIDTH - lpad;
00036     printf("\r%3d%% [%.*s%*s] Item %.2f", val, lpad, PBSTR, rpad, "", performance);
00037     fflush(stdout);
00038 }
```

Here is the caller graph for this function:



4.13.2.10 regenerate_population()

```
void regenerate_population (
            std::vector< std::vector< int > > & population,
            int vector_size,
            int number_of_units)
```

Regenerates the population by introducing new random vectors to replace the less fit individuals, aiming to introduce diversity and prevent premature convergence.

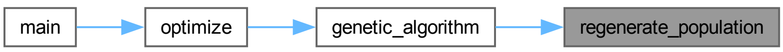
Parameters

population	The population of vectors.
vector_size	The size of each vector.
number_of_units	The maximum value for any gene in the vectors.

Definition at line 248 of file Genetic_Algorithm.cpp.

```
00248
      {
00249     #pragma omp parallel for
00250     for (int i = (int) (population.size() * 0.2); i < population.size(); ++i) {  // Start from 1 to
      keep the first vector unchanged
00251         std::random_device rd;
00252         std::mt19937 gen(rd() + omp_get_thread_num()); // Ensuring unique seed per thread
00253         std::uniform_int_distribution<> distr(0, number_of_units - 1);
00254 
00255         for (int j = 0; j < vector_size; ++j) {
00256             population[i][j] = distr(gen);
00257         }
00258     }
00259 }
```

Here is the caller graph for this function:



4.13.2.11 select()

```
std::vector< std::vector< int > > select (
            const std::vector< std::vector< int > > & population,
            const std::vector< double > & fitness)
```

Selects individuals from the population based on their fitness.

Parameters

population	A reference to the current population.
fitness	A vector containing fitness scores for each individual.

Returns

A vector of vectors containing the selected individuals.

Definition at line 140 of file Genetic_Algorithm.cpp.

```
00140
      {
00141     std::vector<std::vector<int>> selected;
00142     double total_fitness = std::accumulate(fitness.begin(), fitness.end(), 0.0);
00143     std::vector<double> probabilities;
00144 
00145     std::transform(fitness.begin(), fitness.end(), probabilities.begin(), [total_fitness](double f) {
00146         return f / total_fitness;
00147     });
00148 
```

```
    std::discrete_distribution<int> distribution(probabilities.begin(), probabilities.end());
    std::random_device rd;
    std::mt19937 gen(rd());

    for (size_t i = 0; i < population.size(); ++i) {
        selected.push_back(population[distribution(gen)]);
    }

    return selected;
}
```

4.13.2.12 select_index()

```
int select_index (
            const std::vector< double > & cumulative_fitness)
```

Selects an index for roulette wheel selection based on cumulative fitness scores.

Parameters

cumulative_fitness	A vector of cumulative fitness scores.

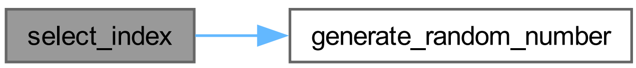
Returns

The selected index based on the random choice in the cumulative distribution.

Definition at line 234 of file Genetic_Algorithm.cpp.

```
00234                                                           {
00235     double rnd = generate_random_number(0.0, cumulative_fitness.back());
00236     return std::lower_bound(cumulative_fitness.begin(), cumulative_fitness.end(), rnd) -
      cumulative_fitness.begin();
00237 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.13.3 Variable Documentation

4.13.3.1 number_of_units

```
int number_of_units = 1
```

Definition at line 20 of file Genetic_Algorithm.cpp.

4.14 Genetic_Algorithm.cpp

Go to the documentation of this file.

```
00001 #include <stdio.h>
00002 #include <cmath>
00003 #include <array>
00004 #include <vector>
00005 #include <iostream>
00006 #include <random>
00007 #include <numeric>
00008 #include <limits>
00009 #include <fstream>
00010 #include <filesystem>
00011 #include <cstdlib>
00012 #include <ctime>
00013 #include <algorithm>
00014 #include <functional>
00015 #include <omp.h>
00016 #include "Genetic_Algorithm.h"
00017
00018 namespace fs = std::filesystem;
00019
00020 int number_of_units = 1;
00021
00022 #define PBSTR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
00023 #define PBWIDTH 60
00024
00025
00026 /**
00027  * Prints the current progress of an operation to the console.
00028  *
00029  * @param percentage The completion percentage of the operation.
00030  * @param performance A floating point value indicating current performance metrics.
00031  */
00032 void printProgress(double percentage, double performance) {
00033     int val = (int)(percentage * 100);
00034     int lpad = (int)(percentage * PBWIDTH);
00035     int rpad = PBWIDTH - lpad;
00036     printf("\r%3d%% [%.*s%*s] Item %.2f", val, lpad, PBSTR, rpad, "", performance);
00037     fflush(stdout);
00038 }
00039
00040
00041 /**
00042  * Finds the maximum value in a double array.
00043  *
00044  * @param array Pointer to the first element of the double array.
00045  * @param size Size of the array.
00046  * @return The maximum value found in the array.
00047  */
00048 double find_max_double(const double* array, int size) {
00049     double max_value = array[0];
00050     for (int i = 1; i < size; ++i) {
00051         if (array[i] > max_value) {
00052             max_value = array[i];
00053         }
00054     }
00055     return max_value;
00056 }
00057
00058
00059 /**
00060  * Generates a random number within a specified range.
00061  *
00062  * @param min The lower bound of the range.
00063  * @param max The upper bound of the range.
00064  * @return A randomly generated number within the specified range.
00065  */
00066 double generate_random_number(double min, double max) {
```

```
    static thread_local std::mt19937 generator(omp_get_thread_num());
    std::uniform_real_distribution<double> distribution(min, max);
    return distribution(generator);
}


/**
 * Initializes a population for the genetic algorithm.
 *
 * @param population_size The size of the population to initialize.
 * @param vector_size The size of each individual in the population.
 * @param initial_vector Initial values for the first individual in the population.
 * @param validity A function that checks the validity of an individual.
 * @param elitism_rate The rate of elitism to apply during evolution.
 * @return A vector of vectors containing the initialized population.
 */
std::vector<std::vector<int» initialize_population(int population_size, int vector_size, const int* initial_vector, std::function<bool(int, int*)> validity, double elitism_rate) {
    std::vector<std::vector<int» population;
    population.reserve(population_size);  // Reserve space to avoid reallocations

    std::vector<int> start_vector(initial_vector, initial_vector + vector_size);
    population.push_back(start_vector);

    number_of_units = *std::max_element(initial_vector, initial_vector + vector_size) + 1;

    #pragma omp parallel
    {
        std::random_device rd;
        std::mt19937 gen(rd() + omp_get_thread_num());  // Unique seed for each thread
        std::uniform_int_distribution<> distr(0, number_of_units - 1);

        #pragma omp for
        for (int i = 1; i < population_size; ++i) {
            std::vector<int> individual(vector_size);
            for (int j = 0; j < vector_size; ++j) {
                individual[j] = distr(gen);

                bool valid = true;
                if (j == 0) {
                    if (individual[j] == number_of_units - 2 || individual[j] == number_of_units - 3) {
                        valid = false;
                    }
                } else if ((j - 1) / 3 == individual[j]) {
                    valid = false;
                }

                if (!valid) {
                    j--;
                }
            }

            #pragma omp critical
            {
                if (validity(vector_size, individual.data()) || population.size() < population_size * 0.8) {
                    population.push_back(individual);
                } else {
                    --i;  // Retry this iteration
                }
            }
        }
    }

    return population;
}


/**
 * Selects individuals from the population based on their fitness.
 *
 * @param population A reference to the current population.
 * @param fitness A vector containing fitness scores for each individual.
 * @return A vector of vectors containing the selected individuals.
 */
std::vector<std::vector<int» select(const std::vector<std::vector<int»& population, const std::vector<double>& fitness) {
    std::vector<std::vector<int» selected;
    double total_fitness = std::accumulate(fitness.begin(), fitness.end(), 0.0);
    std::vector<double> probabilities;

    std::transform(fitness.begin(), fitness.end(), probabilities.begin(), [total_fitness](double f) {
        return f / total_fitness;
    });

    std::discrete_distribution<int> distribution(probabilities.begin(), probabilities.end());
```

```
    std::random_device rd;
    std::mt19937 gen(rd());

    for (size_t i = 0; i < population.size(); ++i) {
        selected.push_back(population[distribution(gen)]);
    }

    return selected;
}


/**
 * Applies a non-uniform mutation to an individual in the population. Mutation depends on the current generation,
 * allowing for finer mutations as the number of generations increases.
 *
 * @param individual A reference to the individual (vector of ints) to mutate.
 * @param mutation_rate The mutation rate to apply.
 * @param max_value The maximum value for any gene in the individual.
 * @param currentGeneration The current generation number in the genetic algorithm.
 * @param maxGenerations The maximum number of generations expected to run.
 */
void NonUniform_Mutation(std::vector<int>& individual, double mutation_rate, int max_value, int currentGeneration, int maxGenerations) {

    for (int& gene : individual) {
        if (generate_random_number(0.0, 1.0) < mutation_rate) {
            double delta = (generate_random_number(0.0, 1.0) < 0.5) ? gene : max_value - gene;
            double b = 5;
            double r = generate_random_number(0.0, 1.0);

            double change = delta * (1 - pow(r, pow((1 - double(currentGeneration) / maxGenerations), b)));
            gene = (generate_random_number(0.0, 1.0) < 0.5) ? gene - static_cast<int>(change) : gene + static_cast<int>(change);

            if (gene < 0) gene = 0;
            if (gene > max_value) gene = max_value;
        }
    }
}


/**
 * Mutates a given vector with a specified mutation rate. Each element in the vector has a chance to be changed
 * based on the mutation rate.
 *
 * @param vector The vector to mutate.
 * @param mutation_rate The probability of mutating each element of the vector.
 * @param max_unit The maximum value any element in the vector can take.
 */
void mutate_vector(std::vector<int>& vector, double mutation_rate, int max_unit) {

    for (int& value : vector) {
        if (generate_random_number(0.0, 1.0) < mutation_rate) {
            value = (value + static_cast<int>(generate_random_number(0, max_unit))) % (max_unit + 1);
        }
    }
}


/**
 * Performs a single-point crossover between two parent vectors.
 *
 * @param parent1 The first parent vector.
 * @param parent2 The second parent vector.
 * @param crossover_rate The probability of performing a crossover.
 * @param max_value The maximum value for any gene in the vectors.
 */
void crossover(std::vector<int>& parent1, std::vector<int>& parent2, double crossover_rate, int max_value) {

    if (generate_random_number(0.0, 1.0) < crossover_rate) {
        std::uniform_int_distribution<> point_dist(1, parent1.size() - 2);
        int crossover_point = static_cast<int>(generate_random_number(1, parent1.size() - 2));

        for (int i = crossover_point; i < parent1.size(); ++i) {
            std::swap(parent1[i], parent2[i]);
        }
    }
}


/**
 * Selects an index for roulette wheel selection based on cumulative fitness scores.
 *
```

```
 * @param cumulative_fitness A vector of cumulative fitness scores.
 * @return The selected index based on the random choice in the cumulative distribution.
 */
int select_index(const std::vector<double>& cumulative_fitness) {
    double rnd = generate_random_number(0.0, cumulative_fitness.back());
    return std::lower_bound(cumulative_fitness.begin(), cumulative_fitness.end(), rnd) - cumulative_fitness.begin();
}


/**
 * Regenerates the population by introducing new random vectors to replace the less fit individuals,
 * aiming to introduce diversity and prevent premature convergence.
 *
 * @param population The population of vectors.
 * @param vector_size The size of each vector.
 * @param number_of_units The maximum value for any gene in the vectors.
 */
void regenerate_population(std::vector<std::vector<int»& population, int vector_size, int number_of_units) {
    #pragma omp parallel for
    for (int i = (int) (population.size() * 0.2); i < population.size(); ++i) {  // Start from 1 to keep the first vector unchanged
        std::random_device rd;
        std::mt19937 gen(rd() + omp_get_thread_num()); // Ensuring unique seed per thread
        std::uniform_int_distribution<> distr(0, number_of_units - 1);

        for (int j = 0; j < vector_size; ++j) {
            population[i][j] = distr(gen);
        }
    }
}


/**
 * Conducts the entire genetic algorithm process, managing the population through multiple generations
 * and applying genetic operations like selection, crossover, and mutation to evolve solutions.
 *
 * @param population The initial population of solutions.
 * @param func A function pointer to the fitness evaluation function.
 * @param validity A function to check the validity of individual solutions.
 * @param parameters Struct containing parameters for the genetic algorithm.
 * @return The maximum fitness achieved by the best solution in the population.
 */
double genetic_algorithm(std::vector<std::vector<int»& population, double (&func) (int, int*),
                         std::function<bool(int, int*)> validity,
                         const Algorithm_Parameters& parameters) {
    int population_size = population.size();
    int vector_size = population[0].size();
    std::vector<double> fitness(population_size);
    double max_fitness = std::numeric_limits<double>::lowest();
    int fitness_unchanged_count = 0;

    int elitism_count = static_cast<int>(population.size() * parameters.elitism_rate);

    for (int generation = 0; generation < parameters.max_iterations; ++generation) {
        // Evaluate fitness for each vector in the population
        #pragma omp parallel for
        for (int i = 0; i < population_size; ++i) {
            if (validity(vector_size, population[i].data())) {
                fitness[i] = func(vector_size, population[i].data());
            } else {
                fitness[i] = std::numeric_limits<double>::lowest();
            }
        }

        // Sort the population based on fitness
        std::vector<int> idx(population_size);
        std::iota(idx.begin(), idx.end(), 0);
        std::sort(idx.begin(), idx.end(), [&](int i1, int i2) { return fitness[i1] > fitness[i2]; });

        printProgress((double)generation / (parameters.max_iterations - 1), fitness[idx[0]]);

        // Implement elitism, save the best individuals
        std::vector<std::vector<int» new_population;
        for (int i = 0; i < elitism_count; ++i) {
            new_population.push_back(population[idx[i]]);
        }

        // Create a cumulative fitness sum for roulette wheel selection
        std::vector<double> cumulative_fitness(population_size);
        std::partial_sum(fitness.begin(), fitness.end(), cumulative_fitness.begin());

        // #pragma omp parallel for
        for (int i = elitism_count; i < population.size(); i+=2) {
            std::vector<int> parent1 = population[select_index(cumulative_fitness)];
            std::vector<int> parent2 = population[select_index(cumulative_fitness)];
```

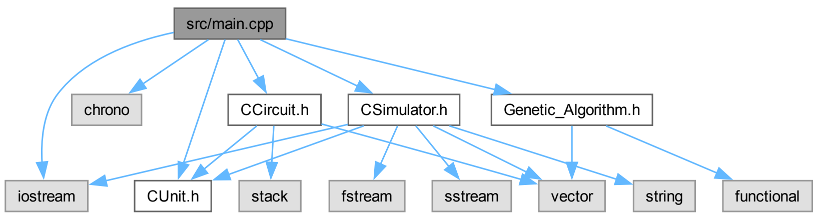
```
            crossover(parent1, parent2, parameters.crossover_rate, number_of_units);
            crossover(parent1, parent2, parameters.crossover_rate, number_of_units);

            // if (vector_size > 100) {
            //     while (!validity(vector_size, parent1.data()) && !validity(vector_size, parent2.data())) {
            //         crossover(parent1, parent2, parameters.crossover_rate, number_of_units);
            //     }
            // }

            NonUniform_Mutation(parent1, parameters.mutation_rate, number_of_units, generation, parameters.max_iterations);
            NonUniform_Mutation(parent2, parameters.mutation_rate, number_of_units, generation, parameters.max_iterations);
            double mutator = 0.0;

            if (fitness_unchanged_count > (parameters.max_iterations * 0.1)) {
                mutator = parameters.mutation_rate + (fitness_unchanged_count * 0.001);
                mutator = mutator < 0.5 ? mutator : 0.5;
            } else {

                mutator = parameters.mutation_rate;
            }
            mutate_vector(parent1, mutator, number_of_units);
            mutate_vector(parent2, mutator, number_of_units);

            new_population.push_back(parent1);
            if (new_population.size() < population_size) {
                new_population.push_back(parent2);
            }
        }

        double temp_fitness = find_max_double(fitness.data(), fitness.size());
        if (temp_fitness - max_fitness < 0.1) {
            fitness_unchanged_count++;
        }
        else{
            fitness_unchanged_count = 0;
        }
        if (temp_fitness > max_fitness) {
          fs::path dir("./output");
          if (!fs::exists(dir)) {
              fs::create_directories(dir);
          }
          std::ofstream vector_file("./output/vector.dat");
          if (vector_file.is_open()) {
              for (int i = 0; i < vector_size; i++) {
                  vector_file « population[0][i] « " ";
              }
              vector_file.close();
          }

        }
        population = new_population;
        if (fitness_unchanged_count > 50) {
            regenerate_population(population, vector_size, number_of_units);
            fitness_unchanged_count = 0;
        }
        #pragma omp barrier
        max_fitness = *std::max_element(fitness.begin(), fitness.end());
    }
    std::cout « std::endl;
    return max_fitness;
}


/**
 * Optimizes a vector using genetic algorithm principles. Initializes a population, runs the genetic algorithm,
 * and stores the best solution back into the original vector.
 *
 * @param vector_size The size of the vector to be optimized.
 * @param vector The vector containing initial values, modified in-place to store the best solution found.
 * @param func A function pointer to the fitness evaluation function.
 * @param validity A function to check the validity of individual solutions.
 * @param parameters Struct containing parameters for the genetic algorithm.
 * @return Returns 0 on successful execution and optimization, -1 if file operation fails.
 */
int optimize(int vector_size, int *vector,
             double (&func) (int, int*),
             std::function<bool(int, int*)> validity,
             struct Algorithm_Parameters parameters) {
    // print the number of threads
    std::cout « "Number of threads: " « omp_get_max_threads() « std::endl;

```

```
    int unit_num = (vector_size - 1) / 3;

    for (int i = 0; i <= unit_num+1; ++i){
        vector[i] = i;
    }
    for (int i = unit_num+2; i < vector_size; ++i){
        vector[i] = 0;
    }

    std::vector<std::vector<int>> population = initialize_population(parameters.initial_pop,
        vector_size, vector, validity, parameters.elitism_rate);
    double max_fitness = genetic_algorithm(population, func, validity, parameters);
    std::copy(population[0].begin(), population[0].end(), vector);

    std::ofstream vector_file("./output/vector.dat");
    if (vector_file.is_open()) {
        for (int i = 0; i < vector_size; i++) {
            vector_file << vector[i] << " ";
        }
        vector_file.close();
    } else {
        return -1;
    }

    return 0;
}
```

4.15 src/main.cpp File Reference

```
#include <iostream>
#include <chrono>
#include "CUnit.h"
#include "CCircuit.h"
#include "CSimulator.h"
#include "Genetic_Algorithm.h"
```

Include dependency graph for main.cpp:



Functions

- int main (int argc, char *argv[ ])

4.15.1 Function Documentation

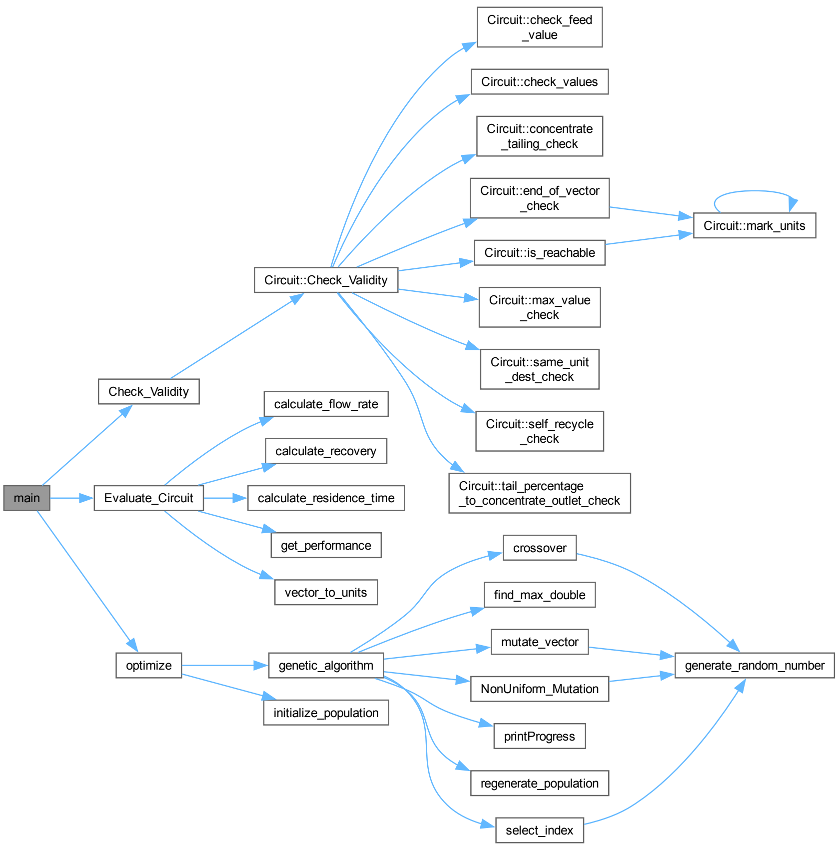
4.15.1.1 main()

```
int main (
        int argc,
        char * argv[ ])
```

Definition at line 10 of file main.cpp.

```
00010                         {
00011 
00012     int units = 42;
00013 
00014     int vector[(units*3)+1];
00015     int n = sizeof(vector) / sizeof(int);
00016 
00017     // Adjust the parameters as needed
00018     Algorithm_Parameters params = {1000, 0.9, 0.01, 0.1, 500};
00019 
00020 
00021     // Measure time for optimize function
00022     auto start_optimize = chrono::high_resolution_clock::now();
00023     optimize(n, vector, Evaluate_Circuit, Check_Validity, params);
00024     auto end_optimize = chrono::high_resolution_clock::now();
00025     chrono::duration<double> duration_optimize = end_optimize - start_optimize;
00026     cout « "Time taken for optimization: " « duration_optimize.count() « " seconds" « endl;
00027 
00028     // Measure time for Evaluate_Circuit function
00029     double evaluation_result = Evaluate_Circuit(n, vector);
00030 
00031     // Generate final output, save to file, etc.
00032     cout « "Evaluation result: " « evaluation_result « endl;
00033 
00034     return 0;
00035 }
```

Here is the call graph for this function:



4.16 main.cpp

Go to the documentation of this file.

```
#include <iostream>
#include <chrono>
#include "CUnit.h"
#include "CCircuit.h"
#include "CSimulator.h"
#include "Genetic_Algorithm.h"

using namespace std;

int main(int argc, char *argv[]) {

    int units = 42;

    int vector[(units*3)+1];
    int n = sizeof(vector) / sizeof(int);

    // Adjust the parameters as needed
    Algorithm_Parameters params = {1000, 0.9, 0.01, 0.1, 500};


    // Measure time for optimize function
    auto start_optimize = chrono::high_resolution_clock::now();
    optimize(n, vector, Evaluate_Circuit, Check_Validity, params);
    auto end_optimize = chrono::high_resolution_clock::now();
    chrono::duration<double> duration_optimize = end_optimize - start_optimize;
    cout « "Time taken for optimization: " « duration_optimize.count() « " seconds" « endl;

    // Measure time for Evaluate_Circuit function
    double evaluation_result = Evaluate_Circuit(n, vector);

    // Generate final output, save to file, etc.
    cout « "Evaluation result: " « evaluation_result « endl;

    return 0;
}
```

Index

Algorithm_Parameters, 5
crossover_rate, 5
elitism_rate, 5
initial_pop, 6
max_iterations, 6
mutation_rate, 6
all_true
Genetic_Algorithm.h, 46

Calculate_constants, 6
k_concentrate_gerardium, 7
k_concentrate_waste, 7
k_inter_gerardium, 7
k_inter_waste, 7
phi, 7
rho, 7
V, 8
calculate_flow_rate
CSimulator.cpp, 65
CSimulator.h, 33
calculate_recovery
CSimulator.cpp, 66
CSimulator.h, 34
calculate_residence_time
CSimulator.cpp, 67
CSimulator.h, 35
CCircuit.cpp
Check_Validity, 58
conc_outlet_reached, 59
inter_outlet_reached, 59
tails_outlet_reached, 60
CCircuit.h
Check_Validity, 28
check_feed_value
Circuit, 9
Check_Validity
CCircuit.cpp, 58
CCircuit.h, 28
Circuit, 10
check_values
Circuit, 12
Circuit, 8
check_feed_value, 9
Check_Validity, 10
check_values, 12
Circuit, 9
concentrate_tailing_check, 13
end_of_vector_check, 14
is_reachable, 15
mark_units, 16
max_value_check, 17
same_unit_dest_check, 18
self_recycle_check, 19
tail_percentage_to_concentrate_outlet_check, 19
units, 20
Circuit_Parameters, 21
max_iterations, 21
tolerance, 21
conc_num
CUnit, 22
conc_outlet_reached
CCircuit.cpp, 59
concentrate_gerardium
Recovery, 26
concentrate_tailing_check
Circuit, 13
concentrate_waste
Recovery, 26
crossover
Genetic_Algorithm.cpp, 76
Genetic_Algorithm.h, 46
crossover_rate
Algorithm_Parameters, 5
CSimulator.cpp
calculate_flow_rate, 65
calculate_recovery, 66
calculate_residence_time, 67
Evaluate_Circuit, 67
get_performance, 70
vector_to_units, 71
CSimulator.h
calculate_flow_rate, 33
calculate_recovery, 34
calculate_residence_time, 35
Evaluate_Circuit, 36
get_performance, 39
vector_to_units, 40
CUnit, 21
conc_num, 22
CUnit, 22
inter_num, 22
mark, 23
new_flow_G, 23
new_flow_W, 23
old_flow_G, 23
old_flow_W, 23
tails_num, 23

DEFAULT_ALGORITHM_PARAMETERS
Genetic_Algorithm.h, 46

Economic_parameters, 24
    penalty, 24
    price, 24
elitism_rate
    Algorithm_Parameters, 5
end_of_vector_check
    Circuit, 14
Evaluate_Circuit
    CSimulator.cpp, 67
    CSimulator.h, 36

find_max_double
    Genetic_Algorithm.cpp, 77
    Genetic_Algorithm.h, 47

generate_random_number
    Genetic_Algorithm.cpp, 78
genetic_algorithm
    Genetic_Algorithm.cpp, 78
    Genetic_Algorithm.h, 48
Genetic_Algorithm.cpp
    crossover, 76
    find_max_double, 77
    generate_random_number, 78
    genetic_algorithm, 78
    initialize_population, 81
    mutate_vector, 82
    NonUniform_Mutation, 83
    number_of_units, 90
    optimize, 84
    PBSTR, 76
    PBWIDTH, 76
    printProgress, 87
    regenerate_population, 87
    select, 88
    select_index, 89
Genetic_Algorithm.h
    all_true, 46
    crossover, 46
    DEFAULT_ALGORITHM_PARAMETERS, 46
    find_max_double, 47
    genetic_algorithm, 48
    initialize_population, 50
    mutate_vector, 52
    NonUniform_Mutation, 53
    optimize, 54
    select_index, 55
get_performance
    CSimulator.cpp, 70
    CSimulator.h, 39

include/CCircuit.h, 27, 29
include/CSimulator.h, 31, 41
include/CUnit.h, 43
include/Genetic_Algorithm.h, 44, 56
init_Fg
    Initial_flow, 25
init_Fw
    Initial_flow, 25
Initial_flow, 25
    init_Fg, 25
    init_Fw, 25
initial_pop
    Algorithm_Parameters, 6
initialize_population
    Genetic_Algorithm.cpp, 81
    Genetic_Algorithm.h, 50
inter_gerardium
    Recovery, 26
inter_num
    CUnit, 22
inter_outlet_reached
    CCircuit.cpp, 59
inter_waste
    Recovery, 26
is_reachable
    Circuit, 15

k_concentrate_gerardium
    Calculate_constants, 7
k_concentrate_waste
    Calculate_constants, 7
k_inter_gerardium
    Calculate_constants, 7
k_inter_waste
    Calculate_constants, 7

main
    main.cpp, 95
main.cpp
    main, 95
mark
    CUnit, 23
mark_units
    Circuit, 16
max_iterations
    Algorithm_Parameters, 6
    Circuit_Parameters, 21
max_value_check
    Circuit, 17
mutate_vector
    Genetic_Algorithm.cpp, 82
    Genetic_Algorithm.h, 52
mutation_rate
    Algorithm_Parameters, 6

new_flow_G
    CUnit, 23
new_flow_W
    CUnit, 23
NonUniform_Mutation
    Genetic_Algorithm.cpp, 83
    Genetic_Algorithm.h, 53
number_of_units
    Genetic_Algorithm.cpp, 90

old_flow_G
    CUnit, 23

old_flow_W
    CUnit, 23
optimize
    Genetic_Algorithm.cpp, 84
    Genetic_Algorithm.h, 54

PBSTR
    Genetic_Algorithm.cpp, 76
PBWIDTH
    Genetic_Algorithm.cpp, 76
penalty
    Economic_parameters, 24
phi
    Calculate_constants, 7
price
    Economic_parameters, 24
printProgress
    Genetic_Algorithm.cpp, 87

Recovery, 25
    concentrate_gerardium, 26
    concentrate_waste, 26
    inter_gerardium, 26
    inter_waste, 26
regenerate_population
    Genetic_Algorithm.cpp, 87
rho
    Calculate_constants, 7

same_unit_dest_check
    Circuit, 18
select
    Genetic_Algorithm.cpp, 88
select_index
    Genetic_Algorithm.cpp, 89
    Genetic_Algorithm.h, 55
self_recycle_check
    Circuit, 19
src/CCircuit.cpp, 57, 60
src/CSimulator.cpp, 64, 72
src/Genetic_Algorithm.cpp, 75, 90
src/main.cpp, 95, 96

tail_percentage_to_concentrate_outlet_check
    Circuit, 19
tails_num
    CUnit, 23
tails_outlet_reached
    CCircuit.cpp, 60
tolerance
    Circuit_Parameters, 21

units
    Circuit, 20

V
    Calculate_constants, 8
vector_to_units
    CSimulator.cpp, 71
    CSimulator.h, 40