

## Gerardium Rush Bauxite Project

Generated by Doxygen 1.9.1



<b>1 Class Index</b>	<b>1</b>
1.1 Class List	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Class Documentation</b>	<b>5</b>
3.1 Algorithm_Parameters Struct Reference	5
3.1.1 Detailed Description	5
3.1.2 Member Data Documentation	5
3.1.2.1 cross_probability	5
3.1.2.2 max_iterations	6
3.1.2.3 mutation_rate	6
3.1.2.4 population_size	6
3.1.2.5 stall_length	6
3.1.2.6 stall_mutation_rate	6
3.2 Circuit Class Reference	6
3.2.1 Detailed Description	7
3.2.2 Constructor & Destructor Documentation	8
3.2.2.1 Circuit()	8
3.2.3 Member Function Documentation	8
3.2.3.1 check_no_all_same_destination()	8
3.2.3.2 check_no_self_recycle()	9
3.2.3.3 Check_Vailidity()	9
3.2.3.4 isReachable()	10
3.2.3.5 mark_units()	11
3.2.3.6 setup_connections()	12
3.2.3.7 update_flows()	13
3.2.4 Member Data Documentation	14
3.2.4.1 concentrate_G	14
3.2.4.2 concentrate_W	14
3.2.4.3 count	14
3.2.4.4 initial_index	15
3.2.4.5 next_feed_G	15
3.2.4.6 next_feed_W	15
3.2.4.7 num_units	15
3.2.4.8 tails_G	15
3.2.4.9 tails_W	15
3.2.4.10 units	16
3.3 Circuit_Parameters Struct Reference	16
3.3.1 Detailed Description	16
3.3.2 Member Data Documentation	16
3.3.2.1 max_iterations	16

3.3.2.2 tolerance	16
3.4 CUnit Class Reference	17
3.4.1 Detailed Description	17
3.4.2 Constructor & Destructor Documentation	18
3.4.2.1 CUnit() [1/2]	18
3.4.2.2 CUnit() [2/2]	18
3.4.3 Member Function Documentation	18
3.4.3.1 calculate_CG()	19
3.4.3.2 calculate_CW()	19
3.4.3.3 calculate_IG()	19
3.4.3.4 calculate_IW()	19
3.4.3.5 Calculate_ResidenceTime()	19
3.4.3.6 Calculate_RHG()	20
3.4.3.7 Calculate_RHW()	20
3.4.3.8 Calculate_RIG()	20
3.4.3.9 Calculate_RIW()	20
3.4.3.10 calculate_TG()	20
3.4.3.11 calculate_TW()	21
3.4.4 Member Data Documentation	21
3.4.4.1 conc_num	21
3.4.4.2 density	21
3.4.4.3 feed_rate_G	21
3.4.4.4 feed_rate_W	21
3.4.4.5 inter_num	22
3.4.4.6 k_cG	22
3.4.4.7 k_cW	22
3.4.4.8 k_iG	22
3.4.4.9 k_iW	22
3.4.4.10 mark	22
3.4.4.11 residence_time	23
3.4.4.12 solid_fraction	23
3.4.4.13 tails_num	23
3.4.4.14 volume	23
3.5 Unit_parameter Struct Reference	23
3.5.1 Detailed Description	24
3.5.2 Member Data Documentation	24
3.5.2.1 CG	24
3.5.2.2 CW	24
3.5.2.3 IG	24
3.5.2.4 IW	24
3.5.2.5 RHG	24
3.5.2.6 RHW	25

3.5.2.7 RIG	25
3.5.2.8 RIW	25
3.5.2.9 tau	25
3.5.2.10 TG	25
3.5.2.11 TW	25
<b>4 File Documentation</b>	<b>27</b>
4.1 /home/bz223/downloads/acs-gerardium-rush-bauxite/include/CCircuit.h File Reference	27
4.2 /home/bz223/downloads/acs-gerardium-rush-bauxite/include/CSimulator.h File Reference	27
4.2.1 Function Documentation	28
4.2.1.1 Evaluate_Circuit() [1/2]	28
4.2.1.2 Evaluate_Circuit() [2/2]	28
4.2.1.3 Evaluate_Circuit_Cell_Calculator()	29
4.2.1.4 Evaluate_Circuit_Grade()	29
4.2.1.5 Evaluate_Circuit_Recovery()	30
4.2.1.6 Save_ResultToTxt()	30
4.3 /home/bz223/downloads/acs-gerardium-rush-bauxite/include/CUnit.h File Reference	31
4.4 /home/bz223/downloads/acs-gerardium-rush-bauxite/include/Genetic_Algorithm.h File Reference	31
4.4.1 Macro Definition Documentation	32
4.4.1.1 DEFAULT_ALGORITHM_PARAMETERS	33
4.4.2 Function Documentation	33
4.4.2.1 all_true()	33
4.4.2.2 crossover()	33
4.4.2.3 evaluate_parents()	34
4.4.2.4 generate_children()	35
4.4.2.5 generate_population()	36
4.4.2.6 inversion()	37
4.4.2.7 multipoint_cross()	37
4.4.2.8 mutate()	38
4.4.2.9 one_point_cross()	39
4.4.2.10 optimize()	40
4.4.2.11 select_parents()	41
4.4.2.12 substitution()	42
4.4.2.13 two_point_cross()	42
4.4.2.14 uniform_cross()	43
4.4.2.15 value_adjustment()	44
4.5 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/CCircuit.cpp File Reference	44
4.6 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/CSimulator.cpp File Reference	44
4.6.1 Function Documentation	45
4.6.1.1 Evaluate_Circuit() [1/2]	45
4.6.1.2 Evaluate_Circuit() [2/2]	46
4.6.1.3 Evaluate_Circuit_Cell_Calculator()	46

4.6.1.4 Evaluate_Circuit_Grade() [1/2]	47
4.6.1.5 Evaluate_Circuit_Grade() [2/2]	47
4.6.1.6 Evaluate_Circuit_Recovery() [1/2]	48
4.6.1.7 Evaluate_Circuit_Recovery() [2/2]	49
4.6.1.8 Save_ResultToTxt()	50
4.6.2 Variable Documentation	50
4.6.2.1 default_circuit_parameters	50
4.6.2.2 dummy_answer_vector	51
4.7 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/CUnit.cpp File Reference	51
4.8 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/Genetic_Algorithm.cpp File Reference	51
4.8.1 Function Documentation	52
4.8.1.1 all_true()	52
4.8.1.2 crossover()	52
4.8.1.3 evaluate_parents()	53
4.8.1.4 generate_children()	54
4.8.1.5 generate_population()	55
4.8.1.6 inversion()	56
4.8.1.7 multipoint_cross()	56
4.8.1.8 mutate()	57
4.8.1.9 one_point_cross()	58
4.8.1.10 optimize()	59
4.8.1.11 select_parents()	61
4.8.1.12 substitution()	61
4.8.1.13 two_point_cross()	62
4.8.1.14 uniform_cross()	62
4.8.1.15 value_adjustment()	63
4.8.1.16 weighted_parent()	63
4.8.1.17 write_output()	64
4.8.2 Variable Documentation	65
4.8.2.1 max_value	65
4.9 /home/bz223/downloads/acs-gerardium-rush-bauxite/src/main.cpp File Reference	65
4.9.1 Function Documentation	65
4.9.1.1 main()	66
<b>Index</b>	<b>67</b>

# Chapter 1

## Class Index

### 1.1 Class List

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

<a href="#">Algorithm_Parameters</a>	5
<a href="#">Circuit</a>	6
<a href="#">Circuit_Parameters</a>	16
<a href="#">CUnit</a>	17
<a href="#">Unit_parameter</a>	23





## Chapter 2

# File Index

### 2.1 File List

Here is a list of all files with brief descriptions:

/home/bz223/downloads/acs-gerardium-rush-bauxite/include/CCircuit.h . . . . .	27
/home/bz223/downloads/acs-gerardium-rush-bauxite/include/CSimulator.h . . . . .	27
/home/bz223/downloads/acs-gerardium-rush-bauxite/include/CUnit.h . . . . .	31
/home/bz223/downloads/acs-gerardium-rush-bauxite/include/Genetic_Algorithm.h . . . . .	31
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/CCircuit.cpp . . . . .	44
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/CSimulator.cpp . . . . .	44
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/CUnit.cpp . . . . .	51
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/Genetic_Algorithm.cpp . . . . .	51
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/main.cpp . . . . .	65



## 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

<i>num_units</i>	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

<i>units</i>	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) {
```

```

230         if ((unit.conc_num == unit.inter_num && unit.inter_num == unit.tails_num) ||
231             (unit.conc_num == unit.tails_num)) {
232             return false; // Here all output streams point to the same unit
233         }
234     }
235     return true;
236 }

```

### 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

<i>units</i>	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.

```

167                                     {
168     for (size_t i = 0; i < units.size(); ++i) {
169         if (units[i].conc_num == i || units[i].inter_num == i || units[i].tails_num == i) {
170             return false;
171         }
172     }
173     return true;
174 }

```

### 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

<i>vector_size</i>	The size of the input vector.
<i>circuit_vector</i>	The input vector containing the connections between units.

Definition at line 83 of file CCircuit.cpp.

```

83                                     {
84     // Check if the vector size is valid
85     if (vector_size < 4 || (vector_size - 1) % 3 != 0) {
86         return false;
87     }
88
89     int num_units = (vector_size - 1) / 3; // (10-1)/3 in this example
90
91     // Set up all units' connections
92     for (int i = 0; i < (vector_size-1) / 3; ++i) {
93         // Add 1 to skip feed unit
94         this->units[i].conc_num = circuit_vector[1 + 3 * i];
95         this->units[i].inter_num = circuit_vector[1 + 3 * i + 1];
96         this->units[i].tails_num = circuit_vector[1 + 3 * i + 2];
97         this->units[i].mark = false;
98
99         // Check the range of conc_num, inter_num, and tails_num
100        // The range should be [0, num_units] for conc_num and inter_num
101        // The range should be [0, num_units + 1] for tails_num, except for the last unit
102        if (
103            this->units[i].conc_num < 0 || this->units[i].conc_num > num_units ||
104            this->units[i].inter_num < 0 || this->units[i].inter_num > num_units + 1 ||
105            this->units[i].tails_num < 0 || this->units[i].tails_num > num_units + 1 ||
106            this->units[i].tails_num == num_units) {
107            return false;
108        }
109    }
110
111    // Check for self-recycle and all output streams pointing to the same unit
112    if (!check_no_self_recycle(this->units) || !check_no_all_same_destination(this->units)) {
113        return false;
114    }
115
116    // Define outlets (Feed -> unit 0, unit 1, unit 2, ...)// e.g. unit 3 and unit 4 are outlets
117    // Check if all units can reach the outlet stream concentrate
118    if (!isReachable(num_units, num_units, this->units)) {
119        return false;
120    }
121
122    // Check if all units can reach the outlet stream tailings
123    if (!isReachable(num_units + 1, num_units, this->units)) {
124        return false;
125    }
126
127    // Initialize all units as unvisited
128    for (int i = 0; i < this->units.size(); i++) {
129        this->units[i].mark = false;
130    }
131
132    // Mark all units accessible from the starting unit (the feed unit)
133    if (circuit_vector[0] < 0 || circuit_vector[0] >= num_units) {
134        return false;
135    }
136    this->mark_units(circuit_vector[0]);
137
138    // Check which units have been marked as visited
139    bool all_accessible = true;
140    for (int i = 0; i < this->units.size(); i++) {
141        if (this->units[i].mark) {
142            //cout << "Unit " << i << " is accessible." << endl;
143        } else {
144            //cout << "Unit " << i << " is not accessible." << endl;
145            all_accessible = false;
146        }
147    }
148
149    // Check if all units are accessible
150    if (!all_accessible) {
151        return false;
152    }
153
154    return true; // If all checks pass, the circuit is valid
155 }

```

### 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

<i>start</i>	The index of the start unit.
<i>num_units</i>	The number of units in the circuit.
<i>units</i>	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

<i>unit_num</i>	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

<i>array</i>	The array representing the circuit connections.
<i>array_size</i>	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];
```

```

259     this->units[unitIndex].tails_num = array[i + 2];
260
261     // Check for special cases where the flow should go to concentrate or tailings
262     if(array[i] == this->num_units) {
263         //Direct the flow to concentrate
264         this->units[unitIndex].conc_num = -1; // -1 indicates that the flow should go to concentrate
265     }
266     if(array[i] == this->num_units + 1){
267         //Direct the flow to tailings
268         this->units[unitIndex].conc_num = -2; // -2 indicates that the flow should go to tailings
269     }
270     if(array[i + 1] == this->num_units){
271         //Direct the flow to concentrate
272         this->units[unitIndex].inter_num = -1; // -1 indicates that the flow should go to concentrate
273     }
274     if(array[i + 1] == this->num_units + 1){
275         //Direct the flow to tailings
276         this->units[unitIndex].inter_num = -2; // -2 indicates that the flow should go to tailings
277     }
278     if(array[i + 2] == this->num_units){
279         //Direct the flow to concentrate
280         this->units[unitIndex].tails_num = -1; // -1 indicates that the flow should go to concentrate
281     }
282     if(array[i + 2] == this->num_units + 1){
283         //Direct the flow to tailings
284         this->units[unitIndex].tails_num = -2; // -2 indicates that the flow should go to tailings
285     }
286 }
287 }
288 }

```

### 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;

```

```

331     concentrate_W += CW;
332 } else if (unit.conc_num == -2){ //Tailings
333     tails_G += CG;
334     tails_W += CW;
335 }
336
337 // Intermediate flow
338 if(unit.inter_num >=0 && unit.inter_num < num_units){
339     next_feed_G[unit.inter_num] += IG;
340     next_feed_W[unit.inter_num] += IW;
341 } else if (unit.inter_num == -1){ //Concentrate
342     concentrate_G += IG;
343     concentrate_W += IW;
344 } else if (unit.inter_num == -2){ //Tailings
345     tails_G += IG;
346     tails_W += IW;
347 }
348
349 // Tailings flow
350 if(unit.tails_num >=0 && unit.tails_num < num_units){
351     next_feed_G[unit.tails_num] += TG;
352     next_feed_W[unit.tails_num] += TW;
353 } else if (unit.tails_num == -1){ //Concentrate
354     concentrate_G += TG;
355     concentrate_W += TW;
356 } else if (unit.tails_num == -2){ //Tailings
357     tails_G += TG;
358     tails_W += TW;
359 }
360 }
361
362 //Update all feed_rate_G and feed_rate_W for the next iteration
363 for (int i = 0; i < this->num_units; ++i){
364     this->units[i].feed_rate_G = next_feed_G[i];
365     this->units[i].feed_rate_W = next_feed_W[i];
366 }
367 }

```

## 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

<i>conc</i>	
<i>inter</i>	
<i>tails</i>	

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.

```
67     {  
68         return feed_rate_G * Calculate_RHG();  
69     }
```

### 3.4.3.2 calculate\_CW()

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

Definition at line 71 of file CUnit.h.

```
71     {  
72         return feed_rate_W * Calculate_RHW();  
73     }
```

### 3.4.3.3 calculate\_IG()

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

Definition at line 75 of file CUnit.h.

```
75     {  
76         return feed_rate_G * Calculate_RIG();  
77     }
```

### 3.4.3.4 calculate\_IW()

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

Definition at line 79 of file CUnit.h.

```
79     {  
80         return feed_rate_W * Calculate_RIW();  
81     }
```

### 3.4.3.5 Calculate\_ResidenceTime()

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

Definition at line 58 of file CUnit.h.

```
58     {  
59         if(total_feed_rate > 0){  
60             residence_time = volume * solid_fraction / (total_feed_rate / density);  
61         } else {  
62             residence_time = 0;  
63         }  
64     };
```

#### 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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	The vector representing the circuit.
<i>parameters</i>	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

<i>parameters</i>	A reference to a <a href="#">Unit_parameter</a> 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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	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

<i>filename</i>	The name of the file to save the results to.
<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	The vector representing the circuit.
<i>performance</i>	The performance value of the circuit.
<i>recovery</i>	The recovery value of the circuit.
<i>grade</i>	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\\_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.*
- 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

<i>parent1</i>	The first parent vector from which genetic material is taken.
<i>parent2</i>	The second parent vector from which genetic material is taken.
<i>type</i>	A string specifying the type of crossover; if "random", a method is randomly selected.
<i>parameters</i>	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.

```

441 {
442     pair<vector<int>, vector<int>> new_children;
443
444     // if the type is random, select a random crossover function, otherwise select the type that was given
445     if (type == "random")
446     {
447         int random = rand() % 4;
448         if (random == 0)
449             new_children = uniform_cross(parent1, parent2);
450         else if (random == 1)
451             new_children = one_point_cross(parent1, parent2);
452         else if (random == 2)
453             new_children = two_point_cross(parent1, parent2);
454         else
455             new_children = multipoint_cross(parent1, parent2);
456     }
457     else if (type == "one_point")
458         new_children = one_point_cross(parent1, parent2);
459     else if (type == "two_point")
460         new_children = two_point_cross(parent1, parent2);
461     else if (type == "multipoint")
462         new_children = multipoint_cross(parent1, parent2);
463     else if (type == "uniform")
464         new_children = uniform_cross(parent1, parent2);
465
466     // return child1 and child2 in a vector
467     vector<vector<int>> children = vector<vector<int>>(new_children.first, new_children.second);
468
469     return children;
470 }

```

#### 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

<i>parents</i>	A reference to the vector of parent vectors.
<i>current_best</i>	A reference to the vector which will store the best vector of the current generation.
<i>best_index</i>	A reference to an integer to store the index of the best vector.
<i>func</i>	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.

```

297 {
298     double current_value = 0;
299     vector<double> fitness(parents.size());
300     // evaluate the fitness of all vectors in the current generation
301     #pragma omp parallel for private(current_value) schedule(dynamic)
302     for (int i = 0; i < parents.size(); i++)
303     {
304         current_value = func(parents[i].size(), parents[i].data());

```



```

305     fitness[i] = current_value;
306 }
307 // find the index and corresponding vector that have the highest fitness
308
309 best_index = distance(fitness.begin(), max_element(fitness.begin(), fitness.end()));
310 current_best = parents[best_index];
311
312 return fitness;
313 }

```

#### 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

<i>stalled</i>	Boolean flag indicating whether the optimization has stalled.
<i>new_population</i>	Reference to the vector that will hold the new population.
<i>parent1</i>	First parent vector.
<i>parent2</i>	Second parent vector.
<i>circuit</i>	An instance of the <a href="#">Circuit</a> class used for validating the vectors.
<i>parameters</i>	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.

```

392 {
393
394     int count = 0;
395     // Decide if parents crossover and create new vectors before applying mutations
396     // only add valid circuits to the new generation
397     if (parameters.cross_probability < (rand() % 100) / 100)
398     {
399         vector<vector<int>> children = crossover(parent1, parent2, "random", parameters);
400         for (auto child : children)
401         {
402             child = mutate(stalled, child, parameters.mutation_rate, parameters.stall_mutation_rate);
403             if (circuit.Check_Validity(child.size(), child.data()))
404             {
405                 new_population.push_back(child);
406                 count++;
407             }
408         }
409     }
410     else
411     {
412         parent1 = mutate(stalled, parent1, parameters.mutation_rate, parameters.stall_mutation_rate);
413         if (circuit.Check_Validity(parent1.size(), parent1.data()))

```

```

414     {
415         new_population.push_back(parent1);
416         count++;
417     }
418     parent2 = mutate(stalled, parent2, parameters.mutation_rate, parameters.stall_mutation_rate);
419     if (circuit.CheckValidity(parent2.size(), parent2.data()))
420     {
421         new_population.push_back(parent2);
422         count++;
423     }
424 }
425 return count;
426 }

```

#### 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

<i>vector_size</i>	The size of vectors in the population.
<i>circuit</i>	An instance of the <a href="#">Circuit</a> class used for validating the vectors.
<i>parameters</i>	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.

```

203 {
204     vector<vector<int> > population(parameters.population_size);
205     bool condition = false;
206     int pop_size = 0;
207     // add vectors to the population until the desired size is reached
208     for (int i = 0; i < parameters.population_size; i++)
209     {
210         vector<int> individual;
211         do
212         {
213             individual = vector<int>(vector_size);
214             // start with each circuit feeding into 0
215             individual[0] = 0;
216             // give all other connections random values
217             for (int j = 1; j < vector_size; j++)
218             {
219                 int value = (rand() % max_value);
220                 if (value != (j-1)/3)
221                 {
222                     individual[j] = value;
223                 } else
224                 {
225                     individual[j] = (value + 1) % max_value;
226                 }
227             }

```

```

228     // check that the generated circuit is valid
229     condition = circuit.Check_Validity(individual.size(), individual.data());
230     } while (!condition);
231
232     population[i] = individual;
233 }
234 return population;
235 }

```

#### 4.4.2.6 inversion()

```

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

```

Definition at line 613 of file Genetic\_Algorithm.cpp.

```

614 {
615     if (mutation_rate > (rand() % 1000) / 1000)
616     {
617         // take a section of the child and invert it
618         int mutation_index = rand() % (child.size()/2);
619         int inversion_size = rand() % (child.size()-mutation_index);
620         reverse(child.begin() + mutation_index, child.begin() + mutation_index + inversion_size);
621     }
622     return child;
623 }

```

#### 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

<i>parent1</i>	First parent vector for crossover.
<i>parent2</i>	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.

```

572 {
573     // crosses m points of the parent vectors
574     vector<int> child1;
575     vector<int> child2;
576     vector<int> points;

```

```

577 // get the crossover points, making sure no duplicates are selected
578 for (int i = 0; i < parent1.size()/2; i++)
579 {
580     int point = 0;
581     do
582     {
583         point = rand() % parent1.size();
584     } while (find(points.begin(), points.end(), point) != points.end());
585     points.push_back(point);
586 }
587 int temp;
588 // only crossover the crossover_point
589 for (int point : points)
590 {
591     temp = parent1[point];
592     parent1[point] = parent2[point];
593     parent2[point] = temp;
594 }
595 return make_pair(parent1, parent2);
596 }

```

#### 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

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

##### Returns

`vector<int>` Returns the mutated vector.

Definition at line 329 of file Genetic\_Algorithm.cpp.

```

330 {
331     // make mutations more aggressive if the improvement has stalled
332     if (stalled){
333         mutation_rate = stall_mutation_rate;
334     }
335     // substitution
336     if (mutation_rate > (rand() % 1000) / 1000.0)
337     {
338         // substitute a random gene with a random value
339         int mutation_index = rand() % child.size();
340         child[mutation_index] = rand() % max_value;
341     }
342     // inversion
343     if (mutation_rate > (rand() % 1000) / 1000.0)
344     {
345         // take a section of the child and invert it
346         int mutation_index = rand() % (child.size()/2);

```

```

347     int inversion_size = rand() % (child.size()-mutation_index);
348     reverse(child.begin() + mutation_index, child.begin() + mutation_index + inversion_size);
349 }
350 // value adjustment
351 if (mutation_rate > (rand() % 1000) / 1000.0)
352 {
353     int mutation_index = rand() % child.size();
354     int mutation_size = rand() % (child.size()-mutation_index);
355     // add 1 to a section of the vector
356     for (int i =0; i<mutation_size; i++)
357     {
358         int index = (mutation_index + i) % child.size();
359         child[index] = (child[index] + 1) %max_value;
360     }
361 }
362 // value adjustment down
363 if (mutation_rate > (rand() % 1000) / 1000.0)
364 {
365     int mutation_index = rand() % child.size();
366     int mutation_size = rand() % (child.size()-mutation_index);
367     // subtract 1 from a section of the vector
368     for (int i =0; i<mutation_size; i++)
369     {
370         int index = (mutation_index + i) % child.size();
371         child[index] = (child[index] - 1) %max_value;
372     }
373 }
374
375 return child;
376 }

```

#### 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

<i>parent1</i>	First parent vector for crossover.
<i>parent2</i>	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.

```

517 {
518     // crosses one point of the parent vectors
519     // get the crossover point
520     int crossover_point = rand() % parent1.size();
521     // only crossover the crossover_point
522     int temp = parent1[crossover_point];
523     parent1[crossover_point] = parent2[crossover_point];
524     parent2[crossover_point] = temp;
525
526     return make_pair(parent1, parent2);
527 }

```

#### 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

<i>vector_size</i>	Size of the input vector.
<i>input_vector</i>	Pointer to the input vector that will be optimized.
<i>func</i>	A function that evaluates the fitness of the vector.
<i>validity</i>	A function that checks the validity of the vector.
<i>parameters</i>	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.

```
82
83 // Initializing variables
84 vector<vector<int>> population;
85 vector<vector<int>> new_population;
86 vector<int> current_best;
87 pair<int, int> parent_pair;
88 vector<double> fitness;
89 bool stalled = false;
90
91
92 // lists used for output generation
93 vector<int> generation_list;
94 vector<double> best_values_list;
95
96
97 Circuit circuit = Circuit((vector_size-1) / 3);
98 // calculate the maximum value
99 max_value = (vector_size -1)/3 + 2;
100 cout << "max_value" << max_value<< endl;
101
102 cout << "printing the vector" << endl;
103 for (int i = 0; i < vector_size; ++i) {
104     cout << input_vector[i] << " ";
105 }
106 cout << endl;
107 // Generate the initial population
108 auto start = std::chrono::high_resolution_clock::now();
109 population = generate_population(vector_size, circuit, parameters);
110 auto end = std::chrono::high_resolution_clock::now();
111
112 double new_best_value = func(vector_size, input_vector);
113 std::chrono::duration<double> elapsed_seconds = end-start;
114 std::cout << "Time to generate population: " << elapsed_seconds.count() << "s\n";
115
116 int current_generation = 0;
117 int stall_count = 0;
118 start = std::chrono::high_resolution_clock::now();
119 do
120 {
```

```

121 // Run the genetic algorithm process
122 int best_index = 0;
123 // Calculate the fitness of the parents to use in the selection of parents
124 fitness = evaluate_parents(population, current_best, best_index, func);
125 for (int child_count = 0; child_count < parameters.population_size;)
126 {
127     // select a pair of parent and create children based on them
128     // only add valid children to the new population
129     parent_pair = select_parents(fitness, parameters);
130     int children = generate_children(stalled, new_population, population[parent_pair.first],
    population[parent_pair.second], circuit, parameters);
131     child_count += children;
132 }
133 // Add the best vector from the previous generation to the new generation unaltered
134 new_population.push_back(current_best);
135
136 // check if the new best value is better than the previous best and swap if it is
137 // if not add to the stall count
138 double value = func(current_best.size(), current_best.data());
139 cout << "\rGeneration " << current_generation << " best value " << value << flush;
140 if (value > new_best_value)
141 {
142     new_best_value = value;
143     stall_count = 0;
144     stalled=false;
145     //generation_list.push_back(current_generation);
146     //best_values_list.push_back(new_best_value);
147 }else
148 {stall_count++;}
149 // change new and old population and clear new population for the next cycle
150 population.swap(new_population);
151 new_population.clear();
152 // check if the calculation has stalled
153 if (stall_count >= parameters.stall_length){
154     stalled = true;
155 }
156 }
157
158 while (parameters.max_iterations > current_generation++ && stall_count < 100);
159 cout << endl;
160 end = std::chrono::high_resolution_clock::now();
161
162 elapsed_seconds = end-start;
163 std::cout << "Time to run simulation: " << elapsed_seconds.count() << "s\n";
164
165 cout << "input vector" << endl;
166 for (int i = 0; i < current_best.size(); i++)
167     cout << input_vector[i] << " ";
168 cout << endl;
169
170 // Update the vector with the best solution found
171 for (int i = 0; i < current_best.size(); i++)
172     input_vector[i] = current_best[i];
173 cout << "Output vector" << endl;
174 for (int i = 0; i < current_best.size(); i++)
175     cout << input_vector[i] << ",";
176 cout << endl;
177 double final_value = func(vector_size, input_vector);
178 cerr << "Final score " << final_value << endl;
179
180 //write_output("output.csv");
181 if (current_generation >= parameters.max_iterations)
182     return 1;
183
184 return 0;
185
186
187 }

```

#### 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**

<i>fitness</i>	A vector of doubles representing the fitness of each vector in the population.
<i>parameters</i>	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.

```

273 {
274     // choose two parents to potentially crossover
275     int new_generation_size = 0;
276     double sum_of_fitness = 0;
277     for (int i = 0; i < fitness.size(); i++)
278         sum_of_fitness += fitness[i];
279
280     // Return the selected parents
281     return make_pair(weighted_parent(sum_of_fitness, fitness), weighted_parent(sum_of_fitness, fitness));
282 }
```

**4.4.2.12 substitution()**

```

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

Definition at line 602 of file Genetic\_Algorithm.cpp.

```

603 {
604     if (mutation_rate > (rand() % 1000) / 1000)
605     {
606         // substitute a random gene with a random value
607         int mutation_index = rand() % child.size();
608         child[mutation_index] = rand() % 10;
609     }
610     return child;
611 }
```

**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**

<i>parent1</i>	First parent vector.
<i>parent2</i>	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.

```

540 {
541     // Similar to one points, but crossover point is two points wide
542     vector<int> points;
543     // get the crossover points
544     points.push_back(rand() % parent1.size());
545     // set the second point to be one higher than the crossover_point1 or the first element depending on
    the boundary
546     points.push_back((points[0]+ 1) % parent1.size());
547     int temp;
548     // only crossover the crossover_point
549     for (int point : points)
550     {
551         temp = parent1[point];
552         parent1[point] = parent2[point];
553         parent2[point] = temp;
554     }
555
556     return make_pair(parent1, parent2);
557 }
```

**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**

<i>parent1</i>	First parent vector for crossover.
<i>parent2</i>	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.

```

483 {
484     // uniform, take two from one, two from other until end
485     // actually just takes one from parent1 and one from parent2 till the end
486     // vice versa for the second child
487     vector<int> child1(parent1.size());
488     vector<int> child2(parent2.size());
489
490     for (int i = 1; i <= parent1.size(); i++)
491     {
492         if (i % 2)
493         {
494             child1[i - 1] = parent1[i - 1];
495             child2[i - 1] = parent2[i - 1];
496         }
```

```

497     else
498     {
499         child1[i - 1] = parent2[i - 1];
500         child2[i - 1] = parent1[i - 1];
501     }
502 }
503
504 return make_pair(child1, child2);
505 }

```

#### 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.

```

626 {
627     if (mutation_rate > (rand() % 1000) / 1000)
628     {
629         int mutation_index = rand() % child.size();
630         int mutation_size = rand() % (child.size()/2);
631         for (int i =0; i<mutation_size; i++)
632         {
633             int index = (mutation_index + i) % child.size();
634             child[index] = (child[index] + 1) %10;
635         }
636     }
637     return child;
638 }

```

## 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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	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 };

```

#### 4.6.1.2 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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	The vector representing the circuit.
<i>parameters</i>	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

<i>parameters</i>	A reference to a <a href="#">Unit_parameter</a> 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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	The vector representing the circuit.
<i>parameters</i>	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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	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

<i>vector_size</i>	The size of the circuit vector.
<i>circuit_vector</i>	The vector representing the circuit.
<i>parameters</i>	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

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

<i>parent1</i>	The first parent vector from which genetic material is taken.
<i>parent2</i>	The second parent vector from which genetic material is taken.
<i>type</i>	A string specifying the type of crossover; if "random", a method is randomly selected.
<i>parameters</i>	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.

```

441 {
442     pair<vector<int>, vector<int>> new_children;
443
444     // if the type is random, select a random crossover function, otherwise select the type that was given
445     if (type == "random")
446     {
447         int random = rand() % 4;
448         if (random == 0)
449             new_children = uniform_cross(parent1, parent2);
450         else if (random == 1)
451             new_children = one_point_cross(parent1, parent2);
452         else if (random == 2)
453             new_children = two_point_cross(parent1, parent2);
454         else
455             new_children = multipoint_cross(parent1, parent2);
456     }
457     else if (type == "one_point")
458         new_children = one_point_cross(parent1, parent2);
459     else if (type == "two_point")
460         new_children = two_point_cross(parent1, parent2);
461     else if (type == "multipoint")
462         new_children = multipoint_cross(parent1, parent2);
463     else if (type == "uniform")
464         new_children = uniform_cross(parent1, parent2);
465
466     // return child1 and child2 in a vector
467     vector<vector<int>> children = vector<vector<int>>(new_children.first, new_children.second);
468
469     return children;
470 }
```

**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**

<i>parents</i>	A reference to the vector of parent vectors.
<i>current_best</i>	A reference to the vector which will store the best vector of the current generation.
<i>best_index</i>	A reference to an integer to store the index of the best vector.
<i>func</i>	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`.

```

297 {
298     double current_value = 0;
299     vector<double> fitness(parents.size());
300     // evaluate the fitness of all vectors in the current generation
301     #pragma omp parallel for private(current_value) schedule(dynamic)
302     for (int i = 0; i < parents.size(); i++)
303     {
304         current_value = func(parents[i].size(), parents[i].data());
305         fitness[i] = current_value;
306     }
307     // find the index and corresponding vector that have the highest fitness
308
309     best_index = distance(fitness.begin(), max_element(fitness.begin(), fitness.end()));
310     current_best = parents[best_index];
311
312     return fitness;
313 }
```

**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**

<i>stalled</i>	Boolean flag indicating whether the optimization has stalled.
<i>new_population</i>	Reference to the vector that will hold the new population.
<i>parent1</i>	First parent vector.
<i>parent2</i>	Second parent vector.
<i>circuit</i>	An instance of the <a href="#">Circuit</a> class used for validating the vectors.
<i>parameters</i>	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`.

```

392 {
393
394     int count = 0;
395     // Decide if parents crossover and create new vectors before applying mutations
396     // only add valid circuits to the new generation
397     if (parameters.cross_probability < (rand() % 100) / 100)
398     {
```

```

399     vector<vector<int> > children = crossover(parent1, parent2, "random", parameters);
400     for (auto child : children)
401     {
402         child = mutate(stalled, child, parameters.mutation_rate, parameters.stall_mutation_rate);
403         if (circuit.Check_Validity(child.size(), child.data()))
404         {
405             new_population.push_back(child);
406             count++;
407         }
408     }
409 }
410 else
411 {
412     parent1 = mutate(stalled, parent1, parameters.mutation_rate, parameters.stall_mutation_rate);
413     if (circuit.Check_Validity(parent1.size(), parent1.data()))
414     {
415         new_population.push_back(parent1);
416         count++;
417     }
418     parent2 = mutate(stalled, parent2, parameters.mutation_rate, parameters.stall_mutation_rate);
419     if (circuit.Check_Validity(parent2.size(), parent2.data()))
420     {
421         new_population.push_back(parent2);
422         count++;
423     }
424 }
425 return count;
426 }

```

#### 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

<i>vector_size</i>	The size of vectors in the population.
<i>circuit</i>	An instance of the <a href="#">Circuit</a> class used for validating the vectors.
<i>parameters</i>	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.

```

203 {
204     vector<vector<int> > population(parameters.population_size);
205     bool condition = false;
206     int pop_size = 0;
207     // add vectors to the population until the desired size is reached
208     for (int i = 0; i < parameters.population_size; i++)
209     {
210         vector<int> individual;
211         do
212         {

```

```

213     individual = vector<int>(vector_size);
214     // start with each circuit feeding into 0
215     individual[0] = 0;
216     // give all other connections random values
217     for (int j = 1; j < vector_size; j++)
218     {
219         int value = (rand() % max_value);
220         if (value != (j-1)/3)
221         {
222             individual[j] = value;
223         } else
224         {
225             individual[j] = (value +1) % max_value;
226         }
227     }
228     // check that the generated circuit is valid
229     condition = circuit.Check_Validity(individual.size(), individual.data());
230     } while (!condition);
231
232     population[i] = individual;
233 }
234 return population;
235 }

```

#### 4.8.1.6 inversion()

```

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

```

Definition at line 613 of file Genetic\_Algorithm.cpp.

```

614 {
615     if (mutation_rate > (rand() % 1000) / 1000)
616     {
617         // take a section of the child and invert it
618         int mutation_index = rand() % (child.size()/2);
619         int inversion_size = rand() % (child.size()-mutation_index);
620         reverse(child.begin() + mutation_index, child.begin() + mutation_index + inversion_size);
621     }
622     return child;
623 }

```

#### 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

<i>parent1</i>	First parent vector for crossover.
<i>parent2</i>	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.

```

572 {
573     // crosses m points of the parent vectors
574     vector<int> child1;
575     vector<int> child2;
576     vector<int> points;
577     // get the crossover points, making sure no duplicates are selected
578     for (int i = 0; i < parent1.size()/2; i++)
579     {
580         int point = 0;
581         do
582         {
583             point = rand() % parent1.size();
584         } while (find(points.begin(), points.end(), point) != points.end());
585         points.push_back(point);
586     }
587     int temp;
588     // only crossover the crossover_point
589     for (int point : points)
590     {
591         temp = parent1[point];
592         parent1[point] = parent2[point];
593         parent2[point] = temp;
594     }
595     return make_pair(parent1, parent2);
596 }
```

### 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

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

## Returns

`vector<int>` Returns the mutated vector.

Definition at line 329 of file Genetic\_Algorithm.cpp.

```

330 {
331     // make mutations more aggressive if the improvement has stalled
332     if (stalled){
```

```

333     mutation_rate = stall_mutation_rate;
334 }
335 // substitution
336 if (mutation_rate > (rand() % 1000) / 1000.0)
337 {
338     // substitute a random gene with a random value
339     int mutation_index = rand() % child.size();
340     child[mutation_index] = rand() % max_value;
341 }
342 // inversion
343 if (mutation_rate > (rand() % 1000) / 1000.0)
344 {
345     // take a section of the child and invert it
346     int mutation_index = rand() % (child.size()/2);
347     int inversion_size = rand() % (child.size()-mutation_index);
348     reverse(child.begin() + mutation_index, child.begin() + mutation_index + inversion_size);
349 }
350 // value adjustment
351 if (mutation_rate > (rand() % 1000) / 1000.0)
352 {
353     int mutation_index = rand() % child.size();
354     int mutation_size = rand() % (child.size()-mutation_index);
355     // add 1 to a section of the vector
356     for (int i=0; i<mutation_size; i++)
357     {
358         int index = (mutation_index + i) % child.size();
359         child[index] = (child[index] + 1) % max_value;
360     }
361 }
362 // value adjustment down
363 if (mutation_rate > (rand() % 1000) / 1000.0)
364 {
365     int mutation_index = rand() % child.size();
366     int mutation_size = rand() % (child.size()-mutation_index);
367     // subtract 1 from a section of the vector
368     for (int i=0; i<mutation_size; i++)
369     {
370         int index = (mutation_index + i) % child.size();
371         child[index] = (child[index] - 1) % max_value;
372     }
373 }
374
375 return child;
376 }

```

#### 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

<i>parent1</i>	First parent vector for crossover.
<i>parent2</i>	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.



```

517 {
518     // crosses one point of the parent vectors
519     // get the crossover point
520     int crossover_point = rand() % parent1.size();
521     // only crossover the crossover_point
522     int temp = parent1[crossover_point];
523     parent1[crossover_point] = parent2[crossover_point];
524     parent2[crossover_point] = temp;
525
526     return make_pair(parent1, parent2);
527 }

```

#### 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

<i>vector_size</i>	Size of the input vector.
<i>input_vector</i>	Pointer to the input vector that will be optimized.
<i>func</i>	A function that evaluates the fitness of the vector.
<i>validity</i>	A function that checks the validity of the vector.
<i>parameters</i>	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.

```

82
83     // Initializing variables
84     vector<vector<int>> population;
85     vector<vector<int>> new_population;
86     vector<int> current_best;
87     pair<int, int> parent_pair;
88     vector<double> fitness;
89     bool stalled = false;
90
91
92     // lists used for output generation
93     vector<int> generation_list;
94     vector<double> best_values_list;
95
96
97     Circuit circuit = Circuit((vector_size-1) / 3);
98     // calculate the maximum value
99     max_value = (vector_size -1)/3 + 2;
100     cout << "max_value" << max_value<< endl;
101
102     cout << "printing the vector" << endl;

```

```

103     for (int i = 0; i < vector_size; ++i) {
104         cout << input_vector[i] << " ";
105     }
106     cout << endl;
107     // Generate the initial population
108     auto start = std::chrono::high_resolution_clock::now();
109     population = generate_population(vector_size, circuit, parameters);
110     auto end = std::chrono::high_resolution_clock::now();
111
112     double new_best_value = func(vector_size, input_vector);
113     std::chrono::duration<double> elapsed_seconds = end-start;
114     std::cout << "Time to generate population: " << elapsed_seconds.count() << "s\n";
115
116     int current_generation = 0;
117     int stall_count = 0;
118     start = std::chrono::high_resolution_clock::now();
119     do
120     {
121         // Run the genetic algorithm process
122         int best_index = 0;
123         // Calculate the fitness of the parents to use in the selection of parents
124         fitness = evaluate_parents(population, current_best, best_index, func);
125         for (int child_count = 0; child_count < parameters.population_size; )
126         {
127             // select a pair of parent and create children based on them
128             // only add valid children to the new population
129             parent_pair = select_parents(fitness, parameters);
130             int children = generate_children(stalled, new_population, population[parent_pair.first],
131             population[parent_pair.second], circuit, parameters);
132             child_count += children;
133         }
134         // Add the best vector from the previous generation to the new generation unaltered
135         new_population.push_back(current_best);
136
137         // check if the new best value is better than the previous best and swap if it is
138         // if not add to the stall count
139         double value = func(current_best.size(), current_best.data());
140         cout << "\rGeneration " << current_generation << " best value " << value << flush;
141         if (value > new_best_value)
142         {
143             new_best_value = value;
144             stall_count = 0;
145             stalled=false;
146             //generation_list.push_back(current_generation);
147             //best_values_list.push_back(new_best_value);
148         }else
149         {stall_count++;}
150         // change new and old population and clear new population for the next cycle
151         population.swap(new_population);
152         new_population.clear();
153         // check if the calculation has stalled
154         if (stall_count >= parameters.stall_length){
155             stalled = true;
156         }
157     }
158     while (parameters.max_iterations > current_generation++ && stall_count < 100);
159     cout << endl;
160     end = std::chrono::high_resolution_clock::now();
161
162     elapsed_seconds = end-start;
163     std::cout << "Time to run simulation: " << elapsed_seconds.count() << "s\n";
164
165     cout << "input vector" << endl;
166     for (int i = 0; i < current_best.size(); i++)
167         cout << input_vector[i] << " ";
168     cout << endl;
169
170     // Update the vector with the best solution found
171     for (int i = 0; i < current_best.size(); i++)
172         input_vector[i] = current_best[i];
173     cout << "Output vector" << endl;
174     for (int i = 0; i < current_best.size(); i++)
175         cout << input_vector[i] << ",";
176     cout << endl;
177     double final_value = func(vector_size, input_vector);
178     cerr << "Final score " << final_value << endl;
179
180     //write_output("output.csv");
181     if (current_generation >= parameters.max_iterations)
182         return 1;
183
184     return 0;
185 }
186
187 }

```

#### 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

<i>fitness</i>	A vector of doubles representing the fitness of each vector in the population.
<i>parameters</i>	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.

```
273 {
274     // choose two parents to potentially crossover
275     int new_generation_size = 0;
276     double sum_of_fitness = 0;
277     for (int i = 0; i < fitness.size(); i++)
278         sum_of_fitness += fitness[i];
279
280     // Return the selected parents
281     return make_pair(weighted_parent(sum_of_fitness, fitness), weighted_parent(sum_of_fitness, fitness));
282 }
```

#### 4.8.1.12 substitution()

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

Definition at line 602 of file Genetic\_Algorithm.cpp.

```
603 {
604     if (mutation_rate > (rand() % 1000) / 1000)
605     {
606         // substitute a random gene with a random value
607         int mutation_index = rand() % child.size();
608         child[mutation_index] = rand() % 10;
609     }
610     return child;
611 }
```

#### 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

<i>parent1</i>	First parent vector.
<i>parent2</i>	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.

```
540 {
541     // Similar to one points, but crossover point is two points wide
542     vector<int> points;
543     // get the crossover points
544     points.push_back(rand() % parent1.size());
545     // set the second point to be one higher than the crossover_point1 or the first element depending on
    the boundary
546     points.push_back((points[0]+ 1) % parent1.size());
547     int temp;
548     // only crossover the crossover_point
549     for (int point : points)
550     {
551         temp = parent1[point];
552         parent1[point] = parent2[point];
553         parent2[point] = temp;
554     }
555
556     return make_pair(parent1, parent2);
557 }
```

#### 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

<i>parent1</i>	First parent vector for crossover.
<i>parent2</i>	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.

```

483 {
484     // uniform, take two from one, two from other until end
485     // actually just takes one from parent1 and one from parent2 till the end
486     // vice versa for the second child
487     vector<int> child1(parent1.size());
488     vector<int> child2(parent2.size());
489
490     for (int i = 1; i <= parent1.size(); i++)
491     {
492         if (i % 2)
493         {
494             child1[i - 1] = parent1[i - 1];
495             child2[i - 1] = parent2[i - 1];
496         }
497         else
498         {
499             child1[i - 1] = parent2[i - 1];
500             child2[i - 1] = parent1[i - 1];
501         }
502     }
503
504     return make_pair(child1, child2);
505 }
```

### 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.

```

626 {
627     if (mutation_rate > (rand() % 1000) / 1000)
628     {
629         int mutation_index = rand() % child.size();
630         int mutation_size = rand() % (child.size()/2);
631         for (int i =0; i<mutation_size; i++)
632         {
633             int index = (mutation_index + i) % child.size();
634             child[index] = (child[index] + 1) %10;
635         }
636     }
637     return child;
638 }
```

### 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**

<i>sum_of_fitness</i>	The sum of all fitness scores in the current population, used to normalize the probabilities.
<i>fitness</i>	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.

```

250 {
251     // get an index using the weighted fitness of each vector
252     int rnd = rand() % fitness.size();
253     for (int i = 0; i < fitness.size(); i++)
254         if(rnd < fitness[i])
255         {
256             return i;
257         }
258     return rnd;
259 }
260 }
```

**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**

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

Definition at line 33 of file Genetic\_Algorithm.cpp.

```

34 {
35     // Create the data folder if it doesn't exist
36
37     filesystem::create_directory("data");
38
39     // Open the file in the data folder
40     ofstream file("data/" + filename);
41
42     // Check if the file is open
43     if (!file.is_open()) {
44         cerr << "Failed to open file for writing." << std::endl;
45         return;
46     }
47
48     // Write the header
49     file << "Generation,Best Value\n";
50
51     // Write the data
52     for (size_t i = 0; i < generation_list.size(); ++i) {
```

```
53     file << generation_list[i] << ", " << best_values_list[i] << "\n";
54 }
55
56 // Close the file
57 file.close();
58
59 // Confirm that the file has been written
60 cout << "Output written to data/" << filename << std::endl;
61 }
```

## 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.

```
10 {
11
12     // set things up
13     // int vector[16] = {0, 1, 1, 2, 2, 3, 3, 0, 4, 1, 0, 2, 6, 5, 0, 6};
14     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};
15
16     const Circuit circuit(8);
17     // circuit.setup_connections(vector, 16);
18     // for (int i = 0; i < 100; i++) {
19     //     circuit.update_flows();
20     // }
21     // double Performance = circuit.concentrate_G*100 - circuit.concentrate_W*750;
22     // double Recovery = circuit.concentrate_G / 10;
23     // double Grade = circuit.concentrate_G / (circuit.concentrate_W+ circuit.concentrate_G);
24     // std::cout << "concentrate_G: " << circuit.concentrate_G << std::endl;
25     // std::cout << "concentrate_W: " << circuit.concentrate_W << std::endl;
26     // std::cout << "tails_G: " << circuit.tails_G << std::endl;
27     // std::cout << "tails_W: " << circuit.tails_W << std::endl;
28     // std::cout << "Performance: " << Performance << std::endl;
29     // std::cout << "Recovery: " << Recovery << std::endl;
30     // std::cout << "Grade: " << Grade << std::endl;
31     // std::cout << "count: " << circuit.count << std::endl;
32
33     // run your code
34     optimize(31, vector, Evaluate_Circuit);
35     // or
36     // optimize(11, vector, Evaluate_Circuit, Circuit::Check_Validity)
37     // etc.
38
39     // generate final output, save to file, etc.
40     std::cout << Evaluate_Circuit(31, vector) << std::endl;
41
42     for (int i = 0; i < sizeof(vector)/sizeof(int); i++) {
43         std::cout << vector[i] << " ";
44     }
45     std::cout << std::endl;
46
47     return 0;
48 }
```



# Index

/home/bz223/downloads/acs-gerardium-rush-bauxite/include/CUnit.h, 27  
/home/bz223/downloads/acs-gerardium-rush-bauxite/include/CUnit.h, 27  
/home/bz223/downloads/acs-gerardium-rush-bauxite/include/CUnit.h, 31  
/home/bz223/downloads/acs-gerardium-rush-bauxite/include/CUnit.h, 31  
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/Circuit.cpp, 44  
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/Circuit.cpp, 44  
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/CUnit.cpp, 51  
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/Genetic\_Algorithm.cpp, 51  
/home/bz223/downloads/acs-gerardium-rush-bauxite/src/main.cpp, 65

Algorithm\_Parameters, 5  
  cross\_probability, 5  
  max\_iterations, 5  
  mutation\_rate, 6  
  population\_size, 6  
  stall\_length, 6  
  stall\_mutation\_rate, 6

all\_true  
  Genetic\_Algorithm.cpp, 52  
  Genetic\_Algorithm.h, 33

calculate\_CG  
  CUnit, 18

calculate\_CW  
  CUnit, 19

calculate\_IG  
  CUnit, 19

calculate\_IW  
  CUnit, 19

Calculate\_ResidenceTime  
  CUnit, 19

Calculate\_RHG  
  CUnit, 19

Calculate\_RHW  
  CUnit, 20

Calculate\_RIG  
  CUnit, 20

Calculate\_RIW  
  CUnit, 20

calculate\_TG  
  CUnit, 20

calculate\_TW  
  CUnit, 20

Circuit.h, 8  
  CUnit, 8  
  check\_no\_all\_same\_destination, 8  
  check\_no\_self\_recycle, 9  
  Check\_Validity, 9  
  Circuit, 9  
  concentrate\_G, 14  
  concentrate\_W, 14  
  count, 14  
  initial\_index, 14  
  isReachable, 10  
  mark\_units, 11  
  next\_feed\_G, 15  
  next\_feed\_W, 15  
  num\_units, 15  
  setup\_connections, 12  
  tails\_G, 15  
  tails\_W, 15  
  units, 15  
  update\_flows, 13

Circuit\_Parameters, 16  
  max\_iterations, 16  
  tolerance, 16

conc\_num  
  CUnit, 21

concentrate\_G  
  Circuit, 14

concentrate\_W  
  Circuit, 14

count  
  Circuit, 14

cross\_probability  
  Algorithm\_Parameters, 5

crossover  
  Genetic\_Algorithm.cpp, 52  
  Genetic\_Algorithm.h, 33

CSimulator.cpp  
  default\_circuit\_parameters, 50  
  dummy\_answer\_vector, 50  
  Evaluate\_Circuit, 45, 46

- Evaluate\_Circuit\_Cell\_Calculator, 46
- Evaluate\_Circuit\_Grade, 47
- Evaluate\_Circuit\_Recovery, 48, 49
- Save\_ResultToTxt, 49
- CSimulator.h
  - Evaluate\_Circuit, 28
  - Evaluate\_Circuit\_Cell\_Calculator, 29
  - Evaluate\_Circuit\_Grade, 29
  - Evaluate\_Circuit\_Recovery, 30
  - Save\_ResultToTxt, 30
- CUnit, 17
  - calculate\_CG, 18
  - calculate\_CW, 19
  - calculate\_IG, 19
  - calculate\_IW, 19
  - Calculate\_ResidenceTime, 19
  - Calculate\_RHG, 19
  - Calculate\_RHW, 20
  - Calculate\_RIG, 20
  - Calculate\_RIW, 20
  - calculate\_TG, 20
  - calculate\_TW, 20
  - conc\_num, 21
  - CUnit, 18
  - density, 21
  - feed\_rate\_G, 21
  - feed\_rate\_W, 21
  - inter\_num, 21
  - k\_cG, 22
  - k\_cW, 22
  - k\_iG, 22
  - k\_iW, 22
  - mark, 22
  - residence\_time, 22
  - solid\_fraction, 23
  - tails\_num, 23
  - volume, 23
- CW
  - Unit\_parameter, 24
- DEFAULT\_ALGORITHM\_PARAMETERS
  - Genetic\_Algorithm.h, 32
- default\_circuit\_parameters
  - CSimulator.cpp, 50
- density
  - CUnit, 21
- dummy\_answer\_vector
  - CSimulator.cpp, 50
- Evaluate\_Circuit
  - CSimulator.cpp, 45, 46
  - CSimulator.h, 28
- Evaluate\_Circuit\_Cell\_Calculator
  - CSimulator.cpp, 46
  - CSimulator.h, 29
- Evaluate\_Circuit\_Grade
  - CSimulator.cpp, 47
  - CSimulator.h, 29
- Evaluate\_Circuit\_Recovery
  - CSimulator.cpp, 48, 49
  - CSimulator.h, 30
- evaluate\_parents
  - Genetic\_Algorithm.cpp, 53
  - Genetic\_Algorithm.h, 34
- feed\_rate\_G
  - CUnit, 21
- feed\_rate\_W
  - CUnit, 21
- generate\_children
  - Genetic\_Algorithm.cpp, 54
  - Genetic\_Algorithm.h, 35
- generate\_population
  - Genetic\_Algorithm.cpp, 55
  - Genetic\_Algorithm.h, 36
- Genetic\_Algorithm.cpp
  - all\_true, 52
  - crossover, 52
  - evaluate\_parents, 53
  - generate\_children, 54
  - generate\_population, 55
  - inversion, 56
  - max\_value, 65
  - multipoint\_cross, 56
  - mutate, 57
  - one\_point\_cross, 58
  - optimize, 59
  - select\_parents, 61
  - substitution, 61
  - two\_point\_cross, 61
  - uniform\_cross, 62
  - value\_adjustment, 63
  - weighted\_parent, 63
  - write\_output, 64
- Genetic\_Algorithm.h
  - all\_true, 33
  - crossover, 33
  - DEFAULT\_ALGORITHM\_PARAMETERS, 32
  - evaluate\_parents, 34
  - generate\_children, 35
  - generate\_population, 36
  - inversion, 37
  - multipoint\_cross, 37
  - mutate, 38
  - one\_point\_cross, 39
  - optimize, 39
  - select\_parents, 41
  - substitution, 42
  - two\_point\_cross, 42
  - uniform\_cross, 43
  - value\_adjustment, 44
- IG
  - Unit\_parameter, 24
- initial\_index
  - Circuit, 14
- inter\_num

- CUnit, 21
- inversion
  - Genetic\_Algorithm.cpp, 56
  - Genetic\_Algorithm.h, 37
- isReachable
  - Circuit, 10
- IW
  - Unit\_parameter, 24
- k\_cG
  - CUnit, 22
- k\_cW
  - CUnit, 22
- k\_iG
  - CUnit, 22
- k\_iW
  - CUnit, 22
- main
  - main.cpp, 65
- main.cpp
  - main, 65
- mark
  - CUnit, 22
- mark\_units
  - Circuit, 11
- max\_iterations
  - Algorithm\_Parameters, 5
  - Circuit\_Parameters, 16
- max\_value
  - Genetic\_Algorithm.cpp, 65
- multipoint\_cross
  - Genetic\_Algorithm.cpp, 56
  - Genetic\_Algorithm.h, 37
- mutate
  - Genetic\_Algorithm.cpp, 57
  - Genetic\_Algorithm.h, 38
- mutation\_rate
  - Algorithm\_Parameters, 6
- next\_feed\_G
  - Circuit, 15
- next\_feed\_W
  - Circuit, 15
- num\_units
  - Circuit, 15
- one\_point\_cross
  - Genetic\_Algorithm.cpp, 58
  - Genetic\_Algorithm.h, 39
- optimize
  - Genetic\_Algorithm.cpp, 59
  - Genetic\_Algorithm.h, 39
- population\_size
  - Algorithm\_Parameters, 6
- residence\_time
  - CUnit, 22
- RHG
  - Unit\_parameter, 24
- RHW
  - Unit\_parameter, 24
- RIG
  - Unit\_parameter, 25
- RIW
  - Unit\_parameter, 25
- Save\_ResultToTxt
  - CSimulator.cpp, 49
  - CSimulator.h, 30
- select\_parents
  - Genetic\_Algorithm.cpp, 61
  - Genetic\_Algorithm.h, 41
- setup\_connections
  - Circuit, 12
- solid\_fraction
  - CUnit, 23
- stall\_length
  - Algorithm\_Parameters, 6
- stall\_mutation\_rate
  - Algorithm\_Parameters, 6
- substitution
  - Genetic\_Algorithm.cpp, 61
  - Genetic\_Algorithm.h, 42
- tails\_G
  - Circuit, 15
- tails\_num
  - CUnit, 23
- tails\_W
  - Circuit, 15
- tau
  - Unit\_parameter, 25
- TG
  - Unit\_parameter, 25
- tolerance
  - Circuit\_Parameters, 16
- TW
  - Unit\_parameter, 25
- two\_point\_cross
  - Genetic\_Algorithm.cpp, 61
  - Genetic\_Algorithm.h, 42
- uniform\_cross
  - Genetic\_Algorithm.cpp, 62
  - Genetic\_Algorithm.h, 43
- Unit\_parameter, 23
  - CG, 24
  - CW, 24
  - IG, 24
  - IW, 24
  - RHG, 24
  - RHW, 24
  - RIG, 25
  - RIW, 25
  - tau, 25
  - TG, 25
  - TW, 25

units

    Circuit, [15](#)

update\_flows

    Circuit, [13](#)

value\_adjustment

    Genetic\_Algorithm.cpp, [63](#)

    Genetic\_Algorithm.h, [44](#)

volume

    CUnit, [23](#)

weighted\_parent

    Genetic\_Algorithm.cpp, [63](#)

write\_output

    Genetic\_Algorithm.cpp, [64](#)