Gerardium Rush Bauxite Project

Generated by Doxygen 1.9.1

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 Algorithm_Parameters Struct Reference

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

Public Attributes

- int max_iterations
- int population_size
- double cross_probability
- double mutation_rate
- int stall_length
- double stall_mutation_rate

3.1.1 Detailed Description

Definition at line 12 of file Genetic_Algorithm.h.

3.1.2 Member Data Documentation

3.1.2.1 cross_probability

```
double Algorithm_Parameters::cross_probability
```

Definition at line 16 of file Genetic_Algorithm.h.

3.1.2.2 max_iterations

```
int Algorithm_Parameters::max_iterations
```

Definition at line 13 of file Genetic_Algorithm.h.

3.1.2.3 mutation_rate

```
double Algorithm_Parameters::mutation_rate
```

Definition at line 17 of file Genetic_Algorithm.h.

3.1.2.4 population_size

```
int Algorithm_Parameters::population_size
```

Definition at line 15 of file Genetic_Algorithm.h.

3.1.2.5 stall_length

```
int Algorithm_Parameters::stall_length
```

Definition at line 18 of file Genetic_Algorithm.h.

3.1.2.6 stall_mutation_rate

```
double Algorithm_Parameters::stall_mutation_rate
```

Definition at line 19 of file Genetic_Algorithm.h.

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

- /home/bz223/downloads/acs-gerardium-rush-bauxite/include/Genetic_Algorithm.h

3.2 Circuit Class Reference

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

Public Member Functions

- Circuit (int num_units)

  Constructs a new Circuit object.
- bool Check_Validity (int vector_size, int ∗circuit_vector)

  Checks the validity of a circuit.
- void setup_connections (int ∗array, int array_size)

  Sets up the connections in the circuit based on the provided array.
- void update_flows ()

  Updates the flows in the circuit.

Public Attributes

- int initial_index
- double concentrate_G = 0
- double concentrate_W = 0
- double tails_G = 0
- double tails_W = 0
- std::vector< double > next_feed_G
- std::vector< double > next_feed_W
- int count = 0

Private Member Functions

- void mark_units (int unit_num)

  Marks a unit in the circuit and recursively marks connected units.

Static Private Member Functions

- static bool isReachable (int start, int num_units, vector< CUnit > &units)

  Checks if all units in the circuit are reachable from a given start unit.
- static bool check_no_self_recycle (const vector< CUnit > &units)

  Checks if there is any self-recycle in the circuit.
- static bool check_no_all_same_destination (const vector< CUnit > &units)

  Checks if there is any unit in the circuit where all output streams point to the same unit.

Private Attributes

- int num_units
- std::vector< CUnit > units

3.2.1 Detailed Description

Definition at line 15 of file CCircuit.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Circuit()

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

Constructs a new Circuit object.

This constructor creates a new Circuit object with a specified number of units. It initializes the 'units' member variable to a vector of 'CUnit' objects of size 'num_units'.

Parameters

num_units	The number of units in the circuit.

Definition at line 20 of file CCircuit.cpp.

```
20                          {
21   this->units.resize(num_units);
22   this->num_units = num_units;
23   this->next_feed_G.resize(num_units);
24   this->next_feed_W.resize(num_units);
25 }
```

3.2.3 Member Function Documentation

3.2.3.1 check_no_all_same_destination()

```
bool Circuit::check_no_all_same_destination (
            const vector< CUnit > & units )  [static], [private]
```

Checks if there is any unit in the circuit where all output streams point to the same unit.

This function checks whether any unit in the circuit has all its output streams (concentrate, intermediate, and tailings) pointing to the same unit.

Parameters

units	The vector of units in the circuit.

Returns

true if there is no unit where all output streams point to the same unit, false otherwise.

Definition at line 228 of file CCircuit.cpp.

```
228                                                               {
229     for (const auto& unit :  units) {
```

```
        if ((unit.conc_num == unit.inter_num && unit.inter_num == unit.tails_num) ||
            (unit.conc_num == unit.tails_num)) {
            return false; // Here all output streams point to the same unit
        }
    }
    return true;
}
```

3.2.3.2 check_no_self_recycle()

```
bool Circuit::check_no_self_recycle (
            const vector< CUnit > & units )  [static], [private]
```

Checks if there is any self-recycle in the circuit.

This function checks whether any unit in the circuit has an output that is connected to itself. A unit is considered to have a self-recycle if its concentrate, intermediate, or tailings output is connected to itself.

Parameters

units	The vector of units in the circuit.

Returns

true if there is no self-recycle in the circuit, false otherwise.

Definition at line 167 of file CCircuit.cpp.

```
                                                                    {
    for (size_t i = 0; i < units.size(); ++i) {
        if (units[i].conc_num == i || units[i].inter_num == i || units[i].tails_num == i) {
            return false;
        }
    }
    return true;
}
```

3.2.3.3 Check_Validity()

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

Checks the validity of a circuit.

This function checks the validity of a circuit based on the connections specified in the input vector. The vector should contain a sequence of integers representing the connections between units in the circuit. The function performs several checks to ensure that the circuit is valid, including checking for self-recycle, checking that all output streams do not point to the same unit, and checking that all units are reachable from the starting unit. If the circuit passes all checks, the function returns true; otherwise, it returns false.

Parameters

vector_size	The size of the input vector.
circuit_vector	The input vector containing the connections between units.

Definition at line 83 of file CCircuit.cpp.

```
                                                                    {
    // Check if the vector size is valid
    if (vector_size < 4 || (vector_size - 1) % 3 != 0) {
        return false;
    }

    int num_units = (vector_size - 1) / 3;  // (10-1)/3 in this example

    // Set up all units' connections
    for (int i = 0; i < (vector_size-1) / 3; ++i) {
        // Add 1 to skip feed unit
        this->units[i].conc_num = circuit_vector[1 + 3 * i];
        this->units[i].inter_num = circuit_vector[1 + 3 * i + 1];
        this->units[i].tails_num = circuit_vector[1 + 3 * i + 2];
        this->units[i].mark = false;

        // Check the range of conc_num, inter_num, and tails_num
         // The range should be [0, num_units] for conc_num and inter_num
         // The range should be [0, num_units + 1] for tails_num, except for the last unit
         if (
           this->units[i].conc_num < 0   || this->units[i].conc_num > num_units||
             this->units[i].inter_num < 0 || this->units[i].inter_num > num_units + 1||
             this->units[i].tails_num < 0 || this->units[i].tails_num > num_units + 1 ||
             this->units[i].tails_num == num_units) {
             return false;
         }
     }

     // Check for self-recycle and all output streams pointing to the same unit
     if (!check_no_self_recycle(this->units) || !check_no_all_same_destination(this->units)) {
         return false;
     }

     // Define outlets (Feed -> unit 0, unit 1, unit 2, ...)// e.g.  unit 3 and unit 4 are outlets
     // Check if all units can reach the outlet stream concentrate
     if (!isReachable(num_units, num_units, this->units)) {
         return false;
     }

     // Check if all units can reach the outlet stream tailings
     if (!isReachable(num_units + 1, num_units, this->units)) {
         return false;
     }

     // Initialize all units as unvisited
     for (int i = 0; i < this->units.size(); i++) {
         this->units[i].mark = false;
     }

     // Mark all units accessible from the starting unit (the feed unit)
     if (circuit_vector[0] < 0 || circuit_vector[0] >= num_units) {
         return false;
     }
     this->mark_units(circuit_vector[0]);

     // Check which units have been marked as visited
     bool all_accessible = true;
     for (int i = 0; i < this->units.size(); i++) {
         if (this->units[i].mark) {
           //cout « "Unit " « i « " is accessible." « endl;
         } else {
           //cout « "Unit " « i « " is not accessible." « endl;
             all_accessible = false;
         }
     }

     // Check if all units are accessible
     if (!all_accessible) {
         return false;
     }

     return true; // If all checks pass, the circuit is valid
}
```

3.2.3.4 isReachable()

```
bool Circuit::isReachable (
               int start,
```

```
            int num_units,
            vector< CUnit > & units )  [static], [private]
```

Checks if all units in the circuit are reachable from a given start unit.

This function uses a breadth-first search algorithm to check whether all units in the circuit are reachable from a given start unit. A unit is considered reachable if there is a path from the start unit to the unit through the connections between units.

Parameters

start	The index of the start unit.
num_units	The number of units in the circuit.
units	The vector of units in the circuit.

Returns

true if all units are reachable from the start unit, false otherwise.

Definition at line 188 of file CCircuit.cpp.

```
188                                                                    {
189     vector<bool> reachable(num_units, false);
190     queue<int> bfs_queue;
191     bfs_queue.push(start);
192
193     while (!bfs_queue.empty()) {
194         int current = bfs_queue.front();
195         bfs_queue.pop();
196
197         for (int i = 0; i < num_units; ++i) {
198             if (!reachable[i]) {
199                 if ((units[i].conc_num == current) ||
200                     (units[i].inter_num == current) ||
201                     (units[i].tails_num == current)) {
202                     reachable[i] = true;
203                     bfs_queue.push(i);
204                 }
205             }
206         }
207     }
208
209     for (int i = 0; i < num_units; ++i) {
210         if (!reachable[i]) {
211             //cout « "Unit " « i « " is the first unit that cannot reach outlet stream " « start « "." «
      endl;
212             return false;
213         }
214     }
215
216     return true;
217 }
```

3.2.3.5 mark_units()

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

Marks a unit in the circuit and recursively marks connected units.

This function marks a unit as visited in the circuit. If the unit has already been marked, the function returns immediately. Otherwise, it marks the unit and then recursively calls itself for each connected unit that is not an outlet of the circuit.

Parameters

unit_num	The index of the unit to mark.

Definition at line 36 of file CCircuit.cpp.

```
36                                          {
37
38   if (this->units[unit_num].mark) return;
39
40   this->units[unit_num].mark = true;
41
42   //If we have seen this unit already exit
43   //Mark that we have now seen the unit
44
45   //If conc_num does not point at a circuit outlet recursively call the function
46   if (this->units[unit_num].conc_num<this->units.size()) {
47     mark_units(this->units[unit_num].conc_num);
48   } else {
49     //cerr « "Concentrate stream is connected to conc outlet." « endl;
50     }
51
52   //If inter_num does not point at a circuit outlet recursively call the function
53   if (this->units[unit_num].inter_num<this->units.size()) {
54     mark_units(this->units[unit_num].inter_num);
55   } else {
56     //cerr « "Intermediate stream is connected to one outlet." « endl;
57   }
58   //If tails_num does not point at a circuit outlet recursively call the function
59
60   if (this->units[unit_num].tails_num<this->units.size()) {
61     mark_units(this->units[unit_num].tails_num);
62   } else {
63     //cerr « "Tails stream is connected to tails outlet." « endl;
64   }
65 }
```

3.2.3.6 setup_connections()

```
void Circuit::setup_connections (
               int * array,
               int array_size )
```

Sets up the connections in the circuit based on the provided array.

This function sets up the connections between the units in the circuit based on the provided array. It interprets the first element of the array as the initial index and the rest of the array as triples representing the connections for each unit.

Parameters

array	The array representing the circuit connections.
array_size	The size of the array.

Definition at line 248 of file CCircuit.cpp.

```
248                                                            {
249   // The first element of the vector indicates the starting unit
250   this->initial_index = array[0];
251   // Process the rest of the vector as connection triples
252   for (int i = 1; i < array_size; i+=3) {
253     int unitIndex = (i - 1) / 3;
254     if (unitIndex < num_units){
255
256       // Set the connections based on the vector values
257       this->units[unitIndex].conc_num = array[i];
258       this->units[unitIndex].inter_num = array[i + 1];
```

```
      this->units[unitIndex].tails_num = array[i + 2];

      // Check for special cases where the flow should go to concentrate or tailings
      if(array[i] == this->num_units) {
        //Direct the flow to concentrate
        this->units[unitIndex].conc_num = -1; // -1 indicates that the flow should go to concentrate
      }
      if(array[i] == this->num_units + 1){
        //Direct the flow to tailings
        this->units[unitIndex].conc_num = -2; // -2 indicates that the flow should go to tailings
      }
      if(array[i + 1] == this->num_units){
        //Direct the flow to concentrate
        this->units[unitIndex].inter_num = -1; // -1 indicates that the flow should go to concentrate
      }
      if(array[i + 1] == this->num_units + 1){
        //Direct the flow to tailings
        this->units[unitIndex].inter_num = -2; // -2 indicates that the flow should go to tailings
      }
      if(array[i + 2] == this->num_units){
        //Direct the flow to concentrate
        this->units[unitIndex].tails_num = -1; // -1 indicates that the flow should go to concentrate
      }
      if(array[i + 2] == this->num_units + 1){
        //Direct the flow to tailings
        this->units[unitIndex].tails_num = -2; // -2 indicates that the flow should go to tailings
      }
    }
  }
}
```

3.2.3.7 update_flows()

```
void Circuit::update_flows ( )
```

Updates the flows in the circuit.

This function calculates and updates the flows in the circuit based on the current state of the units. It calculates the output of each unit and updates the feed rates for the next iteration.

Definition at line 296 of file CCircuit.cpp.

```
296                           {
297   std::vector<double> next_feed_G(num_units, 0);
298   std::vector<double> next_feed_W(num_units, 0);
299 
300   concentrate_G = 0;
301   concentrate_W = 0;
302   tails_G = 0;
303   tails_W = 0;
304 
305   //Calculate every output of the streams and update the next feed
306   for (int i = 0; i < this->num_units; ++i){
307     CUnit &unit = this->units[i];
308 
309     if(i == this->initial_index){
310       unit.feed_rate_G += 10;
311       unit.feed_rate_W += 90;
312     }
313     double total_feed = unit.feed_rate_G + unit.feed_rate_W;
314     unit.Calculate_ResidenceTime(total_feed);
315 
316     //Calculate the Output of the unit
317     double CG = unit.calculate_CG();
318     double CW = unit.calculate_CW();
319     double IG = unit.calculate_IG();
320     double IW = unit.calculate_IW();
321     double TG = unit.calculate_TG();
322     double TW = unit.calculate_TW();
323 
324     //Update the next feed
325     // Concentrate flow
326     if(unit.conc_num >=0 && unit.conc_num < num_units){
327       next_feed_G[unit.conc_num] += CG;
328       next_feed_W[unit.conc_num] += CW;
329     } else if (unit.conc_num == -1){ //Concentrate
330       concentrate_G += CG;
```

```
      concentrate_W += CW;
    } else if (unit.conc_num == -2){ //Tailings
      tails_G += CG;
      tails_W += CW;
    }

    // Intermediate flow
    if(unit.inter_num >=0 && unit.inter_num < num_units){
      next_feed_G[unit.inter_num] += IG;
      next_feed_W[unit.inter_num] += IW;
    } else if (unit.inter_num == -1){ //Concentrate
      concentrate_G += IG;
      concentrate_W += IW;
    } else if (unit.inter_num == -2){ //Tailings
      tails_G += IG;
      tails_W += IW;
    }

    // Tailings flow
    if(unit.tails_num >=0 && unit.tails_num < num_units){
      next_feed_G[unit.tails_num] += TG;
      next_feed_W[unit.tails_num] += TW;
    } else if (unit.tails_num == -1){ //Concentrate
      concentrate_G += TG;
      concentrate_W += TW;
    } else if (unit.tails_num == -2){ //Tailings
      tails_G += TG;
      tails_W += TW;
    }
  }

  //Update all feed_rate_G and feed_rate_W for the next iteration
  for (int i = 0; i < this->num_units; ++i){
    this->units[i].feed_rate_G = next_feed_G[i];
    this->units[i].feed_rate_W = next_feed_W[i];
  }
}
```

3.2.4 Member Data Documentation

3.2.4.1 concentrate_G

`double Circuit::concentrate_G = 0`

Definition at line 22 of file CCircuit.h.

3.2.4.2 concentrate_W

`double Circuit::concentrate_W = 0`

Definition at line 23 of file CCircuit.h.

3.2.4.3 count

`int Circuit::count = 0`

Definition at line 28 of file CCircuit.h.

3.2.4.4 initial_index

```
int Circuit::initial_index
```

Definition at line 21 of file CCircuit.h.

3.2.4.5 next_feed_G

```
std::vector<double> Circuit::next_feed_G
```

Definition at line 26 of file CCircuit.h.

3.2.4.6 next_feed_W

```
std::vector<double> Circuit::next_feed_W
```

Definition at line 27 of file CCircuit.h.

3.2.4.7 num_units

```
int Circuit::num_units [private]
```

Definition at line 31 of file CCircuit.h.

3.2.4.8 tails_G

```
double Circuit::tails_G = 0
```

Definition at line 24 of file CCircuit.h.

3.2.4.9 tails_W

```
double Circuit::tails_W = 0
```

Definition at line 25 of file CCircuit.h.

3.2.4.10 units

```
std::vector<CUnit> Circuit::units [private]
```

Definition at line 33 of file CCircuit.h.

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

- /home/bz223/downloads/acs-gerardium-rush-bauxite/include/CCircuit.h
- /home/bz223/downloads/acs-gerardium-rush-bauxite/src/CCircuit.cpp

3.3 Circuit_Parameters Struct Reference

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

Public Attributes

- double tolerance
- int max_iterations

3.3.1 Detailed Description

header file for the circuit simulator

This header file defines the function that will be used to evaluate the circuit

Definition at line 9 of file CSimulator.h.

3.3.2 Member Data Documentation

3.3.2.1 max_iterations

```
int Circuit_Parameters::max_iterations
```

Definition at line 11 of file CSimulator.h.

3.3.2.2 tolerance

```
double Circuit_Parameters::tolerance
```

Definition at line 10 of file CSimulator.h.

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

- /home/bz223/downloads/acs-gerardium-rush-bauxite/include/CSimulator.h

3.4 CUnit Class Reference

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

Public Member Functions

- CUnit ()
  Construct a default CUnit::CUnit object.
- CUnit (int conc, int inter, int tails)
  Construct a new CUnit::CUnit object with parameters.
- double Calculate_RHG ()
- double Calculate_RHW ()
- double Calculate_RIG ()
- double Calculate_RIW ()
- void Calculate_ResidenceTime (double total_feed_rate)
- double calculate_CG ()
- double calculate_CW ()
- double calculate_IG ()
- double calculate_IW ()
- double calculate_TG ()
- double calculate_TW ()

Public Attributes

- int conc_num
- int inter_num
- int tails_num
- bool mark = false
- double feed_rate_G = 0.0
- double feed_rate_W = 0.0
- double k_cG = 0.004
- double k_cW = 0.0002
- double k_iG = 0.001
- double k_iW = 0.0003
- double residence_time

Static Public Attributes

- static constexpr double density = 3000
- static constexpr double volume = 10
- static constexpr double solid_fraction = 0.1

3.4.1 Detailed Description

Header for the unit class

Definition at line 8 of file CUnit.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 CUnit() [1/2]

```
CUnit::CUnit ( )
```

Construct a default CUnit::CUnit object.

Returns

CUnit

Definition at line 8 of file CUnit.cpp.

```
8 : conc_num(-1), inter_num(-1), tails_num(-1), mark(false) {}
```

3.4.2.2 CUnit() [2/2]

```
CUnit::CUnit (
        int conc,
        int inter,
        int tails )
```

Construct a new CUnit::CUnit object with parameters.

Parameters

conc	
inter	
tails	

Returns

CUnit

Definition at line 18 of file CUnit.cpp.

```
18 : conc_num(conc), inter_num(inter), tails_num(tails), mark(false) {}
```

3.4.3 Member Function Documentation

3.4.3.1 calculate_CG()

double CUnit::calculate_CG ( ) [inline]

Definition at line 67 of file CUnit.h.

```
{
    return feed_rate_G * Calculate_RHG();
  }
```

3.4.3.2 calculate_CW()

double CUnit::calculate_CW ( ) [inline]

Definition at line 71 of file CUnit.h.

```
{
    return feed_rate_W * Calculate_RHW();
  }
```

3.4.3.3 calculate_IG()

double CUnit::calculate_IG ( ) [inline]

Definition at line 75 of file CUnit.h.

```
{
    return feed_rate_G * Calculate_RIG();
  }
```

3.4.3.4 calculate_IW()

double CUnit::calculate_IW ( ) [inline]

Definition at line 79 of file CUnit.h.

```
{
    return feed_rate_W * Calculate_RIW();
  }
```

3.4.3.5 Calculate_ResidenceTime()

void CUnit::Calculate_ResidenceTime (double *total_feed_rate*) [inline]

Definition at line 58 of file CUnit.h.

```
{
    if(total_feed_rate > 0){
      residence_time = volume * solid_fraction / (total_feed_rate / density);
    } else {
      residence_time = 0;
    }
  };
```

3.4.3.6 Calculate_RHG()

double CUnit::Calculate_RHG ( ) [inline]

Definition at line 41 of file CUnit.h.

```
41                  {
42      return k_cG * residence_time / (1 + k_cG * residence_time + k_iG * residence_time);
43  }
```

3.4.3.7 Calculate_RHW()

double CUnit::Calculate_RHW ( ) [inline]

Definition at line 45 of file CUnit.h.

```
45                  {
46      return k_cW * residence_time / (1 + k_cW * residence_time + k_iW * residence_time);
47  }
```

3.4.3.8 Calculate_RIG()

double CUnit::Calculate_RIG ( ) [inline]

Definition at line 49 of file CUnit.h.

```
49                  {
50      return k_iG * residence_time / (1 + k_cG * residence_time + k_iG * residence_time);
51  }
```

3.4.3.9 Calculate_RIW()

double CUnit::Calculate_RIW ( ) [inline]

Definition at line 53 of file CUnit.h.

```
53                  {
54      return k_iW * residence_time / (1 + k_cW * residence_time + k_iW * residence_time);
55  }
```

3.4.3.10 calculate_TG()

double CUnit::calculate_TG ( ) [inline]

Definition at line 84 of file CUnit.h.

```
84                 {
85      return feed_rate_G - calculate_CG() - calculate_IG();
86  }
```

3.4.3.11 calculate_TW()

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

Definition at line 88 of file CUnit.h.

```
88                          {
89     return feed_rate_W - calculate_CW() - calculate_IW();
90   }
```

3.4.4 Member Data Documentation

3.4.4.1 conc_num

```
int CUnit::conc_num
```

Definition at line 16 of file CUnit.h.

3.4.4.2 density

```
constexpr double CUnit::density = 3000 [static], [constexpr]
```

Definition at line 36 of file CUnit.h.

3.4.4.3 feed_rate_G

```
double CUnit::feed_rate_G = 0.0
```

Definition at line 25 of file CUnit.h.

3.4.4.4 feed_rate_W

```
double CUnit::feed_rate_W = 0.0
```

Definition at line 26 of file CUnit.h.

3.4.4.5 inter_num

```
int CUnit::inter_num
```

Definition at line 18 of file CUnit.h.

3.4.4.6 k_cG

```
double CUnit::k_cG = 0.004
```

Definition at line 29 of file CUnit.h.

3.4.4.7 k_cW

```
double CUnit::k_cW = 0.0002
```

Definition at line 30 of file CUnit.h.

3.4.4.8 k_iG

```
double CUnit::k_iG = 0.001
```

Definition at line 31 of file CUnit.h.

3.4.4.9 k_iW

```
double CUnit::k_iW = 0.0003
```

Definition at line 32 of file CUnit.h.

3.4.4.10 mark

```
bool CUnit::mark = false
```

Definition at line 22 of file CUnit.h.

3.4.4.11 residence_time

```
double CUnit::residence_time
```

Definition at line 33 of file CUnit.h.

3.4.4.12 solid_fraction

```
constexpr double CUnit::solid_fraction = 0.1  [static], [constexpr]
```

Definition at line 38 of file CUnit.h.

3.4.4.13 tails_num

```
int CUnit::tails_num
```

Definition at line 20 of file CUnit.h.

3.4.4.14 volume

```
constexpr double CUnit::volume = 10  [static], [constexpr]
```

Definition at line 37 of file CUnit.h.

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

- /home/bz223/downloads/acs-gerardium-rush-bauxite/include/CUnit.h
- /home/bz223/downloads/acs-gerardium-rush-bauxite/src/CUnit.cpp

3.5 Unit_parameter Struct Reference

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

Public Attributes

- double tau
- double CG
- double CW
- double IG
- double IW
- double TG
- double TW
- double RHG
- double RHW
- double RIG
- double RIW

3.5.1 Detailed Description

Definition at line 14 of file CSimulator.h.

3.5.2 Member Data Documentation

3.5.2.1 CG

```
double Unit_parameter::CG
```

Definition at line 16 of file CSimulator.h.

3.5.2.2 CW

```
double Unit_parameter::CW
```

Definition at line 17 of file CSimulator.h.

3.5.2.3 IG

```
double Unit_parameter::IG
```

Definition at line 18 of file CSimulator.h.

3.5.2.4 IW

```
double Unit_parameter::IW
```

Definition at line 19 of file CSimulator.h.

3.5.2.5 RHG

```
double Unit_parameter::RHG
```

Definition at line 22 of file CSimulator.h.

3.5.2.6 RHW

```
double Unit_parameter::RHW
```

Definition at line 23 of file CSimulator.h.

3.5.2.7 RIG

```
double Unit_parameter::RIG
```

Definition at line 24 of file CSimulator.h.

3.5.2.8 RIW

```
double Unit_parameter::RIW
```

Definition at line 25 of file CSimulator.h.

3.5.2.9 tau

```
double Unit_parameter::tau
```

Definition at line 15 of file CSimulator.h.

3.5.2.10 TG

```
double Unit_parameter::TG
```

Definition at line 20 of file CSimulator.h.

3.5.2.11 TW

```
double Unit_parameter::TW
```

Definition at line 21 of file CSimulator.h.

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

- /home/bz223/downloads/acs-gerardium-rush-bauxite/include/CSimulator.h

Chapter 4

File Documentation

4.1 /home/bz223/downloads/acs-gerardium-rush-bauxite/include/↩ CCircuit.h File Reference

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

Include dependency graph for CCircuit.h: This graph shows which files directly or indirectly include this file:

Classes

- class Circuit

4.2 /home/bz223/downloads/acs-gerardium-rush-bauxite/include/↩ CSimulator.h File Reference

```
#include <string>
```

Include dependency graph for CSimulator.h: This graph shows which files directly or indirectly include this file:

Classes

- struct Circuit_Parameters
- struct Unit_parameter

Functions

- double Evaluate_Circuit (int vector_size, int ∗circuit_vector, struct Circuit_Parameters parameters)

  Evaluates the performance of a circuit based on the provided circuit vector and parameters.
- double Evaluate_Circuit (int vector_size, int ∗circuit_vector)

  Evaluates the performance of a circuit using default parameters.
- double Evaluate_Circuit_Recovery (int vector_size, int ∗circuit_vector)

  Evaluates the recovery of a circuit using default parameters.
- double Evaluate_Circuit_Grade (int vector_size, int ∗circuit_vector)

  Evaluates the grade of a circuit using default parameters.
- void Evaluate_Circuit_Cell_Calculator (struct Unit_parameter &parameters)

  Calculates the parameters of a single unit in the circuit.
- void Save_ResultToTxt (const std::string &filename, int vector_size, int ∗circuit_vector, double performance, double recovery, double grade)

  Saves the evaluation results to a text file.

4.2.1 Function Documentation

4.2.1.1 Evaluate_Circuit() [1/2]

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

Evaluates the performance of a circuit using default parameters.

Parameters

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

Returns

double The performance value of the circuit.

Definition at line 197 of file CSimulator.cpp.

```
197                                                          {
198     return Evaluate_Circuit(vector_size, circuit_vector, default_circuit_parameters);
199 };
```

4.2.1.2 Evaluate_Circuit() [2/2]

```
double Evaluate_Circuit (
            int vector_size,
            int * circuit_vector,
            struct Circuit_Parameters parameters )
```

Evaluates the performance of a circuit based on the provided circuit vector and parameters.

This function takes a circuit vector and returns a performance value. The performance is calculated based on the concentrate and waste streams. The function runs for a specified number of iterations or until the change in performance is below the tolerance threshold.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The vector representing the circuit.
parameters	The parameters for the circuit evaluation.

Returns

double The performance value of the circuit.

Definition at line 27 of file CSimulator.cpp.

```
27 {
28   int Unit_Size = int(vector_size-1) / 3;
29   Circuit circuit(Unit_Size);
30   circuit.setup_connections(circuit_vector, vector_size);
31
32   double Performance = 0;
33
34   for (int i = 0; i < parameters.max_iterations; i++) {
35     double total_concentrate = circuit.concentrate_G + circuit.concentrate_W;
36     circuit.update_flows();
37     double new_total_concentrate = circuit.concentrate_G + circuit.concentrate_W;
38
39     Performance = circuit.concentrate_G*100 - circuit.concentrate_W*750;
40
41     if (new_total_concentrate != total_concentrate) {
42       double difference = new_total_concentrate - total_concentrate;
43
44       if (difference <= parameters.tolerance){
45
46         return Performance;
47         break;
48       }
49     }
50   }
51   return -67500.0;
52 }
```

4.2.1.3 Evaluate_Circuit_Cell_Calculator()

```
void Evaluate_Circuit_Cell_Calculator (
        Unit_parameter & parameters )
```

Calculates the parameters of a single unit in the circuit.

This function calculates various parameters of a single unit in the circuit, including residence time, concentrate and waste flows, and recovery values for different grades. The results are stored in the provided parameters structure.

Parameters

parameters	A reference to a Unit_parameter structure to store the calculated values.

Definition at line 133 of file CSimulator.cpp.

```
133 {
134   CUnit unit;
135   unit.feed_rate_G = 10;
136   unit.feed_rate_W = 90;
137   double total_feed = unit.feed_rate_G+unit.feed_rate_W;
138   unit.Calculate_ResidenceTime(total_feed);
139   parameters.tau = unit.residence_time;
140   parameters.CG = unit.calculate_CG();
141   parameters.CW = unit.calculate_CW();
142   parameters.IG = unit.calculate_IG();
143   parameters.IW = unit.calculate_IW();
144   parameters.TG = unit.calculate_TG();
145   parameters.TW = unit.calculate_TW();
146   parameters.RHG = unit.Calculate_RHG();
147   parameters.RHW = unit.Calculate_RHW();
148   parameters.RIG = unit.Calculate_RIG();
149   parameters.RIW = unit.Calculate_RIW();
150 }
```

4.2.1.4 Evaluate_Circuit_Grade()

```
double Evaluate_Circuit_Grade (
```

```
int vector_size,
int * circuit_vector )
```

Evaluates the grade of a circuit using default parameters.

Parameters

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

Returns

double The grade value of the circuit.

Definition at line 219 of file CSimulator.cpp.

```
219 {
220     return Evaluate_Circuit_Grade(vector_size, circuit_vector, default_circuit_parameters);
221 };
```

4.2.1.5 Evaluate_Circuit_Recovery()

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

Evaluates the recovery of a circuit using default parameters.

Parameters

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

Returns

double The recovery value of the circuit.

Definition at line 208 of file CSimulator.cpp.

```
208 {
209     return Evaluate_Circuit_Recovery(vector_size, circuit_vector, default_circuit_parameters);
210 };
```

4.2.1.6 Save_ResultToTxt()

```
void Save_ResultToTxt (
            const std::string & filename,
            int vector_size,
            int * circuit_vector,
            double performance,
```

```
            double recovery,
            double grade )
```

Saves the evaluation results to a text file.

This function writes the input circuit vector and the calculated performance, recovery, and grade values to a text file with the specified filename.

Parameters

filename	The name of the file to save the results to.
vector_size	The size of the circuit vector.
circuit_vector	The vector representing the circuit.
performance	The performance value of the circuit.
recovery	The recovery value of the circuit.
grade	The grade value of the circuit.

Definition at line 165 of file CSimulator.cpp.

```
165
      {
166   std::ofstream out_file(filename, std::ios::out); // Open the file for writing
167
168   if (!out_file) {
169     std::cerr « "Error opening file"« filename « std::endl;
170     return;
171   }
172
173   // Write the input vector to the file
174   out_file « "Input vector:  ";
175   for (int i = 0; i < vector_size; i++) {
176     out_file « circuit_vector[i] « " ";
177   }
178   out_file « std::endl;
179
180   // Write the performance, recovery, and grade to the file
181   out_file « "Performance:  " « performance « std::endl;
182   out_file « "Recovery:  " « recovery « std::endl;
183   out_file « "Grade:  " « grade « std::endl;
184
185   out_file.close(); // Close the file
186 }
```

4.3 /home/bz223/downloads/acs-gerardium-rush-bauxite/include/CUnit.h File Reference

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

Classes

- class CUnit

4.4 /home/bz223/downloads/acs-gerardium-rush-bauxite/include/↩ Genetic_Algorithm.h File Reference

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

Include dependency graph for Genetic_Algorithm.h: This graph shows which files directly or indirectly include this file:

Classes

- struct Algorithm_Parameters

Macros

- #define DEFAULT_ALGORITHM_PARAMETERS Algorithm_Parameters{10000, 500, 0.8, 0.01, 20, 0.5}

Functions

- bool all_true (int vector_size, int ∗vector)
- int optimize (int vector_size, int ∗vector, double(&func)(int, int ∗), bool(&validity)(int, int ∗)=all_true, struct Algorithm_Parameters parameters=DEFAULT_ALGORITHM_PARAMETERS)

  Optimizes an input vector using a genetic algorithm based on a fitness function and validity checks.
- vector< vector< int > > generate_population (int vector_size, Circuit circuit, struct Algorithm_Parameters parameters=DEFAULT_ALGORITHM_PARAMETERS)

  Generates the initial population of vectors for the genetic algorithm.
- pair< int, int > select_parents (vector< double > fitness, struct Algorithm_Parameters parameters=DEFAULT_ALGORITHM_PA

  Selects pairs of parent vectors for breeding based on their fitness scores.
- vector< double > evaluate_parents (vector< vector< int >> parents, vector< int > &current_best, int &best_index, double(&func)(int, int ∗))

  Evaluates the fitness of each vector in the current population and identifies the best one.
- int generate_children (bool stalled, vector< vector< int >> &new_population, vector< int > parent1, vector< int > parent2, Circuit circuit, Algorithm_Parameters parameters=DEFAULT_ALGORITHM_PARAMETERS)

  Generates children from selected parent vectors, applying crossover and mutation operations.
- vector< vector< int > > crossover (vector< int > parent1, vector< int > parent2, string type, struct Algorithm_Parameters parameters=DEFAULT_ALGORITHM_PARAMETERS)

  Performs genetic crossover between two parent vectors based on a specified type or randomly selects a crossover type.
- pair< vector< int >, vector< int > > uniform_cross (vector< int > parent1, vector< int > parent2)

  Performs a uniform crossover between two parent vectors, randomly exchanging genes.
- pair< vector< int >, vector< int > > one_point_cross (vector< int > parent1, vector< int > parent2)

  Executes a one-point crossover between two parent vectors.
- pair< vector< int >, vector< int > > two_point_cross (vector< int > parent1, vector< int > parent2)

  Conducts a two-point crossover between two parent vectors.
- pair< vector< int >, vector< int > > multipoint_cross (vector< int > parent1, vector< int > parent2)

  Performs a multipoint crossover on two parent vectors.
- vector< int > substitution (vector< int > child, double mutation_rate)
- vector< int > inversion (vector< int > child, double mutation_rate)
- vector< int > value_adjustment (vector< int > child, double mutation_rate)
- vector< int > mutate (bool stalled, vector< int > child, double mutation_rate, double stall_mutation_rate)

  Mutates a vector based on a mutation rate, with increased mutation rates if improvements have stalled.

4.4.1 Macro Definition Documentation

4.4.1.1 DEFAULT_ALGORITHM_PARAMETERS

```
#define DEFAULT_ALGORITHM_PARAMETERS Algorithm_Parameters{10000, 500, 0.8, 0.01, 20, 0.5}
```

Definition at line 22 of file Genetic_Algorithm.h.

4.4.2 Function Documentation

4.4.2.1 all_true()

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

Definition at line 63 of file Genetic_Algorithm.cpp.

```
63                                  {
64     return true;
65 }
```

4.4.2.2 crossover()

```
vector<vector<int> > crossover (
            vector< int > parent1,
            vector< int > parent2,
            string type,
            Algorithm_Parameters parameters )
```

Performs genetic crossover between two parent vectors based on a specified type or randomly selects a crossover type.

This function chooses a crossover strategy (uniform, one-point, two-point, or multipoint) to combine genetic material from two parent vectors. The choice of crossover strategy can be specified or randomly determined, affecting how segments of the parents' vectors are mixed to produce new child vectors.

Parameters

parent1	The first parent vector from which genetic material is taken.
parent2	The second parent vector from which genetic material is taken.
type	A string specifying the type of crossover; if "random", a method is randomly selected.
parameters	Struct containing parameters for the genetic algorithm, influencing how crossover is performed.

Returns

vector<vector<int>> Returns two new child vectors resulting from the crossover operation.

Definition at line 440 of file Genetic_Algorithm.cpp.

```
{
  pair<vector<int>, vector<int» new_children;

  // if the type is random, select a random crossover function, otherwise select the type that was given
  if (type == "random")
  {
    int random = rand() % 4;
    if (random == 0)
      new_children = uniform_cross(parent1, parent2);
    else if (random == 1)
      new_children = one_point_cross(parent1, parent2);
    else if (random == 2)
      new_children = two_point_cross(parent1, parent2);
    else
      new_children = multipoint_cross(parent1, parent2);
  }
  else if (type == "one_point")
    new_children = one_point_cross(parent1, parent2);
  else if (type == "two_point")
    new_children = two_point_cross(parent1, parent2);
  else if (type == "multipoint")
    new_children = multipoint_cross(parent1, parent2);
  else if (type == "uniform")
    new_children = uniform_cross(parent1, parent2);

  // return child1 and child2 in a vector
  vector<vector<int» children = vector<vector<int»{new_children.first, new_children.second};

  return children;
}
```

4.4.2.3 evaluate_parents()

```
vector<double> evaluate_parents (
            vector< vector< int >> parents,
            vector< int > & current_best,
            int & best_index,
            double(&)(int, int *) func )
```

Evaluates the fitness of each vector in the current population and identifies the best one.

The function calculates the fitness for each vector in the population using a provided fitness function. It updates the best vector found in the current generation based on the highest fitness score.

Parameters

parents	A reference to the vector of parent vectors.
current_best	A reference to the vector which will store the best vector of the current generation.
best_index	A reference to an integer to store the index of the best vector.
func	A function pointer to the fitness function used to evaluate vectors.

Returns

vector<double> Returns a vector of fitness scores corresponding to each parent vector.

Definition at line 296 of file Genetic_Algorithm.cpp.

```
{
  double current_value = 0;
  vector<double> fitness(parents.size());
  // evaluate the fitness of all vectors in the current generation
#pragma omp parallel for private(current_value) schedule(dynamic)
  for (int i = 0; i < parents.size(); i++)
  {
    current_value = func(parents[i].size(), parents[i].data());
```

```
        fitness[i] = current_value;
    }
    // find the index and corresponding vector that have the highest fitness

    best_index = distance(fitness.begin(), max_element(fitness.begin(), fitness.end()));
    current_best = parents[best_index];

    return fitness;
}
```

4.4.2.4 generate_children()

```
int generate_children (
            bool stalled,
            vector< vector< int >> & new_population,
            vector< int > parent1,
            vector< int > parent2,
            Circuit circuit,
            Algorithm_Parameters parameters )
```

Generates children from selected parent vectors, applying crossover and mutation operations.

This function performs crossover and mutation to generate new vectors from the selected parents. It further ensures that only valid vectors are added to the new population.

Parameters

stalled	Boolean flag indicating whether the optimization has stalled.
new_population	Reference to the vector that will hold the new population.
parent1	First parent vector.
parent2	Second parent vector.
circuit	An instance of the Circuit class used for validating the vectors.
parameters	The parameters of the genetic algorithm, including crossover probability and mutation rates.

Returns

int The number of valid children added to the new population.

Definition at line 391 of file Genetic_Algorithm.cpp.

```
{

    int count = 0;
    // Decide if parents crossover and create new vectors before applying mutations
    // only add valid circuits to the new generation
    if (parameters.cross_probability < (rand() % 100) / 100)
    {
        vector<vector<int» children = crossover(parent1, parent2, "random", parameters);
        for (auto child :  children)
        {
            child = mutate(stalled, child, parameters.mutation_rate, parameters.stall_mutation_rate);
            if (circuit.Check_Validity(child.size(), child.data()))
            {
                new_population.push_back(child);
                count++;
            }
        }
    }
    else
    {
        parent1 = mutate(stalled, parent1, parameters.mutation_rate, parameters.stall_mutation_rate);
        if (circuit.Check_Validity(parent1.size(), parent1.data()))
```

```
      {
        new_population.push_back(parent1);
        count++;
      }
    parent2 = mutate(stalled, parent2, parameters.mutation_rate, parameters.stall_mutation_rate);
    if (circuit.Check_Validity(parent2.size(), parent2.data()))
      {
        new_population.push_back(parent2);
        count++;
      }
  }
  return count;
}
```

4.4.2.5 generate_population()

```
vector<vector<int> > generate_population (
              int vector_size,
              Circuit circuit,
              Algorithm_Parameters parameters )
```

Generates the initial population of vectors for the genetic algorithm.

Initializes the population for the genetic algorithm. Each vector within the population is generated to meet specific validity criteria as determined by the circuit validity function. Random values are assigned to each element of the vector except the first, which is set to zero. There is an additional check to ensure that no unit is looping back on itself

Parameters

vector_size	The size of vectors in the population.
circuit	An instance of the Circuit class used for validating the vectors.
parameters	The parameters of the genetic algorithm including population size.

Returns

vector<vector<int>> Returns a population of valid vectors.

Definition at line 202 of file Genetic_Algorithm.cpp.

```
{
  vector<vector<int» population(parameters.population_size);
  bool condition = false;
  int pop_size = 0;
  // add vectors to the population until the desired size is reached
  for (int i = 0; i < parameters.population_size; i++)
  {
    vector<int> individual;
    do
    {
      individual = vector<int>(vector_size);
      // start with each circuit feeding into 0
      individual[0] = 0;
      // give all other connections random values
      for (int j = 1; j < vector_size; j++)
      {
        int value = (rand() % max_value);
        if (value != (j-1)/3)
        {
          individual[j] = value;
        } else
        {
          individual[j] = (value +1) % max_value;
        }
      }
```

```
      // check that the generated circuit is valid
      condition = circuit.Check_Validity(individual.size(), individual.data());
    } while (!condition);

    population[i] = individual;
  }
  return population;
}
```

4.4.2.6 inversion()

```
vector<int> inversion (
            vector< int > child,
            double mutation_rate )
```

Definition at line 613 of file Genetic_Algorithm.cpp.

```
{
  if (mutation_rate > (rand() % 1000) / 1000)
  {
    // take a section of the child and invert it
    int mutation_index = rand() % (child.size()/2);
    int inversion_size =  rand() % (child.size()-mutation_index);
    reverse(child.begin() + mutation_index, child.begin() + mutation_index + inversion_size);
  }
  return child;
}
```

4.4.2.7 multipoint_cross()

```
pair<vector<int>, vector<int> > multipoint_cross (
            vector< int > parent1,
            vector< int > parent2 )
```

Performs a multipoint crossover on two parent vectors.

Several crossover points are randomly determined along the length of the vectors. The genes located at these points are then swapped between the two parents, resulting in a high level of gene shuffling. This approach can create significant genetic diversity within the offspring.

Parameters

parent1	First parent vector for crossover.
parent2	Second parent vector for crossover.

Returns

pair<vector<int>, vector<int>> Returns two child vectors with highly mixed genetic material from both parents.

Definition at line 571 of file Genetic_Algorithm.cpp.

```
{
  // crosses m points of the parent vectors
  vector<int> child1;
  vector<int> child2;
  vector<int> points;
```

```
  // get the crossover points, making sure no duplicates are selected
  for (int i = 0; i < parent1.size()/2; i++)
  {
    int point = 0;
    do
    {
      point = rand() % parent1.size();
    } while (find(points.begin(), points.end(), point) != points.end());
    points.push_back(point);
  }
  int temp;
  // only crossover the crossover_point
  for (int point :  points)
  {
    temp = parent1[point];
    parent1[point] = parent2[point];
    parent2[point] = temp;
  }
  return make_pair(parent1, parent2);
}
```

4.4.2.8 mutate()

```
vector<int> mutate (
            bool stalled,
            vector< int > child,
            double mutation_rate,
            double stall_mutation_rate )
```

Mutates a vector based on a mutation rate, with increased mutation rates if improvements have stalled.

This function applies mutations to a vector, potentially altering its elements to explore new genetic combinations. The type and intensity of mutations may increase if progress towards optimization has stalled. Note it is possible for more than one mutation to take place on the same vector.

Parameters

stalled	Boolean flag indicating whether the optimization has stalled.
child	The vector to mutate.
mutation_rate	The base rate of mutation.
stall_mutation_rate	The increased mutation rate to apply if stalled.

Returns

vector<int> Returns the mutated vector.

Definition at line 329 of file Genetic_Algorithm.cpp.

```
{
  // make mutations more aggressive if the improvement has stalled
  if (stalled){
    mutation_rate = stall_mutation_rate;
  }
  // substitution
  if (mutation_rate > (rand() % 1000) / 1000.0)
  {
    // substitute a random gene with a random value
    int mutation_index = rand() % child.size();
    child[mutation_index] = rand() % max_value;
  }
  // inversion
  if (mutation_rate > (rand() % 1000) / 1000.0)
  {
    // take a section of the child and invert it
    int mutation_index = rand() % (child.size()/2);
```

```
    int inversion_size =  rand() % (child.size()-mutation_index);
    reverse(child.begin() + mutation_index, child.begin() + mutation_index + inversion_size);
  }
  // value adjustment
  if (mutation_rate > (rand() % 1000) / 1000.0)
  {
    int mutation_index = rand() % child.size();
    int mutation_size = rand() % (child.size()-mutation_index);
    // add 1 to a section of the vector
    for (int i =0; i<mutation_size; i++)
    {
      int index = (mutation_index + i) % child.size();
      child[index] = (child[index] + 1) %max_value;
    }
  }
  // value adjustment down
  if (mutation_rate > (rand() % 1000) / 1000.0)
  {
    int mutation_index = rand() % child.size();
    int mutation_size = rand() % (child.size()-mutation_index);
    // subtract 1 from a section of the vector
    for (int i =0; i<mutation_size; i++)
    {
      int index = (mutation_index + i) % child.size();
      child[index] = (child[index] - 1) %max_value;
    }
  }

  return child;
}
```

4.4.2.9 one_point_cross()

```
pair<vector<int>, vector<int> > one_point_cross (
              vector< int > parent1,
              vector< int > parent2 )
```

Executes a one-point crossover between two parent vectors.

This function selects a random point in the vector, and all the genes (vector elements) beyond that point are swapped between the two parents. This results in two offspring, each sharing a block of genes with each parent.

Parameters

parent1	First parent vector for crossover.
parent2	Second parent vector for crossover.

Returns

pair<vector<int>, vector<int>> Returns two child vectors, each inheriting a contiguous block of genes from both parents.

Definition at line 516 of file Genetic_Algorithm.cpp.

```
{
  // crosses one point of the parent vectors
  // get the crossover point
  int crossover_point = rand() % parent1.size();
  // only crossover the crossover_point
  int temp = parent1[crossover_point];
  parent1[crossover_point] = parent2[crossover_point];
  parent2[crossover_point] = temp;

  return make_pair(parent1, parent2);
}
```

4.4.2.10 optimize()

```
int optimize (
            int vector_size,
            int * input_vector,
            double(&)(int, int *) func,
            bool(&)(int, int *) validity,
            struct Algorithm_Parameters parameters )
```

Optimizes an input vector using a genetic algorithm based on a fitness function and validity checks.

This function simulates a genetic algorithm process to find the optimal vectore to maximize a given function. It generates an initial population, evaluates fitness, selects parents, generates children, and mutates them across several generations until the maximum number of iterations is reached or improvement stalls. The final best solution found replaces the original input vector.

Parameters

vector_size	Size of the input vector.
input_vector	Pointer to the input vector that will be optimized.
func	A function that evaluates the fitness of the vector.
validity	A function that checks the validity of the vector.
parameters	Struct containing parameters for the genetic algorithm (e.g., population size, mutation rate).

Returns

int Returns 0 if the optimization completes before reaching the maximum iteration limit, 1 otherwise.

Definition at line 80 of file Genetic_Algorithm.cpp.

```
                                                            {
// Initializing variables
vector<vector<int» population;
vector<vector<int» new_population;
vector<int> current_best;
pair<int, int> parent_pair;
vector<double> fitness;
bool stalled = false;


// lists used for output generation
vector<int> generation_list;
vector<double> best_values_list;


Circuit circuit = Circuit((vector_size-1) / 3);
// calculate the maximum value
max_value = (vector_size -1)/3 + 2;
 cout « "max_value" « max_value« endl;

 cout « "printing the vector" « endl;
 for (int i = 0; i < vector_size; ++i) {
     cout « input_vector[i] « " ";
 }
 cout « endl;
 // Generate the initial population
 auto start = std::chrono::high_resolution_clock::now();
 population = generate_population(vector_size, circuit, parameters);
 auto end = std::chrono::high_resolution_clock::now();

 double new_best_value = func(vector_size, input_vector);
 std::chrono::duration<double> elapsed_seconds = end-start;
 std::cout « "Time to generate population:  " « elapsed_seconds.count() « "s\n";

 int current_generation = 0;
 int stall_count = 0;
 start = std::chrono::high_resolution_clock::now();
 do
 {
```

```
    // Run the genetic algorithm process
    int best_index = 0;
    // Calculate the fitness of the parents to use in the selection of parents
    fitness = evaluate_parents(population, current_best, best_index, func);
    for (int child_count = 0; child_count < parameters.population_size;)
    {
      // select a pair of parent and create children based on them
      // only add valid children to the new population
      parent_pair = select_parents(fitness, parameters);
      int children = generate_children(stalled, new_population, population[parent_pair.first],
  population[parent_pair.second], circuit, parameters);
      child_count += children;
    }
    // Add the best vector from the previous generation to the new generation unaltered
    new_population.push_back(current_best);

    // check if the new best value is better than the previous best and swap if it is
    // if not add to the stall count
    double value = func(current_best.size(), current_best.data());
    cout « "\rGeneration " « current_generation « " best value " « value « flush;
    if (value > new_best_value)
    {
      new_best_value = value;
      stall_count = 0;
      stalled=false;
      //generation_list.push_back(current_generation);
      //best_values_list.push_back(new_best_value);
    }else
    {stall_count++;}
    // change new and old population and clear new population for the next cycle
    population.swap(new_population);
    new_population.clear();
    // check if the calculation has stalled
    if (stall_count >= parameters.stall_length){
      stalled = true;
    }
  }

  while (parameters.max_iterations > current_generation++ && stall_count < 100);
  cout « endl;
  end = std::chrono::high_resolution_clock::now();


  elapsed_seconds = end-start;
  std::cout « "Time to run simulation:  " « elapsed_seconds.count() « "s\n";

  cout « "input vector" « endl;
  for (int i = 0; i < current_best.size(); i++)
    cout « input_vector[i] « " ";
  cout « endl;

  // Update the vector with the best solution found
  for (int i = 0; i < current_best.size(); i++)
    input_vector[i] = current_best[i];
  cout « "Output vector" « endl;
  for (int i = 0; i < current_best.size(); i++)
    cout « input_vector[i] « ",";
  cout « endl;
  double final_value = func(vector_size, input_vector);
  cerr « "Final score " « final_value « endl;

  //write_output("output.csv");
  if (current_generation >= parameters.max_iterations)
    return 1;

  return 0;

}
```

4.4.2.11 select_parents()

```
pair<int, int> select_parents (
            vector< double > fitness,
            struct Algorithm_Parameters parameters )
```

Selects pairs of parent vectors for breeding based on their fitness scores.

This function uses a weighted probability method to select two parents from the current population. The fitness of each vector influences its likelihood of being chosen as a parent.

Parameters

fitness	A vector of doubles representing the fitness of each vector in the population.
parameters	Struct containing parameters for the genetic algorithm, including selection details.

Returns

pair<int, int> Returns a pair of indices representing the selected parents from the population.

Definition at line 272 of file Genetic_Algorithm.cpp.

```
{
  // choose two parents to potentially crossover
  int new_generation_size = 0;
  double sum_of_fitness = 0;
  for (int i = 0; i < fitness.size(); i++)
    sum_of_fitness += fitness[i];

  // Return the selected parents
  return make_pair(weighted_parent(sum_of_fitness, fitness), weighted_parent(sum_of_fitness, fitness));
}
```

4.4.2.12 substitution()

```
vector<int> substitution (
            vector< int > child,
            double mutation_rate )
```

Definition at line 602 of file Genetic_Algorithm.cpp.

```
{
  if (mutation_rate > (rand() % 1000) / 1000)
  {
    // substitute a random gene with a random value
    int mutation_index = rand() % child.size();
    child[mutation_index] = rand() % 10;
  }
  return child;
}
```

4.4.2.13 two_point_cross()

```
pair<vector<int>, vector<int> > two_point_cross (
            vector< int > parent1,
            vector< int > parent2 )
```

Conducts a two-point crossover between two parent vectors.

Two random points are chosen within the vectors, and the genetic material enclosed by these points is swapped between the two parents. This method allows for a more localized alteration of the genetic structure compared to one-point crossover.

Parameters

parent1	First parent vector.
parent2	Second parent vector.

Returns

pair<vector<int>, vector<int>> Returns two new child vectors, each containing segments of genes exchanged between the parents.

Definition at line 539 of file Genetic_Algorithm.cpp.

```
{
  // Similar to one points, but crossover point is two points wide
  vector<int> points;
  // get the crossover points
  points.push_back(rand() % parent1.size());
  // set the second point to be one higher than the crossover_point1 or the first element depending on the boundary
  points.push_back((points[0]+ 1) % parent1.size());
  int temp;
  // only crossover the crossover_point
  for (int point :  points)
  {
    temp = parent1[point];
    parent1[point] = parent2[point];
    parent2[point] = temp;
  }

  return make_pair(parent1, parent2);
}
```

4.4.2.14 uniform_cross()

```
pair<vector<int>, vector<int> > uniform_cross (
            vector< int > parent1,
            vector< int > parent2 )
```

Performs a uniform crossover between two parent vectors, randomly exchanging genes.

In uniform crossover, each gene (vector element) from the parents has a 50% chance of being swapped. This method provides a high degree of mixing and can generate highly varied offspring. Each gene is considered independently, making it possible to achieve diverse genetic combinations.

Parameters

parent1	First parent vector for crossover.
parent2	Second parent vector for crossover.

Returns

pair<vector<int>, vector<int>> Returns a pair of child vectors, each containing mixed genes from both parents.

Definition at line 482 of file Genetic_Algorithm.cpp.

```
{
  // uniform, take two from one, two from other until end
  // actually just takes one from parent1 and one from parent2 till the end
  // vice verca for the second child
  vector<int> child1(parent1.size());
  vector<int> child2(parent2.size());

  for (int i = 1; i <= parent1.size(); i++)
  {
    if (i % 2)
    {
      child1[i - 1] = parent1[i - 1];
      child2[i - 1] = parent2[i - 1];
    }
```

```
    else
    {
      child1[i - 1] = parent2[i - 1];
      child2[i - 1] = parent1[i - 1];
    }
  }

  return make_pair(child1, child2);
}
```

4.4.2.15 value_adjustment()

```
vector<int> value_adjustment (
            vector< int > child,
            double mutation_rate )
```

Definition at line 625 of file Genetic_Algorithm.cpp.

```
{
  if (mutation_rate > (rand() % 1000) / 1000)
  {
    int mutation_index = rand() % child.size();
    int mutation_size = rand() % (child.size()/2);
    for (int i =0; i<mutation_size; i++)
    {
      int index = (mutation_index + i) % child.size();
      child[index] = (child[index] + 1) %10;
    }
  }
  return child;
}
```

4.5 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/↵ CCircuit.cpp File Reference

```
#include <vector>
#include <iostream>
#include <cstdlib>
#include <queue>
#include <stdio.h>
#include <CUnit.h>
#include <CCircuit.h>
```

Include dependency graph for CCircuit.cpp:

4.6 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/↵ CSimulator.cpp File Reference

```
#include "CUnit.h"
#include "CCircuit.h"
#include "CSimulator.h"
#include <iostream>
#include <cmath>
#include <fstream>
#include <string.h>
```

Include dependency graph for CSimulator.cpp:

Functions

- double Evaluate_Circuit (int vector_size, int ∗circuit_vector, struct Circuit_Parameters parameters)
  Evaluates the performance of a circuit based on the provided circuit vector and parameters.
- double Evaluate_Circuit_Recovery (int vector_size, int ∗circuit_vector, struct Circuit_Parameters parameters)
  Evaluates the recovery of a circuit based on the provided circuit vector and parameters.
- double Evaluate_Circuit_Grade (int vector_size, int ∗circuit_vector, struct Circuit_Parameters parameters)
  Evaluates the grade of a circuit based on the provided circuit vector and parameters.
- void Evaluate_Circuit_Cell_Calculator (Unit_parameter &parameters)
  Calculates the parameters of a single unit in the circuit.
- void Save_ResultToTxt (const std::string &filename, int vector_size, int ∗circuit_vector, double performance, double recovery, double grade)
  Saves the evaluation results to a text file.
- double Evaluate_Circuit (int vector_size, int ∗circuit_vector)
  Evaluates the performance of a circuit using default parameters.
- double Evaluate_Circuit_Recovery (int vector_size, int ∗circuit_vector)
  Evaluates the recovery of a circuit using default parameters.
- double Evaluate_Circuit_Grade (int vector_size, int ∗circuit_vector)
  Evaluates the grade of a circuit using default parameters.

Variables

- struct Circuit_Parameters default_circuit_parameters = {1e-6, 1000}
- int dummy_answer_vector [ ] = {0, 1, 2, 2, 3, 4, 2, 1, 4, 4, 5, 4, 4, 1, 2, 6}

4.6.1 Function Documentation

4.6.1.1 Evaluate_Circuit() [1/2]

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

Evaluates the performance of a circuit using default parameters.

Parameters

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

Returns

double The performance value of the circuit.

Definition at line 197 of file CSimulator.cpp.

```
197 {
```

```
    return Evaluate_Circuit(vector_size, circuit_vector, default_circuit_parameters);
};
```

4.6.1.2 Evaluate_Circuit() [2/2]

```
double Evaluate_Circuit (
            int vector_size,
            int * circuit_vector,
            struct Circuit_Parameters parameters )
```

Evaluates the performance of a circuit based on the provided circuit vector and parameters.

This function takes a circuit vector and returns a performance value. The performance is calculated based on the concentrate and waste streams. The function runs for a specified number of iterations or until the change in performance is below the tolerance threshold.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The vector representing the circuit.
parameters	The parameters for the circuit evaluation.

Returns

double The performance value of the circuit.

Definition at line 27 of file CSimulator.cpp.

```
27                                                                                                      {
28   int Unit_Size = int(vector_size-1) / 3;
29   Circuit circuit(Unit_Size);
30   circuit.setup_connections(circuit_vector, vector_size);
31
32   double Performance = 0;
33
34   for (int i = 0; i < parameters.max_iterations; i++) {
35     double total_concentrate = circuit.concentrate_G + circuit.concentrate_W;
36     circuit.update_flows();
37     double new_total_concentrate = circuit.concentrate_G + circuit.concentrate_W;
38
39     Performance = circuit.concentrate_G*100 - circuit.concentrate_W*750;
40
41     if (new_total_concentrate != total_concentrate) {
42       double difference = new_total_concentrate - total_concentrate;
43
44       if (difference <= parameters.tolerance){
45
46         return Performance;
47         break;
48       }
49     }
50   }
51   return -67500.0;
52 }
```

4.6.1.3 Evaluate_Circuit_Cell_Calculator()

```
void Evaluate_Circuit_Cell_Calculator (
            Unit_parameter & parameters )
```

Calculates the parameters of a single unit in the circuit.

This function calculates various parameters of a single unit in the circuit, including residence time, concentrate and waste flows, and recovery values for different grades. The results are stored in the provided parameters structure.

Parameters

parameters	A reference to a Unit_parameter structure to store the calculated values.

Definition at line 133 of file CSimulator.cpp.

```
133                                                              {
134   CUnit unit;
135   unit.feed_rate_G = 10;
136   unit.feed_rate_W = 90;
137   double total_feed = unit.feed_rate_G+unit.feed_rate_W;
138   unit.Calculate_ResidenceTime(total_feed);
139   parameters.tau = unit.residence_time;
140   parameters.CG = unit.calculate_CG();
141   parameters.CW = unit.calculate_CW();
142   parameters.IG = unit.calculate_IG();
143   parameters.IW = unit.calculate_IW();
144   parameters.TG = unit.calculate_TG();
145   parameters.TW = unit.calculate_TW();
146   parameters.RHG = unit.Calculate_RHG();
147   parameters.RHW = unit.Calculate_RHW();
148   parameters.RIG = unit.Calculate_RIG();
149   parameters.RIW = unit.Calculate_RIW();
150 }
```

4.6.1.4 Evaluate_Circuit_Grade() [1/2]

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

Evaluates the grade of a circuit using default parameters.

Parameters

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

Returns

double The grade value of the circuit.

Definition at line 219 of file CSimulator.cpp.

```
219                                                               {
220     return Evaluate_Circuit_Grade(vector_size, circuit_vector, default_circuit_parameters);
221 };
```

4.6.1.5 Evaluate_Circuit_Grade() [2/2]

```
double Evaluate_Circuit_Grade (
            int vector_size,
```

```
            int * circuit_vector,
            struct Circuit_Parameters parameters )
```

Evaluates the grade of a circuit based on the provided circuit vector and parameters.

This function takes a circuit vector and returns a grade value. The grade is calculated based on the concentrate and waste streams. The function runs for a specified number of iterations or until the change in grade is below the tolerance threshold.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The vector representing the circuit.
parameters	The parameters for the circuit evaluation.

Returns

double The grade value of the circuit.

Definition at line 101 of file CSimulator.cpp.

```
101
    {
102   int Unit_Size = int(vector_size-1) / 3;
103   Circuit circuit(Unit_Size);
104   circuit.setup_connections(circuit_vector, vector_size);
105
106   double Grade = 0;
107
108   for (int i = 0; i < parameters.max_iterations; i++) {
109     double total_concentrate = circuit.concentrate_G + circuit.concentrate_W;
110     circuit.update_flows();
111     double new_total_concentrate = circuit.concentrate_G + circuit.concentrate_W;
112     Grade = circuit.concentrate_G / (circuit.concentrate_W+ circuit.concentrate_G);
113     if (new_total_concentrate != total_concentrate) {
114       double difference = new_total_concentrate - total_concentrate;
115       if (difference < parameters.tolerance){
116         return Grade;
117         break;
118       }
119     }
120   }
121   return Grade;
122 }
```

4.6.1.6 Evaluate_Circuit_Recovery() [1/2]

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

Evaluates the recovery of a circuit using default parameters.

Parameters

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

Returns

double The recovery value of the circuit.

Definition at line 208 of file CSimulator.cpp.

```
208 {
209     return Evaluate_Circuit_Recovery(vector_size, circuit_vector, default_circuit_parameters);
210 };
```

4.6.1.7 Evaluate_Circuit_Recovery() [2/2]

```
double Evaluate_Circuit_Recovery (
            int vector_size,
            int * circuit_vector,
            struct Circuit_Parameters parameters )
```

Evaluates the recovery of a circuit based on the provided circuit vector and parameters.

This function takes a circuit vector and returns a recovery value. The recovery is calculated based on the concentrate stream. The function runs for a specified number of iterations or until the change in recovery is below the tolerance threshold.

Parameters

vector_size	The size of the circuit vector.
circuit_vector	The vector representing the circuit.
parameters	The parameters for the circuit evaluation.

Returns

double The recovery value of the circuit.

Definition at line 66 of file CSimulator.cpp.

```
66 {
67   int Unit_Size = int(vector_size-1) / 3;
68   Circuit circuit(Unit_Size);
69   circuit.setup_connections(circuit_vector, vector_size);
70
71   double Recovery = 0;
72
73   for (int i = 0; i < parameters.max_iterations; i++) {
74     double total_concentrate = circuit.concentrate_G + circuit.concentrate_W;
75     circuit.update_flows();
76     double new_total_concentrate = circuit.concentrate_G + circuit.concentrate_W;
77     Recovery = circuit.concentrate_G/10;
78     if (new_total_concentrate != total_concentrate) {
79       double difference = new_total_concentrate - total_concentrate;
80       if (difference < parameters.tolerance){
81         return Recovery;
82         break;
83       }
84     }
85   }
86   return Recovery;
87 }
```

4.6.1.8 Save_ResultToTxt()

```
void Save_ResultToTxt (
            const std::string & filename,
            int vector_size,
            int * circuit_vector,
            double performance,
            double recovery,
            double grade )
```

Saves the evaluation results to a text file.

This function writes the input circuit vector and the calculated performance, recovery, and grade values to a text file with the specified filename.

Parameters

filename	The name of the file to save the results to.
vector_size	The size of the circuit vector.
circuit_vector	The vector representing the circuit.
performance	The performance value of the circuit.
recovery	The recovery value of the circuit.
grade	The grade value of the circuit.

Definition at line 165 of file CSimulator.cpp.

```
165
    {
166   std::ofstream out_file(filename, std::ios::out); // Open the file for writing
167
168   if (!out_file) {
169     std::cerr « "Error opening file"« filename « std::endl;
170     return;
171   }
172
173   // Write the input vector to the file
174   out_file « "Input vector:  ";
175   for (int i = 0; i < vector_size; i++) {
176     out_file « circuit_vector[i] « " ";
177   }
178   out_file « std::endl;
179
180   // Write the performance, recovery, and grade to the file
181   out_file « "Performance:  " « performance « std::endl;
182   out_file « "Recovery:  " « recovery « std::endl;
183   out_file « "Grade:  " « grade « std::endl;
184
185   out_file.close(); // Close the file
186 }
```

4.6.2 Variable Documentation

4.6.2.1 default_circuit_parameters

```
struct Circuit_Parameters default_circuit_parameters = {1e-6, 1000}
```

Definition at line 1 of file CSimulator.cpp.

4.6.2.2 dummy_answer_vector

```
int dummy_answer_vector[] = {0, 1, 2, 2, 3, 4, 2, 1, 4, 4, 5, 4, 4, 1, 2, 6}
```

Definition at line 13 of file CSimulator.cpp.

4.7 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/CUnit.cpp File Reference

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

Include dependency graph for CUnit.cpp:

4.8 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/Genetic_↩ Algorithm.cpp File Reference

```
#include <stdio.h>
#include <cmath>
#include <array>
#include <omp.h>
#include <chrono>
#include <iostream>
#include <algorithm>
#include <fstream>
#include <filesystem>
#include "Genetic_Algorithm.h"
#include "CSimulator.h"
#include "CCircuit.h"
```

Include dependency graph for Genetic_Algorithm.cpp:

Functions

- void write_output (const string &filename, vector< int > generation_list, vector< double > best_values_list)

  Output the best value of each generation to a file.
- bool all_true (int vector_size, int ∗vector)
- int optimize (int vector_size, int ∗input_vector, double(&func)(int, int ∗), bool(&validity)(int, int ∗), struct Algorithm_Parameters parameters)

  Optimizes an input vector using a genetic algorithm based on a fitness function and validity checks.
- vector< vector< int > > generate_population (int vector_size, Circuit circuit, Algorithm_Parameters parameters)

  Generates the initial population of vectors for the genetic algorithm.
- int weighted_parent (double sum_of_fitness, vector< double > fitness)

  Selects a parent index for breeding based on a weighted probability distribution of fitness scores.
- pair< int, int > select_parents (vector< double > fitness, struct Algorithm_Parameters parameters)

  Selects pairs of parent vectors for breeding based on their fitness scores.
- vector< double > evaluate_parents (vector< vector< int >> parents, vector< int > &current_best, int &best_index, double(&func)(int, int ∗))

  Evaluates the fitness of each vector in the current population and identifies the best one.
- vector< int > mutate (bool stalled, vector< int > child, double mutation_rate, double stall_mutation_rate)

Mutates a vector based on a mutation rate, with increased mutation rates if improvements have stalled.
- int generate_children (bool stalled, vector< vector< int >> &new_population, vector< int > parent1, vector< int > parent2, Circuit circuit, Algorithm_Parameters parameters)

  Generates children from selected parent vectors, applying crossover and mutation operations.
- vector< vector< int > > crossover (vector< int > parent1, vector< int > parent2, string type, Algorithm_Parameters parameters)

  Performs genetic crossover between two parent vectors based on a specified type or randomly selects a crossover type.
- pair< vector< int >, vector< int > > uniform_cross (vector< int > parent1, vector< int > parent2)

  Performs a uniform crossover between two parent vectors, randomly exchanging genes.
- pair< vector< int >, vector< int > > one_point_cross (vector< int > parent1, vector< int > parent2)

  Executes a one-point crossover between two parent vectors.
- pair< vector< int >, vector< int > > two_point_cross (vector< int > parent1, vector< int > parent2)

  Conducts a two-point crossover between two parent vectors.
- pair< vector< int >, vector< int > > multipoint_cross (vector< int > parent1, vector< int > parent2)

  Performs a multipoint crossover on two parent vectors.
- vector< int > substitution (vector< int > child, double mutation_rate)
- vector< int > inversion (vector< int > child, double mutation_rate)
- vector< int > value_adjustment (vector< int > child, double mutation_rate)

Variables

- int max_value = 0

4.8.1 Function Documentation

4.8.1.1 all_true()

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

Definition at line 63 of file Genetic_Algorithm.cpp.

```
63                                          {
64   return true;
65 }
```

4.8.1.2 crossover()

```
vector<vector<int> > crossover (
            vector< int > parent1,
            vector< int > parent2,
            string type,
            Algorithm_Parameters parameters )
```

Performs genetic crossover between two parent vectors based on a specified type or randomly selects a crossover type.

This function chooses a crossover strategy (uniform, one-point, two-point, or multipoint) to combine genetic material from two parent vectors. The choice of crossover strategy can be specified or randomly determined, affecting how segments of the parents' vectors are mixed to produce new child vectors.

Parameters

parent1	The first parent vector from which genetic material is taken.
parent2	The second parent vector from which genetic material is taken.
type	A string specifying the type of crossover; if "random", a method is randomly selected.
parameters	Struct containing parameters for the genetic algorithm, influencing how crossover is performed.

Returns

vector<vector<int>> Returns two new child vectors resulting from the crossover operation.

Definition at line 440 of file Genetic_Algorithm.cpp.

```
{
  pair<vector<int>, vector<int» new_children;

  // if the type is random, select a random crossover function, otherwise select the type that was given
  if (type == "random")
  {
    int random = rand() % 4;
    if (random == 0)
      new_children = uniform_cross(parent1, parent2);
    else if (random == 1)
      new_children = one_point_cross(parent1, parent2);
    else if (random == 2)
      new_children = two_point_cross(parent1, parent2);
    else
      new_children = multipoint_cross(parent1, parent2);
  }
  else if (type == "one_point")
    new_children = one_point_cross(parent1, parent2);
  else if (type == "two_point")
    new_children = two_point_cross(parent1, parent2);
  else if (type == "multipoint")
    new_children = multipoint_cross(parent1, parent2);
  else if (type == "uniform")
    new_children = uniform_cross(parent1, parent2);

  // return child1 and child2 in a vector
  vector<vector<int» children = vector<vector<int»{new_children.first, new_children.second};

  return children;
}
```

4.8.1.3 evaluate_parents()

```
vector<double> evaluate_parents (
            vector< vector< int >> parents,
            vector< int > & current_best,
            int & best_index,
            double(&)(int, int *) func )
```

Evaluates the fitness of each vector in the current population and identifies the best one.

The function calculates the fitness for each vector in the population using a provided fitness function. It updates the best vector found in the current generation based on the highest fitness score.

Parameters

parents	A reference to the vector of parent vectors.
current_best	A reference to the vector which will store the best vector of the current generation.
best_index	A reference to an integer to store the index of the best vector.
func	A function pointer to the fitness function used to evaluate vectors.

Returns

vector<double> Returns a vector of fitness scores corresponding to each parent vector.

Definition at line 296 of file Genetic_Algorithm.cpp.

```
{
  double current_value = 0;
  vector<double> fitness(parents.size());
  // evaluate the fitness of all vectors in the current generation
#pragma omp parallel for private(current_value) schedule(dynamic)
  for (int i = 0; i < parents.size(); i++)
  {
    current_value = func(parents[i].size(), parents[i].data());
    fitness[i] = current_value;
  }
  // find the index and corresponding vector that have the highest fitness

  best_index = distance(fitness.begin(), max_element(fitness.begin(), fitness.end()));
  current_best = parents[best_index];

  return fitness;
}
```

4.8.1.4 generate_children()

```
int generate_children (
            bool stalled,
            vector< vector< int >> & new_population,
            vector< int > parent1,
            vector< int > parent2,
            Circuit circuit,
            Algorithm_Parameters parameters )
```

Generates children from selected parent vectors, applying crossover and mutation operations.

This function performs crossover and mutation to generate new vectors from the selected parents. It further ensures that only valid vectors are added to the new population.

Parameters

stalled	Boolean flag indicating whether the optimization has stalled.
new_population	Reference to the vector that will hold the new population.
parent1	First parent vector.
parent2	Second parent vector.
circuit	An instance of the Circuit class used for validating the vectors.
parameters	The parameters of the genetic algorithm, including crossover probability and mutation rates.

Returns

int The number of valid children added to the new population.

Definition at line 391 of file Genetic_Algorithm.cpp.

```
{

  int count = 0;
  // Decide if parents crossover and create new vectors before applying mutations
  // only add valid circuits to the new generation
  if (parameters.cross_probability < (rand() % 100) / 100)
  {
```

```
    vector<vector<int>> children = crossover(parent1, parent2, "random", parameters);
    for (auto child :  children)
    {
      child = mutate(stalled, child, parameters.mutation_rate, parameters.stall_mutation_rate);
      if (circuit.Check_Validity(child.size(), child.data()))
      {
        new_population.push_back(child);
        count++;
      }
    }
  }
  else
  {
    parent1 = mutate(stalled, parent1, parameters.mutation_rate, parameters.stall_mutation_rate);
    if (circuit.Check_Validity(parent1.size(), parent1.data()))
      {
        new_population.push_back(parent1);
        count++;
      }
    parent2 = mutate(stalled, parent2, parameters.mutation_rate, parameters.stall_mutation_rate);
    if (circuit.Check_Validity(parent2.size(), parent2.data()))
      {
        new_population.push_back(parent2);
        count++;
      }
  }
  return count;
}
```

4.8.1.5 generate_population()

```
vector<vector<int> > generate_population (
            int vector_size,
            Circuit circuit,
            Algorithm_Parameters parameters )
```

Generates the initial population of vectors for the genetic algorithm.

Initializes the population for the genetic algorithm. Each vector within the population is generated to meet specific validity criteria as determined by the circuit validity function. Random values are assigned to each element of the vector except the first, which is set to zero. There is an additional check to ensure that no unit is looping back on itself

Parameters

vector_size	The size of vectors in the population.
circuit	An instance of the Circuit class used for validating the vectors.
parameters	The parameters of the genetic algorithm including population size.

Returns

vector<vector<int>> Returns a population of valid vectors.

Definition at line 202 of file Genetic_Algorithm.cpp.

```
{
  vector<vector<int>> population(parameters.population_size);
  bool condition = false;
  int pop_size = 0;
  // add vectors to the population until the desired size is reached
  for (int i = 0; i < parameters.population_size; i++)
  {
    vector<int> individual;
    do
    {
```

```
      individual = vector<int>(vector_size);
      // start with each circuit feeding into 0
      individual[0] = 0;
      // give all other connections random values
      for (int j = 1; j < vector_size; j++)
      {
        int value = (rand() % max_value);
        if (value != (j-1)/3)
        {
          individual[j] = value;
        } else
        {
          individual[j] = (value +1) % max_value;
        }
      }
      // check that the generated circuit is valid
      condition = circuit.Check_Validity(individual.size(), individual.data());
    } while (!condition);

    population[i] = individual;
  }
  return population;
}
```

4.8.1.6 inversion()

```
vector<int> inversion (
              vector< int > child,
              double mutation_rate )
```

Definition at line 613 of file Genetic_Algorithm.cpp.

```
{
  if (mutation_rate > (rand() % 1000) / 1000)
  {
    // take a section of the child and invert it
    int mutation_index = rand() % (child.size()/2);
    int inversion_size =  rand() % (child.size()-mutation_index);
    reverse(child.begin() + mutation_index, child.begin() + mutation_index + inversion_size);
  }
  return child;
}
```

4.8.1.7 multipoint_cross()

```
pair<vector<int>, vector<int> > multipoint_cross (
              vector< int > parent1,
              vector< int > parent2 )
```

Performs a multipoint crossover on two parent vectors.

Several crossover points are randomly determined along the length of the vectors. The genes located at these points are then swapped between the two parents, resulting in a high level of gene shuffling. This approach can create significant genetic diversity within the offspring.

Parameters

parent1	First parent vector for crossover.
parent2	Second parent vector for crossover.

Returns

pair<vector<int>, vector<int>> Returns two child vectors with highly mixed genetic material from both parents.

Definition at line 571 of file Genetic_Algorithm.cpp.

```
{
  // crosses m points of the parent vectors
  vector<int> child1;
  vector<int> child2;
  vector<int> points;
  // get the crossover points, making sure no duplicates are selected
  for (int i = 0; i < parent1.size()/2; i++)
  {
    int point = 0;
    do
    {
      point = rand() % parent1.size();
    } while (find(points.begin(), points.end(), point) != points.end());
    points.push_back(point);
  }
  int temp;
  // only crossover the crossover_point
  for (int point :  points)
  {
    temp = parent1[point];
    parent1[point] = parent2[point];
    parent2[point] = temp;
  }
  return make_pair(parent1, parent2);
}
```

4.8.1.8 mutate()

```
vector<int> mutate (
            bool stalled,
            vector< int > child,
            double mutation_rate,
            double stall_mutation_rate )
```

Mutates a vector based on a mutation rate, with increased mutation rates if improvements have stalled.

This function applies mutations to a vector, potentially altering its elements to explore new genetic combinations. The type and intensity of mutations may increase if progress towards optimization has stalled. Note it is possible for more than one mutation to take place on the same vector.

Parameters

stalled	Boolean flag indicating whether the optimization has stalled.
child	The vector to mutate.
mutation_rate	The base rate of mutation.
stall_mutation_rate	The increased mutation rate to apply if stalled.

Returns

vector<int> Returns the mutated vector.

Definition at line 329 of file Genetic_Algorithm.cpp.

```
{
  // make mutations more aggressive if the improvement has stalled
  if (stalled){
```

```
    mutation_rate = stall_mutation_rate;
  }
  // substitution
  if (mutation_rate > (rand() % 1000) / 1000.0)
  {
    // substitute a random gene with a random value
    int mutation_index = rand() % child.size();
    child[mutation_index] = rand() % max_value;
  }
  // inversion
  if (mutation_rate > (rand() % 1000) / 1000.0)
  {
    // take a section of the child and invert it
    int mutation_index = rand() % (child.size()/2);
    int inversion_size =  rand() % (child.size()-mutation_index);
    reverse(child.begin() + mutation_index, child.begin() + mutation_index + inversion_size);
  }
  // value adjustment
  if (mutation_rate > (rand() % 1000) / 1000.0)
  {
    int mutation_index = rand() % child.size();
    int mutation_size = rand() % (child.size()-mutation_index);
    // add 1 to a section of the vector
    for (int i =0; i<mutation_size; i++)
    {
      int index = (mutation_index + i) % child.size();
      child[index] = (child[index] + 1) %max_value;
    }
  }
  // value adjustment down
  if (mutation_rate > (rand() % 1000) / 1000.0)
  {
    int mutation_index = rand() % child.size();
    int mutation_size = rand() % (child.size()-mutation_index);
    // subtract 1 from a section of the vector
    for (int i =0; i<mutation_size; i++)
    {
      int index = (mutation_index + i) % child.size();
      child[index] = (child[index] - 1) %max_value;
    }
  }

  return child;
}
```

4.8.1.9 one_point_cross()

```
pair<vector<int>, vector<int> > one_point_cross (
            vector< int > parent1,
            vector< int > parent2 )
```

Executes a one-point crossover between two parent vectors.

This function selects a random point in the vector, and all the genes (vector elements) beyond that point are swapped between the two parents. This results in two offspring, each sharing a block of genes with each parent.

Parameters

parent1	First parent vector for crossover.
parent2	Second parent vector for crossover.

Returns

pair<vector<int>, vector<int>> Returns two child vectors, each inheriting a contiguous block of genes from both parents.

Definition at line 516 of file Genetic_Algorithm.cpp.

```
{
  // crosses one point of the parent vectors
  // get the crossover point
  int crossover_point = rand() % parent1.size();
  // only crossover the crossover_point
  int temp = parent1[crossover_point];
  parent1[crossover_point] = parent2[crossover_point];
  parent2[crossover_point] = temp;

  return make_pair(parent1, parent2);
}
```

4.8.1.10 optimize()

```
int optimize (
            int vector_size,
            int * input_vector,
            double(&)(int, int *) func,
            bool(&)(int, int *) validity,
            struct Algorithm_Parameters parameters )
```

Optimizes an input vector using a genetic algorithm based on a fitness function and validity checks.

This function simulates a genetic algorithm process to find the optimal vectore to maximize a given function. It generates an initial population, evaluates fitness, selects parents, generates children, and mutates them across several generations until the maximum number of iterations is reached or improvement stalls. The final best solution found replaces the original input vector.

Parameters

vector_size	Size of the input vector.
input_vector	Pointer to the input vector that will be optimized.
func	A function that evaluates the fitness of the vector.
validity	A function that checks the validity of the vector.
parameters	Struct containing parameters for the genetic algorithm (e.g., population size, mutation rate).

Returns

int Returns 0 if the optimization completes before reaching the maximum iteration limit, 1 otherwise.

Definition at line 80 of file Genetic_Algorithm.cpp.

```
                                                                  {
  // Initializing variables
  vector<vector<int>> population;
  vector<vector<int>> new_population;
  vector<int> current_best;
  pair<int, int> parent_pair;
  vector<double> fitness;
  bool stalled = false;


  // lists used for output generation
  vector<int> generation_list;
  vector<double> best_values_list;


  Circuit circuit = Circuit((vector_size-1) / 3);
  // calculate the maximum value
  max_value = (vector_size -1)/3 + 2;
  cout << "max_value" << max_value<< endl;

  cout << "printing the vector" << endl;
```

```
  for (int i = 0; i < vector_size; ++i) {
      cout « input_vector[i] « " ";
  }
  cout « endl;
  // Generate the initial population
  auto start = std::chrono::high_resolution_clock::now();
  population = generate_population(vector_size, circuit, parameters);
  auto end = std::chrono::high_resolution_clock::now();

  double new_best_value = func(vector_size, input_vector);
  std::chrono::duration<double> elapsed_seconds = end-start;
  std::cout « "Time to generate population:  " « elapsed_seconds.count() « "s\n";

  int current_generation = 0;
  int stall_count = 0;
  start = std::chrono::high_resolution_clock::now();
  do
  {
    // Run the genetic algorithm process
    int best_index = 0;
    // Calculate the fitness of the parents to use in the selection of parents
    fitness = evaluate_parents(population, current_best, best_index, func);
    for (int child_count = 0; child_count < parameters.population_size;)
    {
      // select a pair of parent and create children based on them
      // only add valid children to the new population
      parent_pair = select_parents(fitness, parameters);
      int children = generate_children(stalled, new_population, population[parent_pair.first],
  population[parent_pair.second], circuit, parameters);
      child_count += children;
    }
    // Add the best vector from the previous generation to the new generation unaltered
    new_population.push_back(current_best);

    // check if the new best value is better than the previous best and swap if it is
    // if not add to the stall count
    double value = func(current_best.size(), current_best.data());
    cout « "\rGeneration " « current_generation « " best value " « value « flush;
    if (value > new_best_value)
    {
      new_best_value = value;
      stall_count = 0;
      stalled=false;
      //generation_list.push_back(current_generation);
      //best_values_list.push_back(new_best_value);
    }else
    {stall_count++;}
    // change new and old population and clear new population for the next cycle
    population.swap(new_population);
    new_population.clear();
    // check if the calculation has stalled
    if (stall_count >= parameters.stall_length){
      stalled = true;
    }
  }

  while (parameters.max_iterations > current_generation++ && stall_count < 100);
  cout « endl;
  end = std::chrono::high_resolution_clock::now();


  elapsed_seconds = end-start;
  std::cout « "Time to run simulation:  " « elapsed_seconds.count() « "s\n";

  cout « "input vector" « endl;
  for (int i = 0; i < current_best.size(); i++)
    cout « input_vector[i] « " ";
  cout « endl;

  // Update the vector with the best solution found
  for (int i = 0; i < current_best.size(); i++)
    input_vector[i] = current_best[i];
  cout « "Output vector" « endl;
  for (int i = 0; i < current_best.size(); i++)
    cout « input_vector[i] « ",";
  cout « endl;
  double final_value = func(vector_size, input_vector);
  cerr « "Final score " « final_value « endl;

  //write_output("output.csv");
  if (current_generation >= parameters.max_iterations)
    return 1;

  return 0;

}
```

4.8.1.11 select_parents()

```
pair<int, int> select_parents (
            vector< double > fitness,
            struct Algorithm_Parameters parameters )
```

Selects pairs of parent vectors for breeding based on their fitness scores.

This function uses a weighted probability method to select two parents from the current population. The fitness of each vector influences its likelihood of being chosen as a parent.

Parameters

fitness	A vector of doubles representing the fitness of each vector in the population.
parameters	Struct containing parameters for the genetic algorithm, including selection details.

Returns

pair<int, int> Returns a pair of indices representing the selected parents from the population.

Definition at line 272 of file Genetic_Algorithm.cpp.

```
{
  // choose two parents to potentially crossover
  int new_generation_size = 0;
  double sum_of_fitness = 0;
  for (int i = 0; i < fitness.size(); i++)
    sum_of_fitness += fitness[i];

  // Return the selected parents
  return make_pair(weighted_parent(sum_of_fitness, fitness), weighted_parent(sum_of_fitness, fitness));
}
```

4.8.1.12 substitution()

```
vector<int> substitution (
            vector< int > child,
            double mutation_rate )
```

Definition at line 602 of file Genetic_Algorithm.cpp.

```
{
  if (mutation_rate > (rand() % 1000) / 1000)
  {
    // substitute a random gene with a random value
    int mutation_index = rand() % child.size();
    child[mutation_index] = rand() % 10;
  }
  return child;
}
```

4.8.1.13 two_point_cross()

```
pair<vector<int>, vector<int> > two_point_cross (
            vector< int > parent1,
            vector< int > parent2 )
```

Conducts a two-point crossover between two parent vectors.

Two random points are chosen within the vectors, and the genetic material enclosed by these points is swapped between the two parents. This method allows for a more localized alteration of the genetic structure compared to one-point crossover.

Parameters

parent1	First parent vector.
parent2	Second parent vector.

Returns

pair<vector<int>, vector<int>> Returns two new child vectors, each containing segments of genes exchanged between the parents.

Definition at line 539 of file Genetic_Algorithm.cpp.

```
{
  // Similar to one points, but crossover point is two points wide
  vector<int> points;
  // get the crossover points
  points.push_back(rand() % parent1.size());
  // set the second point to be one higher than the crossover_point1 or the first element depending on the boundary
  points.push_back((points[0]+ 1) % parent1.size());
  int temp;
  // only crossover the crossover_point
  for (int point :  points)
  {
    temp = parent1[point];
    parent1[point] = parent2[point];
    parent2[point] = temp;
  }

  return make_pair(parent1, parent2);
}
```

4.8.1.14 uniform_cross()

```
pair<vector<int>, vector<int> > uniform_cross (
            vector< int > parent1,
            vector< int > parent2 )
```

Performs a uniform crossover between two parent vectors, randomly exchanging genes.

In uniform crossover, each gene (vector element) from the parents has a 50% chance of being swapped. This method provides a high degree of mixing and can generate highly varied offspring. Each gene is considered independently, making it possible to achieve diverse genetic combinations.

Parameters

parent1	First parent vector for crossover.
parent2	Second parent vector for crossover.

Returns

pair<vector<int>, vector<int>> Returns a pair of child vectors, each containing mixed genes from both parents.

Definition at line 482 of file Genetic_Algorithm.cpp.

```
{
  // uniform, take two from one, two from other until end
  // actually just takes one from parent1 and one from parent2 till the end
  // vice verca for the second child
  vector<int> child1(parent1.size());
  vector<int> child2(parent2.size());

  for (int i = 1; i <= parent1.size(); i++)
  {
    if (i % 2)
    {
      child1[i - 1] = parent1[i - 1];
      child2[i - 1] = parent2[i - 1];
    }
    else
    {
      child1[i - 1] = parent2[i - 1];
      child2[i - 1] = parent1[i - 1];
    }
  }

  return make_pair(child1, child2);
}
```

4.8.1.15 value_adjustment()

```
vector<int> value_adjustment (
            vector< int > child,
            double mutation_rate )
```

Definition at line 625 of file Genetic_Algorithm.cpp.

```
{
  if (mutation_rate > (rand() % 1000) / 1000)
  {
    int mutation_index = rand() % child.size();
    int mutation_size = rand() % (child.size()/2);
    for (int i =0; i<mutation_size; i++)
    {
      int index = (mutation_index + i) % child.size();
      child[index] = (child[index] + 1) %10;
    }
  }
  return child;
}
```

4.8.1.16 weighted_parent()

```
int weighted_parent (
            double sum_of_fitness,
            vector< double > fitness )
```

Selects a parent index for breeding based on a weighted probability distribution of fitness scores.

This function uses the fitness scores of each vector in the population to determine the likelihood of each vector being selected as a parent. The method involves computing a cumulative distribution of fitness scores, then selecting a parent based on random sampling from this distribution. The function ensures that vectors with higher fitness have a higher chance of being selected, promoting the propagation of favorable genes.

Parameters

sum_of_fitness	The sum of all fitness scores in the current population, used to normalize the probabilities.
fitness	A vector of doubles representing the fitness scores of each individual vector in the population.

Returns

int Returns the index of the selected parent vector based on the weighted fitness probability.

Definition at line 249 of file Genetic_Algorithm.cpp.

```
{
  // get an index using the weighted fitness of each vector
  int rnd = rand() % fitness.size();
  for (int i = 0; i < fitness.size(); i++)
    if(rnd < fitness[i])
    {
      return i;
    }

  return rnd;
}
```

4.8.1.17 write_output()

```
void write_output (
            const string & filename,
            vector< int > generation_list,
            vector< double > best_values_list )
```

Output the best value of each generation to a file.

This function writes the best value of each generation to a file. The file is saved in the data folder and is named output.csv.

Parameters

filename	The name of the file to write the output to.
generation_list	A list of the generation numbers.
best_values_list	A list of the best values of each generation.

Definition at line 33 of file Genetic_Algorithm.cpp.

```
{
  // Create the data folder if it doesn't exist

  filesystem::create_directory("data");

  // Open the file in the data folder
  ofstream file("data/" + filename);

  // Check if the file is open
  if (!file.is_open()) {
      cerr « "Failed to open file for writing." « std::endl;
      return;
  }

  // Write the header
  file « "Generation,Best Value\n";

  // Write the data
  for (size_t i = 0; i < generation_list.size(); ++i) {
```

```
        file « generation_list[i] « "," « best_values_list[i] « "\n";
    }

    // Close the file
    file.close();

    // Confirm that the file has been written
    cout « "Output written to data/" « filename « std::endl;
}
```

4.8.2 Variable Documentation

4.8.2.1 max_value

```
int max_value = 0
```

Definition at line 20 of file Genetic_Algorithm.cpp.

4.9 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/main.cpp File Reference

```
#include <iostream>
#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[ ])

4.9.1 Function Documentation

4.9.1.1 main()

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

Definition at line 9 of file main.cpp.

```
{

    // set things up
    // int vector[16] = {0, 1, 1, 2, 2, 3, 3, 0, 4, 1, 0, 2, 6, 5, 0, 6};
    int vector[31] = {0, 1, 3, 2, 4, 4, 3, 1, 3, 6, 1, 1, 0, 5, 1, 1,1,2,3,4,5,6,7,8,9,1,2,3,1,2,3};

    const Circuit circuit(8);
    // circuit.setup_connections(vector, 16);
    // for (int i = 0; i < 100; i++) {
    //     circuit.update_flows();
    // }
    // double Performance = circuit.concentrate_G*100 - circuit.concentrate_W*750;
    // double Recovery = circuit.concentrate_G / 10;
    // double Grade = circuit.concentrate_G / (circuit.concentrate_W+ circuit.concentrate_G);
    // std::cout « "concentrate_G: " « circuit.concentrate_G « std::endl;
    // std::cout « "concentrate_W: " « circuit.concentrate_W « std::endl;
    // std::cout « "tails_G: " « circuit.tails_G « std::endl;
    // std::cout « "tails_W: " « circuit.tails_W « std::endl;
    // std::cout « "Performance:  " « Performance « std::endl;
    // std::cout « "Recovery:  " « Recovery « std::endl;
    // std::cout « "Grade:  " « Grade « std::endl;
    // std::cout « "count:  " « circuit.count « std::endl;

    // run your code
    optimize(31, vector, Evaluate_Circuit);
    // or
    // optimize(11, vector, Evaluate_Circuit, Circuit::Check_Validity)
    // etc.

    // generate final output, save to file, etc.
    std::cout « Evaluate_Circuit(31, vector) « std::endl;

    for (int i = 0; i < sizeof(vector)/sizeof(int); i++) {
        std::cout « vector[i] « " ";
    }
    std::cout « std::endl;

    return 0;
}
```

Index