

Palusznium-Rush Ilmenite Optimiser

1.0

Generated on Fri May 23 2025 14:41:18 for Palusznium-Rush Ilmenite Optimiser by Doxygen
1.9.8

Fri May 23 2025 14:41:18

1 Palusznium-Rush Ilmenite Optimiser

A self-contained C++17 + OpenMP toolkit that designs mineral-processing circuits with a genetic-algorithm core, plus Python helpers for visualisation.

1.1 1 Problem in a nutshell

We must configure a circuit of identical separation units so that two valuable minerals—**palusznium (P)** and **gormanium (G)**—are recovered profitably while punishing waste entrainment and oversized equipment. The design space (both topology **and** unit volumes) is combinatorial, so we use a **genetic algorithm** (GA) to search it.

Full background → [docs/Problem Statement for Genetic Algorithms Project 2025.pdf](#).

1.2 2 Repository layout

```
CMakeLists.txt          # top-level build
include/               # public headers used by src/
    CCircuit.h         # circuit class
    CUnit.h            # single separation unit
    CSimulator.h       # helper for testing/plotting
    Genetic_Algorithm.h # GA interface
...
src/                  # implementation (.cpp files)
    CCircuit.cpp
    CUnit.cpp
    CSimulator.cpp
    Genetic_Algorithm.cpp
    main.cpp           # CLI entry point
docs/                 # PDF + Markdown design docs
plotting/              # **generated** on first run
    circuit_results.csv # GA output (append-only)
    plot.py             # matplotlib helper → png/pdf
    cplot.cpp           # minimal C++ visualiser (optional)
tests/                # GoogleTest unit-tests & CTest driver
rng_examples/          # tiny demos comparing RNG quality
hooks/                # pre-commit & install helper for git hooks
parameters.txt         # runtime GA settings (human-readable)
```

Tip: The project builds out-of-tree; build/ is ignored by git.

1.3 </blockquote>

1.4 3 Build & run

1.4.1 3.1 Prerequisites

Tool	Minimum	Tested on
CMake	3.12	3.27
C++ compiler	C++17 + OpenMP	GCC 9, Clang 14, MSVC 19.36
Python (visualisation)	3.8	3.11

Install Python deps with pip install -r requirements.txt (matplotlib + pandas).

1.4.2 3.2 Build *(one-liner)*

Run the helper script; it creates the build/ directory, configures CMake for a **Release** build and compiles the optimiser.

```
./build.sh      # → build/bin/Optimizer (plus unit-test binaries)
```

If you need a clean rebuild:

```
./build.sh clean # wipes previous build/ then recompiles
```

1.4.3 3.3 Run

```
./run.sh        # builds + runs optimiser, then auto-plots results
```

- Appends one line to plotting/circuit_results.csv.

- Calls the Python helper `plotting/main.py -f` which reads that CSV and writes a **PNG flow-sheet diagram + vector table** to `output/flowchart.png`.

Optional flags:

```
./run.sh d      # discrete only (recommended)
./run.sh h      # hybrid (shape + volumes)
./run.sh c      # continuous DEV-ONLY
```

Rendering requires **Graphviz**, **Pillow** and **pandas**; install once with `pip install -r plotting/requirements.txt`.

----- ----- ----- d connections only explore profitable flowsheets c -volumes only (connections frozen) – DEV-ONLY . \ This mode does <i>not</i> find profitable solutions; it is kept for unit-testing kinetics & cost functions h alternates d c end-to-end optimisation
--

1.5 4 <tt>parameters.txt</tt> — full reference

Every run-time option is in `parameters.txt` so you can tune the optimiser without recompiling.

Key	Type / Range	Default	Description
<code>num_units</code>	integer 2	6	Number of separation units *(vector length=2·n+1)*
<code>mode</code>	d c h	h	GA operating mode: discrete, continuous (dev-only), or hybrid
<code>max_iterations</code>	integer	100	GA generations per optimisation call
<code>population_size</code>	integer	600	Individuals per generation
<code>elite_count</code>	integer	2	Best genomes copied unchanged each generation
<code>tournament_size</code>	integer	3	k-way tournament selection pressure
<code>crossover_probability</code>	0–1	0.9	Chance two parents cross
<code>mutation_probability</code>	0–1	0.08	Per-gene mutation chance (all modes)
<code>mutation_step_size</code>	integer 1	3	Max ± step for discrete "creep"
<code>use_inversion</code>	bool	true	Enable contiguous slice reversal (discrete)
<code>inversion_probability</code>	0–1	0.2	Chance <i>per child</i> that inversion occurs
<code>use_scaling_mutation</code>	bool	true	Enable multiplicative tweak for genes
<code>scaling_mutation_prob</code>	0–1	0.3	Probability a child gets scaling mutation
<code>scaling_mutation_min</code>	>0	0.7	Lower bound of scaling factor
<code>scaling_mutation_max</code>	>1	1.3	Upper bound of scaling factor
<code>convergence_threshold</code>	real 0	0.1	fitness below which a change is deemed "no improvement"
<code>stall_generations</code>	integer	50	Stop if no improvement for this many generations
<code>verbose</code>	bool	true	Print progress every 10 generations
<code>log_results</code>	bool	false	Append CSV copy of every generation to <code>log_file</code>
<code>log_file</code>	filename	<code>ga_run.log</code>	Only used if <code>log_results = true</code>
<code>random_seed</code>	integer 1	42	0 → deterministic RNG, 1 → random seed

Tip: change a value, save the file, re-run `./run.sh` — no rebuild is needed.

1.6 </blockquote>

1.7 5 Interpreting output Analysing results

1.7.1 5 Analysing results

5.1 CSV format

Every optimiser run appends to plotting/circuit_results.csv:

```
[int vector ...] , [concentration flow/unit , tailings flow/unit , ...]
```

5.2 Auto-generated flowchart

Running ./run.sh (or python plotting/main.py -f) produces output/flowchart.png:

- directed graph of the circuit with blue/red edge labels (concentrate / tails flow)
- beneath it: a table showing the integer vector laid out by unit

Open the PNG directly, or embed it in documentation.

1.8 6 Developers' guide

- **Unit kinetics** – edit `src/CUnit.cpp` (`CUnit::process`).
 - **Economic model** – tune coefficients in `src/CCircuit.cpp` (`get_economic_value`).
 - **GA extensions** – new operators live in `src/Genetic_Algorithm.cpp` (see the three `optimize` overloads).
 - **Unit tests** – add cases in `tests/`; they build and run automatically with `./build.sh test` or `ctest`.
 - **Git hooks** – `hooks/install.sh` installs clang-format, static-analysis and pre-commit checks.
-

1.9 7 Licence & citation

The code is released under the **MIT Licence** (see `LICENSE`). If you use it in academic work, please cite the original *Palusznium Rush 2025* coursework and this repository.
Happy circuit hunting!

2 Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Constants	??
Constants::Circuit	??
Constants::Economic	??
Constants::Feed	??
Constants::GA	??
Constants::Physical	??
Constants::Simulation	??
Constants::Test	??

3 Class Index

3.1 Class List

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

Algorithm_Parameters	??
Circuit	??
CircuitVector	??
CUnit	??
OptimizationResult	??
Simulator_Parameters	??

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

include/CCircuit.h	Declares the Circuit class – a mineral-processing circuit	??
include/circuit_vector.h	Circuit Vector Header	??
include/Config.h	Configuration file for the Genetic Algorithm	??
include/constants.h	Header file for project-wide constants	??
include/CSimulator.h	C++ header file for the circuit simulator	??
include/CUnit.h	Declares the CUnit class – a single separation unit in the mineral-processing circuit (e.g.	??
include/Genetic_Algorithm.h	Genetic Algorithm Header	??
src/CCircuit.cpp	Implementation of the Circuit class	??
src/CSimulator.cpp	C++ source file for the circuit simulator	??
src/CUnit.cpp	Implementation of the CUnit class	??
src/Genetic_Algorithm.cpp	Genetic Algorithm Implementation	??
src/main.cpp		??

5 Namespace Documentation

5.1 Constants Namespace Reference

Namespaces

- namespace [Circuit](#)
- namespace [Economic](#)
- namespace [Feed](#)
- namespace [GA](#)
- namespace [Physical](#)
- namespace [Simulation](#)
- namespace [Test](#)

5.2 Constants::Circuit Namespace Reference

Variables

- `constexpr double DEFAULT_UNIT_VOLUME = 10.0`
- `constexpr double MIN_UNIT_VOLUME = 2.5`
- `constexpr double MAX_UNIT_VOLUME = 20.0`
- `constexpr double MAX_CIRCUIT_VOLUME = 150.0`
- `constexpr int DEFAULT_NUM_UNITS = 10`

5.2.1 Variable Documentation

DEFAULT_NUM_UNITS

```
constexpr int Constants::Circuit::DEFAULT_NUM_UNITS = 10 [constexpr]  
Definition at line 97 of file constants.h.
```

DEFAULT_UNIT_VOLUME

```
constexpr double Constants::Circuit::DEFAULT_UNIT_VOLUME = 10.0 [constexpr]  
Definition at line 92 of file constants.h.
```

MAX_CIRCUIT_VOLUME

```
constexpr double Constants::Circuit::MAX_CIRCUIT_VOLUME = 150.0 [constexpr]  
Definition at line 95 of file constants.h.
```

MAX_UNIT_VOLUME

```
constexpr double Constants::Circuit::MAX_UNIT_VOLUME = 20.0 [constexpr]  
Definition at line 94 of file constants.h.
```

MIN_UNIT_VOLUME

```
constexpr double Constants::Circuit::MIN_UNIT_VOLUME = 2.5 [constexpr]  
Definition at line 93 of file constants.h.
```

5.3 Constants::Economic Namespace Reference

Variables

- `constexpr double PALUSNIUM_VALUE_IN_PALUSNIUM_STREAM = 120.0`
- `constexpr double GORMANIUM_VALUE_IN_PALUSNIUM_STREAM = -20.0`
- `constexpr double WASTE_PENALTY_IN_PALUSNIUM_STREAM = -300.0`
- `constexpr double PALUSNIUM_VALUE_IN_GORMANIUM_STREAM = 0.0`
- `constexpr double GORMANIUM_VALUE_IN_GORMANIUM_STREAM = 80.0`

- `constexpr double WASTE_PENALTY_IN_GORMANIUM_STREAM = -25.0`
- `constexpr double COST_COEFFICIENT = 5.0`
- `constexpr double VOLUME_PENALTY_COEFFICIENT = 1000.0`

5.3.1 Variable Documentation

COST_COEFFICIENT

```
constexpr double Constants::Economic::COST_COEFFICIENT = 5.0 [constexpr]  
Definition at line 77 of file constants.h.
```

GORMANIUM_VALUE_IN_GORMANIUM_STREAM

```
constexpr double Constants::Economic::GORMANIUM_VALUE_IN_GORMANIUM_STREAM = 80.0 [constexpr]  
Definition at line 73 of file constants.h.
```

GORMANIUM_VALUE_IN_PALUSZNIA_STREAM

```
constexpr double Constants::Economic::GORMANIUM_VALUE_IN_PALUSZNIA_STREAM = -20.0 [constexpr]  
Definition at line 69 of file constants.h.
```

PALUSZNIA_VALUE_IN_GORMANIUM_STREAM

```
constexpr double Constants::Economic::PALUSZNIA_VALUE_IN_GORMANIUM_STREAM = 0.0 [constexpr]  
Definition at line 72 of file constants.h.
```

PALUSZNIA_VALUE_IN_PALUSZNIA_STREAM

```
constexpr double Constants::Economic::PALUSZNIA_VALUE_IN_PALUSZNIA_STREAM = 120.0 [constexpr]  
Definition at line 68 of file constants.h.
```

VOLUME_PENALTY_COEFFICIENT

```
constexpr double Constants::Economic::VOLUME_PENALTY_COEFFICIENT = 1000.0 [constexpr]  
Definition at line 78 of file constants.h.
```

WASTE_PENALTY_IN_GORMANIUM_STREAM

```
constexpr double Constants::Economic::WASTE_PENALTY_IN_GORMANIUM_STREAM = -25.0 [constexpr]  
Definition at line 74 of file constants.h.
```

WASTE_PENALTY_IN_PALUSZNIA_STREAM

```
constexpr double Constants::Economic::WASTE_PENALTY_IN_PALUSZNIA_STREAM = -300.0 [constexpr]  
Definition at line 70 of file constants.h.
```

5.4 Constants::Feed Namespace Reference

Variables

- `constexpr double DEFAULT_PALUSZNIA_FEED = 8.0`
- `constexpr double DEFAULT_GORMANIUM_FEED = 12.0`
- `constexpr double DEFAULT_WASTE_FEED = 80.0`

5.4.1 Variable Documentation

DEFAULT_GORMANIUM_FEED

```
constexpr double Constants::Feed::DEFAULT_GORMANIUM_FEED = 12.0 [constexpr]  
Definition at line 85 of file constants.h.
```

DEFAULT_PALUSZNIUM_FEED

```
constexpr double Constants::Feed::DEFAULT_PALUSZNIUM_FEED = 8.0 [constexpr]  
Definition at line 84 of file constants.h.
```

DEFAULT_WASTE_FEED

```
constexpr double Constants::Feed::DEFAULT_WASTE_FEED = 80.0 [constexpr]  
Definition at line 86 of file constants.h.
```

5.5 Constants::GA Namespace Reference

Variables

- `constexpr int DEFAULT_POPULATION_SIZE = 100`
- `constexpr int DEFAULT_MAX_GENERATIONS = 1000`
- `constexpr double DEFAULT_CROSSOVER_RATE = 0.8`
- `constexpr double DEFAULT_MUTATION_RATE = 0.01`
- `constexpr int DEFAULT_ELITE_COUNT = 1`

5.5.1 Variable Documentation

DEFAULT_CROSSOVER_RATE

```
constexpr double Constants::GA::DEFAULT_CROSSOVER_RATE = 0.8 [constexpr]  
Definition at line 113 of file constants.h.
```

DEFAULT_ELITE_COUNT

```
constexpr int Constants::GA::DEFAULT_ELITE_COUNT = 1 [constexpr]  
Definition at line 115 of file constants.h.
```

DEFAULT_MAX_GENERATIONS

```
constexpr int Constants::GA::DEFAULT_MAX_GENERATIONS = 1000 [constexpr]  
Definition at line 112 of file constants.h.
```

DEFAULT_MUTATION_RATE

```
constexpr double Constants::GA::DEFAULT_MUTATION_RATE = 0.01 [constexpr]  
Definition at line 114 of file constants.h.
```

DEFAULT_POPULATION_SIZE

```
constexpr int Constants::GA::DEFAULT_POPULATION_SIZE = 100 [constexpr]  
Definition at line 111 of file constants.h.
```

5.6 Constants::Physical Namespace Reference

Variables

- `constexpr double MATERIAL_DENSITY = 3000.0`
- `constexpr double SOLIDS_CONTENT = 0.1`
- `constexpr double K_PALUSZNIUM = 0.008`
- `constexpr double K_GORMANIUM = 0.004`
- `constexpr double K_WASTE = 0.0005`

5.6.1 Variable Documentation

K_GORMANIUM

```
constexpr double Constants::Physical::K_GORMANIUM = 0.004 [constexpr]  
Definition at line 60 of file constants.h.
```

K_PALUSZNIUM

```
constexpr double Constants::Physical::K_PALUSZNIUM = 0.008 [constexpr]  
Definition at line 59 of file constants.h.
```

K_WASTE

```
constexpr double Constants::Physical::K_WASTE = 0.0005 [constexpr]  
Definition at line 61 of file constants.h.
```

MATERIAL_DENSITY

```
constexpr double Constants::Physical::MATERIAL_DENSITY = 3000.0 [constexpr]  
Definition at line 55 of file constants.h.
```

SOLIDS_CONTENT

```
constexpr double Constants::Physical::SOLIDS_CONTENT = 0.1 [constexpr]  
Definition at line 56 of file constants.h.
```

5.7 Constants::Simulation Namespace Reference

Variables

- `constexpr double DEFAULT_TOLERANCE = 1e-6`
- `constexpr int DEFAULT_MAX_ITERATIONS = 1000`
- `constexpr double MIN_FLOW_RATE = 1e-6`

5.7.1 Variable Documentation

DEFAULT_MAX_ITERATIONS

```
constexpr int Constants::Simulation::DEFAULT_MAX_ITERATIONS = 1000 [constexpr]  
Definition at line 104 of file constants.h.
```

DEFAULT_TOLERANCE

```
constexpr double Constants::Simulation::DEFAULT_TOLERANCE = 1e-6 [constexpr]  
Definition at line 103 of file constants.h.
```

MIN_FLOW_RATE

```
constexpr double Constants::Simulation::MIN_FLOW_RATE = 1e-6 [constexpr]  
Definition at line 105 of file constants.h.
```

5.8 Constants::Test Namespace Reference

Variables

- `constexpr double DEFAULT_PALUSZNIUM_FEED = 10.0`
- `constexpr double DEFAULT_GORMANIUM_FEED = 10.0`
- `constexpr double DEFAULT_WASTE_FEED = 10.0`
- `constexpr double PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM = 100.0`
- `constexpr double GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM = 0.0`

- `constexpr double WASTE_PENALTY_IN_PALUSZNIUM_STREAM = 0.0`
- `constexpr double PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM = 0.0`
- `constexpr double GORMANIUM_VALUE_IN_GORMANIUM_STREAM = 100.0`
- `constexpr double WASTE_PENALTY_IN_GORMANIUM_STREAM = 0.0`
- `constexpr double COST_COEFFICIENT = 5.0`
- `constexpr double VOLUME_PENALTY_COEFFICIENT = 1000.0`
- `constexpr double MATERIAL_DENSITY = 3000.0`
- `constexpr double SOLIDS_CONTENT = 0.1`
- `constexpr double K_PALUSZNIUM = 0.008`
- `constexpr double K_GORMANIUM = 0.004`
- `constexpr double K_WASTE = 0.0005`
- `constexpr double DEFAULT_UNIT_VOLUME = 5.0`
- `constexpr double MIN_UNIT_VOLUME = 2.5`
- `constexpr double MAX_UNIT_VOLUME = 20.0`
- `constexpr double MAX_CIRCUIT_VOLUME = 150.0`
- `constexpr int DEFAULT_NUM_UNITS = 10`

5.8.1 Variable Documentation

COST_COEFFICIENT

```
constexpr double Constants::Test::COST_COEFFICIENT = 5.0 [constexpr]
Definition at line 30 of file constants.h.
```

DEFAULT_GORMANIUM_FEED

```
constexpr double Constants::Test::DEFAULT_GORMANIUM_FEED = 10.0 [constexpr]
Definition at line 17 of file constants.h.
Referenced by Circuit::Circuit().
```

DEFAULT_NUM_UNITS

```
constexpr int Constants::Test::DEFAULT_NUM_UNITS = 10 [constexpr]
Definition at line 47 of file constants.h.
```

DEFAULT_PALUSZNIUM_FEED

```
constexpr double Constants::Test::DEFAULT_PALUSZNIUM_FEED = 10.0 [constexpr]
Definition at line 16 of file constants.h.
Referenced by Circuit::Circuit().
```

DEFAULT_UNIT_VOLUME

```
constexpr double Constants::Test::DEFAULT_UNIT_VOLUME = 5.0 [constexpr]
Definition at line 42 of file constants.h.
Referenced by CUnit::CUnit().
```

DEFAULT_WASTE_FEED

```
constexpr double Constants::Test::DEFAULT_WASTE_FEED = 10.0 [constexpr]
Definition at line 18 of file constants.h.
Referenced by Circuit::Circuit().
```

GORMANIUM_VALUE_IN_GORMANIUM_STREAM

```
constexpr double Constants::Test::GORMANIUM_VALUE_IN_GORMANIUM_STREAM = 100.0 [constexpr]
Definition at line 26 of file constants.h.
Referenced by Circuit::Circuit().
```

GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM

```
constexpr double Constants::Test::GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM = 0.0 [constexpr]
Definition at line 22 of file constants.h.
Referenced by Circuit::Circuit().
```

K_GORMANIUM

```
constexpr double Constants::Test::K_GORMANIUM = 0.004 [constexpr]
Definition at line 39 of file constants.h.
Referenced by CUnit::CUnit().
```

K_PALUSZNIUM

```
constexpr double Constants::Test::K_PALUSZNIUM = 0.008 [constexpr]
Definition at line 38 of file constants.h.
Referenced by CUnit::CUnit().
```

K_WASTE

```
constexpr double Constants::Test::K_WASTE = 0.0005 [constexpr]
Definition at line 40 of file constants.h.
Referenced by CUnit::CUnit().
```

MATERIAL_DENSITY

```
constexpr double Constants::Test::MATERIAL_DENSITY = 3000.0 [constexpr]
Definition at line 34 of file constants.h.
Referenced by CUnit::CUnit().
```

MAX_CIRCUIT_VOLUME

```
constexpr double Constants::Test::MAX_CIRCUIT_VOLUME = 150.0 [constexpr]
Definition at line 45 of file constants.h.
```

MAX_UNIT_VOLUME

```
constexpr double Constants::Test::MAX_UNIT_VOLUME = 20.0 [constexpr]
Definition at line 44 of file constants.h.
Referenced by CUnit::CUnit().
```

MIN_UNIT_VOLUME

```
constexpr double Constants::Test::MIN_UNIT_VOLUME = 2.5 [constexpr]
Definition at line 43 of file constants.h.
Referenced by CUnit::CUnit().
```

PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM

```
constexpr double Constants::Test::PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM = 0.0 [constexpr]
Definition at line 25 of file constants.h.
Referenced by Circuit::Circuit().
```

PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM

```
constexpr double Constants::Test::PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM = 100.0 [constexpr]
Definition at line 21 of file constants.h.
Referenced by Circuit::Circuit().
```

SOLIDS_CONTENT

```
constexpr double Constants::Test::SOLIDS_CONTENT = 0.1 [constexpr]
```

Definition at line [35](#) of file [constants.h](#).

Referenced by [CUnit::CUnit\(\)](#).

VOLUME_PENALTY_COEFFICIENT

```
constexpr double Constants::Test::VOLUME_PENALTY_COEFFICIENT = 1000.0 [constexpr]
```

Definition at line [31](#) of file [constants.h](#).

WASTE_PENALTY_IN_GORMANIUM_STREAM

```
constexpr double Constants::Test::WASTE_PENALTY_IN_GORMANIUM_STREAM = 0.0 [constexpr]
```

Definition at line [27](#) of file [constants.h](#).

Referenced by [Circuit::Circuit\(\)](#).

WASTE_PENALTY_IN_PALUSZNIAUM_STREAM

```
constexpr double Constants::Test::WASTE_PENALTY_IN_PALUSZNIAUM_STREAM = 0.0 [constexpr]
```

Definition at line [23](#) of file [constants.h](#).

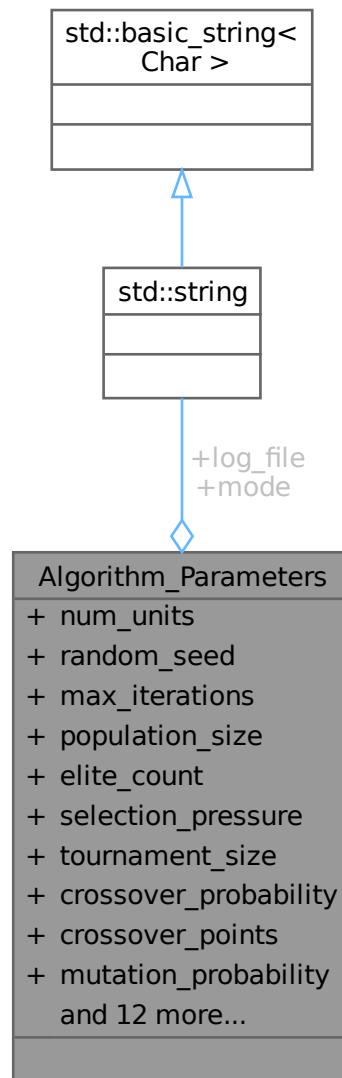
Referenced by [Circuit::Circuit\(\)](#).

6 Class Documentation

6.1 Algorithm_Parameters Struct Reference

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

Collaboration diagram for Algorithm_Parameters:



Public Attributes

- int `num_units` = 10
- std::string `mode` = "h"
- int `random_seed` = -1
- int `max_iterations` = 1000
- int `population_size` = 100
- int `elite_count` = 1
- double `selection_pressure` = 1.5
- int `tournament_size` = 2
- double `crossover_probability` = 0.8
- int `crossover_points` = 1
- double `mutation_probability` = 0.01
- int `mutation_step_size` = 2

- bool `allow_mutation_wrapping` = true
- bool `use_inversion` = true
- double `inversion_probability` = 0.05
- bool `use_scaling_mutation` = true
- double `scaling_mutation_prob` = 0.2
- double `scaling_mutation_min` = 0.8
- double `scaling_mutation_max` = 1.2
- double `convergence_threshold` = 1e-6
- int `stall_generations` = 50
- bool `verbose` = false
- bool `log_results` = false
- std::string `log_file` = "ga_log.txt"

6.1.1 Detailed Description

Definition at line 18 of file [Genetic_Algorithm.h](#).

6.1.2 Member Data Documentation

`allow_mutation_wrapping`

bool Algorithm_Parameters::allow_mutation_wrapping = true

Definition at line 43 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), and [main\(\)](#).

`convergence_threshold`

double Algorithm_Parameters::convergence_threshold = 1e-6

Definition at line 56 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), [main\(\)](#), [optimize\(\)](#), and [optimize\(\)](#).

`crossover_points`

int Algorithm_Parameters::crossover_points = 1

Definition at line 38 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), and [main\(\)](#).

`crossover_probability`

double Algorithm_Parameters::crossover_probability = 0.8

Definition at line 37 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), [main\(\)](#), [optimize\(\)](#), and [optimize\(\)](#).

`elite_count`

int Algorithm_Parameters::elite_count = 1

Definition at line 30 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), and [main\(\)](#).

`inversion_probability`

double Algorithm_Parameters::inversion_probability = 0.05

Definition at line 47 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), [main\(\)](#), and [optimize\(\)](#).

`log_file`

std::string Algorithm_Parameters::log_file = "ga_log.txt"

Definition at line 62 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), and [main\(\)](#).

log_results

```
bool Algorithm_Parameters::log_results = false
```

Definition at line 61 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), and [main\(\)](#).

max_iterations

```
int Algorithm_Parameters::max_iterations = 1000
```

Definition at line 28 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), [main\(\)](#), [optimize\(\)](#), and [optimize\(\)](#).

mode

```
std::string Algorithm_Parameters::mode = "h"
```

Definition at line 24 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), and [main\(\)](#).

mutation_probability

```
double Algorithm_Parameters::mutation_probability = 0.01
```

Definition at line 41 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), [main\(\)](#), [optimize\(\)](#), and [optimize\(\)](#).

mutation_step_size

```
int Algorithm_Parameters::mutation_step_size = 2
```

Definition at line 42 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), [main\(\)](#), [optimize\(\)](#), and [optimize\(\)](#).

num_units

```
int Algorithm_Parameters::num_units = 10
```

Definition at line 21 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), and [main\(\)](#).

population_size

```
int Algorithm_Parameters::population_size = 100
```

Definition at line 29 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), [main\(\)](#), [optimize\(\)](#), and [optimize\(\)](#).

random_seed

```
int Algorithm_Parameters::random_seed = -1
```

Definition at line 27 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), and [main\(\)](#).

scaling_mutation_max

```
double Algorithm_Parameters::scaling_mutation_max = 1.2
```

Definition at line 53 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), [main\(\)](#), and [optimize\(\)](#).

scaling_mutation_min

```
double Algorithm_Parameters::scaling_mutation_min = 0.8
```

Definition at line 52 of file [Genetic_Algorithm.h](#).

Referenced by [load_parameters\(\)](#), [main\(\)](#), and [optimize\(\)](#).

scaling_mutation_prob

```
double Algorithm_Parameters::scaling_mutation_prob = 0.2
Definition at line 51 of file Genetic\_Algorithm.h.
Referenced by load\_parameters\(\), main\(\), and optimize\(\).
```

selection_pressure

```
double Algorithm_Parameters::selection_pressure = 1.5
Definition at line 33 of file Genetic\_Algorithm.h.
Referenced by load\_parameters\(\), and main\(\).
```

stall_generations

```
int Algorithm_Parameters::stall_generations = 50
Definition at line 57 of file Genetic\_Algorithm.h.
Referenced by load\_parameters\(\), main\(\), optimize\(\), and optimize\(\).
```

tournament_size

```
int Algorithm_Parameters::tournament_size = 2
Definition at line 34 of file Genetic\_Algorithm.h.
Referenced by load\_parameters\(\), main\(\), optimize\(\), and optimize\(\).
```

use_inversion

```
bool Algorithm_Parameters::use_inversion = true
Definition at line 46 of file Genetic\_Algorithm.h.
Referenced by load\_parameters\(\), main\(\), and optimize\(\).
```

use_scaling_mutation

```
bool Algorithm_Parameters::use_scaling_mutation = true
Definition at line 50 of file Genetic\_Algorithm.h.
Referenced by load\_parameters\(\), main\(\), and optimize\(\).
```

verbose

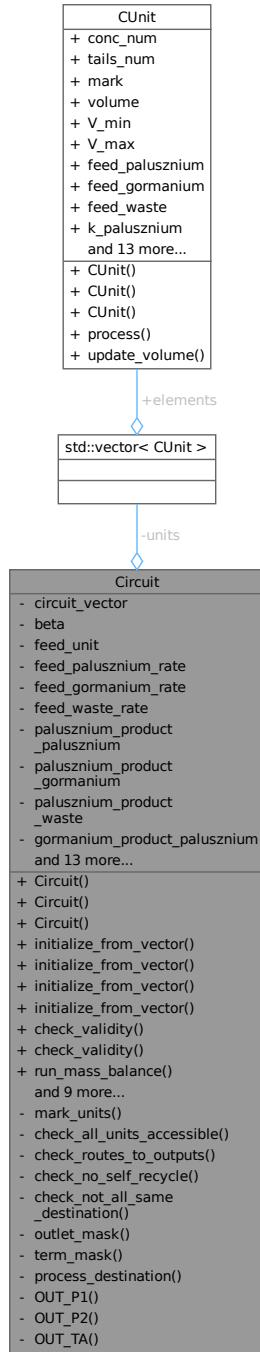
```
bool Algorithm_Parameters::verbose = false
Definition at line 60 of file Genetic\_Algorithm.h.
Referenced by load\_parameters\(\), main\(\), optimize\(\), and optimize\(\).
The documentation for this struct was generated from the following file:
```

- [include/Genetic_Algorithm.h](#)

6.2 Circuit Class Reference

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

Collaboration diagram for Circuit:



Public Member Functions

- [Circuit \(int num_units\)](#)
Constructor for the Circuit class.
- [Circuit \(int num_units, double *beta\)](#)
Constructor for the Circuit class.
- [Circuit \(int num_units, double *beta, bool testFlag\)](#)
Constructor for the Circuit class.

- bool `initialize_from_vector` (int vector_size, const int *`circuit_vector`)

Initialize the circuit from a circuit vector.
- bool `initialize_from_vector` (int vector_size, const int *`circuit_vector`, const double *`beta`)

Initialize the circuit from a circuit vector.
- bool `initialize_from_vector` (int vector_size, const int *`circuit_vector`, bool testFlag)

Initialize the circuit from a circuit vector.
- bool `initialize_from_vector` (int vector_size, const int *`circuit_vector`, const double *`beta`, bool testFlag)

Initialize the circuit from a circuit vector.
- bool `check_validity` (int vector_size, const int *`circuit_vector`)

Check the validity of the circuit.
- bool `check_validity` (int vector_size, const int *`circuit_vector`, int unit_parameters_size, double *unit_parameters)

Check the validity of the circuit vector and its parameters.
- bool `run_mass_balance` (double tolerance=1e-6, int max_iterations=1000)

Run mass balance calculations for the circuit.
- double `get_economic_value` () const

Get the economic value of the circuit.
- double `get_palusznium_recovery` () const

Get the recovery of valuable materials.
- double `get_germanium_recovery` () const

Get the recovery of germanium.
- double `get_palusznium_grade` () const

Get the grade of palusznium.
- double `get_germanium_grade` () const

Get the grade of germanium.
- bool `export_to_dot` (const std::string &filename) const

Export the circuit to a DOT file.
- bool `save_all_units_to_csv` (const std::string &filename)

Save all units to a CSV file.
- bool `save_vector_to_csv` (const std::string &filename)

Save a vector to a CSV file.
- bool `save_output_info` (const std::string &filename)

Save the circuit data to a CSV file.

Private Member Functions

- void `mark_units` (int unit_num)

Mark the units in the circuit.
- bool `check_all_units_accessible` () const
- bool `check_routes_to_outputs` () const
- bool `check_no_self_recycle` () const
- bool `check_not_all_same_destination` () const
- uint8_t `outlet_mask` (int unit_idx, std::vector< int8_t > &cache) const

Get the terminal mask for a given unit.
- uint8_t `term_mask` (int start) const

Get the terminal mask for a given unit.
- void `process_destination` (int dest, uint8_t &mask, std::vector< bool > &visited, std::queue< int > &q) const

Process the destination unit.
- int `OUT_P1` () const
- int `OUT_P2` () const
- int `OUT_TA` () const

Private Attributes

- std::vector< CUnit > units
- const int * circuit_vector
- double * beta
- int feed_unit
- double feed_palusznium_rate
- double feed_germanium_rate
- double feed_waste_rate
- double palusznium_product_palusznium
- double palusznium_product_germanium
- double palusznium_product_waste
- double germanium_product_palusznium
- double germanium_product_germanium
- double germanium_product_waste
- double tailings_palusznium
- double tailings_germanium
- double tailings_waste
- double palusznium_value
- double germanium_value
- double germanium_value_in_palusznium
- double palusznium_value_in_germanium
- double waste_penalty_palusznium
- double waste_penalty_germanium
- int n
- int feed_dest = 0

6.2.1 Detailed Description

Definition at line 44 of file [CCircuit.h](#).

6.2.2 Constructor & Destructor Documentation

Circuit() [1/3]

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

Constructor for the [Circuit](#) class.

Definition at line 26 of file [CCircuit.cpp](#).

Circuit() [2/3]

```
Circuit::Circuit (
    int num_units,
    double * beta )
```

Constructor for the [Circuit](#) class.

This constructor initializes the circuit with the given number of units and a pointer to the beta array.

Parameters

<i>num_units</i>	Number of units in the circuit
<i>beta</i>	Pointer to the beta array

Definition at line 252 of file [CCircuit.cpp](#).

Circuit() [3/3]

```
Circuit::Circuit (
```

```
    int num_units,
    double * beta,
    bool testFlag )
```

Constructor for the [Circuit](#) class.

This constructor initializes the circuit with the given number of units, a pointer to the beta array, and a test flag.

Parameters

<i>num_units</i>	Number of units in the circuit
<i>beta</i>	Pointer to the beta array
<i>testFlag</i>	Test flag to indicate whether to use test parameters

Definition at line 279 of file [CCircuit.cpp](#).

References [Constants::Test::DEFAULT_GORMANIUM_FEED](#), [Constants::Test::DEFAULT_PALUSZNIUM_FEED](#), [Constants::Test::DEFAULT_WASTE_FEED](#), [feed_gormanium_rate](#), [feed_paluszniun_rate](#), [feed_waste_rate](#), [gormanium_value](#), [Constants::Test::GORMANIUM_VALUE_IN_GORMANIUM_STREAM](#), [gormanium_value_in_paluszniun](#), [Constants::Test::GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM](#), [paluszniun_value](#), [paluszniun_value_in_gormanium](#), [Constants::Test::PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM](#), [Constants::Test::PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM](#), [waste_penalty_gormanium](#), [Constants::Test::WASTE_PENALTY_IN_GORMANIUM_STREAM](#), [Constants::Test::WASTE_PENALTY_IN_PALUSZNIUM_STREAM](#) and [waste_penalty_paluszniun](#).

6.2.3 Member Function Documentation

[check_all_units_accessible\(\)](#)

```
bool Circuit::check_all_units_accessible ( ) const [private]
```

[check_no_self_recycle\(\)](#)

```
bool Circuit::check_no_self_recycle ( ) const [private]
```

[check_not_all_same_destination\(\)](#)

```
bool Circuit::check_not_all_same_destination ( ) const [private]
```

[check_routes_to_outputs\(\)](#)

```
bool Circuit::check_routes_to_outputs ( ) const [private]
```

[check_validity\(\) \[1/2\]](#)

```
bool Circuit::check_validity (
    int vector_size,
    const int * vec )
```

Check the validity of the circuit.

This function checks the validity of the circuit vector by performing various checks, including length check, feed check, index check, self-loop check, same output check, reachability check, terminal check, and mass balance convergence check.

Parameters

<i>vector_size</i>	Size of the circuit vector
<i>vec</i>	Circuit vector

Returns

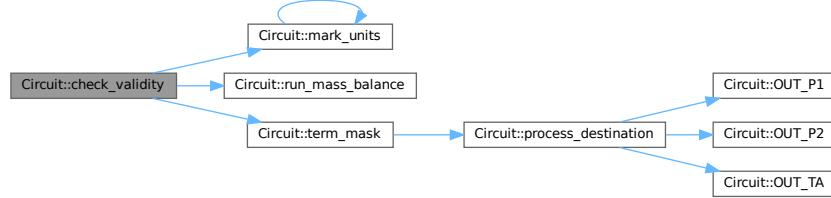
true if the circuit is valid, false otherwise

Definition at line 54 of file [CCircuit.cpp](#).

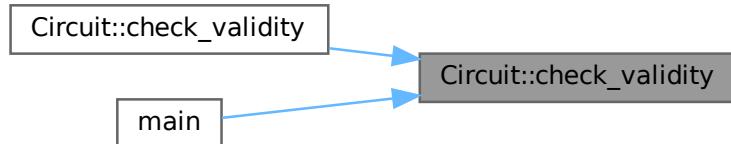
References `feed_dest`, `mark_units()`, `n`, `run_mass_balance()`, `term_mask()`, and `units`.

Referenced by `check_validity()`, and `main()`.

Here is the call graph for this function:



Here is the caller graph for this function:



`check_validity()` [2/2]

```
bool Circuit::check_validity (
    int vector_size,
    const int * circuit_vector,
    int unit_parameters_size,
    double * unit_parameters )
```

Check the validity of the circuit vector and its parameters.

This function checks the validity of the circuit vector and its parameters by performing various checks, including length check, parameter range check, and validity of the circuit vector.

Parameters

<code>vector_size</code>	Size of the circuit vector
<code>circuit_vector</code>	<code>Circuit</code> vector
<code>unit_parameters_size</code>	Size of the unit parameters
<code>unit_parameters</code>	Unit parameters

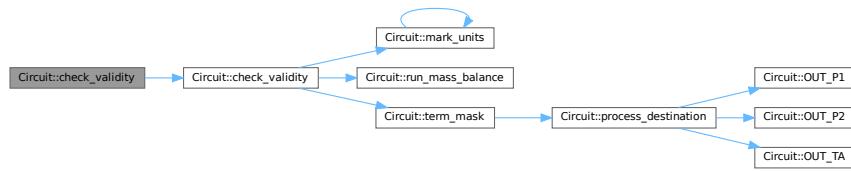
Returns

true if the circuit vector and parameters are valid, false otherwise

Definition at line 173 of file [CCircuit.cpp](#).

References [beta](#), [check_validity\(\)](#), [circuit_vector](#), and [n](#).

Here is the call graph for this function:

**export_to_dot()**

```
bool Circuit::export_to_dot (
    const std::string & filename ) const
```

Export the circuit to a DOT file.

This function exports the circuit to a DOT file for visualization.

Parameters

<i>filename</i>	The name of the output DOT file
-----------------	---------------------------------

Returns

true if export is successful, false otherwise

Definition at line 700 of file [CCircuit.cpp](#).

References [GORMANIUM_PRODUCT](#), [PALUSZNIA_PRODUCT](#), [TAILINGS_OUTPUT](#), and [units](#).

get_economic_value()

```
double Circuit::get_economic_value ( ) const
```

Get the economic value of the circuit.

This function calculates the economic value of the circuit based on the product flow rates and the values of the materials.

Returns

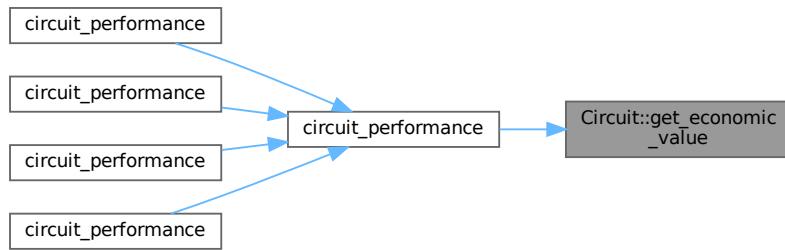
The economic value of the circuit

Definition at line 598 of file [CCircuit.cpp](#).

References [gormanium_product_gormanium](#), [gormanium_product_palusznium](#), [gormanium_product_waste](#), [gormanium_value](#), [gormanium_value_in_palusznium](#), [palusznium_product_gormanium](#), [palusznium_product_palusznium](#), [palusznium_product_waste](#), [palusznium_value](#), [palusznium_value_in_gormanium](#), [units](#), [waste_penalty_gormanium](#), and [waste_penalty_palusznium](#).

Referenced by [circuit_performance\(\)](#).

Here is the caller graph for this function:

**get_gormanium_grade()**

```
double Circuit::get_gormanium_grade ( ) const
Get the grade of gormanium.
```

This function calculates the grade of gormanium in the circuit based on the product flow rates.

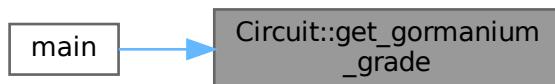
Returns

The grade of gormanium

Definition at line 685 of file [CCircuit.cpp](#).

References [gormanium_product_gormanium](#), [gormanium_product_palusznium](#), and [gormanium_product_waste](#). Referenced by [main\(\)](#).

Here is the caller graph for this function:

**get_gormanium_recovery()**

```
double Circuit::get_gormanium_recovery ( ) const
Get the recovery of gormanium.
```

This function calculates the recovery of gormanium in the circuit based on the product flow rates and the feed rates.

Returns

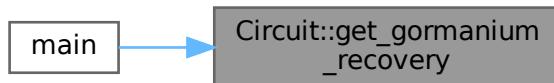
The recovery of gormanium

Definition at line 652 of file [CCircuit.cpp](#).

References [feed_gormanium_rate](#), and [gormanium_product_gormanium](#).

Referenced by [main\(\)](#).

Here is the caller graph for this function:

**get_palusznium_grade()**

double Circuit::get_palusznium_grade () const

Get the grade of palusznium.

This function calculates the grade of palusznium in the circuit based on the product flow rates.

Returns

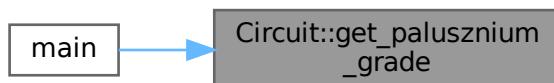
The grade of palusznium

Definition at line 670 of file [CCircuit.cpp](#).

References [palusznium_product_gormanium](#), [palusznium_product_palusznium](#), and [palusznium_product_waste](#).

Referenced by [main\(\)](#).

Here is the caller graph for this function:

**get_palusznium_recovery()**

double Circuit::get_palusznium_recovery () const

Get the recovery of valuable materials.

This function calculates the recovery of valuable materials in the circuit based on the product flow rates and the feed rates.

Returns

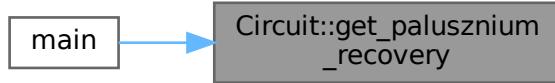
The recovery of valuable materials

Definition at line 633 of file [CCircuit.cpp](#).

References [feed_palusznium_rate](#), and [palusznium_product_palusznium](#).

Referenced by [main\(\)](#).

Here is the caller graph for this function:



initialize_from_vector() [1/4]

```
bool Circuit::initialize_from_vector (
    int vector_size,
    const int * circuit_vector )
```

Initialize the circuit from a circuit vector.

This function initializes the circuit from a circuit vector. It takes the size of the vector and the vector itself as input parameters.

Parameters

<code>vector_size</code>	Size of the circuit vector
<code>circuit_vector</code>	Circuit vector

Returns

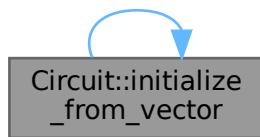
true if initialization is successful, false otherwise

Definition at line 320 of file [CCircuit.cpp](#).

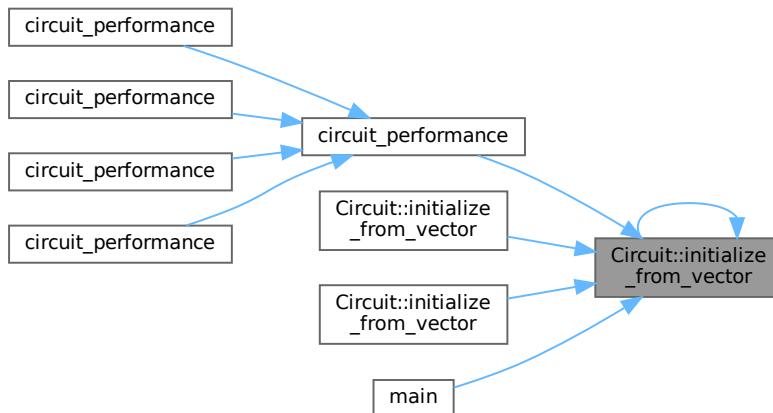
References [circuit_vector](#), and [initialize_from_vector\(\)](#).

Referenced by [circuit_performance\(\)](#), [initialize_from_vector\(\)](#), [initialize_from_vector\(\)](#), [initialize_from_vector\(\)](#), and [main\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



initialize_from_vector() [2/4]

```
    bool Circuit::initialize_from_vector (  
        int vector_size,  
        const int * circuit_vector,  
        bool testFlag )
```

Initialize the circuit from a circuit vector.

This function initializes the circuit from a circuit vector. It takes the size of the vector, the vector itself, and a test flag as input parameters.

Parameters

<code>vector_size</code>	Size of the circuit vector
<code>circuit_vector</code>	Circuit vector
<code>testFlag</code>	Test flag to indicate whether to use test parameters

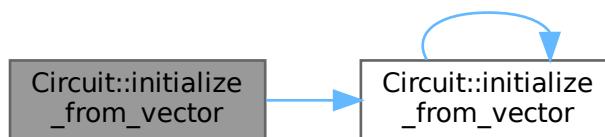
Returns

true if initialization is successful, false otherwise

Definition at line 357 of file CCircuit.cpp.

References `circuit_vector`, and `initialize_from_vector()`.

Here is the call graph for this function:



initialize_from_vector() [3/4]

```
bool Circuit::initialize_from_vector (
    int vector_size,
    const int * circuit_vector,
    const double * beta )
```

Initialize the circuit from a circuit vector.

This function initializes the circuit from a circuit vector. It takes the size of the vector, the vector itself, and a pointer to the beta array as input parameters.

Parameters

<i>vector_size</i>	Size of the circuit vector
<i>circuit_vector</i>	Circuit vector
<i>beta</i>	Pointer to the beta array

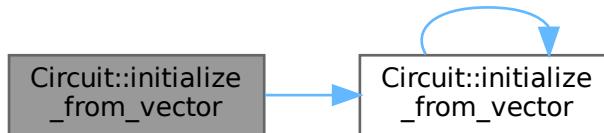
Returns

true if initialization is successful, false otherwise

Definition at line 338 of file [CCircuit.cpp](#).

References [beta](#), [circuit_vector](#), and [initialize_from_vector\(\)](#).

Here is the call graph for this function:

**initialize_from_vector() [4/4]**

```
bool Circuit::initialize_from_vector (
    int vector_size,
    const int * circuit_vector,
    const double * beta,
    bool testFlag )
```

Initialize the circuit from a circuit vector.

This function initializes the circuit from a circuit vector. It takes the size of the vector, the vector itself, a pointer to the beta array, and a test flag as input parameters.

Parameters

<i>vector_size</i>	Size of the circuit vector
<i>circuit_vector</i>	Circuit vector
<i>beta</i>	Pointer to the beta array
<i>testFlag</i>	Test flag to indicate whether to use test parameters

Returns

true if initialization is successful, false otherwise

Definition at line 378 of file [CCircuit.cpp](#).

References [beta](#), [circuit_vector](#), [feed_unit](#), [GORMANIUM_PRODUCT](#), [PALUSZNIA_PRODUCT](#), [TAILINGS_OUTPUT](#), and [units](#).

mark_units()

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

Mark the units in the circuit.

This function marks the units in the circuit as visited. It recursively traverses the circuit starting from the given unit number and marks each unit as visited.

Parameters

<code>unit_num</code>	The unit number to start marking from
-----------------------	---------------------------------------

Definition at line 216 of file [CCircuit.cpp](#).

References [mark_units\(\)](#), and [units](#).

Referenced by [check_validity\(\)](#), and [mark_units\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**OUT_P1()**

```
int Circuit::OUT_P1 ( ) const [inline], [private]
```

Definition at line 164 of file [CCircuit.h](#).

References [n](#).

Referenced by [process_destination\(\)](#).

Here is the caller graph for this function:



OUT_P2()

`int Circuit::OUT_P2 () const [inline], [private]`
Definition at line 168 of file [CCircuit.h](#).

References [n](#).

Referenced by [process_destination\(\)](#).

Here is the caller graph for this function:



OUT_TA()

`int Circuit::OUT_TA () const [inline], [private]`
Definition at line 172 of file [CCircuit.h](#).
References [n](#).
Referenced by [process_destination\(\)](#).

Here is the caller graph for this function:



outlet_mask()

```
uint8_t Circuit::outlet_mask (
    int unit_idx,
    std::vector< int8_t > & cache ) const [private]
```

process_destination()

```
void Circuit::process_destination (
    int dest,
    uint8_t & mask,
    std::vector< bool > & visited,
    std::queue< int > & q ) const [private]
```

Process the destination unit.

This function processes the destination unit and updates the mask accordingly. It also adds the destination unit to the queue for further processing.

Parameters

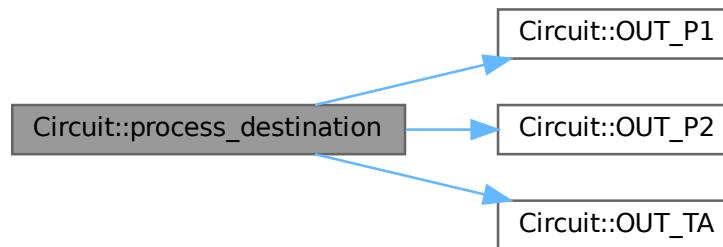
<i>dest</i>	The destination unit number
<i>mask</i>	The terminal mask
<i>visited</i>	Vector to keep track of visited units
<i>q</i>	Queue for breadth-first search

Definition at line 785 of file [CCircuit.cpp](#).

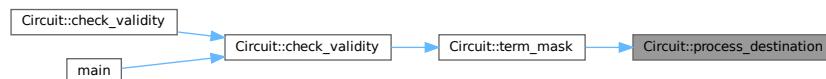
References [n](#), [OUT_P1\(\)](#), [OUT_P2\(\)](#), and [OUT_TA\(\)](#).

Referenced by [term_mask\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**run_mass_balance()**

```
bool Circuit::run_mass_balance (
    double tolerance = 1e-6,
    int max_iterations = 1000 )
```

Run mass balance calculations for the circuit.

This function runs mass balance calculations for the circuit. It takes a tolerance and a maximum number of iterations as input parameters.

Parameters

<i>tolerance</i>	Tolerance for convergence
<i>max_iterations</i>	Maximum number of iterations

Returns

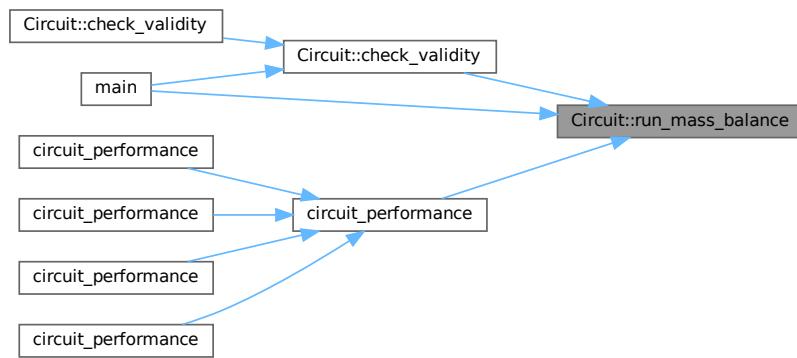
true if mass balance converges, false otherwise

Definition at line 432 of file [CCircuit.cpp](#).

References [feed_gormanium_rate](#), [feed_palusznium_rate](#), [feed_unit](#), [feed_waste_rate](#), [GORMANIUM_PRODUCT](#), [gormanium_product_gormanium](#), [gormanium_product_palusznium](#), [gormanium_product_waste](#), [PALUSZNIA_PRODUCT](#), [palusznium_product_gormanium](#), [palusznium_product_palusznium](#), [palusznium_product_waste](#), [tailings_gormanium](#), [TAILINGS_OUTPUT](#), [tailings_palusznium](#), [tailings_waste](#), and [units](#).

Referenced by [check_validity\(\)](#), [circuit_performance\(\)](#), and [main\(\)](#).

Here is the caller graph for this function:



save_all_units_to_csv()

```
bool Circuit::save_all_units_to_csv (
    const std::string & filename )
```

Save all units to a CSV file.

Save the circuit output information to a CSV file.

Parameters

<i>filename</i>	The name of the output CSV file.
-----------------	----------------------------------

Returns

true if save is successful, false otherwise

This function saves the circuit output information to a CSV file. It appends the data to the file if it already exists.

Parameters

<i>filename</i>	The name of the output CSV file
-----------------	---------------------------------

Returns

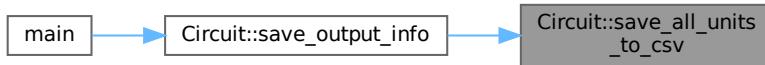
true if saving is successful, false otherwise

Definition at line 816 of file CCircuit.cpp.

References CUnit::conc_gormanium, CUnit::conc_palusznium, CUnit::conc_waste, CUnit::tails_gormanium, CUnit::tails_palusznium, CUnit::tails_waste, and units.

Referenced by [save_output_info\(\)](#).

Here is the caller graph for this function:

**save_output_info()**

```
bool Circuit::save_output_info (
    const std::string & filename )
```

Save the circuit data to a CSV file.

Save the circuit output information to a CSV file.

Parameters

<i>filename</i>	The name of the output CSV file.
-----------------	----------------------------------

Returns

true if save is successful, false otherwise

This function saves the circuit output information to a CSV file. It appends the data to the file if it already exists.

Parameters

<i>filename</i>	The name of the output CSV file
-----------------	---------------------------------

Returns

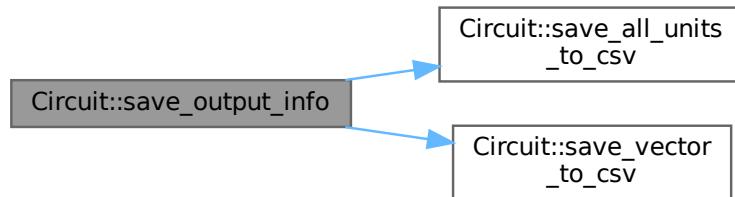
true if saving is successful, false otherwise

Definition at line 891 of file [CCircuit.cpp](#).

References [save_all_units_to_csv\(\)](#), and [save_vector_to_csv\(\)](#).

Referenced by [main\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**save_vector_to_csv()**

```
bool Circuit::save_vector_to_csv (
    const std::string & filename )
```

Save a vector to a CSV file.

Save the circuit output information to a CSV file.

Parameters

<i>filename</i>	The name of the output CSV file.
-----------------	----------------------------------

Returns

true if save is successful, false otherwise

This function saves the circuit output information to a CSV file. It appends the data to the file if it already exists.

Parameters

<i>filename</i>	The name of the output CSV file
-----------------	---------------------------------

Returns

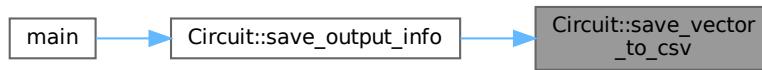
true if saving is successful, false otherwise

Definition at line 856 of file [CCircuit.cpp](#).

References [circuit_vector](#), and [units](#).

Referenced by [save_output_info\(\)](#).

Here is the caller graph for this function:

**term_mask()**

```
uint8_t Circuit::term_mask (
    int start ) const [private]
```

Get the terminal mask for a given unit.

This function calculates the terminal mask for a given unit. It uses breadth-first search to traverse the circuit and find the terminals.

Parameters

<i>start</i>	The starting unit number
--------------	--------------------------

Returns

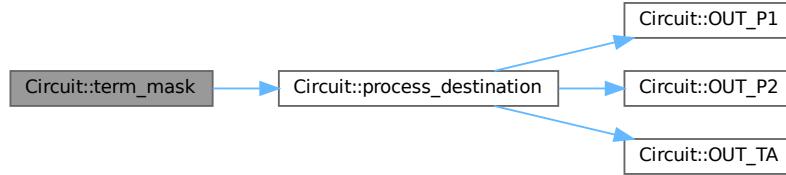
The terminal mask

Definition at line 745 of file [CCircuit.cpp](#).

References [n](#), [process_destination\(\)](#), and [units](#).

Referenced by [check_validity\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.4 Member Data Documentation

beta

double* `Circuit::beta` [private]

Definition at line 113 of file [CCircuit.h](#).

Referenced by [check_validity\(\)](#), [initialize_from_vector\(\)](#), and [initialize_from_vector\(\)](#).

circuit_vector

const int* `Circuit::circuit_vector` [private]

Definition at line 110 of file [CCircuit.h](#).

Referenced by [check_validity\(\)](#), [initialize_from_vector\(\)](#), [initialize_from_vector\(\)](#), [initialize_from_vector\(\)](#), [initialize_from_vector\(\)](#), and [save_vector_to_csv\(\)](#).

feed_dest

int `Circuit::feed_dest` = 0 [private]

Definition at line 159 of file [CCircuit.h](#).

Referenced by [check_validity\(\)](#).

feed_gormanium_rate

double `Circuit::feed_gormanium_rate` [private]

Definition at line 119 of file [CCircuit.h](#).

Referenced by [Circuit\(\)](#), [get_gormanium_recovery\(\)](#), and [run_mass_balance\(\)](#).

feed_palusznium_rate

double `Circuit::feed_palusznium_rate` [private]

Definition at line 118 of file [CCircuit.h](#).

Referenced by [Circuit\(\)](#), [get_palusznium_recovery\(\)](#), and [run_mass_balance\(\)](#).

feed_unit

int `Circuit::feed_unit` [private]

Definition at line 117 of file [CCircuit.h](#).

Referenced by [initialize_from_vector\(\)](#), and [run_mass_balance\(\)](#).

feed_waste_rate

double `Circuit::feed_waste_rate` [private]

Definition at line 120 of file [CCircuit.h](#).

Referenced by [Circuit\(\)](#), and [run_mass_balance\(\)](#).

gormanium_product_gormanium

```
double Circuit::gormanium_product_gormanium [private]
```

Definition at line 128 of file [CCircuit.h](#).

Referenced by [get_economic_value\(\)](#), [get_gormanium_grade\(\)](#), [get_gormanium_recovery\(\)](#), and [run_mass_balance\(\)](#).

gormanium_product_palusznium

```
double Circuit::gormanium_product_palusznium [private]
```

Definition at line 127 of file [CCircuit.h](#).

Referenced by [get_economic_value\(\)](#), [get_gormanium_grade\(\)](#), and [run_mass_balance\(\)](#).

gormanium_product_waste

```
double Circuit::gormanium_product_waste [private]
```

Definition at line 129 of file [CCircuit.h](#).

Referenced by [get_economic_value\(\)](#), [get_gormanium_grade\(\)](#), and [run_mass_balance\(\)](#).

gormanium_value

```
double Circuit::gormanium_value [private]
```

Definition at line 138 of file [CCircuit.h](#).

Referenced by [Circuit\(\)](#), and [get_economic_value\(\)](#).

gormanium_value_in_palusznium

```
double Circuit::gormanium_value_in_palusznium [private]
```

Definition at line 139 of file [CCircuit.h](#).

Referenced by [Circuit\(\)](#), and [get_economic_value\(\)](#).

n

```
int Circuit::n [private]
```

Definition at line 158 of file [CCircuit.h](#).

Referenced by [check_validity\(\)](#), [check_validity\(\)](#), [OUT_P1\(\)](#), [OUT_P2\(\)](#), [OUT_TA\(\)](#), [process_destination\(\)](#), and [term_mask\(\)](#).

palusznium_product_gormanium

```
double Circuit::palusznium_product_gormanium [private]
```

Definition at line 124 of file [CCircuit.h](#).

Referenced by [get_economic_value\(\)](#), [get_palusznium_grade\(\)](#), and [run_mass_balance\(\)](#).

palusznium_product_palusznium

```
double Circuit::palusznium_product_palusznium [private]
```

Definition at line 123 of file [CCircuit.h](#).

Referenced by [get_economic_value\(\)](#), [get_palusznium_grade\(\)](#), [get_palusznium_recovery\(\)](#), and [run_mass_balance\(\)](#).

palusznium_product_waste

```
double Circuit::palusznium_product_waste [private]
```

Definition at line 125 of file [CCircuit.h](#).

Referenced by [get_economic_value\(\)](#), [get_palusznium_grade\(\)](#), and [run_mass_balance\(\)](#).

palusznium_value

```
double Circuit::palusznium_value [private]
```

Definition at line 137 of file [CCircuit.h](#).

Referenced by [Circuit\(\)](#), and [get_economic_value\(\)](#).

palusznium_value_in_germanium

```
double Circuit::palusznium_value_in_germanium [private]
Definition at line 140 of file CCircuit.h.
Referenced by Circuit\(\), and get\_economic\_value\(\).
```

tailings_germanium

```
double Circuit::tailings_germanium [private]
Definition at line 132 of file CCircuit.h.
Referenced by run\_mass\_balance\(\).
```

tailings_palusznium

```
double Circuit::tailings_palusznium [private]
Definition at line 131 of file CCircuit.h.
Referenced by run\_mass\_balance\(\).
```

tailings_waste

```
double Circuit::tailings_waste [private]
Definition at line 133 of file CCircuit.h.
Referenced by run\_mass\_balance\(\).
```

units

```
std::vector<CUUnit> Circuit::units [private]
Definition at line 107 of file CCircuit.h.
Referenced by check\_validity\(\), export\_to\_dot\(\), get\_economic\_value\(\), initialize\_from\_vector\(\), mark\_units\(\), run\_mass\_balance\(\), save\_all\_units\_to\_csv\(\), save\_vector\_to\_csv\(\), and term\_mask\(\).
```

waste_penalty_germanium

```
double Circuit::waste_penalty_germanium [private]
Definition at line 142 of file CCircuit.h.
Referenced by Circuit\(\), and get\_economic\_value\(\).
```

waste_penalty_palusznium

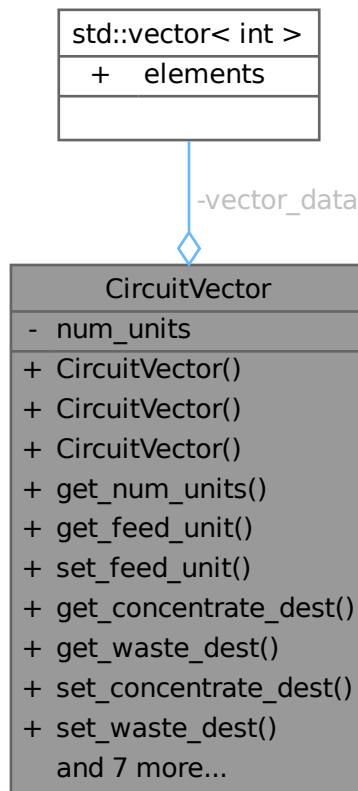
```
double Circuit::waste_penalty_palusznium [private]
Definition at line 141 of file CCircuit.h.
Referenced by Circuit\(\), and get\_economic\_value\(\).
The documentation for this class was generated from the following files:
```

- [include/CCircuit.h](#)
- [src/CCircuit.cpp](#)

6.3 CircuitVector Class Reference

```
#include <circuit_vector.h>
```

Collaboration diagram for CircuitVector:



Public Member Functions

- `CircuitVector ()=default`
- `CircuitVector (int num_units)`
- `CircuitVector (int vector_size, const int *data)`
- `int get_num_units () const`
- `int get_feed_unit () const`
- `void set_feed_unit (int unit)`
- `int get_concentrate_dest (int unit) const`
- `int get_waste_dest (int unit) const`
- `void set_concentrate_dest (int unit, int dest)`
- `void set_waste_dest (int unit, int dest)`
- `const std::vector< int > & get_data () const`
- `const int * data () const`
- `int size () const`
- `void randomize ()`
- `void print (std::ostream &os=std::cout) const`
- `bool save_to_file (const std::string &filename) const`
- `bool load_from_file (const std::string &filename)`

Private Attributes

- std::vector< int > `vector_data`
- int `num_units` = 0

6.3.1 Detailed Description

Definition at line 30 of file [circuit_vector.h](#).

6.3.2 Constructor & Destructor Documentation**CircuitVector() [1/3]**

```
CircuitVector::CircuitVector ( ) [default]
```

CircuitVector() [2/3]

```
CircuitVector::CircuitVector (
    int num_units )
```

CircuitVector() [3/3]

```
CircuitVector::CircuitVector (
    int vector_size,
    const int * data )
```

6.3.3 Member Function Documentation**data()**

```
const int * CircuitVector::data ( ) const
```

get_concentrate_dest()

```
int CircuitVector::get_concentrate_dest (
    int unit ) const
```

get_data()

```
const std::vector< int > & CircuitVector::get_data ( ) const
```

get_feed_unit()

```
int CircuitVector::get_feed_unit ( ) const
```

get_num_units()

```
int CircuitVector::get_num_units ( ) const
```

get_waste_dest()

```
int CircuitVector::get_waste_dest (
    int unit ) const
```

load_from_file()

```
bool CircuitVector::load_from_file (
    const std::string & filename )
```

```
print()

void CircuitVector::print (
    std::ostream & os = std::cout ) const

randomize()

void CircuitVector::randomize ( )

save_to_file()

bool CircuitVector::save_to_file (
    const std::string & filename ) const

set_concentrate_dest()

void CircuitVector::set_concentrate_dest (
    int unit,
    int dest )

set_feed_unit()

void CircuitVector::set_feed_unit (
    int unit )

set_waste_dest()

void CircuitVector::set_waste_dest (
    int unit,
    int dest )

size()

int CircuitVector::size ( ) const
```

6.3.4 Member Data Documentation

num_units

```
int CircuitVector::num_units = 0 [private]
Definition at line 86 of file circuit\_vector.h.
```

vector_data

```
std::vector<int> CircuitVector::vector_data [private]
Definition at line 85 of file circuit\_vector.h.
The documentation for this class was generated from the following file:
```

- [include/circuit_vector.h](#)

6.4 CUnit Class Reference

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

Collaboration diagram for CUnit:

CUnit
+ conc_num
+ tails_num
+ mark
+ volume
+ V_min
+ V_max
+ feed_palusznium
+ feed_gormanium
+ feed_waste
+ k_palusznium
and 13 more...
+ CUnit()
+ CUnit()
+ CUnit()
+ process()
+ update_volume()

Public Member Functions

- `CUnit ()`
Default constructor – initialises all numeric members to zero and routes to invalid destinations (e.g.
- `CUnit (int conc, int tails)`
Convenience constructor – sets outlet destinations; remaining parameters are pulled from `constants.h` defaults.
- `CUnit (int conc, int tails, bool testFlag)`
Perform unit calculation for the current feed.
- `void process ()`
Process the unit.
- `void update_volume (double beta)`
Check if the unit is valid.

Public Attributes

- `int conc_num`
Unit volume V (m^3) – default 10 m^3
- `int tails_num`
Minimum volume (m^3) – default 2.5 m^3
- `bool mark`
- `double volume`
Maximum volume (m^3) – default 20 m^3

- double `feed_palusznium`
- double `feed_gormanium`
- double `feed_waste`
- double `k_palusznium`
- double `k_gormanium`
- double `k_waste`
- double `conc_palusznium`
 $C_P \text{ (kg s}^{-1}\text{)}$
- double `conc_gormanium`
 $C_G \text{ (kg s}^{-1}\text{)}$
- double `conc_waste`
 $C_W \text{ (kg s}^{-1}\text{)}$
- double `tails_palusznium`
 $T_P \text{ (kg s}^{-1}\text{)}$
- double `tails_gormanium`
 $T_G \text{ (kg s}^{-1}\text{)}$
- double `tails_waste`
 $T_W \text{ (kg s}^{-1}\text{)}$
- double `rho`
- double `phi`
- double `Rp`
- double `Rg`
- double `Rw`

Recoveries for each component.

6.4.1 Detailed Description

Definition at line 31 of file [CUnit.h](#).

6.4.2 Constructor & Destructor Documentation

CUnit() [1/3]

`CUnit::CUnit ()`

Default constructor – initialises all numeric members to zero and routes to invalid destinations (e.g. Constructors for the [CUnit](#) class.

-1) until set by GA vector.

Definition at line 17 of file [CUnit.cpp](#).

CUnit() [2/3]

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

Convenience constructor – sets outlet destinations; remaining parameters are pulled from [constants.h](#) defaults.

Parameters

<code>conc</code>	Destination index for concentrate
<code>tails</code>	Destination index for tails

Definition at line 25 of file [CUnit.cpp](#).

CUnit() [3/3]

`CUnit::CUnit (`

```
    int conc,
    int tails,
    bool testFlag )
```

Perform unit calculation for the current feed.

Steps:

1. Compute residence time = $V / (F_i)$
2. Evaluate recoveries $R_i^C = k_i / (1 + k_i)$
3. Split feed into concentrate & tails streams
4. Store outlet flowrates in the public members above

No return value – results are written into `conc_*` and `tails_*`. Caller is responsible for ensuring `feed_*` are populated beforehand.

Definition at line 35 of file [CUnit.cpp](#).

References `Constants::Test::DEFAULT_UNIT_VOLUME`, `Constants::Test::K_GORMANIUM`, `k_gormanium`, `Constants::Test::K_PALUSZNIUM`, `k_paluszniun`, `Constants::Test::K_WASTE`, `k_waste`, `Constants::Test::MATERIAL_DENSITY`, `Constants::Test::MAX_UNIT_VOLUME`, `Constants::Test::MIN_UNIT_VOLUME`, `phi`, `rho`, `Constants::Test::SOLIDS_CONTENT`, `V_max`, `V_min`, and `volume`.

6.4.3 Member Function Documentation

process()

```
void CUnit::process ( )
```

Process the unit.

This function processes the unit by calculating the residence time, recoveries, and splitting the feed into products.

Definition at line 64 of file [CUnit.cpp](#).

References `conc_gormanium`, `conc_paluszniun`, `conc_waste`, `feed_gormanium`, `feed_paluszniun`, `feed_waste`, `k_gormanium`, `k_paluszniun`, `k_waste`, `phi`, `Rg`, `rho`, `Rp`, `Rw`, `tails_gormanium`, `tails_paluszniun`, `tails_waste`, and `volume`.

update_volume()

```
void CUnit::update_volume (
    double beta )
```

Check if the unit is valid.

Update the volume of the unit.

A unit is valid if:

1. It has a valid destination for both concentrate and tails
2. It has a non-zero volume
3. It has a non-zero k-value for at least one component

Returns

true if valid, false otherwise.

Update the volume of the unit.

Parameters

<code>beta</code>	The new volume of the unit.
-------------------	-----------------------------

This function updates the volume of the unit based on the given beta value.

Parameters

<code>beta</code>	The beta value to update the volume
-------------------	-------------------------------------

Definition at line 102 of file [CUnit.cpp](#).
References [V_max](#), [V_min](#), and [volume](#).

6.4.4 Member Data Documentation

conc_gormanium

```
double CUnit::conc_gormanium
C_G (kg s-1)
Definition at line 60 of file CUnit.h.
Referenced by process\(\), and Circuit::save\_all\_units\_to\_csv\(\).
```

conc_num

```
int CUnit::conc_num
Definition at line 35 of file CUnit.h.
```

conc_palusznium

```
double CUnit::conc_palusznium
C_P (kg s-1)
Definition at line 59 of file CUnit.h.
Referenced by process\(\), and Circuit::save\_all\_units\_to\_csv\(\).
```

conc_waste

```
double CUnit::conc_waste
C_W (kg s-1)
Definition at line 61 of file CUnit.h.
Referenced by process\(\), and Circuit::save\_all\_units\_to\_csv\(\).
```

feed_gormanium

```
double CUnit::feed_gormanium
Definition at line 49 of file CUnit.h.
Referenced by process\(\).
```

feed_palusznium

```
double CUnit::feed_palusznium
Definition at line 48 of file CUnit.h.
Referenced by process\(\).
```

feed_waste

```
double CUnit::feed_waste
Definition at line 50 of file CUnit.h.
Referenced by process\(\).
```

k_gormanium

```
double CUnit::k_gormanium
Definition at line 54 of file CUnit.h.
Referenced by CUnit\(\), and process\(\).
```

k_palusznium

```
double CUnit::k_palusznium
Definition at line 53 of file CUnit.h.
Referenced by CUnit\(\), and process\(\).
```

k_waste

```
double CUnit::k_waste
Definition at line 55 of file CUnit.h.
Referenced by CUnit(), and process().
```

mark

```
bool CUnit::mark
Definition at line 40 of file CUnit.h.
```

phi

```
double CUnit::phi
Definition at line 68 of file CUnit.h.
Referenced by CUnit(), and process().
```

Rg

```
double CUnit::Rg
Definition at line 71 of file CUnit.h.
Referenced by process().
```

rho

```
double CUnit::rho
Definition at line 67 of file CUnit.h.
Referenced by CUnit(), and process().
```

Rp

```
double CUnit::Rp
Definition at line 71 of file CUnit.h.
Referenced by process().
```

Rw

```
double CUnit::Rw
Recoveries for each component.
Definition at line 71 of file CUnit.h.
Referenced by process().
```

tails_gormanium

```
double CUnit::tails_gormanium
T_G (kg s-1)
Definition at line 64 of file CUnit.h.
Referenced by process(), and Circuit::save_all_units_to_csv().
```

tails_num

```
int CUnit::tails_num
Definition at line 37 of file CUnit.h.
```

tails_palusznium

```
double CUnit::tails_palusznium
T_P (kg s-1)
Definition at line 63 of file CUnit.h.
Referenced by process(), and Circuit::save_all_units_to_csv().
```

tails_waste

```
double CUnit::tails_waste
T_W (kg s-1)
Definition at line 65 of file CUnit.h.
Referenced by process(), and Circuit::save_all_units_to_csv().
```

V_max

```
double CUnit::V_max
Maximum volume (m3) – default 20 m3
Definition at line 45 of file CUnit.h.
Referenced by CUnit(), and update_volume().
```

V_min

```
double CUnit::V_min
Minimum volume (m3) – default 2.5 m3
Definition at line 44 of file CUnit.h.
Referenced by CUnit(), and update_volume().
```

volume

```
double CUnit::volume
Unit volume V (m3) – default 10 m3
Definition at line 43 of file CUnit.h.
Referenced by CUnit(), process(), and update_volume().
The documentation for this class was generated from the following files:
```

- include/CUnit.h
- src/CUnit.cpp

6.5 OptimizationResult Struct Reference

```
#include <Genetic_Algorithm.h>
Collaboration diagram for OptimizationResult:
```

OptimizationResult
+ best_fitness
+ generations
+ avg_fitness
+ std_fitness
+ time_taken
+ converged
+ OptimizationResult()

Public Member Functions

- OptimizationResult()

Public Attributes

- double `best_fitness`
- int `generations`
- double `avg_fitness`
- double `std_fitness`
- double `time_taken`
- bool `converged`

6.5.1 Detailed Description

Definition at line 91 of file [Genetic_Algorithm.h](#).

6.5.2 Constructor & Destructor Documentation

`OptimizationResult()`

```
OptimizationResult::OptimizationResult ( ) [inline]  
Definition at line 101 of file Genetic\_Algorithm.h.
```

6.5.3 Member Data Documentation

`avg_fitness`

```
double OptimizationResult::avg_fitness  
Definition at line 95 of file Genetic\_Algorithm.h.
```

`best_fitness`

```
double OptimizationResult::best_fitness  
Definition at line 93 of file Genetic\_Algorithm.h.  
Referenced by optimize\(\), and optimize\(\).
```

`converged`

```
bool OptimizationResult::converged  
Definition at line 98 of file Genetic\_Algorithm.h.
```

`generations`

```
int OptimizationResult::generations  
Definition at line 94 of file Genetic\_Algorithm.h.  
Referenced by optimize\(\), and optimize\(\).
```

`std_fitness`

```
double OptimizationResult::std_fitness  
Definition at line 96 of file Genetic\_Algorithm.h.
```

`time_taken`

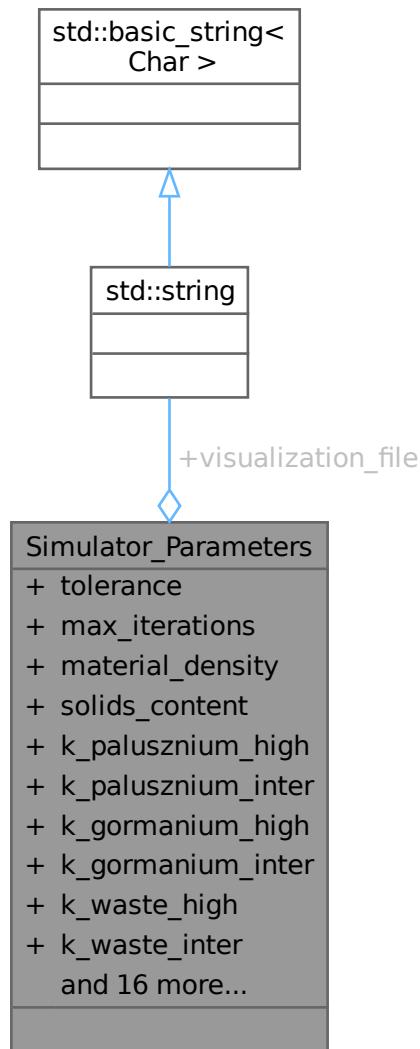
```
double OptimizationResult::time_taken  
Definition at line 97 of file Genetic\_Algorithm.h.  
The documentation for this struct was generated from the following file:
```

- [include/Genetic_Algorithm.h](#)

6.6 Simulator_Parameters Struct Reference

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

Collaboration diagram for Simulator_Parameters:



Public Attributes

- double `tolerance` = 1e-6
- int `max_iterations` = 1000
- double `material_density` = 3000.0
- double `solids_content` = 0.1
- double `k_palusznium_high` = 0.008
- double `k_palusznium_inter` = 0.004
- double `k_gormanium_high` = 0.004
- double `k_gormanium_inter` = 0.002
- double `k_waste_high` = 0.0005
- double `k_waste_inter` = 0.00025

- double `feed_palusznium` = 8.0
- double `feed_gormanium` = 12.0
- double `feed_waste` = 80.0
- double `palusznium_value_in_palusznium_stream` = 120.0
- double `gormanium_value_in_palusznium_stream` = -20.0
- double `waste_penalty_in_palusznium_stream` = -300.0
- double `palusznium_value_in_gormanium_stream` = 0.0
- double `gormanium_value_in_gormanium_stream` = 80.0
- double `waste_penalty_in_gormanium_stream` = -25.0
- double `fixed_unit_volume` = 10.0
- double `min_unit_volume` = 2.5
- double `max_unit_volume` = 20.0
- double `max_circuit_volume` = 150.0
- double `cost_coefficient` = 5.0
- double `volume_penalty_coefficient` = 1000.0
- bool `generate_visualization` = false
- std::string `visualization_file` = "circuit.dot"

6.6.1 Detailed Description

Definition at line 13 of file [CSimulator.h](#).

6.6.2 Member Data Documentation

`cost_coefficient`

double Simulator_Parameters::cost_coefficient = 5.0

Definition at line 52 of file [CSimulator.h](#).

`feed_gormanium`

double Simulator_Parameters::feed_gormanium = 12.0

Definition at line 33 of file [CSimulator.h](#).

`feed_palusznium`

double Simulator_Parameters::feed_palusznium = 8.0

Definition at line 32 of file [CSimulator.h](#).

`feed_waste`

double Simulator_Parameters::feed_waste = 80.0

Definition at line 34 of file [CSimulator.h](#).

`fixed_unit_volume`

double Simulator_Parameters::fixed_unit_volume = 10.0

Definition at line 46 of file [CSimulator.h](#).

`generate_visualization`

bool Simulator_Parameters::generate_visualization = false

Definition at line 56 of file [CSimulator.h](#).

`gormanium_value_in_gormanium_stream`

double Simulator_Parameters::gormanium_value_in_gormanium_stream = 80.0

Definition at line 42 of file [CSimulator.h](#).

gormanium_value_in_palusznium_stream

```
double Simulator_Parameters::gormanium_value_in_palusznium_stream = -20.0
Definition at line 38 of file CSimulator.h.
```

k_gormanium_high

```
double Simulator_Parameters::k_gormanium_high = 0.004
Definition at line 26 of file CSimulator.h.
```

k_gormanium_inter

```
double Simulator_Parameters::k_gormanium_inter = 0.002
Definition at line 27 of file CSimulator.h.
```

k_palusznium_high

```
double Simulator_Parameters::k_palusznium_high = 0.008
Definition at line 24 of file CSimulator.h.
```

k_palusznium_inter

```
double Simulator_Parameters::k_palusznium_inter = 0.004
Definition at line 25 of file CSimulator.h.
```

k_waste_high

```
double Simulator_Parameters::k_waste_high = 0.0005
Definition at line 28 of file CSimulator.h.
```

k_waste_inter

```
double Simulator_Parameters::k_waste_inter = 0.00025
Definition at line 29 of file CSimulator.h.
```

material_density

```
double Simulator_Parameters::material_density = 3000.0
Definition at line 20 of file CSimulator.h.
```

max_circuit_volume

```
double Simulator_Parameters::max_circuit_volume = 150.0
Definition at line 49 of file CSimulator.h.
```

max_iterations

```
int Simulator_Parameters::max_iterations = 1000
Definition at line 17 of file CSimulator.h.
Referenced by circuit_performance().
```

max_unit_volume

```
double Simulator_Parameters::max_unit_volume = 20.0
Definition at line 48 of file CSimulator.h.
```

min_unit_volume

```
double Simulator_Parameters::min_unit_volume = 2.5
Definition at line 47 of file CSimulator.h.
```

palusznium_value_in_gormanium_stream

```
double Simulator_Parameters::palusznium_value_in_gormanium_stream = 0.0
Definition at line 41 of file CSimulator.h.
```

palusznium_value_in_palusznium_stream

```
double Simulator_Parameters::palusznium_value_in_palusznium_stream = 120.0
Definition at line 37 of file CSimulator.h.
```

solids_content

```
double Simulator_Parameters::solids_content = 0.1
Definition at line 21 of file CSimulator.h.
```

tolerance

```
double Simulator_Parameters::tolerance = 1e-6
Definition at line 16 of file CSimulator.h.
Referenced by circuit_performance().
```

visualization_file

```
std::string Simulator_Parameters::visualization_file = "circuit.dot"
Definition at line 57 of file CSimulator.h.
```

volume_penalty_coefficient

```
double Simulator_Parameters::volume_penalty_coefficient = 1000.0
Definition at line 53 of file CSimulator.h.
```

waste_penalty_in_gormanium_stream

```
double Simulator_Parameters::waste_penalty_in_gormanium_stream = -25.0
Definition at line 43 of file CSimulator.h.
```

waste_penalty_in_palusznium_stream

```
double Simulator_Parameters::waste_penalty_in_palusznium_stream = -300.0
Definition at line 39 of file CSimulator.h.
```

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

- include/CSimulator.h

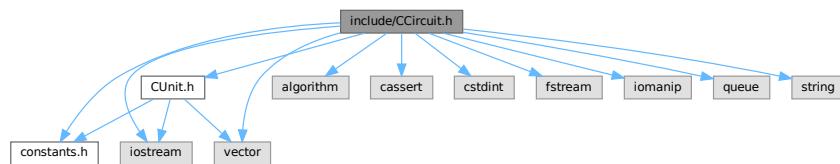
7 File Documentation

7.1 include/CCircuit.h File Reference

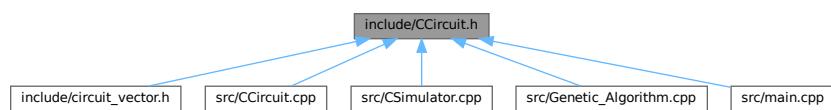
Declares the [Circuit](#) class – a mineral-processing circuit.

```
#include "CUnit.h"
#include "constants.h"
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <queue>
#include <string>
```

```
#include <vector>
Include dependency graph for CCircuit.h:
```



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



Classes

- class [Circuit](#)

Enumerations

- enum [CircuitDestination](#) { [PALUSZNIUM_PRODUCT](#) = -1 , [GORMANIUM_PRODUCT](#) = -2 , [TAILINGS_OUTPUT](#) = -3 }

7.1.1 Detailed Description

Declares the [Circuit](#) class – a mineral-processing circuit.

A circuit consists of a number of separation units ([CUnit](#)) connected together. Each unit receives a mixed feed stream and produces two output streams: a concentrate and a tailings stream.

The class stores: • The array of units in the circuit • The feed unit number and feed rates • The final product stream flow rates • Economic parameters

Definition in file [CCircuit.h](#).

7.1.2 Enumeration Type Documentation

[CircuitDestination](#)

enum [CircuitDestination](#)

Enumerator

PALUSZNIUM_PRODUCT	
GORMANIUM_PRODUCT	
TAILINGS_OUTPUT	

Definition at line 33 of file [CCircuit.h](#).

7.2 CCircuit.h

[Go to the documentation of this file.](#)

00001 */***

```

00002 * @file CCircuit.h
00003 * @brief Declares the Circuit class - a mineral-processing circuit
00004 *
00005 * A circuit consists of a number of separation units (CUnit) connected
00006 * together. Each unit receives a mixed feed stream and produces two
00007 * output streams: a concentrate and a tailings stream.
00008 *
00009 * The class stores:
00010 *   - The array of units in the circuit
00011 *   - The feed unit number and feed rates
00012 *   - The final product stream flow rates
00013 *   - Economic parameters
00014 *
00015 */
00016
00017 #pragma once
00018 #include "CUnit.h"
00019 #include "constants.h"
00020 #include <algorithm>
00021 #include <cassert>
00022 #include <cstdint>
00023 #include <fstream>
00024 #include <iomanip>
00025 #include <iostream>
00026 #include <queue>
00027 #include <string>
00028 #include <vector>
00029
00030 // Constants for the circuit outlet destinations
00031 // These values will be used in the circuit vector to indicate the final product
00032 // streams
00033 enum CircuitDestination
00034 {
00035     PALUSZNIA_PRODUCT = -1, // Final Palusznium concentrate product
00036     GORMANIUM_PRODUCT = -2, // Final Gormanium concentrate product
00037     TAILINGS_OUTPUT = -3    // Final tailings output
00038 };
00039
00040 /* ----- */
00041 /*          Circuit class           */
00042 /* ----- */
00043
00044 class Circuit
00045 {
00046 public:
00047     // Constructor that takes the number of units in the circuit
00048     Circuit(int num_units);
00049
00050     // Constructor with beta values for unit volumes
00051     Circuit(int num_units, double* beta);
00052
00053     // Test constructor with beta values and test flag
00054     Circuit(int num_units, double* beta, bool testFlag);
00055
00056     // Initialize the circuit from a circuit vector
00057     bool initialize_from_vector(int vector_size, const int* circuit_vector);
00058     bool initialize_from_vector(int vector_size, const int* circuit_vector, const double* beta);
00059
00060     bool initialize_from_vector(int vector_size, const int* circuit_vector, bool testFlag);
00061     bool initialize_from_vector(int vector_size, const int* circuit_vector, const double* beta, bool
00062     testFlag);
00063
00064     // Check validity of a circuit vector
00065     bool check_validity(int vector_size, const int* circuit_vector);
00066     bool check_validity(int vector_size, const int* circuit_vector, int unit_parameters_size, double*
00067     unit_parameters);
00068
00069     // Run a mass balance calculation on the circuit
00070     bool run_mass_balance(double tolerance = 1e-6, int max_iterations = 1000);
00071
00072     // Get the economic value of the circuit
00073     double get_economic_value() const;
00074
00075     // Get the recovery of valuable materials
00076     double get_palusznium_recovery() const;
00077     double get_gormanium_recovery() const;
00078
00079     // Get the grade of valuable materials in products
00080     double get_palusznium_grade() const;
00081     double get_gormanium_grade() const;
00082
00083     // Export the circuit to a dot file for visualization
00084     bool export_to_dot(const std::string& filename) const;
00085
00086 /**
00087 * @brief Save all units to a CSV file.
00088 * @param filename The name of the output CSV file.

```

```

00087     * @return true if save is successful, false otherwise
00088     */
00089     bool save_all_units_to_csv(const std::string& filename);
00090
00091 /**
00092     * @brief Save a vector to a CSV file.
00093     * @param filename The name of the output CSV file.
00094     * @return true if save is successful, false otherwise
00095     */
00096     bool save_vector_to_csv(const std::string& filename);
00097
00098 /**
00099     * @brief Save the circuit data to a CSV file.
00100     * @param filename The name of the output CSV file.
00101     * @return true if save is successful, false otherwise
00102     */
00103     bool save_output_info(const std::string& filename);
00104
00105 private:
00106     // The array of units in the circuit
00107     std::vector<CUnit> units;
00108
00109     // Circuit vector (for output)
00110     const int* circuit_vector;
00111
00112     /* ----- volume information ----- */
00113     double* beta; // Array of beta values for unit volumes
00114
00115     /* ----- flow information ----- */
00116     // Feed unit number and feed rates
00117     int feed_unit;
00118     double feed_palusznium_rate; // kg/s
00119     double feed_germanium_rate; // kg/s
00120     double feed_waste_rate; // kg/s
00121
00122     // Final product stream flow rates
00123     double palusznium_product_palusznium; // kg/s
00124     double palusznium_product_germanium; // kg/s
00125     double palusznium_product_waste; // kg/s
00126
00127     double germanium_product_palusznium; // kg/s
00128     double germanium_product_germanium; // kg/s
00129     double germanium_product_waste; // kg/s
00130
00131     double tailings_palusznium; // kg/s
00132     double tailings_germanium; // kg/s
00133     double tailings_waste; // kg/s
00134
00135     /* ----- economic parameters ----- */
00136     // Economic parameters
00137     double palusznium_value; // £/kg in Palusznium stream
00138     double germanium_value; // £/kg in Germanium stream
00139     double germanium_value_in_palusznium; // £/kg in Palusznium stream
00140     double palusznium_value_in_germanium; // £/kg in Germanium stream
00141     double waste_penalty_palusznium; // £/kg waste in Palusznium stream
00142     double waste_penalty_germanium; // £/kg waste in Germanium stream
00143
00144     // Mark units that are accessible from a given unit (for validity checking)
00145     void mark_units(int unit_num);
00146
00147     // Check if all units are accessible from the feed
00148     bool check_all_units_accessible() const;
00149
00150     // Check if all units have routes to at least two output streams
00151     bool check_routes_to_outputs() const;
00152
00153     // Check for self-recycle and other invalid configurations
00154     bool check_no_self_recycle() const;
00155     bool check_not_all_same_destination() const;
00156
00157     /* ----- Circuit validity checking ----- */
00158     int n;
00159     int feed_dest = 0;
00160     uint8_t outlet_mask(int unit_idx, std::vector<int8_t>& cache) const;
00161     uint8_t term_mask(int start) const;
00162     void process_destination(int dest, uint8_t& mask, std::vector<bool>& visited, std::queue<int>& q)
00163     const;
00164     inline int OUT_P1() const
00165     {
00166         return n;
00167     } // palusznium
00168     inline int OUT_P2() const
00169     {
00170         return n + 1;
00171     } // germanium
00172     inline int OUT_TA() const

```

```

00173     {
00174         return n + 2;
00175     } // tails
00176 };

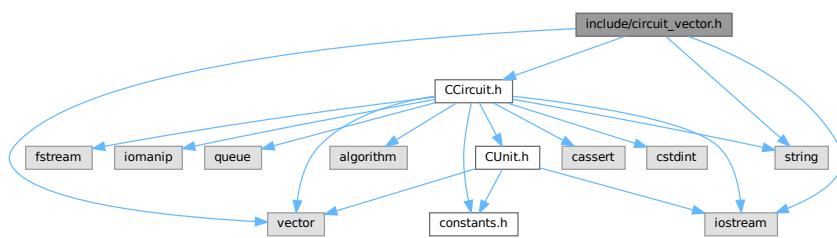
```

7.3 include/circuit_vector.h File Reference

[Circuit](#) Vector Header.

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

Include dependency graph for circuit_vector.h:



Classes

- class [CircuitVector](#)

7.3.1 Detailed Description

[Circuit](#) Vector Header.

This header defines the format and operations related to circuit vectors, which describe the connections in a mineral processing circuit.

A circuit vector has the following format: [feed_unit, unit0_conc, unit0_waste, unit1_conc, unit1_waste, ...]
where:

- feed_unit: Index of the unit receiving the circuit feed (0 to num_units-1)
- unitX_conc: Destination of the concentrate stream from unit X
- unitX_waste: Destination of the waste stream from unit X

Destination can be:

- 0 to (num_units-1): Index of the unit receiving the stream
- PALUSZNIUM_PRODUCT (-1): Final Palusznium concentrate product
- GORMANIUM_PRODUCT (-2): Final Gormanium concentrate product
- TAILINGS_OUTPUT (-3): Final tailings output

Definition in file [circuit_vector.h](#).

7.4 circuit_vector.h

[Go to the documentation of this file.](#)

```

00001 /**
00002 * @file circuit_vector.h
00003 * @brief Circuit Vector Header
00004 *
00005 * This header defines the format and operations related to circuit vectors,
00006 * which describe the connections in a mineral processing circuit.

```

```

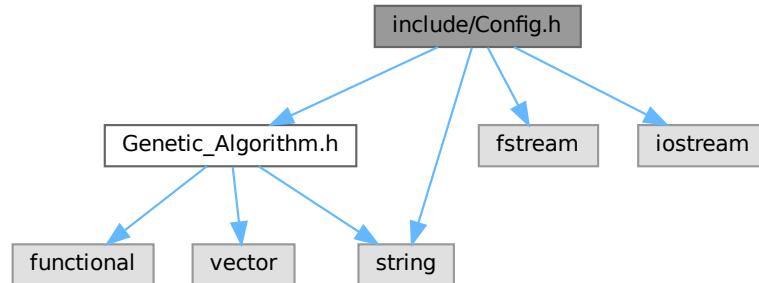
00007  *
00008  * A circuit vector has the following format:
00009  * [feed_unit, unit0_conc, unit0_waste, unit1_conc, unit1_waste, ...]
00010  *
00011  * where:
00012  * - feed_unit: Index of the unit receiving the circuit feed (0 to num_units-1)
00013  * - unitX_conc: Destination of the concentrate stream from unit X
00014  * - unitX_waste: Destination of the waste stream from unit X
00015  *
00016  * Destination can be:
00017  * - 0 to (num_units-1): Index of the unit receiving the stream
00018  * - PALUSZNIUM_PRODUCT (-1): Final Palusznium concentrate product
00019  * - GORMANIUM_PRODUCT (-2): Final Gormanium concentrate product
00020  * - TAILINGS_OUTPUT (-3): Final tailings output
00021 */
00022
00023 #pragma once
00024
00025 #include "CCircuit.h" // Include for CircuitDestination enum
00026 #include <iostream>
00027 #include <string>
00028 #include <vector>
00029
00030 class CircuitVector
00031 {
00032 public:
00033     // Constructor for empty circuit vector
00034     CircuitVector() = default;
00035
00036     // Constructor for circuit vector with specified number of units
00037     CircuitVector(int num_units);
00038
00039     // Constructor from existing circuit vector data
00040     CircuitVector(int vector_size, const int* data);
00041
00042     // Get number of units in the circuit
00043     int get_num_units() const;
00044
00045     // Get feed unit
00046     int get_feed_unit() const;
00047
00048     // Set feed unit
00049     void set_feed_unit(int unit);
00050
00051     // Get concentrate destination for unit
00052     int get_concentrate_dest(int unit) const;
00053
00054     // Get waste destination for unit
00055     int get_waste_dest(int unit) const;
00056
00057     // Set concentrate destination for unit
00058     void set_concentrate_dest(int unit, int dest);
00059
00060     // Set waste destination for unit
00061     void set_waste_dest(int unit, int dest);
00062
00063     // Get the raw vector data
00064     const std::vector<int>& get_data() const;
00065
00066     // Convert to raw array (for compatibility with existing functions)
00067     const int* data() const;
00068
00069     // Get the size of the vector
00070     int size() const;
00071
00072     // Randomize the circuit vector with valid values
00073     void randomize();
00074
00075     // Print the circuit vector
00076     void print(std::ostream& os = std::cout) const;
00077
00078     // Save to file
00079     bool save_to_file(const std::string& filename) const;
00080
00081     // Load from file
00082     bool load_from_file(const std::string& filename);
00083
00084 private:
00085     std::vector<int> vector_data;
00086     int num_units = 0;
00087 };

```

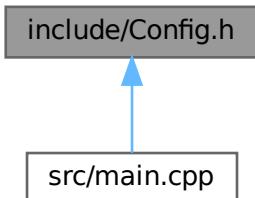
7.5 include/Config.h File Reference

Configuration file for the Genetic Algorithm.

```
#include "Genetic_Algorithm.h"
#include <fstream>
#include <iostream>
#include <string>
Include dependency graph for Config.h:
```



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



Functions

- void `load_parameters` (const std::string &file, `Algorithm_Parameters` &p)

7.5.1 Detailed Description

Configuration file for the Genetic Algorithm.

This file contains the definition of the `Algorithm_Parameters` structure and the function to load parameters from a configuration file.

The `Algorithm_Parameters` structure holds various parameters for the genetic algorithm.

Definition in file `Config.h`.

7.5.2 Function Documentation

`load_parameters()`

```
void load_parameters (
    const std::string & file,
    Algorithm_Parameters & p ) [inline]
```

Definition at line 20 of file `Config.h`.

References `Algorithm_Parameters::allow_mutation_wrapping`, `Algorithm_Parameters::convergence_threshold`, `Algorithm_Parameters::crossover_points`, `Algorithm_Parameters::crossover_probability`, `Algorithm_Parameters::elite_count`, `Algorithm_Parameters::inversion_probability`, `Algorithm_Parameters::log_file`, `Algorithm_Parameters::log_results`, `Algorithm_Parameters::max_iterations`, `Algorithm_Parameters::mode`, `Algorithm_Parameters::mutation_probability`, `Algorithm_Parameters::mutation_step_size`, `Algorithm_Parameters::num_units`, `Algorithm_Parameters::population_size`, `Algorithm_Parameters::random_seed`, `Algorithm_Parameters::scaling_mutation_max`, `Algorithm_Parameters::scaling_mutation_min`, `Algorithm_Parameters::scaling_mutation_prob`, `Algorithm_Parameters::selection_pressure`, `Algorithm_Parameters::stall_generations`, `Algorithm_Parameters::tournament_size`, `Algorithm_Parameters::use_inversion`, `Algorithm_Parameters::use_scaling_mutation`, and `Algorithm_Parameters::verbose`.

Referenced by `main()`.

Here is the caller graph for this function:



7.6 Config.h

[Go to the documentation of this file.](#)

```

00001 /**
00002 * @file Config.h
00003 * @brief Configuration file for the Genetic Algorithm
00004 *
00005 * This file contains the definition of the Algorithm_Parameters structure
00006 * and the function to load parameters from a configuration file.
00007 *
00008 * The Algorithm_Parameters structure holds various parameters for the
00009 * genetic algorithm.
00010 *
00011 */
00012 #pragma once
00013
00014 #include "Genetic_Algorithm.h"
00015 #include <fstream>
00016 #include <iostream>
00017 #include <string>
00018
00019 // Load GA parameters from a simple key=value text file
00020 inline void load_parameters(const std::string& file, Algorithm_Parameters& p)
00021 {
00022     std::ifstream in(file);
00023     if (!in)
00024     {
00025         std::cerr << "Warning: could not open " << file << " -- using default parameters.\n";
00026         return;
00027     }
00028     // Read the file line by line
00029     std::string line;
00030     auto trim = [&](std::string& s)
00031     {
00032         const char* ws = "\t\n\r";
00033         size_t start = s.find_first_not_of(ws);
00034         size_t end = s.find_last_not_of(ws);
00035         s = (start == std::string::npos) ? "" : s.substr(start, end - start + 1);
00036     };
00037     while (std::getline(in, line))
00038     {
00039         // remove comments
00040         if (auto pos = line.find('#'); pos != std::string::npos)
00041             line.resize(pos);
00042         trim(line);
00043         if (line.empty())
00044             continue;
00045         // split key and value
00046         auto pos = line.find('=');
00047         if (pos == std::string::npos)
00048             continue;
00049         std::string key = line.substr(0, pos);
00050         std::string val = line.substr(pos + 1);
  
```

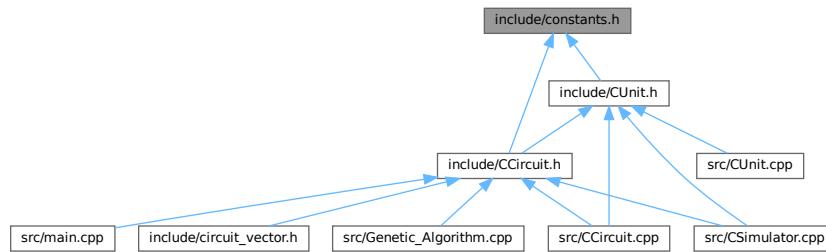
```

00051     trim(key);
00052     trim(val);
00053     try
00054     {
00055         if (key == "random_seed") // Seed for the random number generator
00056             p.random_seed = std::stoi(val);
00057         else if (key == "num_units") // Number of units in the circuit
00058             p.num_units = std::stoi(val);
00059         else if (key == "mode") // Optimization mode: discrete, continuous, or hybrid
00060             p.mode = val; // d, c, or h
00061         else if (key == "max_iterations") // Maximum number of generations
00062             p.max_iterations = std::stoi(val);
00063         else if (key == "population_size") // Number of individuals in the population
00064             p.population_size = std::stoi(val);
00065         else if (key == "elite_count") // Number of best individuals to keep unchanged
00066             p.elite_count = std::stoi(val);
00067         else if (key == "tournament_size") // Number of contenders per tournament
00068             p.tournament_size = std::stoi(val);
00069         else if (key == "selection_pressure") // Linear rank selection pressure parameter
00070             p.selection_pressure = std::stod(val);
00071         else if (key == "crossover_probability") // Probability of crossover
00072             p.crossover_probability = std::stod(val);
00073         else if (key == "crossover_points") // Number of crossover points (1 or 2)
00074             p.crossover_points = std::stoi(val);
00075         else if (key == "mutation_probability") // Probability of mutation per gene
00076             p.mutation_probability = std::stod(val);
00077         else if (key == "mutation_step_size") // Maximum change in value during mutation
00078             p.mutation_step_size = std::stoi(val);
00079         else if (key == "allow_mutation_wrapping") // Allow mutations to wrap around
00080             p.allow_mutation_wrapping = (val == "true" || val == "1");
00081         else if (key == "use_inversion") // Use inversion mutation
00082             p.use_inversion = (val == "true" || val == "1");
00083         else if (key == "inversion_probability") // Probability of inversion mutation
00084             p.inversion_probability = std::stod(val);
00085         else if (key == "use_scaling_mutation") // Use scaling mutation
00086             p.use_scaling_mutation = (val == "true" || val == "1");
00087         else if (key == "scaling_mutation_prob") // Probability of scaling mutation
00088             p.scaling_mutation_prob = std::stod(val);
00089         else if (key == "scaling_mutation_min") // Minimum scaling factor
00090             p.scaling_mutation_min = std::stod(val);
00091         else if (key == "scaling_mutation_max") // Maximum scaling factor
00092             p.scaling_mutation_max = std::stod(val);
00093         else if (key == "convergence_threshold") // Convergence threshold
00094             p.convergence_threshold = std::stod(val);
00095         else if (key == "stall_generations") // Max generations with no improvement
00096             p.stall_generations = std::stoi(val);
00097         else if (key == "verbose") // Print progress information
00098             p.verbose = (val == "true" || val == "1");
00099         else if (key == "log_results") // Log results to file
00100             p.log_results = (val == "true" || val == "1");
00101         else if (key == "log_file") // Log file name
00102             p.log_file = val;
00103     }
00104     else
00105     {
00106         std::cerr << "Warning: unknown parameter '" << key << "' in " << file << "\n";
00107     }
00108     catch (...)
00109     {
00110         std::cerr << "Warning: could not parse '" << key << "=" << val << "'\n";
00111     }
00112 }
00113 }
```

7.7 include/constants.h File Reference

Header file for project-wide constants.

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



Namespaces

- namespace [Constants](#)
- namespace [Constants::Test](#)
- namespace [Constants::Physical](#)
- namespace [Constants::Economic](#)
- namespace [Constants::Feed](#)
- namespace [Constants::Circuit](#)
- namespace [Constants::Simulation](#)
- namespace [Constants::GA](#)

Variables

- `constexpr double Constants::Test::DEFAULT_PALUSZNIUM_FEED = 10.0`
- `constexpr double Constants::Test::DEFAULT_GORMANIUM_FEED = 10.0`
- `constexpr double Constants::Test::DEFAULT_WASTE_FEED = 10.0`
- `constexpr double Constants::Test::PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM = 100.0`
- `constexpr double Constants::Test::GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM = 0.0`
- `constexpr double Constants::Test::WASTE_PENALTY_IN_PALUSZNIUM_STREAM = 0.0`
- `constexpr double Constants::Test::PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM = 0.0`
- `constexpr double Constants::Test::GORMANIUM_VALUE_IN_GORMANIUM_STREAM = 100.0`
- `constexpr double Constants::Test::WASTE_PENALTY_IN_GORMANIUM_STREAM = 0.0`
- `constexpr double Constants::Test::COST_COEFFICIENT = 5.0`
- `constexpr double Constants::Test::VOLUME_PENALTY_COEFFICIENT = 1000.0`
- `constexpr double Constants::Test::MATERIAL_DENSITY = 3000.0`
- `constexpr double Constants::Test::SOLIDS_CONTENT = 0.1`
- `constexpr double Constants::Test::K_PALUSZNIUM = 0.008`
- `constexpr double Constants::Test::K_GORMANIUM = 0.004`
- `constexpr double Constants::Test::K_WASTE = 0.0005`
- `constexpr double Constants::Test::DEFAULT_UNIT_VOLUME = 5.0`
- `constexpr double Constants::Test::MIN_UNIT_VOLUME = 2.5`
- `constexpr double Constants::Test::MAX_UNIT_VOLUME = 20.0`
- `constexpr double Constants::Test::MAX_CIRCUIT_VOLUME = 150.0`
- `constexpr int Constants::Test::DEFAULT_NUM_UNITS = 10`
- `constexpr double Constants::Physical::MATERIAL_DENSITY = 3000.0`
- `constexpr double Constants::Physical::SOLIDS_CONTENT = 0.1`
- `constexpr double Constants::Physical::K_PALUSZNIUM = 0.008`
- `constexpr double Constants::Physical::K_GORMANIUM = 0.004`
- `constexpr double Constants::Physical::K_WASTE = 0.0005`
- `constexpr double Constants::Economic::PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM = 120.0`
- `constexpr double Constants::Economic::GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM = -20.0`

- `constexpr double Constants::Economic::WASTE_PENALTY_IN_PALUSZNIA_STREAM = -300.0`
- `constexpr double Constants::Economic::PALUSZNIA_VALUE_IN_GORMANIUM_STREAM = 0.0`
- `constexpr double Constants::Economic::GORMANIUM_VALUE_IN_GORMANIUM_STREAM = 80.0`
- `constexpr double Constants::Economic::WASTE_PENALTY_IN_GORMANIUM_STREAM = -25.0`
- `constexpr double Constants::Economic::COST_COEFFICIENT = 5.0`
- `constexpr double Constants::Economic::VOLUME_PENALTY_COEFFICIENT = 1000.0`
- `constexpr double Constants::Feed::DEFAULT_PALUSZNIA_FEED = 8.0`
- `constexpr double Constants::Feed::DEFAULT_GORMANIUM_FEED = 12.0`
- `constexpr double Constants::Feed::DEFAULT_WASTE_FEED = 80.0`
- `constexpr double Constants::Circuit::DEFAULT_UNIT_VOLUME = 10.0`
- `constexpr double Constants::Circuit::MIN_UNIT_VOLUME = 2.5`
- `constexpr double Constants::Circuit::MAX_UNIT_VOLUME = 20.0`
- `constexpr double Constants::Circuit::MAX_CIRCUIT_VOLUME = 150.0`
- `constexpr int Constants::Circuit::DEFAULT_NUM_UNITS = 10`
- `constexpr double Constants::Simulation::DEFAULT_TOLERANCE = 1e-6`
- `constexpr int Constants::Simulation::DEFAULT_MAX_ITERATIONS = 1000`
- `constexpr double Constants::Simulation::MIN_FLOW_RATE = 1e-6`
- `constexpr int Constants::GA::DEFAULT_POPULATION_SIZE = 100`
- `constexpr int Constants::GA::DEFAULT_MAX_GENERATIONS = 1000`
- `constexpr double Constants::GA::DEFAULT_CROSSOVER_RATE = 0.8`
- `constexpr double Constants::GA::DEFAULT_MUTATION_RATE = 0.01`
- `constexpr int Constants::GA::DEFAULT_ELITE_COUNT = 1`

7.7.1 Detailed Description

Header file for project-wide constants.

This file defines physical constants, economic parameters, and other global values used throughout the project.

Definition in file `constants.h`.

7.8 constants.h

[Go to the documentation of this file.](#)

```
00001 /**
00002 * @file constants.h
00003 * @brief Header file for project-wide constants
00004 *
00005 * This file defines physical constants, economic parameters,
00006 * and other global values used throughout the project.
00007 */
00008
00009 #pragma once
00010
00011 namespace Constants
00012 {
00013 namespace Test
00014 {
00015 // Feed rates (kg/s)
00016 constexpr double DEFAULT_PALUSZNIA_FEED = 10.0;
00017 constexpr double DEFAULT_GORMANIUM_FEED = 10.0;
00018 constexpr double DEFAULT_WASTE_FEED = 10.0;
00019
00020 // Values in £/kg
00021 constexpr double PALUSZNIA_VALUE_IN_PALUSZNIA_STREAM = 100.0;
00022 constexpr double GORMANIUM_VALUE_IN_PALUSZNIA_STREAM = 0.0;
00023 constexpr double WASTE_PENALTY_IN_PALUSZNIA_STREAM = 0.0;
00024
00025 constexpr double PALUSZNIA_VALUE_IN_GORMANIUM_STREAM = 0.0;
00026 constexpr double GORMANIUM_VALUE_IN_GORMANIUM_STREAM = 100.0;
00027 constexpr double WASTE_PENALTY_IN_GORMANIUM_STREAM = 0.0;
00028
00029 // Operating cost parameters
00030 constexpr double COST_COEFFICIENT = 5.0;
00031 constexpr double VOLUME_PENALTY_COEFFICIENT = 1000.0;
00032
00033 // Material properties
00034 constexpr double MATERIAL_DENSITY = 3000.0; // kg/m3, density of all solid materials
00035 constexpr double SOLIDS_CONTENT = 0.1; // Fraction of solids by volume
00036
00037 // Rate constants (s-1)
00038 constexpr double K_PALUSZNIA = 0.008; // Rate constant for Palusznium
```

```

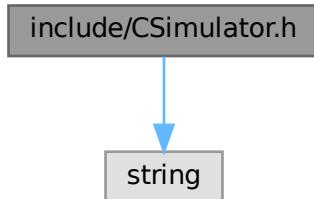
00039 constexpr double K_GORMANIUM = 0.004; // Rate constant for Gormanium
00040 constexpr double K_WASTE = 0.0005; // Rate constant for Waste
00041
00042 constexpr double DEFAULT_UNIT_VOLUME = 5.0; // m3
00043 constexpr double MIN_UNIT_VOLUME = 2.5; // m3 (for variable case)
00044 constexpr double MAX_UNIT_VOLUME = 20.0; // m3 (for variable case)
00045 constexpr double MAX_CIRCUIT_VOLUME = 150.0; // m3
00046
00047 constexpr int DEFAULT_NUM_UNITS = 10; // Default number of units in circuit
00048
00049 } // namespace Test
00050
00051 // Physical constants
00052 namespace Physical
00053 {
00054 // Material properties
00055 constexpr double MATERIAL_DENSITY = 3000.0; // kg/m3, density of all solid materials
00056 constexpr double SOLIDS_CONTENT = 0.1; // Fraction of solids by volume
00057
00058 // Rate constants (s-1)
00059 constexpr double K_PALUSZNIAU = 0.008; // Rate constant for Palusznium
00060 constexpr double K_GORMANIUM = 0.004; // Rate constant for Gormanium
00061 constexpr double K_WASTE = 0.0005; // Rate constant for Waste
00062 } // namespace Physical
00063
00064 // Economic parameters
00065 namespace Economic
00066 {
00067 // Values in £/kg
00068 constexpr double PALUSZNIAU_VALUE_IN_PALUSZNIAU_STREAM = 120.0;
00069 constexpr double GORMANIUM_VALUE_IN_PALUSZNIAU_STREAM = -20.0;
00070 constexpr double WASTE_PENALTY_IN_PALUSZNIAU_STREAM = -300.0;
00071
00072 constexpr double PALUSZNIAU_VALUE_IN_GORMANIUM_STREAM = 0.0;
00073 constexpr double GORMANIUM_VALUE_IN_GORMANIUM_STREAM = 80.0;
00074 constexpr double WASTE_PENALTY_IN_GORMANIUM_STREAM = -25.0;
00075
00076 // Operating cost parameters
00077 constexpr double COST_COEFFICIENT = 5.0;
00078 constexpr double VOLUME_PENALTY_COEFFICIENT = 1000.0;
00079 } // namespace Economic
00080
00081 // Default feed rates (kg/s)
00082 namespace Feed
00083 {
00084 constexpr double DEFAULT_PALUSZNIAU_FEED = 8.0;
00085 constexpr double DEFAULT_GORMANIUM_FEED = 12.0;
00086 constexpr double DEFAULT_WASTE_FEED = 80.0;
00087 } // namespace Feed
00088
00089 // Circuit parameters
00090 namespace Circuit
00091 {
00092 constexpr double DEFAULT_UNIT_VOLUME = 10.0; // m3
00093 constexpr double MIN_UNIT_VOLUME = 2.5; // m3 (for variable case)
00094 constexpr double MAX_UNIT_VOLUME = 20.0; // m3 (for variable case)
00095 constexpr double MAX_CIRCUIT_VOLUME = 150.0; // m3
00096
00097 constexpr int DEFAULT_NUM_UNITS = 10; // Default number of units in circuit
00098 } // namespace Circuit
00099
00100 // Simulation parameters
00101 namespace Simulation
00102 {
00103 constexpr double DEFAULT_TOLERANCE = 1e-6;
00104 constexpr int DEFAULT_MAX_ITERATIONS = 1000;
00105 constexpr double MIN_FLOW_RATE = 1e-6; // Minimum flow rate to prevent numerical issues
00106 } // namespace Simulation
00107
00108 // Genetic algorithm parameters
00109 namespace GA
00110 {
00111 constexpr int DEFAULT_POPULATION_SIZE = 100;
00112 constexpr int DEFAULT_MAX_GENERATIONS = 1000;
00113 constexpr double DEFAULT_CROSSOVER_RATE = 0.8;
00114 constexpr double DEFAULT_MUTATION_RATE = 0.01;
00115 constexpr int DEFAULT_ELITE_COUNT = 1;
00116 } // namespace GA
00117 } // namespace Constants

```

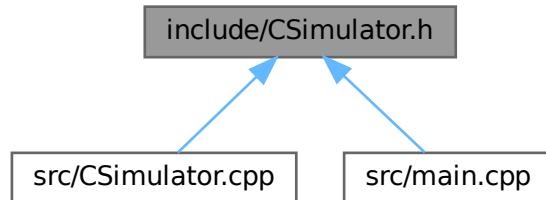
7.9 include/CSimulator.h File Reference

C++ header file for the circuit simulator.

```
#include <string>
Include dependency graph for CSimulator.h:
```



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



Classes

- struct `Simulator_Parameters`

Functions

- double `circuit_performance` (int vector_size, int *circuit_vector, int unit_parameters_size, double *unit_parameters, `Simulator_Parameters` simulator_parameters)
- double `circuit_performance` (int vector_size, int *circuit_vector, int unit_parameters_size, double *unit_parameters)
- double `circuit_performance` (int vector_size, int *circuit_vector, `Simulator_Parameters` simulator_parameters)
- double `circuit_performance` (int vector_size, int *circuit_vector)
- double `circuit_performance` (int vector_size, int *circuit_vector, bool testFlag)
- double `circuit_performance` (int vector_size, int *circuit_vector, int unit_parameters_size, double *unit_parameters, bool testFlag)

Variables

- `Simulator_Parameters default_simulator_parameters`

7.9.1 Detailed Description

C++ header file for the circuit simulator.

This header file defines the function that will be used to evaluate the circuit and the parameters for the simulation Definition in file `CSimulator.h`.

7.9.2 Function Documentation

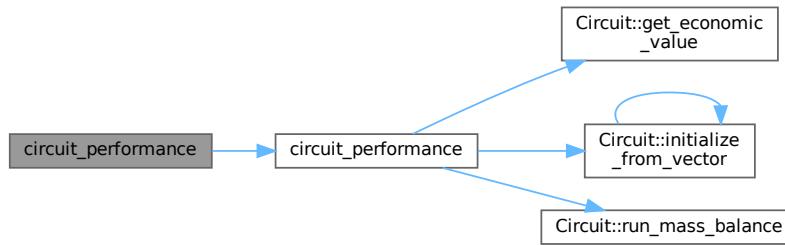
circuit_performance() [1/6]

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

Definition at line 79 of file [CSimulator.cpp](#).

References [circuit_performance\(\)](#), and [default_simulator_parameters](#).

Here is the call graph for this function:



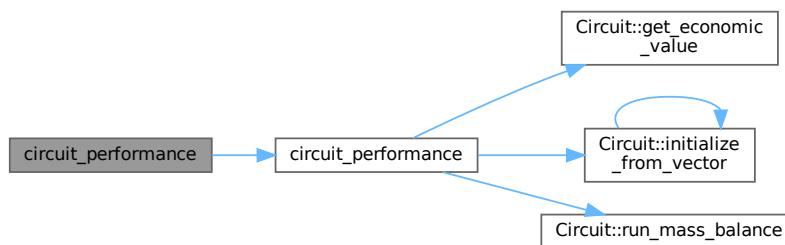
circuit_performance() [2/6]

```
double circuit_performance (
    int vector_size,
    int * circuit_vector,
    bool testFlag )
```

Definition at line 95 of file [CSimulator.cpp](#).

References [circuit_performance\(\)](#), and [default_simulator_parameters](#).

Here is the call graph for this function:



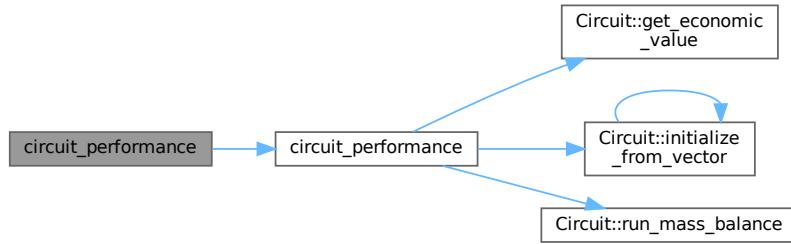
circuit_performance() [3/6]

```
double circuit_performance (
    int vector_size,
    int * circuit_vector,
    int unit_parameters_size,
    double * unit_parameters )
```

Definition at line 73 of file [CSimulator.cpp](#).

References [circuit_performance\(\)](#), and [default_simulator_parameters](#).

Here is the call graph for this function:



`circuit_performance()` [4/6]

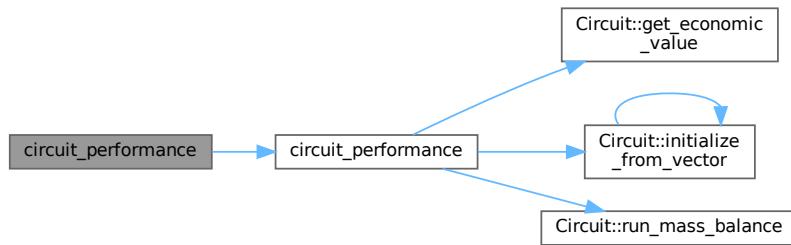
```

double circuit_performance (
    int vector_size,
    int * circuit_vector,
    int unit_parameters_size,
    double * unit_parameters,
    bool testFlag )
  
```

Definition at line 88 of file [CSimulator.cpp](#).

References [circuit_performance\(\)](#), and [default_simulator_parameters](#).

Here is the call graph for this function:



`circuit_performance()` [5/6]

```

double circuit_performance (
    int vector_size,
    int * circuit_vector,
    int unit_parameters_size,
    double * unit_parameters,
    Simulator_Parameters simulator_parameters )
  
```

Referenced by [main\(\)](#).

Here is the caller graph for this function:



circuit_performance() [6/6]

```
double circuit_performance (
    int vector_size,
    int * circuit_vector,
    Simulator_Parameters simulator_parameters )
```

7.9.3 Variable Documentation

default_simulator_parameters

`Simulator_Parameters default_simulator_parameters [extern]`

Definition at line 17 of file [CSimulator.cpp](#).

Referenced by [circuit_performance\(\)](#), [circuit_performance\(\)](#), [circuit_performance\(\)](#), and [circuit_performance\(\)](#).

7.10 CSimulator.h

[Go to the documentation of this file.](#)

```
00001 /**
00002  * @file CSimulator.h
00003  * @brief C++ header file for the circuit simulator
00004  *
00005  * This header file defines the function that will be used to evaluate the
00006  * circuit and the parameters for the simulation
00007 */
00008 #pragma once
00009
00010 #include <string>
00011
00012 // Structure to hold the simulation parameters
00013 struct Simulator_Parameters
00014 {
00015     // Convergence parameters
00016     double tolerance = 1e-6;
00017     int max_iterations = 1000;
00018
00019     // Material properties
00020     double material_density = 3000.0; // kg/m3, density of all solid materials
00021     double solids_content = 0.1;      // Fraction of solids by volume
00022
00023     // Rate constants (s-1)
00024     double k_palusznium_high = 0.008;
00025     double k_palusznium_inter = 0.004;
00026     double k_germanium_high = 0.004;
00027     double k_germanium_inter = 0.002;
00028     double k_waste_high = 0.0005;
00029     double k_waste_inter = 0.00025;
00030
00031     // Feed rates (kg/s)
00032     double feed_palusznium = 8.0;
00033     double feed_germanium = 12.0;
00034     double feed_waste = 80.0;
00035
00036     // Economic parameters (f/kg)
00037     double palusznium_value_in_palusznium_stream = 120.0;
00038     double germanium_value_in_palusznium_stream = -20.0;
00039     double waste_penalty_in_palusznium_stream = -300.0;
00040
00041     double palusznium_value_in_germanium_stream = 0.0;
00042     double germanium_value_in_germanium_stream = 80.0;
```

```

00043     double waste_penalty_in_gormanium_stream = -25.0;
00044
00045     // Unit volume parameters
00046     double fixed_unit_volume = 10.0;      // m3
00047     double min_unit_volume = 2.5;        // m3 (for variable case)
00048     double max_unit_volume = 20.0;       // m3 (for variable case)
00049     double max_circuit_volume = 150.0;    // m3
00050
00051     // Circuit operating cost parameters
00052     double cost_coefficient = 5.0;
00053     double volume_penalty_coefficient = 1000.0;
00054
00055     // Visualization options
00056     bool generate_visualization = false;
00057     std::string visualization_file = "circuit.dot";
00058 };
00059
00060 // Default simulation parameters
00061 extern Simulator_Parameters default_simulator_parameters;
00062
00063 // Main circuit performance evaluation function
00064 double circuit_performance(int vector_size, int* circuit_vector, int unit_parameters_size, double*
00065     unit_parameters,
00066             Simulator_Parameters simulator_parameters);
00067
00068 // Overloaded functions for convenience
00069 double circuit_performance(int vector_size, int* circuit_vector, int unit_parameters_size, double*
00070     unit_parameters);
00071 double circuit_performance(int vector_size, int* circuit_vector, Simulator_Parameters
00072     simulator_parameters);
00073 double circuit_performance(int vector_size, int* circuit_vector);
00074 double circuit_performance(int vector_size, int* circuit_vector, bool testFlag);
00075 double circuit_performance(int vector_size, int* circuit_vector, int unit_parameters_size, double*
00076     unit_parameters,
00077             bool testFlag);

```

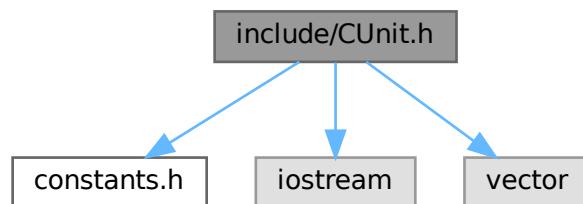
7.11 include/CUnit.h File Reference

Declares the **CUnit** class – a single separation unit in the mineral-processing circuit (e.g.

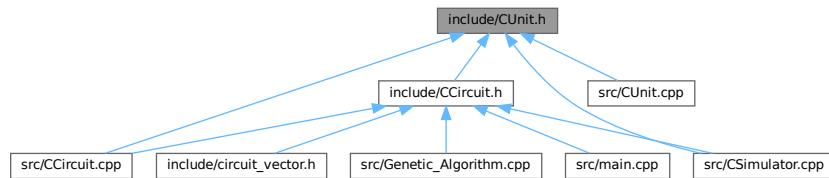
```
#include "constants.h"
#include <iostream>
```

```
#include <vector>
```

Include dependency graph for CUnit.h:



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



Classes

- class [CUnit](#)

7.11.1 Detailed Description

Declares the [CUnit](#) class – a single separation unit in the mineral-processing circuit (e.g. flotation cell / centrifuge).

A unit receives one mixed feed stream and produces two output streams:

- Concentrate ("high-grade") -> directed to conc_num
- Tails ("low-grade") -> directed to tails_num

The class stores: – Topology information (where each outlet goes) – Kinetic/geometry constants – Current iteration's mass-flow state (feed & product streams) – A traversal flag used by validity checks / graph search

It also provides: • Constructors to initialise a unit • process() – computes residence time, recoveries, and updates the outlet flowrates given the current feed.

Definition in file [CUnit.h](#).

7.12 CUnit.h

[Go to the documentation of this file.](#)

```

00001 /**
00002 * @file CUnit.h
00003 * @brief Declares the CUnit class - a single separation unit in the
00004 *        mineral-processing circuit (e.g. flotation cell / centrifuge).
00005 *
00006 * A unit receives one mixed feed stream and produces two output streams:
00007 *   - Concentrate ("high-grade") -> directed to conc_num
00008 *   - Tails      ("low-grade") -> directed to tails_num
00009 *
00010 * The class stores:
00011 *   - Topology information (where each outlet goes)
00012 *   - Kinetic/geometry constants
00013 *   - Current iteration's mass-flow state (feed & product streams)
00014 *   - A traversal flag used by validity checks / graph search
00015 *
00016 * It also provides:
00017 *   - Constructors to initialise a unit
00018 *   - process() - computes residence time, recoveries, and updates the
00019 *                 outlet flowrates given the current feed.
00020 */
00021
00022 #pragma once
00023
00024 #include "constants.h" // contains default k-values, density, , etc.
00025 #include <iostream>
00026 #include <vector>
00027
00028 /* ----- */
00029 /*           Separation-unit class           */
00030 /* ----- */
00031 class CUnit
00032 {
00033 public:
00034     // Index of the unit to which this unit's concentrate stream is connected
00035     int conc_num;
00036     // Index of the unit to which this unit's tailings stream is connected
  
```

```

00037     int tails_num;
00038     // A Boolean that is changed to true if the unit has been seen during graph
00039     // traversal
00040     bool mark;
00041
00042     /* ----- Physical / kinetic parameters ----- */
00043     double volume; // Unit volume V (m3) - default 10 m3
00044     double V_min; // Minimum volume (m3) - default 2.5 m3
00045     double V_max; // Maximum volume (m3) - default 20 m3
00046
00047     // Material flow rates (kg/s)
00048     double feed_palusznium; // Palusznium in feed
00049     double feed_gormanium; // Gormanium in feed
00050     double feed_waste; // Waste material in feed
00051
00052     // Rate constants for separation (s-1)
00053     double k_palusznium; // Rate constant for Palusznium
00054     double k_gormanium; // Rate constant for Gormanium
00055     double k_waste; // Rate constant for Waste
00056
00057     /* ----- Computed outlet mass flowrates ----- */
00058     // Concentrate stream
00059     double conc_palusznium; // C_P (kg s-1)
00060     double conc_gormanium; // C_G (kg s-1)
00061     double conc_waste; // C_W (kg s-1)
00062     // Tails stream
00063     double tails_palusznium; // T_P (kg s-1)
00064     double tails_gormanium; // T_G (kg s-1)
00065     double tails_waste; // T_W (kg s-1)
00066
00067     double rho;
00068     double phi;
00069
00070     /* ----- Computed recoveries for each component ----- */
00071     double Rp, Rg, Rw; // Recoveries for each component
00072
00073     /* ----- Constructors ----- */
00074     /// Default constructor - initialises all numeric members to zero and
00075     /// routes to invalid destinations (e.g. -1) until set by GA vector.
00076     CUnit();
00077
00078     /// Convenience constructor - sets outlet destinations; remaining
00079     /// parameters are pulled from constants.h defaults.
00080     /// @param conc Destination index for concentrate
00081     /// @param tails Destination index for tails
00082     CUnit(int conc, int tails);
00083
00084     /* ----- Methods ----- */
00085     /**
00086     * @brief Perform unit calculation for the current feed.
00087     *
00088     * Steps:
00089     *   1. Compute residence time = V / ( F_i )
00090     *   2. Evaluate recoveries R_i^C = k_i / (1 + k_i )
00091     *   3. Split feed into concentrate & tails streams
00092     *   4. Store outlet flowrates in the public members above
00093     *
00094     * No return value - results are written into conc_* and tails_*.
00095     * Caller is responsible for ensuring feed_* are populated beforehand.
00096     */
00097
00098     CUnit(int conc, int tails, bool testFlag);
00099
00100     void process();
00101
00102     /**
00103     * @brief Check if the unit is valid.
00104     *
00105     * A unit is valid if:
00106     *   1. It has a valid destination for both concentrate and tails
00107     *   2. It has a non-zero volume
00108     *   3. It has a non-zero k-value for at least one component
00109     *
00110     * @return true if valid, false otherwise.
00111     */
00112     // double calculate_recovery(const string& component, double feed_rate) const;
00113
00114     /**
00115     * @brief Update the volume of the unit.
00116     *
00117     * @param beta The new volume of the unit.
00118     */
00119     void update_volume(double beta);
00120 };

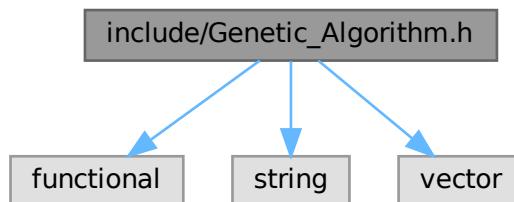
```

7.13 include/Genetic_Algorithm.h File Reference

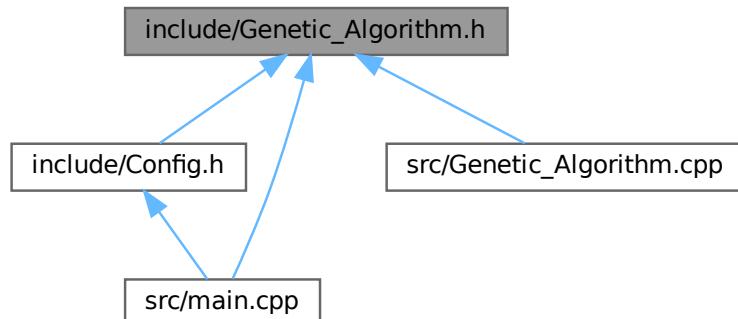
Genetic Algorithm Header.

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

Include dependency graph for Genetic_Algorithm.h:



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



Classes

- struct [Algorithm_Parameters](#)
- struct [OptimizationResult](#)

Macros

- #define [DEFAULT_ALGORITHM_PARAMETERS](#) Algorithm_Parameters {}

Functions

- std::vector< int > [generate_valid_circuit_template](#) (int num_units)
Generate a valid circuit template.
- bool [all_true](#) (int int_vector_size, int *int_vector, int real_vector_size, double *real_vector)
Check if all values in the vector are true.

- bool `all_true_ints` (int int_vector_size, int *vector)
- bool `all_true_reals` (int real_vector_size, double *vector)
- void `set_random_seed` (int seed)

Set the random seed for the random number generator.
- int `optimize` (int int_vector_size, int *int_vector, std::function< double(int, int *)> func, std::function< bool(int, int *)> validity=`all_true_ints`, `Algorithm_Parameters` algorithm_parameters=`Algorithm_Parameters` {})

Optimize a discrete vector using a genetic algorithm.
- int `optimize` (int real_vector_size, double *real_vector, std::function< double(int, double *)> func, std::function< bool(int, double *)> validity=`all_true_reals`, `Algorithm_Parameters` algorithm_parameters=`Algorithm_Parameters` {})

Optimize a continuous vector using a genetic algorithm.
- int `optimize` (int int_vector_size, int *int_vector, int real_vector_size, double *real_vector, std::function< double(int, int *, int, double *)> func, std::function< bool(int, int *, int, double *)> validity=`all_true`, `Algorithm_Parameters` algorithm_parameters=`Algorithm_Parameters` {})

Optimize a mixed discrete-continuous vector using a genetic algorithm.
- `OptimizationResult get_last_optimization_result ()`

Get the last optimization result.

7.13.1 Detailed Description

Genetic Algorithm Header.

Author

This header defines the interface for the genetic algorithm optimization
Definition in file `Genetic_Algorithm.h`.

7.13.2 Macro Definition Documentation

DEFAULT_ALGORITHM_PARAMETERS

```
#define DEFAULT_ALGORITHM_PARAMETERS Algorithm_Parameters {}
```

Definition at line 66 of file `Genetic_Algorithm.h`.

7.13.3 Function Documentation

all_true()

```
bool all_true (
    int iv,
    int * ivs,
    int rv,
    double * rvs )
```

Check if all values in the vector are true.

This function checks if all values in the vector are true.

Parameters

<code>iv</code>	Size of the vector
<code>ivs</code>	Pointer to the vector
<code>rv</code>	Size of the real vector
<code>rvs</code>	Pointer to the real vector

Definition at line 315 of file `Genetic_Algorithm.cpp`.

all_true_ints()

```
bool all_true_ints (
    int int_vector_size,
    int * vector )
```

Definition at line 319 of file [Genetic_Algorithm.cpp](#).

all_true_reals()

```
bool all_true_reals (
    int real_vector_size,
    double * vector )
```

Definition at line 323 of file [Genetic_Algorithm.cpp](#).

generate_valid_circuit_template()

```
std::vector< int > generate_valid_circuit_template (
    int num_units )
```

Generate a valid circuit template.

This function generates a valid circuit template based on the number of units. It creates a vector of integers representing the circuit connections.

Parameters

<code>num_units</code>	Number of units in the circuit
------------------------	--------------------------------

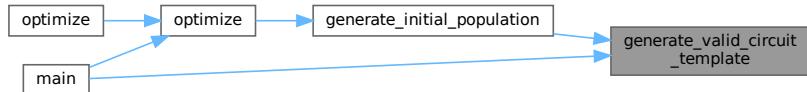
Returns

A vector of integers representing the circuit connections

Definition at line 59 of file [Genetic_Algorithm.cpp](#).

Referenced by [generate_initial_population\(\)](#), and [main\(\)](#).

Here is the caller graph for this function:

**get_last_optimization_result()**

```
OptimizationResult get_last_optimization_result ( )
```

Get the last optimization result.

This function returns the last optimization result. This structure holds the results of the optimization process, including the best fitness, number of generations, average fitness, standard deviation of fitness, time taken, and convergence status.

Returns

The last optimization result

Definition at line 340 of file [Genetic_Algorithm.cpp](#).

References [last_result](#).

optimize() [1/3]

```
int optimize (
    int int_vector_size,
    int * int_vector,
    int real_vector_size,
    double * real_vector,
    std::function< double(int, int *)> hybrid_func,
    std::function< bool(int, int *)> hybrid_validity,
    Algorithm_Parameters params )
```

Optimize a mixed discrete-continuous vector using a genetic algorithm.

This function optimizes a mixed discrete-continuous vector using a genetic algorithm. It evaluates the fitness of the population in parallel and applies selection, crossover, and mutation to generate new populations.

Parameters

<i>int_vector_size</i>	Size of the integer vector
<i>int_vector</i>	Pointer to the integer vector
<i>real_vector_size</i>	Size of the real vector
<i>real_vector</i>	Pointer to the real vector
<i>hybrid_func</i>	Function to evaluate the fitness of the circuit
<i>hybrid_validity</i>	Function to check the validity of the circuit
<i>params</i>	Algorithm parameters for the optimization process

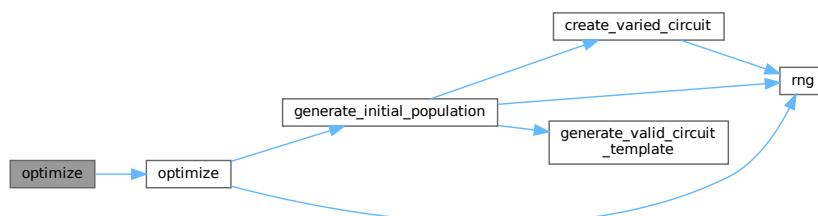
Returns

The best fitness value found during optimization

Definition at line 836 of file [Genetic_Algorithm.cpp](#).

References [optimize\(\)](#).

Here is the call graph for this function:

**optimize() [2/3]**

```
int optimize (
    int int_vector_size,
    int * int_vector,
    std::function< double(int, int *)> func,
    std::function< bool(int, int *)> validity,
    Algorithm_Parameters params )
```

Optimize a discrete vector using a genetic algorithm.

This function optimizes a discrete vector using a genetic algorithm. It evaluates the fitness of the population in parallel and applies selection, crossover, and mutation to generate new populations.

Parameters

<i>int_vector_size</i>	Size of the integer vector
<i>int_vector</i>	Pointer to the integer vector
<i>func</i>	Function to evaluate the fitness of the circuit
<i>validity</i>	Function to check the validity of the circuit
<i>params</i>	Algorithm parameters for the optimization process

Returns

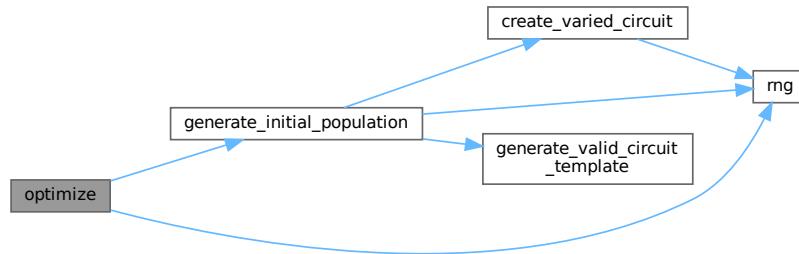
The best fitness value found during optimization

Definition at line 364 of file [Genetic_Algorithm.cpp](#).

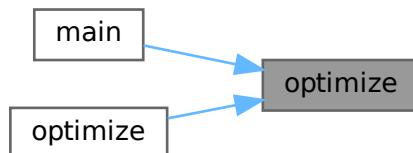
References [OptimizationResult::best_fitness](#), [Algorithm_Parameters::convergence_threshold](#), [Algorithm_Parameters::crossover_prob](#), [generate_initial_population\(\)](#), [OptimizationResult::generations](#), [Algorithm_Parameters::inversion_probability](#), [last_result](#), [Algorithm_Parameters::max_iterations](#), [Algorithm_Parameters::mutation_probability](#), [Algorithm_Parameters::mutation_steepest_descent](#), [Algorithm_Parameters::population_size](#), [rng\(\)](#), [Algorithm_Parameters::stall_generations](#), [Algorithm_Parameters::tournament_size](#), [Algorithm_Parameters::use_inversion](#), and [Algorithm_Parameters::verbose](#).

Referenced by [main\(\)](#), and [optimize\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**optimize() [3/3]**

```

int optimize (
    int real_vector_size,
    double * real_vector,
    std::function< double(int, double *)> func,
  
```

```
    std::function< bool(int, double *)> validity,
    Algorithm_Parameters params )
```

Optimize a continuous vector using a genetic algorithm.

This function optimizes a continuous vector using a genetic algorithm. It evaluates the fitness of the population in parallel and applies selection, crossover, and mutation to generate new populations.

Parameters

<i>real_vector_size</i>	Size of the real vector
<i>real_vector</i>	Pointer to the real vector
<i>func</i>	Function to evaluate the fitness of the circuit
<i>validity</i>	Function to check the validity of the circuit
<i>params</i>	Algorithm parameters for the optimization process

Returns

The best fitness value found during optimization

Definition at line 620 of file [Genetic_Algorithm.cpp](#).

References [OptimizationResult::best_fitness](#), [Algorithm_Parameters::convergence_threshold](#), [Algorithm_Parameters::crossover_prob](#), [OptimizationResult::generations](#), [last_result](#), [Algorithm_Parameters::max_iterations](#), [Algorithm_Parameters::mutation_probability](#), [Algorithm_Parameters::mutation_step_size](#), [Algorithm_Parameters::population_size](#), [rng\(\)](#), [Algorithm_Parameters::scaling_mutation](#), [Algorithm_Parameters::scaling_mutation_min](#), [Algorithm_Parameters::scaling_mutation_prob](#), [Algorithm_Parameters::stall_generations](#), [Algorithm_Parameters::tournament_size](#), [Algorithm_Parameters::use_scaling_mutation](#), and [Algorithm_Parameters::verbose](#).

Here is the call graph for this function:



set_random_seed()

```
void set_random_seed (
    int seed )
```

Set the random seed for the random number generator.

Definition at line 27 of file [Genetic_Algorithm.cpp](#).

References [g_random_seed](#).

Referenced by [main\(\)](#).

Here is the caller graph for this function:



7.14 Genetic_Algorithm.h

[Go to the documentation of this file.](#)

```

00001 /**
00002  * @file Genetic_Algorithm.h
00003  * @brief Genetic Algorithm Header
00004  * @author
00005  *
00006  * This header defines the interface for the genetic algorithm optimization
00007 */
00008
00009 #pragma once
00010
00011 #include <functional>
00012 #include <string>
00013 #include <vector>
00014
00015 std::vector<int> generate_valid_circuit_template(int num_units);
00016
00017 // Structure to hold genetic algorithm parameters
00018 struct Algorithm_Parameters
00019 {
00020     // Number of units
00021     int num_units = 10;
00022
00023     // Optimization mode: : "d", "c", or "h"
00024     std::string mode = "h";
00025
00026     // General parameters
00027     int random_seed = -1;
00028     int max_iterations = 1000; // Maximum number of generations
00029     int population_size = 100; // Number of individuals in the population
00030     int elite_count = 1; // Number of best individuals to keep unchanged
00031
00032     // Selection parameters
00033     double selection_pressure = 1.5; // Linear rank selection pressure parameter
00034     int tournament_size = 2; // Number of contenders per tournament
00035
00036     // Crossover parameters
00037     double crossover_probability = 0.8; // Probability of crossover
00038     int crossover_points = 1; // Number of crossover points (1 or 2)
00039
00040     // Mutation parameters
00041     double mutation_probability = 0.01; // Probability of mutation per gene
00042     int mutation_step_size = 2; // Maximum change in value during mutation
00043     bool allow_mutation_wrapping = true; // Allow mutations to wrap around
00044
00045     // Inversion-mutation parameters
00046     bool use_inversion = true; // turn inversion on/off
00047     double inversion_probability = 0.05; // chance to invert per child
00048
00049     // Scaling-mutation parameters
00050     bool use_scaling_mutation = true;
00051     double scaling_mutation_prob = 0.2; // how often to apply a scale mutation
00052     double scaling_mutation_min = 0.8; // lower bound on the scale factor
00053     double scaling_mutation_max = 1.2; // upper bound on the scale factor
00054
00055     // Termination criteria
00056     double convergence_threshold = 1e-6; // Convergence threshold
00057     int stall_generations = 50; // Max generations with no improvement
00058
00059     // Debug options
00060     bool verbose = false; // Print progress information
00061     bool log_results = false; // Log results to file
00062     std::string log_file = "ga_log.txt"; // Log file name
00063 };
00064
00065 // Default algorithm parameters
00066 #define DEFAULT_ALGORITHM_PARAMETERS
00067 \
00068     Algorithm_Parameters {}
00069
00070 // Validity checking functions
00071 bool all_true(int int_vector_size, int* int_vector, int real_vector_size, double* real_vector);
00072 bool all_true_ints(int int_vector_size, int* vector);
00073 void set_random_seed(int seed);
00074 // Optimization function for discrete vector
00075 int optimize(int int_vector_size, int* int_vector, std::function<double(int, int*)> func,
00076                 std::function<bool(int, int*)> validity = all_true_ints,
00077                 Algorithm_Parameters algorithm_parameters = DEFAULT_ALGORITHM_PARAMETERS);
00078
00079 // Optimization function for continuous vector
00080 int optimize(int real_vector_size, double* real_vector, std::function<double(int, double*)> func,
00081                 std::function<bool(int, double*)> validity = all_true_reals,
00082                 Algorithm_Parameters algorithm_parameters = DEFAULT_ALGORITHM_PARAMETERS);
00083

```

```

00084 // Optimization function for mixed discrete-continuous vector
00085 int optimize(int int_vector_size, int* int_vector, int real_vector_size, double* real_vector,
00086     std::function<double(int, int*, int, double*)> func,
00087     std::function<bool(int, int*, int, double*)> validity = all_true,
00088     Algorithm_Parameters algorithm_parameters = DEFAULT_ALGORITHM_PARAMETERS);
00089
00090 // Structure to hold statistics about the optimization process
00091 struct OptimizationResult
00092 {
00093     double best_fitness; // Best fitness value found
00094     int generations; // Number of generations run
00095     double avg_fitness; // Average fitness of final population
00096     double std_fitness; // Standard deviation of final population fitness
00097     double time_taken; // Time taken for optimization (seconds)
00098     bool converged; // Whether algorithm converged
00099
00100    // Default constructor
00101    OptimizationResult()
00102        : best_fitness(0), generations(0), avg_fitness(0), std_fitness(0), time_taken(0),
00103        converged(false)
00104    {
00105    }
00106
00107 // Get the last optimization result
00108 OptimizationResult get_last_optimization_result();

```

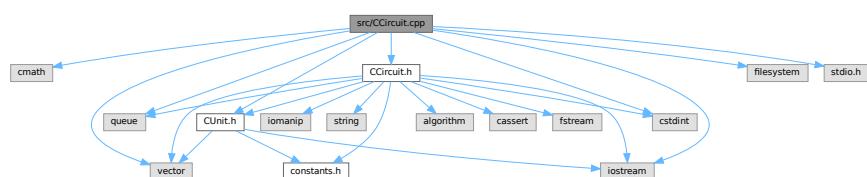
7.15 README.md File Reference

7.16 src/CCircuit.cpp File Reference

Implementation of the [Circuit](#) class.

```
#include <cmath>
#include <queue>
#include <vector>
#include <CCircuit.h>
#include <CUnit.h>
#include <cstdint>
#include <filesystem>
#include <iostream>
#include <stdio.h>
```

Include dependency graph for CCircuit.cpp:



7.16.1 Detailed Description

Implementation of the [Circuit](#) class.

This file contains the implementation of the [Circuit](#) class, which represents a mineral-processing circuit. The class includes methods for checking the validity of the circuit, marking units, running mass balance calculations, and exporting the circuit to a dot file for visualization.

Definition in file [CCircuit.cpp](#).

7.17 CCircuit.cpp

[Go to the documentation of this file.](#)

```

00001 /**
00002 * @file CCircuit.cpp
00003 * @brief Implementation of the Circuit class
00004 *
00005 * This file contains the implementation of the Circuit class, which represents
00006 * a mineral-processing circuit. The class includes methods for checking

```

```

00007 * the validity of the circuit, marking units, running mass balance
00008 * calculations, and exporting the circuit to a dot file for visualization.
00009 *
00010 */
00011 #include <cmath>
00012 #include <queue>
00013 #include <vector>
00014
00015 #include <CCircuit.h>
00016 #include <CUnit.h>
00017 #include <cstdint>
00018 #include <filesystem>
00019 #include <iostream>
00020 #include <stdio.h>
00021
00022 /**
00023 * @brief Constructor for the Circuit class
00024 *
00025 */
00026 Circuit::Circuit(int num_units)
00027     : units(num_units), feed_unit(0), feed_palusznium_rate(Constants::Feed::DEFAULT_PALUSZNIAUM_FEED),
00028         feed_germanium_rate(Constants::Feed::DEFAULT_GORMANIUM_FEED),
00029         feed_waste_rate(Constants::Feed::DEFAULT_WASTE_FEED), palusznium_product_palusznium(0.0),
00030         palusznium_product_germanium(0.0), palusznium_product_waste(0.0),
00031         germanium_product_palusznium(0.0),
00032         germanium_product_germanium(0.0), germanium_product_waste(0.0), tailings_palusznium(0.0),
00033         tailings_germanium(0.0),
00034         tailings_waste(0.0),
00035         palusznium_value(Constants::Economic::PALUSZNIAUM_VALUE_IN_PALUSZNIAUM_STREAM),
00036         germanium_value(Constants::Economic::GORMANIUM_VALUE_IN_GORMANIUM_STREAM),
00037         waste_penalty_palusznium(Constants::Economic::WASTE_PENALTY_IN_PALUSZNIAUM_STREAM),
00038         waste_penalty_germanium(Constants::Economic::WASTE_PENALTY_IN_GORMANIUM_STREAM), beta(nullptr),
00039         circuit_vector(nullptr), n(num_units)
00040 {
00041 }
00042
00043 /**
00044 * @brief Check the validity of the circuit
00045 *
00046 * This function checks the validity of the circuit vector by performing
00047 * various checks, including length check, feed check, index check,
00048 * self-loop check, same output check, reachability check, terminal check,
00049 * and mass balance convergence check.
00050 *
00051 * @param vector_size Size of the circuit vector
00052 * @param vec Circuit vector
00053 */
00054 bool Circuit::check_validity(int vector_size, const int* vec)
00055 {
00056     // 1. length must be 2*n+1
00057     int expected = 2 * n + 1;
00058     if (vector_size != expected)
00059     {
00060         return false;
00061     }
00062
00063     // 2. feed check: feed cannot directly feed to terminal
00064     feed_dest = vec[0]; // feed points to the unit
00065     if (feed_dest < 0 || feed_dest >= n)
00066     {
00067         return false;
00068     }
00069
00070     // read each unit's concentrate and tailing and do static check
00071     struct Dest
00072     {
00073         int conc;
00074         int tail;
00075     };
00076     std::vector<Dest> dest(n);
00077
00078     int max_idx = n + 2; // the last valid index
00079
00080     for (int i = 0; i < n; ++i)
00081     {
00082         int conc = vec[1 + 2 * i]; // conc points to the unit
00083         int tail = vec[2 + 2 * i]; // tail points to the unit
00084
00085         // 3. index check: conc must be in (0, n+2), tail must be in (0, n+2)
00086         if (conc < 0 || conc > max_idx)
00087         {
00088             return false;
00089         }
00090         if (tail < 0 || tail > max_idx)

```

```

00091         {
00092             return false;
00093         }
00094
00095         // 4. no self-loop: conc cannot be equal to i, tail cannot be equal to i
00096         if (conc == i || tail == i)
00097     {
00098         return false;
00099     }
00100
00101         // 5. same output: conc cannot be equal to tail
00102         if (conc == tail)
00103     {
00104         return false;
00105     }
00106
00107         dest[i] = {conc, tail};
00108
00109         units[i].conc_num = conc;
00110         units[i].tails_num = tail;
00111         units[i].mark = false;
00112     }
00113
00114         // 6. reachability check: all units must be reachable from feed
00115         this->mark_units(feed_dest);
00116
00117         for (int i = 0; i < n; ++i)
00118     {
00119         if (!units[i].mark)
00120     {
00121         return false;
00122     }
00123 }
00124
00125         // 7. two terminals check: each unit must reach at least 2 different terminals
00126         std::vector<int8_t> cache(n, -1);
00127         uint8_t global_mask = 0;
00128         for (int i = 0; i < n; ++i)
00129     {
00130             uint8_t mask = this->term_mask(i);
00131             global_mask |= mask;
00132             int cnt = (mask & 1) + ((mask >> 1) & 1) + ((mask >> 2) & 1);
00133             if (cnt < 2)
00134     {
00135                 return false;
00136             }
00137 }
00138
00139         // 8. final terminal check: P1/P2, TA must be present
00140         if ((global_mask & (0b001 | 0b010)) == 0)
00141     {
00142         return false;
00143     }
00144
00145         if ((global_mask & 0b100) == 0)
00146     {
00147         return false;
00148     }
00149
00150         // 9. mass balance check: mass balance must converge
00151         if (!run_mass_balance(1e-6, 100))
00152     {
00153         return false;
00154     }
00155
00156         return true;
00157     }
00158
00159 /**
00160 * @brief Check the validity of the circuit vector and its parameters
00161 *
00162 * This function checks the validity of the circuit vector and its parameters
00163 * by performing various checks, including length check, parameter range check,
00164 * and validity of the circuit vector.
00165 *
00166 * @param vector_size Size of the circuit vector
00167 * @param circuit_vector Circuit vector
00168 * @param unit_parameters_size Size of the unit parameters
00169 * @param unit_parameters Unit parameters
00170 *
00171 * @return true if the circuit vector and parameters are valid, false otherwise
00172 */
00173 bool Circuit::check_validity(int vector_size, const int* circuit_vector, int unit_parameters_size,
00174                             double* unit_parameters)
00175 {
00176     bool valid = check_validity(vector_size, circuit_vector);
00177     // check the validity of the circuit vector

```

```

00178     if (!valid)
00179     {
00180         return false;
00181     }
00182 // check the validity of the unit parameters
00183 if (unit_parameters == nullptr)
00184 {
00185     return valid;
00186 }
00187
00188 // the length of the continuous parameters must be exactly n units
00189 if (unit_parameters_size != n)
00190 {
00191     return false;
00192 }
00193
00194 // each parameter must be in [0,1] (or other physical range)
00195 for (int i = 0; i < unit_parameters_size; ++i)
00196 {
00197     double beta = unit_parameters[i];
00198     if (beta < 0.0 || beta > 1.0 || std::isnan(beta))
00199     {
00200         return false;
00201     }
00202 }
00203
00204 return true;
00205 }
00206
00207 /**
00208 * @brief Mark the units in the circuit
00209 *
00210 * This function marks the units in the circuit as visited. It recursively
00211 * traverses the circuit starting from the given unit number and marks each
00212 * unit as visited.
00213 *
00214 * @param unit_num The unit number to start marking from
00215 */
00216 void Circuit::mark_units(int unit_num)
00217 {
00218
00219     if (this->units[unit_num].mark)
00220         return;
00221
00222     this->units[unit_num].mark = true;
00223
00224     // If we have seen this unit already exit
00225     // Mark that we have now seen the unit
00226
00227     // If conc_num does not point at a circuit outlet recursively call the
00228     // function
00229     if (this->units[unit_num].conc_num < this->units.size())
00230     {
00231         mark_units(this->units[unit_num].conc_num);
00232     }
00233
00234     // If tails_num does not point at a circuit outlet recursively call the
00235     // function
00236
00237     if (this->units[unit_num].tails_num < this->units.size())
00238     {
00239         mark_units(this->units[unit_num].tails_num);
00240     }
00241 }
00242
00243 /**
00244 * @brief Constructor for the Circuit class
00245 *
00246 * This constructor initializes the circuit with the given number of units
00247 * and a pointer to the beta array.
00248 *
00249 * @param num_units Number of units in the circuit
00250 * @param beta Pointer to the beta array
00251 */
00252 Circuit::Circuit(int num_units, double* beta)
00253     : units(num_units), feed_unit(0), n(num_units),
00254
00255     feed_palusznium_rate(Constants::Feed::DEFAULT_PALUSZNIA_FEED),
00256     feed_germanium_rate(Constants::Feed::DEFAULT_GORMANIUM_FEED),
00257     feed_waste_rate(Constants::Feed::DEFAULT_WASTE_FEED), palusznium_product_palusznium(0.0),
00258     palusznium_product_germanium(0.0), palusznium_product_waste(0.0),
00259     germanium_product_palusznium(0.0),
00260     germanium_product_germanium(0.0), germanium_product_waste(0.0), tailings_palusznium(0.0),
00261     tailings_germanium(0.0),
00262     tailings_waste(0.0), beta(beta),
00263     palusznium_value(Constants::Economic::PALUSZNIA_VALUE_IN_PALUSZNIA_STREAM),
00264     germanium_value(Constants::Economic::GORMANIUM_VALUE_IN_GORMANIUM_STREAM),
00265

```

```

00262     waste_penalty_palusznium(Constants::Economic::WASTE_PENALTY_IN_PALUSZNIAUM_STREAM),
00263     waste_penalty_gormaniun(Constants::Economic::WASTE_PENALTY_IN_GORMANIUM_STREAM),
00264     palusznium_value_in_gormaniun(Constants::Economic::PALUSZNIAUM_VALUE_IN_GORMANIUM_STREAM),
00265     gormaniun_value_in_palusznium(Constants::Economic::GORMANIUM_VALUE_IN_PALUSZNIAUM_STREAM)
00266 {
00267 }
00268
00269 /**
00270 * @brief Constructor for the Circuit class
00271 *
00272 * This constructor initializes the circuit with the given number of units,
00273 * a pointer to the beta array, and a test flag.
00274 *
00275 * @param num_units Number of units in the circuit
00276 * @param beta Pointer to the beta array
00277 * @param testFlag Test flag to indicate whether to use test parameters
00278 */
00279 Circuit::Circuit(int num_units, double* beta, bool testFlag)
00280 : units(num_units), feed_unit(0), n(num_units),
00281
00282     feed_palusznium_rate(Constants::Feed::DEFAULT_PALUSZNIAUM_FEED),
00283     feed_gormaniun_rate(Constants::Feed::DEFAULT_GORMANIUM_FEED),
00284     feed_waste_rate(Constants::Feed::DEFAULT_WASTE_FEED), palusznium_product_palusznium(0.0),
00285     palusznium_product_gormaniun(0.0), palusznium_product_waste(0.0),
00286     gormaniun_product_gormaniun(0.0), gormaniun_product_waste(0.0), tailings_palusznium(0.0),
00287     tailings_gormaniun(0.0),
00288     tailings_waste(0.0), beta(beta),
00289     palusznium_value(Constants::Economic::PALUSZNIAUM_VALUE_IN_PALUSZNIAUM_STREAM),
00290     gormaniun_value(Constants::Economic::GORMANIUM_VALUE_IN_GORMANIUM_STREAM),
00291     waste_penalty_palusznium(Constants::Economic::WASTE_PENALTY_IN_PALUSZNIAUM_STREAM),
00292     waste_penalty_gormaniun(Constants::Economic::WASTE_PENALTY_IN_GORMANIUM_STREAM),
00293     palusznium_value_in_gormaniun(Constants::Economic::PALUSZNIAUM_VALUE_IN_GORMANIUM_STREAM),
00294     gormaniun_value_in_palusznium(Constants::Economic::GORMANIUM_VALUE_IN_PALUSZNIAUM_STREAM)
00295 {
00296     if (testFlag)
00297     {
00298         this->feed_palusznium_rate = Constants::Test::DEFAULT_PALUSZNIAUM_FEED;
00299         this->feed_gormaniun_rate = Constants::Test::DEFAULT_GORMANIUM_FEED;
00300         this->feed_waste_rate = Constants::Test::DEFAULT_WASTE_FEED;
00301
00302         this->palusznium_value = Constants::Test::PALUSZNIAUM_VALUE_IN_PALUSZNIAUM_STREAM;
00303         this->gormaniun_value = Constants::Test::GORMANIUM_VALUE_IN_GORMANIUM_STREAM;
00304         this->waste_penalty_palusznium = Constants::Test::WASTE_PENALTY_IN_PALUSZNIAUM_STREAM;
00305         this->waste_penalty_gormaniun = Constants::Test::WASTE_PENALTY_IN_GORMANIUM_STREAM;
00306         this->palusznium_value_in_gormaniun = Constants::Test::PALUSZNIAUM_VALUE_IN_GORMANIUM_STREAM;
00307         this->gormaniun_value_in_palusznium = Constants::Test::GORMANIUM_VALUE_IN_PALUSZNIAUM_STREAM;
00308     }
00309 /**
00310 * @brief Initialize the circuit from a circuit vector
00311 *
00312 * This function initializes the circuit from a circuit vector. It takes
00313 * the size of the vector and the vector itself as input parameters.
00314 *
00315 * @param vector_size Size of the circuit vector
00316 * @param circuit_vector Circuit vector
00317 *
00318 * @return true if initialization is successful, false otherwise
00319 */
00320 bool Circuit::initialize_from_vector(int vector_size, const int* circuit_vector)
00321 {
00322     return initialize_from_vector(vector_size, circuit_vector, nullptr, false);
00323 }
00324
00325 /**
00326 * @brief Initialize the circuit from a circuit vector
00327 *
00328 * This function initializes the circuit from a circuit vector. It takes
00329 * the size of the vector, the vector itself, and a pointer to the beta
00330 * array as input parameters.
00331 *
00332 * @param vector_size Size of the circuit vector
00333 * @param circuit_vector Circuit vector
00334 * @param beta Pointer to the beta array
00335 *
00336 * @return true if initialization is successful, false otherwise
00337 */
00338 bool Circuit::initialize_from_vector(int vector_size, const int* circuit_vector, const double* beta)
00339 {
00340     return initialize_from_vector(vector_size, circuit_vector, beta, false);
00341 }
00342
00343 /**
00344 * @brief Initialize the circuit from a circuit vector
00345 *

```

```

00346 * This function initializes the circuit from a circuit vector. It takes
00347 * the size of the vector, the vector itself, and a test flag as input
00348 * parameters.
00349 *
00350 * @param vector_size Size of the circuit vector
00351 * @param circuit_vector Circuit vector
00352 * @param testFlag Test flag to indicate whether to use test parameters
00353 *
00354 * @return true if initialization is successful, false otherwise
00355 *
00356 */
00357 bool Circuit::initialize_from_vector(int vector_size, const int* circuit_vector, bool testFlag)
00358 {
00359     // Initialize the circuit from the circuit vector
00360     return initialize_from_vector(vector_size, circuit_vector, nullptr, testFlag);
00362 }
00363
00364 /**
00365 * @brief Initialize the circuit from a circuit vector
00366 *
00367 * This function initializes the circuit from a circuit vector. It takes
00368 * the size of the vector, the vector itself, a pointer to the beta array,
00369 * and a test flag as input parameters.
00370 *
00371 * @param vector_size Size of the circuit vector
00372 * @param circuit_vector Circuit vector
00373 * @param beta Pointer to the beta array
00374 * @param testFlag Test flag to indicate whether to use test parameters
00375 *
00376 * @return true if initialization is successful, false otherwise
00377 */
00378 bool Circuit::initialize_from_vector(int vector_size, const int* circuit_vector, const double* beta,
00379                                     bool testFlag)
00380 {
00381     // num_units = n
00382     int num_units = (vector_size - 1) / 2;
00383     if (vector_size != 2 * num_units + 1)
00384         return false;
00384     units.resize(num_units);
00385     this->circuit_vector = circuit_vector;
00386
00387     // feed_unit is the first element of the circuit vector
00388     feed_unit = circuit_vector[0];
00389
00390     // Map the target units to corresponding unit numbers
00391     for (int i = 0; i < num_units; ++i)
00392     {
00393         int conc = circuit_vector[1 + 2 * i];
00394         int tails = circuit_vector[1 + 2 * i + 1];
00395
00396         // transform the unit numbers from n, n+1, n+2 to -1, -2, -3
00397         if (conc == num_units)
00398             conc = PALUSZNIUM_PRODUCT;
00399         else if (conc == num_units + 1)
00400             conc = GORMANIUM_PRODUCT;
00401         else if (conc == num_units + 2)
00402             conc = TAILINGS_OUTPUT;
00403
00404         if (tails == num_units)
00405             tails = PALUSZNIUM_PRODUCT;
00406         else if (tails == num_units + 1)
00407             tails = GORMANIUM_PRODUCT;
00408         else if (tails == num_units + 2)
00409             tails = TAILINGS_OUTPUT;
00410
00411         units[i] = CUnit(conc, tails, testFlag);
00412         if (beta != nullptr)
00413         {
00414             units[i].update_volume(beta[i]);
00416         }
00417     }
00418     return true;
00419 }
00420
00421 /**
00422 * @brief Run mass balance calculations for the circuit
00423 *
00424 * This function runs mass balance calculations for the circuit. It takes
00425 * a tolerance and a maximum number of iterations as input parameters.
00426 *
00427 * @param tolerance Tolerance for convergence
00428 * @param max_iterations Maximum number of iterations
00429 *
00430 * @return true if mass balance converges, false otherwise
00431 */

```

```

00432 bool Circuit::run_mass_balance(double tolerance, int max_iterations)
00433 {
00434     // Initialize feed for all the units
00435     for (auto& u : units)
00436     {
00437         u.feed_palusznium = 0.0;
00438         u.feed_gormanium = 0.0;
00439         u.feed_waste = 0.0;
00440     }
00441     // Initialize feed for the first unit
00442     units[feed_unit].feed_palusznium = feed_palusznium_rate;
00443     units[feed_unit].feed_gormanium = feed_gormanium_rate;
00444     units[feed_unit].feed_waste = feed_waste_rate;
00445
00446     // Record the last feed for each unit for convergence check
00447     std::vector<double> last_feed_p(units.size(), 0.0);
00448     std::vector<double> last_feed_g(units.size(), 0.0);
00449     std::vector<double> last_feed_w(units.size(), 0.0);
00450     // std::cout << "Unit number: " << units.size() << std::endl;
00451
00452     for (int iter = 0; iter < max_iterations; ++iter)
00453     {
00454
00455         // Record the current feed
00456         // Record the current feed to last_feed and clear the current feed
00457         if (iter == 0)
00458         {
00459             for (size_t i = 0; i < units.size(); ++i)
00460             {
00461                 last_feed_p[i] = units[i].feed_palusznium;
00462                 last_feed_g[i] = units[i].feed_gormanium;
00463                 last_feed_w[i] = units[i].feed_waste;
00464                 // clear the current feed
00465                 units[i].feed_palusznium = 0.0;
00466                 units[i].feed_gormanium = 0.0;
00467                 units[i].feed_waste = 0.0;
00468             }
00469         }
00470         else
00471         {
00472             for (size_t i = 0; i < units.size(); ++i)
00473             {
00474                 last_feed_p[i] = units[i].feed_palusznium;
00475                 last_feed_g[i] = units[i].feed_gormanium;
00476                 last_feed_w[i] = units[i].feed_waste;
00477             }
00478         }
00479         // Initialize feed for the first unit
00480         units[feed_unit].feed_palusznium = feed_palusznium_rate;
00481         units[feed_unit].feed_gormanium = feed_gormanium_rate;
00482         units[feed_unit].feed_waste = feed_waste_rate;
00483
00484         // Process all units
00485         for (size_t i = 0; i < units.size(); ++i)
00486         {
00487             units[i].process();
00488         }
00489
00490         // This vector is used to mark whether the feed for each unit has been
00491         // cleared We need to make sure that the feed for each unit is cleared only
00492         // once
00493         std::vector<bool> feedCleared(units.size(), false);
00494
00495         // Initialize the product flow rates
00496         palusznium_product_palusznium = palusznium_product_gormanium = palusznium_product_waste = 0.0;
00497         gormanium_product_palusznium = gormanium_product_gormanium = gormanium_product_waste = 0.0;
00498         tailings_palusznium = tailings_gormanium = tailings_waste = 0.0;
00499
00500         // =====Distributing downstream data=====<<std::endl;
00501         for (size_t i = 0; i < units.size(); ++i)
00502         {
00503             // concentrate flow
00504             int concDest = units[i].conc_num;
00505             if (concDest == PALUSZNIA_PRODUCT)
00506             {
00507                 palusznium_product_palusznium += units[i].conc_palusznium;
00508                 palusznium_product_gormanium += units[i].conc_gormanium;
00509                 palusznium_product_waste += units[i].conc_waste;
00510             }
00511             else if (concDest == GORMANIUM_PRODUCT)
00512             {
00513                 gormanium_product_palusznium += units[i].conc_palusznium;
00514                 gormanium_product_gormanium += units[i].conc_gormanium;
00515                 gormanium_product_waste += units[i].conc_waste;
00516             }
00517             else if (concDest == TAILINGS_OUTPUT)
00518             {

```

```

00519         tailings_palusznium += units[i].conc_palusznium;
00520         tailings_gormanium += units[i].conc_gormanium;
00521         tailings_waste += units[i].conc_waste;
00522     }
00523     else if (concDest >= 0 && concDest < (int)units.size())
00524     {
00525         if (!feedCleared[concDest])
00526         {
00527             feedCleared[concDest] = true;
00528             units[concDest].feed_palusznium = 0.0;
00529             units[concDest].feed_gormanium = 0.0;
00530             units[concDest].feed_waste = 0.0;
00531         }
00532         units[concDest].feed_palusznium += units[i].conc_palusznium;
00533         units[concDest].feed_gormanium += units[i].conc_gormanium;
00534         units[concDest].feed_waste += units[i].conc_waste;
00535     }
00536
00537     // tailings flow
00538     int tailsDest = units[i].tails_num;
00539     if (tailsDest == PALUSZNIA_PRODUCT)
00540     {
00541         palusznia_product_palusznium += units[i].tails_palusznium;
00542         palusznia_product_gormanium += units[i].tails_gormanium;
00543         palusznia_product_waste += units[i].tails_waste;
00544     }
00545     else if (tailsDest == GORMANIUM_PRODUCT)
00546     {
00547         gormanium_product_palusznium += units[i].tails_palusznium;
00548         gormanium_product_gormanium += units[i].tails_gormanium;
00549         gormanium_product_waste += units[i].tails_waste;
00550     }
00551     else if (tailsDest == TAILINGS_OUTPUT)
00552     {
00553         tailings_palusznium += units[i].tails_palusznium;
00554         tailings_gormanium += units[i].tails_gormanium;
00555         tailings_waste += units[i].tails_waste;
00556     }
00557     else if (tailsDest >= 0 && tailsDest < (int)units.size())
00558     {
00559         if (!feedCleared[tailsDest])
00560         {
00561             feedCleared[tailsDest] = true;
00562             units[tailsDest].feed_palusznium = 0.0;
00563             units[tailsDest].feed_gormanium = 0.0;
00564             units[tailsDest].feed_waste = 0.0;
00565         }
00566
00567         units[tailsDest].feed_palusznium += units[i].tails_palusznium;
00568         units[tailsDest].feed_gormanium += units[i].tails_gormanium;
00569         units[tailsDest].feed_waste += units[i].tails_waste;
00570     }
00571 }
00572
00573     // convergence check
00574     double max_rel_change = 0.0;
00575     for (size_t i = 0; i < units.size(); ++i)
00576     {
00577         double rel_p = std::abs(units[i].feed_palusznium - last_feed_p[i]) /
00578             std::max(last_feed_p[i], 1e-12);
00579         double rel_g = std::abs(units[i].feed_gormanium - last_feed_g[i]) /
00580             std::max(last_feed_g[i], 1e-12);
00581         double rel_w = std::abs(units[i].feed_waste - last_feed_w[i]) / std::max(last_feed_w[i],
00582             1e-12);
00583         max_rel_change = std::max({max_rel_change, rel_p, rel_g, rel_w});
00584     }
00585     if (max_rel_change < tolerance)
00586         return true;
00587 }
00588
00589 /**
00590 * @brief Get the economic value of the circuit
00591 *
00592 * This function calculates the economic value of the circuit based on
00593 * the product flow rates and the values of the materials.
00594 *
00595 * @return The economic value of the circuit
00596 *
00597 */
00598 double Circuit::get_economic_value() const
00599 {
00600     double value = 0.0;
00601
00602     // Palusznium product

```

```

00603     value += palusznium_product_palusznium * palusznium_value;
00604     value += palusznium_product_gormanium * gormanium_value_in_palusznium;
00605     value += palusznium_product_waste * waste_penalty_palusznium;
00606
00607     // Gormanium product
00608     value += gormanium_product_gormanium * gormanium_value;
00609     value += gormanium_product_palusznium * palusznium_value_in_gormanium;
00610     value += gormanium_product_waste * waste_penalty_gormanium;
00611
00612     double total_volume = 0.0;
00613     for (const auto& u : units)
00614         total_volume += u.volume;
00615     double cost = 5.0 * std::pow(total_volume, 2.0 / 3.0);
00616     if (total_volume >= 150.0)
00617     {
00618         cost += 1000.0 * std::pow(total_volume - 150.0, 2.0);
00619     }
00620     value -= cost; // cost of the circuit
00621     return value;
00622 }
00623
00624 /**
00625 * @brief Get the recovery of valuable materials
00626 *
00627 * This function calculates the recovery of valuable materials in the
00628 * circuit based on the product flow rates and the feed rates.
00629 *
00630 * @return The recovery of valuable materials
00631 *
00632 */
00633 double Circuit::get_palusznium_recovery() const
00634 {
00635
00636     double total_feed = feed_palusznium_rate;
00637     double recovered = palusznium_product_palusznium;
00638     if (total_feed < 1e-12)
00639         return 0.0;
00640     return recovered / total_feed;
00641 }
00642
00643 /**
00644 * @brief Get the recovery of gormanium
00645 *
00646 * This function calculates the recovery of gormanium in the circuit
00647 * based on the product flow rates and the feed rates.
00648 *
00649 * @return The recovery of gormanium
00650 *
00651 */
00652 double Circuit::get_gormanium_recovery() const
00653 {
00654     double total_feed = feed_gormanium_rate;
00655     double recovered = gormanium_product_gormanium;
00656     if (total_feed < 1e-12)
00657         return 0.0;
00658     return recovered / total_feed;
00659 }
00660
00661 /**
00662 * @brief Get the grade of palusznium
00663 *
00664 * This function calculates the grade of palusznium in the circuit
00665 * based on the product flow rates.
00666 *
00667 * @return The grade of palusznium
00668 *
00669 */
00670 double Circuit::get_palusznium_grade() const
00671 {
00672     double total = palusznium_product_palusznium + palusznium_product_gormanium +
00673     palusznium_product_waste;
00674     return (total > 0) ? (palusznium_product_palusznium / total) : 0.0;
00675 }
00676 /**
00677 * @brief Get the grade of gormanium
00678 *
00679 * This function calculates the grade of gormanium in the circuit
00680 * based on the product flow rates.
00681 *
00682 * @return The grade of gormanium
00683 *
00684 */
00685 double Circuit::get_gormanium_grade() const
00686 {
00687     double total = gormanium_product_palusznium + gormanium_product_gormanium +
00688     gormanium_product_waste;

```

```

00688     return (total > 0) ? (gormanium_product_gormanium / total) : 0.0;
00689 }
00690
00691 /**
00692 * @brief Export the circuit to a DOT file
00693 *
00694 * This function exports the circuit to a DOT file for visualization.
00695 *
00696 * @param filename The name of the output DOT file
00697 *
00698 * @return true if export is successful, false otherwise
00699 */
00700 bool Circuit::export_to_dot(const std::string& filename) const
00701 {
00702     std::ofstream ofs(filename);
00703     if (!ofs)
00704         return false;
00705     ofs << "digraph Circuit {\n";
00706     for (size_t i = 0; i < units.size(); ++i)
00707     {
00708         ofs << "    unit" << i << "[label="Unit " << i << ""];\n";
00709         // concentrate flow
00710         if (units[i].conc_num >= 0)
00711             ofs << "    unit" << i << " -> unit" << units[i].conc_num << "[label="conc\"];\\n";
00712         else if (units[i].conc_num == PALUSZNIUM_PRODUCT)
00713             ofs << "    unit" << i << " -> palusznium_product [label="conc\"];\\n";
00714         else if (units[i].conc_num == GORMANIUM_PRODUCT)
00715             ofs << "    unit" << i << " -> gormanium_product [label="conc\"];\\n";
00716         else if (units[i].conc_num == TAILINGS_OUTPUT)
00717             ofs << "    unit" << i << " -> tailings [label="conc\"];\\n";
00718         // tailings flow
00719         if (units[i].tails_num >= 0)
00720             ofs << "    unit" << i << " -> unit" << units[i].tails_num << "[label="tails\"];\\n";
00721         else if (units[i].tails_num == PALUSZNIUM_PRODUCT)
00722             ofs << "    unit" << i << " -> palusznium_product [label="tails\"];\\n";
00723         else if (units[i].tails_num == GORMANIUM_PRODUCT)
00724             ofs << "    unit" << i << " -> gormanium_product [label="tails\"];\\n";
00725         else if (units[i].tails_num == TAILINGS_OUTPUT)
00726             ofs << "    unit" << i << " -> tailings [label="tails\"];\\n";
00727     }
00728     ofs << "    palusznium_product [shape=box, label="Palusznium Product\"];\\n";
00729     ofs << "    gormanium_product [shape=box, label="Gormanium Product\"];\\n";
00730     ofs << "    tailings [shape=box, label="Tailings\"];\\n";
00731     ofs << "}\n";
00732     return true;
00733 }
00734
00735 /**
00736 * @brief Get the terminal mask for a given unit
00737 *
00738 * This function calculates the terminal mask for a given unit. It uses
00739 * breadth-first search to traverse the circuit and find the terminals.
00740 *
00741 * @param start The starting unit number
00742 *
00743 * @return The terminal mask
00744 */
00745 uint8_t Circuit::term_mask(int start) const
00746 {
00747     uint8_t mask = 0;
00748     std::vector<bool> visited(n, false);
00749
00750     // Use queue for breadth-first search
00751     std::queue<int> q;
00752     q.push(start);
00753     visited[start] = true;
00754
00755     while (!q.empty())
00756     {
00757         int current = q.front();
00758         q.pop();
00759
00760         const int conc_dest = units[current].conc_num;
00761         const int tail_dest = units[current].tails_num;
00762
00763         process_destination(conc_dest, mask, visited, q);
00764         process_destination(tail_dest, mask, visited, q);
00765
00766         if ((mask & (mask - 1)) >= 3)
00767             break;
00768     }
00769
00770     return mask;
00771 }
00772
00773 /**
00774 * @brief Process the destination unit

```

```

00775 *
00776 * This function processes the destination unit and updates the mask
00777 * accordingly. It also adds the destination unit to the queue for further
00778 * processing.
00779 *
00780 * @param dest The destination unit number
00781 * @param mask The terminal mask
00782 * @param visited Vector to keep track of visited units
00783 * @param q Queue for breadth-first search
00784 */
00785 void Circuit::process_destination(int dest, uint8_t& mask, std::vector<bool>& visited,
00786   std::queue<int>& q) const
00787 {
00788     if (dest >= n)
00789     {
00790         if (dest == OUT_P1())
00791             mask |= 0b001;
00792         else if (dest == OUT_P2())
00793             mask |= 0b010;
00794         else if (dest == OUT_TA())
00795             mask |= 0b100;
00796     }
00797     else
00798     {
00799         if (!visited[dest])
00800         {
00801             visited[dest] = true;
00802             q.push(dest);
00803         }
00804     }
00805
00806 /**
00807 * @brief Save the circuit output information to a CSV file
00808 *
00809 * This function saves the circuit output information to a CSV file.
00810 * It appends the data to the file if it already exists.
00811 *
00812 * @param filename The name of the output CSV file
00813 *
00814 * @return true if saving is successful, false otherwise
00815 */
00816 bool Circuit::save_all_units_to_csv(const std::string& filename)
00817 {
00818     std::ofstream ofs(filename, std::ios::app);
00819     if (!ofs.is_open())
00820     {
00821         std::cerr << "Error: Unable to open file " << filename << std::endl;
00822         return false;
00823     }
00824
00825 // output in a single line
00826 ofs << std::fixed << std::setprecision(2);
00827
00828 for (size_t i = 0; i < units.size(); ++i)
00829 {
00830     const CUnit& unit = units[i];
00831
00832     ofs << unit.conc_palusznium + unit.conc_germanium + unit.conc_waste << ","
00833     << unit.tails_palusznium + unit.tails_germanium + unit.tails_waste;
00834
00835     if (i < units.size() - 1)
00836     {
00837         ofs << ",";
00838     }
00839 }
00840 ofs << "\n";
00841
00842 ofs.close();
00843 return true;
00844 }
00845
00846 /**
00847 * @brief Save the circuit output information to a CSV file
00848 *
00849 * This function saves the circuit output information to a CSV file.
00850 * It appends the data to the file if it already exists.
00851 *
00852 * @param filename The name of the output CSV file
00853 *
00854 * @return true if saving is successful, false otherwise
00855 */
00856 bool Circuit::save_vector_to_csv(const std::string& filename)
00857 {
00858     std::ofstream ofs(filename, std::ios::app);
00859     if (!ofs.is_open())
00860     {

```

```

00861         std::cerr << "Error: Unable to open file " << filename << std::endl;
00862         return false;
00863     }
00864
00865     int length = static_cast<int>(units.size()) * 2 + 1;
00866
00867     for (int i = 0; i < length; ++i)
00868     {
00869         ofs << circuit_vector[i];
00870         if (i < length - 1)
00871         {
00872             ofs << ",";
00873         }
00874     }
00875     ofs << "\n";
00876
00877     ofs.close();
00878     return true;
00879 }
00880
00881 /**
00882 * @brief Save the circuit output information to a CSV file
00883 *
00884 * This function saves the circuit output information to a CSV file.
00885 * It appends the data to the file if it already exists.
00886 *
00887 * @param filename The name of the output CSV file
00888 *
00889 * @return true if saving is successful, false otherwise
00890 */
00891 bool Circuit::save_output_info(const std::string& filename)
00892 {
00893     namespace fs = std::filesystem; // If you see an error here it is because of the C++ version, it
00894     will compile fine
00895     fs::path p(filename);
00896
00897     if (p.has_parent_path())
00898     {
00899         std::error_code ec;
01000         fs::create_directories(p.parent_path(), ec); // ok if it already exists
01001         if (ec)
01002         {
01003             std::cerr << "Cannot create directory " << p.parent_path() << " : " << ec.message() << '\n';
01004             return false;
01005         }
01006     }
01007
01008     // truncate the file, then append the two blocks of data
01009     std::ofstream{filename, std::ios::trunc};
01010     return save_vector_to_csv(filename) && save_all_units_to_csv(filename);
01011 }

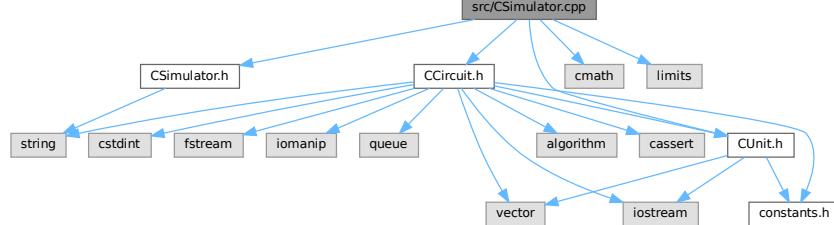
```

7.18 src/CSimulator.cpp File Reference

C++ source file for the circuit simulator.

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

Include dependency graph for CSimulator.cpp:



Functions

- double `circuit_performance` (int vector_size, int *circuit_vector, int unit_parameters_size, double *unit_parameters, struct `Simulator_Parameters` simulator_parameters, bool testFlag)

Evaluate the circuit performance.
- double `circuit_performance` (int vector_size, int *circuit_vector, int unit_parameters_size, double *unit_parameters)
- double `circuit_performance` (int vector_size, int *circuit_vector)
- double `circuit_performance` (int vector_size, int *circuit_vector, int unit_parameters_size, double *unit_parameters, bool testFlag)
- double `circuit_performance` (int vector_size, int *circuit_vector, bool testFlag)

Variables

- struct `Simulator_Parameters default_simulator_parameters` = {1e-6, 100}

7.18.1 Detailed Description

C++ source file for the circuit simulator.

Author

This source file contains the implementation of the function that will be used to evaluate the circuit and the parameters for the simulation.

Definition in file [CSimulator.cpp](#).

7.18.2 Function Documentation

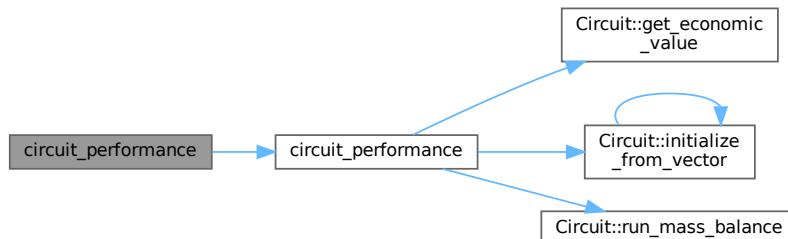
`circuit_performance()` [1/5]

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

Definition at line 79 of file [CSimulator.cpp](#).

References `circuit_performance()`, and `default_simulator_parameters`.

Here is the call graph for this function:

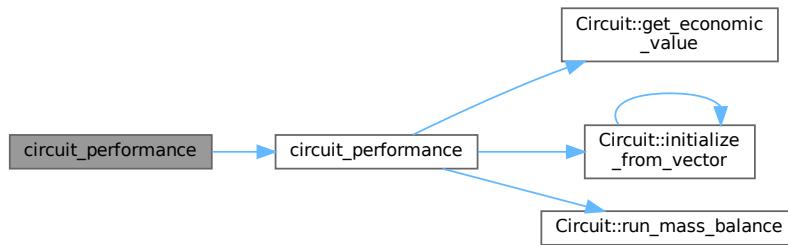


`circuit_performance()` [2/5]

```
double circuit_performance (
    int vector_size,
    int * circuit_vector,
    bool testFlag )
```

Definition at line 95 of file [CSimulator.cpp](#).

References [circuit_performance\(\)](#), and [default_simulator_parameters](#).
 Here is the call graph for this function:



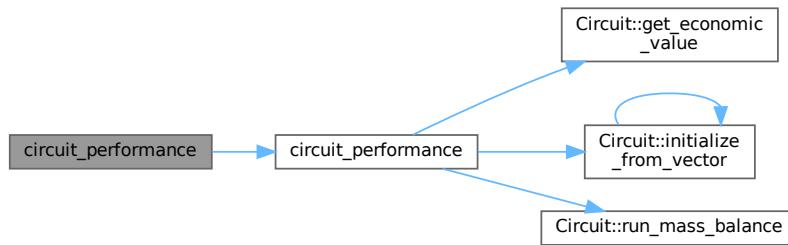
circuit_performance() [3/5]

```
double circuit_performance (
    int vector_size,
    int * circuit_vector,
    int unit_parameters_size,
    double * unit_parameters )
```

Definition at line 73 of file [CSimulator.cpp](#).

References [circuit_performance\(\)](#), and [default_simulator_parameters](#).

Here is the call graph for this function:



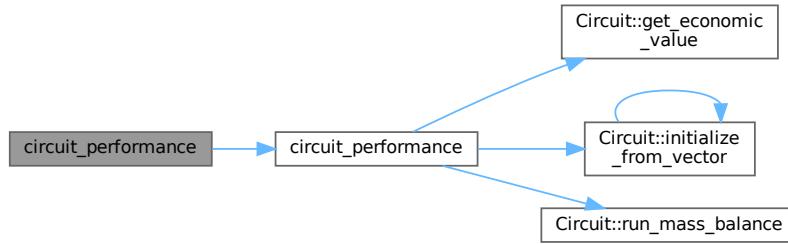
circuit_performance() [4/5]

```
double circuit_performance (
    int vector_size,
    int * circuit_vector,
    int unit_parameters_size,
    double * unit_parameters,
    bool testFlag )
```

Definition at line 88 of file [CSimulator.cpp](#).

References [circuit_performance\(\)](#), and [default_simulator_parameters](#).

Here is the call graph for this function:



circuit_performance() [5/5]

```
double circuit_performance (
    int vector_size,
    int * circuit_vector,
    int unit_parameters_size,
    double * unit_parameters,
    struct Simulator_Parameters simulator_parameters,
    bool testFlag )
```

Evaluate the circuit performance.

This function evaluates the performance of the circuit based on the circuit vector and the unit parameters. It initializes the circuit, runs the mass balance, and returns the economic value of the circuit.

Parameters

<i>vector_size</i>	Size of the circuit vector
<i>circuit_vector</i>	Circuit vector
<i>unit_parameters_size</i>	Size of the unit parameters
<i>unit_parameters</i>	Unit parameters
<i>simulator_parameters</i>	Simulation parameters
<i>testFlag</i>	Test flag to indicate whether to use test parameters

Returns

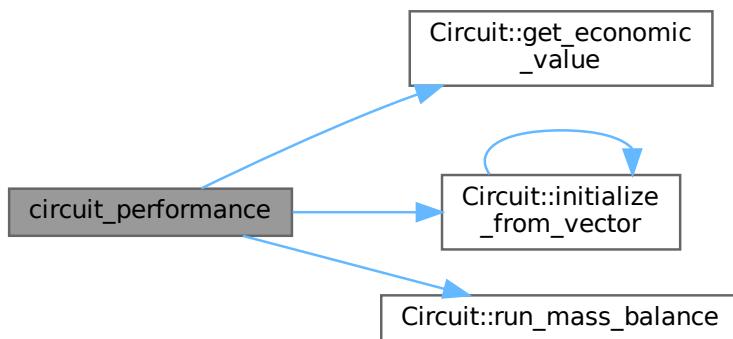
Economic value of the circuit

Definition at line 36 of file [CSimulator.cpp](#).

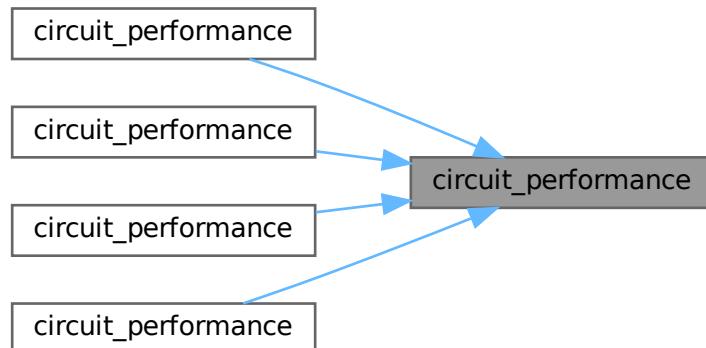
References [Circuit::get_economic_value\(\)](#), [Circuit::initialize_from_vector\(\)](#), [Simulator_Parameters::max_iterations](#), [Circuit::run_mass_balance\(\)](#), and [Simulator_Parameters::tolerance](#).

Referenced by [circuit_performance\(\)](#), [circuit_performance\(\)](#), [circuit_performance\(\)](#), and [circuit_performance\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**7.18.3 Variable Documentation****default_simulator_parameters**

```
struct Simulator_Parameters default_simulator_parameters = {1e-6, 100}
```

Definition at line 17 of file [CSimulator.cpp](#).

Referenced by [circuit_performance\(\)](#), [circuit_performance\(\)](#), [circuit_performance\(\)](#), and [circuit_performance\(\)](#).

7.19 CSimulator.cpp

[Go to the documentation of this file.](#)

```

00001 /**
00002 * @file CSimulator.cpp
00003 * @brief C++ source file for the circuit simulator
00004 * @author
00005 *
00006 * This source file contains the implementation of the function that will be
00007 * used to evaluate the circuit and the parameters for the simulation.
00008 *
00009 */
00010 #include "CSimulator.h"
00011 #include "CCircuit.h"
00012 #include "CUnit.h"
00013 #include <cmath>
00014 #include <limits>
00015
00016 // Default simulation parameters
00017 struct Simulator_Parameters default_simulator_parameters = {1e-6, 100};
00018
00019 /**
00020 * @brief Evaluate the circuit performance
00021 *
00022 * This function evaluates the performance of the circuit based on the
00023 * circuit vector and the unit parameters. It initializes the circuit,
00024 * runs the mass balance, and returns the economic value of the circuit.
00025 *
00026 * @param vector_size Size of the circuit vector
00027 * @param circuit_vector Circuit vector
00028 * @param unit_parameters_size Size of the unit parameters
00029 * @param unit_parameters Unit parameters
00030 * @param simulator_parameters Simulation parameters
00031 * @param testFlag Test flag to indicate whether to use test parameters
00032 *
00033 * @return Economic value of the circuit
00034 *
00035 */
00036 double circuit_performance(int vector_size, int* circuit_vector,
00037
00038                         int unit_parameters_size, double* unit_parameters,
00039                         struct Simulator_Parameters simulator_parameters, bool testFlag)
00040 {
00041
00042     // Calculate the number of units
00043     int num_units = (vector_size - 1) / 2;
00044     // Check if the vector size is valid
00045     if (vector_size != 2 * num_units + 1 || num_units <= 0)
00046     {
00047         // Invalid vector size
00048         return -1e12;
00049     }
00050
00051     // Initialize the circuit
00052     Circuit circuit(num_units, unit_parameters, testFlag);
00053     if (!circuit.initialize_from_vector(vector_size, circuit_vector, unit_parameters, testFlag))
00054     {
00055         // Invalid structure
00056         return -1e12;
00057     }
00058
00059     // Run the mass balance
00060     bool converged = circuit.run_mass_balance(simulator_parameters.tolerance,
00061                                               simulator_parameters.max_iterations);
00062     if (!converged)
00063     {
00064         // Not converged, consider invalid
00065         return -1e12;
00066     }
00067
00068     // Return performance
00069     return circuit.get_economic_value();
00070 }
00071 // Overloads for other input
00072
00073 double circuit_performance(int vector_size, int* circuit_vector, int unit_parameters_size, double*
00074                             unit_parameters)
00075 {
00076     return circuit_performance(vector_size, circuit_vector, unit_parameters_size, unit_parameters,
00077                               default_simulator_parameters, false);
00078 }
00079 double circuit_performance(int vector_size, int* circuit_vector)
00080 {
00081     int num_parameters = (vector_size - 1) / 2;
00082     double result =

```

```

00083     circuit_performance(vector_size, circuit_vector, num_parameters, nullptr,
00084     default_simulator_parameters, false);
00085     return result;
00086 }
00087
00088 double circuit_performance(int vector_size, int* circuit_vector, int unit_parameters_size, double*
00089     unit_parameters,
00090     bool testFlag)
00091 {
00092     return circuit_performance(vector_size, circuit_vector, unit_parameters_size, unit_parameters,
00093     default_simulator_parameters, testFlag);
00094 }
00095 double circuit_performance(int vector_size, int* circuit_vector, bool testFlag)
00096 {
00097     int num_parameters = (vector_size - 1) / 2;
00098     double* parameters = new double[num_parameters];
00099     for (int i = 0; i < num_parameters; i++)
00100     {
00101         parameters[i] = 1.0;
00102     }
00103     double result = circuit_performance(vector_size, circuit_vector, num_parameters, nullptr,
00104     default_simulator_parameters, testFlag);
00105
00106     // Clean up
00107     delete[] parameters;
00108     return result;
00109 }

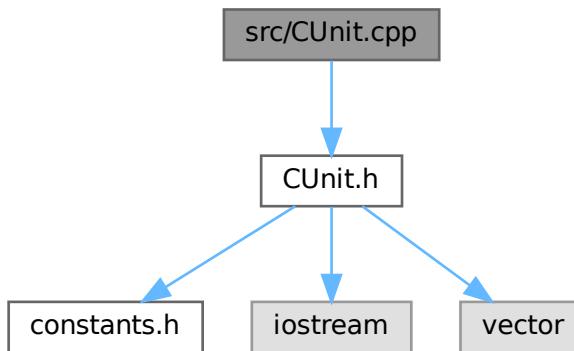
```

7.20 src/CUnit.cpp File Reference

Implementation of the [CUnit](#) class.

```
#include "CUnit.h"
```

Include dependency graph for CUnit.cpp:



7.20.1 Detailed Description

Implementation of the [CUnit](#) class.

This file contains the implementation of the [CUnit](#) class, which represents a single separation unit in a mineral-processing circuit. The class includes methods for processing the unit, updating its volume, and calculating recoveries for different components.

Definition in file [CUnit.cpp](#).

7.21 CUnit.cpp

[Go to the documentation of this file.](#)

```
00001 /**
00002 * @file CUnit.cpp
```

```

00003 * @brief Implementation of the CUnit class
00004 *
00005 * This file contains the implementation of the CUnit class, which represents
00006 * a single separation unit in a mineral-processing circuit. The class
00007 * includes methods for processing the unit, updating its volume, and
00008 * calculating recoveries for different components.
00009 *
00010 */
00011 #include "CUnit.h"
00012
00013 /**
00014 * @brief Constructors for the CUnit class
00015 *
00016 */
00017 CUnit::CUnit()
00018     : conc_num(0), tails_num(0), mark(false), volume(10.0), feed_palusznium(0.0), feed_germanium(0.0),
00019         k_palusznium(0.008), k_germanium(0.004), k_waste(0.0005), conc_palusznium(0.0),
00020         conc_germanium(0.0),
00021         conc_waste(0.0), rho(0.0), phi(0.0), tails_palusznium(0.0), tails_germanium(0.0),
00022         tails_waste(0.0), V_min(2.5),
00023         V_max(20.0)
00024 {
00025     CUnit::CUnit(int conc, int tails)
00026         : conc_num(conc), tails_num(tails), mark(false), volume(Constants::Circuit::DEFAULT_UNIT_VOLUME),
00027             feed_palusznium(0.0), feed_germanium(0.0), feed_waste(0.0),
00028             k_palusznium(Constants::Physical::K_PALUSZNIUM),
00029             k_germanium(Constants::Physical::K_GORMANIUM), k_waste(Constants::Physical::K_WASTE),
00030             rho(Constants::Physical::MATERIAL_DENSITY), phi(Constants::Physical::SOLIDS_CONTENT),
00031             conc_palusznium(0.0),
00032             conc_germanium(0.0), conc_waste(0.0), tails_palusznium(0.0), tails_germanium(0.0),
00033             tails_waste(0.0),
00034             V_min(Constants::Circuit::MIN_UNIT_VOLUME), V_max(Constants::Circuit::MAX_UNIT_VOLUME)
00035 {
00036     CUnit::CUnit(int conc, int tails, bool testFlag)
00037         : conc_num(conc), tails_num(tails), mark(false), volume(Constants::Circuit::DEFAULT_UNIT_VOLUME),
00038             feed_palusznium(0.0), feed_germanium(0.0), feed_waste(0.0),
00039             k_palusznium(Constants::Physical::K_PALUSZNIUM),
00040             k_germanium(Constants::Physical::K_GORMANIUM), k_waste(Constants::Physical::K_WASTE),
00041             rho(Constants::Physical::MATERIAL_DENSITY), phi(Constants::Physical::SOLIDS_CONTENT),
00042             conc_palusznium(0.0),
00043             conc_germanium(0.0), conc_waste(0.0), tails_palusznium(0.0), tails_germanium(0.0),
00044             tails_waste(0.0),
00045             V_min(Constants::Circuit::MIN_UNIT_VOLUME), V_max(Constants::Circuit::MAX_UNIT_VOLUME)
00046 {
00047     if (testFlag)
00048     {
00049         this->k_palusznium = Constants::Test::K_PALUSZNIUM;
00050         this->k_germanium = Constants::Test::K_GORMANIUM;
00051         this->k_waste = Constants::Test::K_WASTE;
00052         this->rho = Constants::Test::MATERIAL_DENSITY;
00053         this->phi = Constants::Test::SOLIDS_CONTENT;
00054         this->V_min = Constants::Test::MIN_UNIT_VOLUME;
00055         this->V_max = Constants::Test::MAX_UNIT_VOLUME;
00056         this->volume = Constants::Test::DEFAULT_UNIT_VOLUME;
00057     }
00058 /**
00059 * @brief Process the unit
00060 * This function processes the unit by calculating the residence time,
00061 * recoveries, and splitting the feed into products.
00062 */
00063 */
00064 void CUnit::process()
00065 {
00066     /* ----- 1. Residence time ----- */
00067
00068     // total solids feed (kg/s)
00069     const double Ftot = feed_palusznium + feed_germanium + feed_waste;
00070     // guard against division-by-zero / vanishing flow
00071     const double minFlow = 1e-10;
00072     const double tau = phi * this->volume / (std::max(Ftot, minFlow) / rho);
00073
00074     /* ----- 2. Recoveries R_i^C ----- */
00075     Rp = k_palusznium * tau / (1.0 + k_palusznium * tau);
00076     Rg = k_germanium * tau / (1.0 + k_germanium * tau);
00077     Rw = k_waste * tau / (1.0 + k_waste * tau);
00078
00079     /* ----- 3. Split feed into products --- */
00080     // Palusznium

```

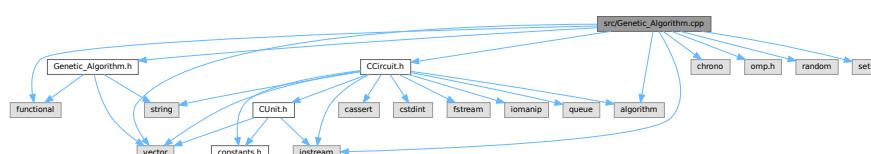
```
00081     conc_palusznium = feed_palusznium * Rp;
00082     tails_palusznium = feed_palusznium - conc_palusznium;
00083
00084     // Gormanium
00085     conc_germanium = feed_germanium * Rg;
00086     tails_germanium = feed_germanium - conc_germanium;
00087
00088     // Waste
00089     conc_waste = feed_waste * Rw;
00090     tails_waste = feed_waste - conc_waste;
00091 }
00092
00093 /**
00094 * @brief Update the volume of the unit
00095 *
00096 * This function updates the volume of the unit based on the given beta
00097 * value.
00098 *
00099 * @param beta The beta value to update the volume
00100 *
00101 */
00102 void CUnit::update_volume(double beta)
00103 {
00104     this->volume = this->V_min + (this->V_max - this->V_min) * beta;
00105 }
```

7.22 src/Genetic_Algorithm.cpp File Reference

Genetic Algorithm Implementation.

```
#include "Genetic_Algorithm.h"
#include "CCircuit.h"
#include <algorithm>
#include <chrono>
#include <functional>
#include <iostream>
#include <omp.h>
#include <random>
#include <set>
#include <vector>
```

Include dependency graph for Genetic_Algorithm.cpp:



Functions

- void `set_random_seed` (int seed)
Set the random seed for the random number generator.
 - static std::mt19937 & `rng` ()
 - std::vector< int > `generate_valid_circuit_template` (int num_units)
Generate a valid circuit template.
 - std::vector< int > `create_varied_circuit` (const std::vector< int > &template_vec, int num_units, std::function< bool(int, int *)> validity_check)
Create a varied circuit based on a template.
 - std::vector< std::vector< int > > `generate_initial_population` (int population_size, int num_units, std::function< bool(int, int *)> validity_check)
Generate an initial population of valid circuits.
 - bool `all_true` (int iv, int *ivs, int rv, double *rvs)
Check if all values in the vector are true.

- bool [all_true_ints](#) (int iv, int *ivs)
- bool [all_true_reals](#) (int iv, double *rvs)
- [OptimizationResult get_last_optimization_result](#) ()
Get the last optimization result.
- int [optimize](#) (int int_vector_size, int *int_vector, std::function< double(int, int *)> func, std::function< bool(int, int *)> validity, [Algorithm_Parameters](#) params)
Optimize a discrete vector using a genetic algorithm.
- int [optimize](#) (int real_vector_size, double *real_vector, std::function< double(int, double *)> func, std::function< bool(int, double *)> validity, [Algorithm_Parameters](#) params)
Optimize a continuous vector using a genetic algorithm.
- int [optimize](#) (int int_vector_size, int *int_vector, int real_vector_size, double *real_vector, std::function< double(int, int *, int, double *)> hybrid_func, std::function< bool(int, int *, int, double *)> hybrid_validity, [Algorithm_Parameters](#) params)
Optimize a mixed discrete-continuous vector using a genetic algorithm.

Variables

- static int [g_random_seed](#) = -1
- static [OptimizationResult](#) [last_result](#)

7.22.1 Detailed Description

Genetic Algorithm Implementation.

This file contains the implementation of the genetic algorithm for optimizing the circuit design. It includes functions for generating valid circuits, evaluating fitness, and performing genetic operations such as selection, crossover, and mutation.

Definition in file [Genetic_Algorithm.cpp](#).

7.22.2 Function Documentation

[all_true\(\)](#)

```
bool all_true (
    int iv,
    int * ivs,
    int rv,
    double * rvs )
```

Check if all values in the vector are true.

This function checks if all values in the vector are true.

Parameters

<i>iv</i>	Size of the vector
<i>ivs</i>	Pointer to the vector
<i>rv</i>	Size of the real vector
<i>rvs</i>	Pointer to the real vector

Definition at line 315 of file [Genetic_Algorithm.cpp](#).

[all_true_ints\(\)](#)

```
bool all_true_ints (
    int iv,
    int * ivs )
```

Definition at line 319 of file [Genetic_Algorithm.cpp](#).

all_true_reals()

```
bool all_true_reals (
    int iv,
    double * rvs )
```

Definition at line 323 of file [Genetic_Algorithm.cpp](#).

create_varied_circuit()

```
std::vector< int > create_varied_circuit (
    const std::vector< int > & template_vec,
    int num_units,
    std::function< bool(int, int *)> validity_check )
```

Create a varied circuit based on a template.

This function creates a varied circuit based on a given template vector. It modifies the template by changing a few connections while ensuring the circuit remains valid.

Parameters

<i>template_vec</i>	The template vector to modify
<i>num_units</i>	Number of units in the circuit
<i>validity_check</i>	Function to check the validity of the circuit

Returns

A vector representing the varied circuit

Definition at line 119 of file [Genetic_Algorithm.cpp](#).

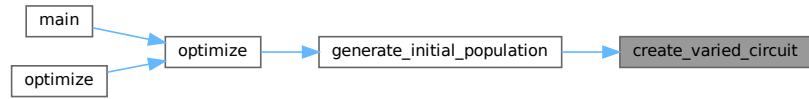
References [rng\(\)](#).

Referenced by [generate_initial_population\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**generate_initial_population()**

```
std::vector< std::vector< int > > generate_initial_population (
    int population_size,
```

```
int num_units,
std::function< bool(int, int *)> validity_check )
```

Generate an initial population of valid circuits.

This function generates an initial population of valid circuits based on a set of templates.

Parameters

<i>population_size</i>	Size of the population to generate
<i>num_units</i>	Number of units in the circuit
<i>validity_check</i>	Function to check the validity of the circuit

Returns

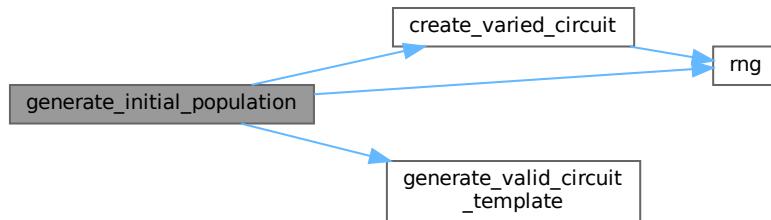
A vector of vectors representing the initial population

Definition at line 212 of file [Genetic_Algorithm.cpp](#).

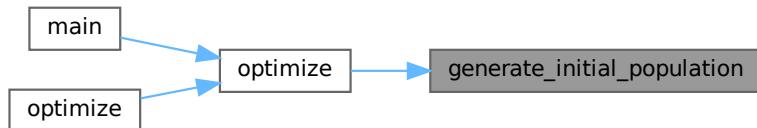
References [create_varied_circuit\(\)](#), [generate_valid_circuit_template\(\)](#), and [rng\(\)](#).

Referenced by [optimize\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



[generate_valid_circuit_template\(\)](#)

```
std::vector< int > generate_valid_circuit_template (
    int num_units )
```

Generate a valid circuit template.

This function generates a valid circuit template based on the number of units. It creates a vector of integers representing the circuit connections.

Parameters

<code>num_units</code>	Number of units in the circuit
------------------------	--------------------------------

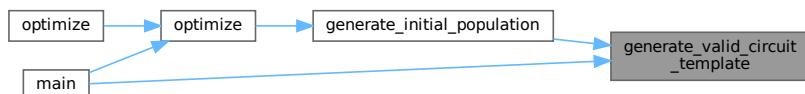
Returns

A vector of integers representing the circuit connections

Definition at line 59 of file [Genetic_Algorithm.cpp](#).

Referenced by [generate_initial_population\(\)](#), and [main\(\)](#).

Here is the caller graph for this function:

**get_last_optimization_result()**

`OptimizationResult get_last_optimization_result ()`

Get the last optimization result.

This function returns the last optimization result. This structure holds the results of the optimization process, including the best fitness, number of generations, average fitness, standard deviation of fitness, time taken, and convergence status.

Returns

The last optimization result

Definition at line 340 of file [Genetic_Algorithm.cpp](#).

References [last_result](#).

optimize() [1/3]

```

int optimize (
    int int_vector_size,
    int * int_vector,
    int real_vector_size,
    double * real_vector,
    std::function< double(int, int *, int, double *)> hybrid_func,
    std::function< bool(int, int *, int, double *)> hybrid_validity,
    Algorithm_Parameters params )

```

Optimize a mixed discrete-continuous vector using a genetic algorithm.

This function optimizes a mixed discrete-continuous vector using a genetic algorithm. It evaluates the fitness of the population in parallel and applies selection, crossover, and mutation to generate new populations.

Parameters

<code>int_vector_size</code>	Size of the integer vector
<code>int_vector</code>	Pointer to the integer vector
<code>real_vector_size</code>	Size of the real vector
<code>real_vector</code>	Pointer to the real vector
<code>hybrid_func</code>	Function to evaluate the fitness of the circuit
<code>hybrid_validity</code>	Function to check the validity of the circuit
<code>params</code>	Algorithm parameters for the optimization process

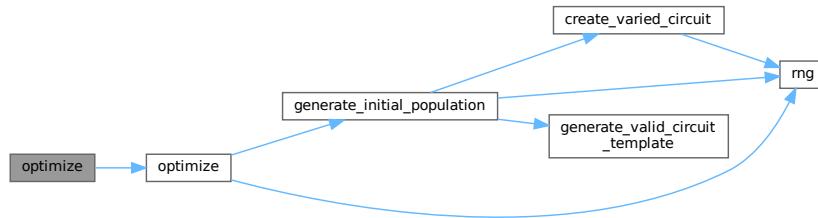
Returns

The best fitness value found during optimization

Definition at line 836 of file [Genetic_Algorithm.cpp](#).

References [optimize\(\)](#).

Here is the call graph for this function:

**optimize() [2/3]**

```
int optimize (
    int int_vector_size,
    int * int_vector,
    std::function< double(int, int *)> func,
    std::function< bool(int, int *)> validity,
    Algorithm_Parameters params )
```

Optimize a discrete vector using a genetic algorithm.

This function optimizes a discrete vector using a genetic algorithm. It evaluates the fitness of the population in parallel and applies selection, crossover, and mutation to generate new populations.

Parameters

<i>int_vector_size</i>	Size of the integer vector
<i>int_vector</i>	Pointer to the integer vector
<i>func</i>	Function to evaluate the fitness of the circuit
<i>validity</i>	Function to check the validity of the circuit
<i>params</i>	Algorithm parameters for the optimization process

Returns

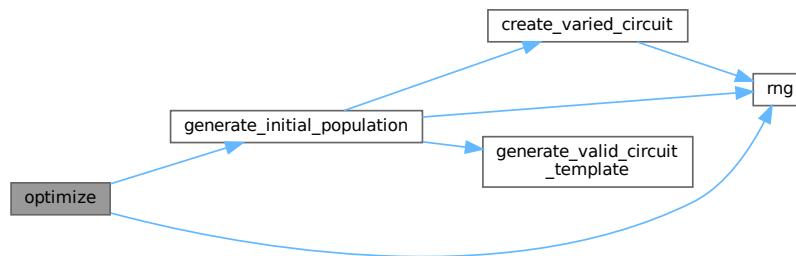
The best fitness value found during optimization

Definition at line 364 of file [Genetic_Algorithm.cpp](#).

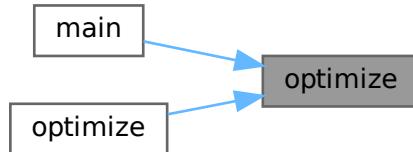
References [OptimizationResult::best_fitness](#), [Algorithm_Parameters::convergence_threshold](#), [Algorithm_Parameters::crossover_prob](#), [generate_initial_population\(\)](#), [OptimizationResult::generations](#), [Algorithm_Parameters::inversion_probability](#), [last_result](#), [Algorithm_Parameters::max_iterations](#), [Algorithm_Parameters::mutation_probability](#), [Algorithm_Parameters::mutation_step](#), [Algorithm_Parameters::population_size](#), [rng\(\)](#), [Algorithm_Parameters::stall_generations](#), [Algorithm_Parameters::tournament_size](#), [Algorithm_Parameters::use_inversion](#), and [Algorithm_Parameters::verbose](#).

Referenced by [main\(\)](#), and [optimize\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**optimize() [3/3]**

```

int optimize (
    int real_vector_size,
    double * real_vector,
    std::function< double(int, double *)> func,
    std::function< bool(int, double *)> validity,
    Algorithm_Parameters params )
  
```

Optimize a continuous vector using a genetic algorithm.

This function optimizes a continuous vector using a genetic algorithm. It evaluates the fitness of the population in parallel and applies selection, crossover, and mutation to generate new populations.

Parameters

<code>real_vector_size</code>	Size of the real vector
<code>real_vector</code>	Pointer to the real vector

Parameters

<i>func</i>	Function to evaluate the fitness of the circuit
<i>validity</i>	Function to check the validity of the circuit
<i>params</i>	Algorithm parameters for the optimization process

Returns

The best fitness value found during optimization

Definition at line 620 of file [Genetic_Algorithm.cpp](#).

References `OptimizationResult::best_fitness`, `Algorithm_Parameters::convergence_threshold`, `Algorithm_Parameters::crossover_prob`, `OptimizationResult::generations`, `last_result`, `Algorithm_Parameters::max_iterations`, `Algorithm_Parameters::mutation_probability`, `Algorithm_Parameters::mutation_step_size`, `Algorithm_Parameters::population_size`, `rng()`, `Algorithm_Parameters::scaling_mutation`, `Algorithm_Parameters::scaling_mutation_min`, `Algorithm_Parameters::scaling_mutation_prob`, `Algorithm_Parameters::stall_generations`, `Algorithm_Parameters::tournament_size`, `Algorithm_Parameters::use_scaling_mutation`, and `Algorithm_Parameters::verbose`.

Here is the call graph for this function:



`rng()`

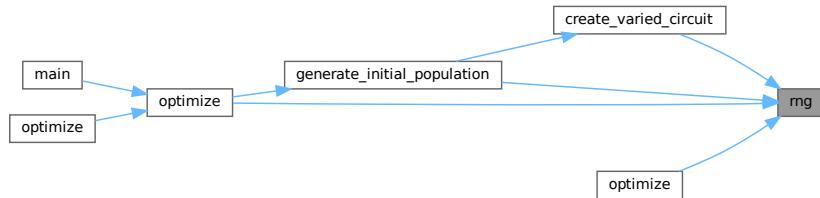
```
static std::mt19937 & rng ( ) [static]
```

Definition at line 33 of file [Genetic_Algorithm.cpp](#).

References [g_random_seed](#).

Referenced by `create_varied_circuit()`, `generate_initial_population()`, `optimize()`, and `optimize()`.

Here is the caller graph for this function:



`set_random_seed()`

```
void set_random_seed (
    int seed )
```

Set the random seed for the random number generator.

Definition at line 27 of file [Genetic_Algorithm.cpp](#).

References [g_random_seed](#).

Referenced by `main()`.

Here is the caller graph for this function:



7.22.3 Variable Documentation

g_random_seed

```
int g_random_seed = -1 [static]
```

Definition at line 22 of file [Genetic_Algorithm.cpp](#).
Referenced by [rng\(\)](#), and [set_random_seed\(\)](#).

last_result

```
OptimizationResult last_result [static]
```

Definition at line 328 of file [Genetic_Algorithm.cpp](#).
Referenced by [get_last_optimization_result\(\)](#), [optimize\(\)](#), and [optimize\(\)](#).

7.23 Genetic_Algorithm.cpp

[Go to the documentation of this file.](#)

```

00001 /**
00002 * @file Genetic_Algorithm.cpp
00003 * @brief Genetic Algorithm Implementation
00004 *
00005 * This file contains the implementation of the genetic algorithm for optimizing
00006 * the circuit design. It includes functions for generating valid circuits,
00007 * evaluating fitness, and performing genetic operations such as selection,
00008 * crossover, and mutation.
00009 *
00010 */
00011 #include "Genetic_Algorithm.h"
00012 #include "CCircuit.h"
00013 #include <algorithm>
00014 #include <chrono>
00015 #include <functional>
00016 #include <iostream>
00017 #include <omp.h>
00018 #include <random>
00019 #include <set>
00020 #include <vector>
00021
00022 static int g_random_seed = -1; // -1 means use random seed
00023
00024 /**
00025 * @brief Set the random seed for the random number generator
00026 */
00027 void set_random_seed(int seed)
00028 {
00029     g_random_seed = seed;
00030 }
00031
00032 // Modified rng() function - properly thread-safe
00033 static std::mt19937& rng()
00034 {
00035     if (g_random_seed >= 0)
00036     {
00037         // Deterministic mode - use thread ID to ensure different seeds per thread
00038         static thread_local std::mt19937 gen(g_random_seed + omp_get_thread_num());
00039         return gen;
00040     }
00041     else
00042     {
00043         // Non-deterministic mode
00044         static thread_local std::mt19937 gen(std::random_device{}());
00045         return gen;
00046     }
00047 }

```

```

00046      }
00047  }
00048
00049 /**
00050 * @brief Generate a valid circuit template
00051 *
00052 * This function generates a valid circuit template based on the number of units.
00053 * It creates a vector of integers representing the circuit connections.
00054 *
00055 * @param num_units Number of units in the circuit
00056 *
00057 * @return A vector of integers representing the circuit connections
00058 */
00059 std::vector<int> generate_valid_circuit_template(int num_units)
00060 {
00061     const int n = num_units;
00062     const int vec_size = 2 * n + 1;
00063     std::vector<int> vec(vec_size);
00064
00065     // Set feed to unit 0 (most common valid configuration)
00066     vec[0] = 0;
00067
00068     // Basic linear flow pattern with some recycling
00069     for (int i = 0; i < n; i++)
00070     {
00071         // For the concentrate stream (high-grade)
00072         if (i < n - 1)
00073         {
00074             // Forward flow to next unit for most units
00075             vec[2 * i + 1] = i + 1;
00076         }
00077         else
00078         {
00079             // Last unit sends concentrate to Palusznium product (n)
00080             vec[2 * i + 1] = n;
00081         }
00082
00083         // For the tailings stream
00084         if (i % 3 == 0)
00085         {
00086             // Every 3rd unit sends tailings to Gormanium product
00087             vec[2 * i + 2] = n + 1;
00088         }
00089         else if (i % 3 == 1)
00090         {
00091             // Every 3rd+1 unit sends tailings to final tailings
00092             vec[2 * i + 2] = n + 2;
00093         }
00094         else
00095         {
00096             // Other units recycle tailings back to unit 0
00097             vec[2 * i + 2] = 0;
00098         }
00099     }
00100
00101     return vec;
00102 }
00103
00104 /**
00105 *
00106 * @brief Create a varied circuit based on a template
00107 *
00108 * This function creates a varied circuit based on a given template vector.
00109 * It modifies the template by changing a few connections while ensuring
00110 * the circuit remains valid.
00111 *
00112 * @param template_vec The template vector to modify
00113 * @param num_units Number of units in the circuit
00114 * @param validity_check Function to check the validity of the circuit
00115 *
00116 * @return A vector representing the varied circuit
00117 *
00118 */
00119 std::vector<int> create_varied_circuit(const std::vector<int>& template_vec, int num_units,
00120                                         std::function<bool(int, int*)> validity_check)
00121 {
00122     const int n = num_units;
00123     std::vector<int> result = template_vec;
00124
00125     // Apply a few random changes
00126     int num_changes = std::uniform_int_distribution<int>(1, n)(rng());
00127
00128     // Try multiple times to create a valid variation
00129     for (int attempt = 0; attempt < 20; attempt++)
00130     {
00131         std::vector<int> candidate = template_vec;
00132

```

```

00133     for (int i = 0; i < num_changes; i++)
00134     {
00135         // Pick a random position to modify (excluding feed position)
00136         int pos = std::uniform_int_distribution<int>(1, 2 * n)(rng());
00137
00138         // Determine which unit this connection belongs to
00139         int unit_idx = (pos - 1) / 2;
00140
00141         // Pick a valid destination (any unit or terminal except self)
00142         std::vector<int> valid_dests;
00143         for (int dest = 0; dest < n + 3; dest++)
00144         {
00145             if (dest != unit_idx)
00146                 // Avoid self-loop
00147                 valid_dests.push_back(dest);
00148         }
00149
00150
00151         // Shuffle possible destinations
00152         std::shuffle(valid_dests.begin(), valid_dests.end(), rng());
00153
00154         // Try each possible destination until we find one that works
00155         bool found_valid = false;
00156         for (int dest : valid_dests)
00157         {
00158             int old_val = candidate[pos];
00159             candidate[pos] = dest;
00160
00161             // Check if both connections from this unit point to the same place
00162             int other_conn = (pos % 2 == 1) ? pos + 1 : pos - 1;
00163             if (other_conn < candidate.size() && candidate[other_conn] == dest)
00164             {
00165                 candidate[pos] = old_val; // Restore old value
00166                 continue;           // Can't have both connections to same place
00167             }
00168
00169             // Check if the circuit is valid with this change
00170             if (validity_check(candidate.size(), candidate.data()))
00171             {
00172                 found_valid = true;
00173                 break;
00174             }
00175             else
00176             {
00177                 candidate[pos] = old_val; // Restore old value
00178             }
00179         }
00180
00181         if (!found_valid)
00182         {
00183             // If we couldn't find a valid change for this position, try another
00184             continue;
00185         }
00186     }
00187
00188     // Check if the final candidate is valid
00189     if (validity_check(candidate.size(), candidate.data()))
00190     {
00191         return candidate;
00192     }
00193 }
00194
00195     // If all attempts failed, return the original template
00196     return template_vec;
00197 }

00198 /**
00199 * @brief Generate an initial population of valid circuits
00200 *
00201 * This function generates an initial population of valid circuits
00202 * based on a set of templates.
00203 *
00204 * @param population_size Size of the population to generate
00205 * @param num_units Number of units in the circuit
00206 * @param validity_check Function to check the validity of the circuit
00207 *
00208 * @return A vector of vectors representing the initial population
00209 */
00210
00211 */
00212 std::vector<std::vector<int>> generate_initial_population(int population_size, int num_units,
00213                                         std::function<bool(int, int*)>
00214                                         validity_check)
00215 {
00216     std::vector<std::vector<int>> population;
00217     std::set<std::vector<int>> unique_circuits; // To ensure uniqueness
00218
00219     // Create base templates

```

```

00219     std::vector<std::vector<int>> templates;
00220
00221     // Template 1: Linear flow with recycling
00222     templates.push_back(generate_valid_circuit_template(num_units));
00223
00224     // Template 2: Alternating product outputs
00225     auto template2 = generate_valid_circuit_template(num_units);
00226     for (int i = 0; i < num_units; i++)
00227     {
00228         if (i % 2 == 0)
00229         {
00230             template2[2 * i + 1] = num_units;      // Even units send concentrate to Palusznium
00231             template2[2 * i + 2] = num_units + 2; // and tailings to final tailings
00232         }
00233         else
00234         {
00235             template2[2 * i + 1] = num_units + 1; // Odd units send concentrate to Gormanium
00236             template2[2 * i + 2] = 0;           // and tailings back to first unit
00237         }
00238     }
00239     if (validity_check(template2.size(), template2.data()))
00240     {
00241         templates.push_back(template2);
00242     }
00243
00244     // Template 3: Butterfly pattern
00245     auto template3 = generate_valid_circuit_template(num_units);
00246     for (int i = 0; i < num_units; i++)
00247     {
00248         if (i < num_units / 2)
00249         {
00250             template3[2 * i + 1] = i + num_units / 2; // First half feed to second half
00251             template3[2 * i + 2] = num_units + 2;    // Tailings to final tailings
00252         }
00253         else
00254         {
00255             template3[2 * i + 1] = num_units; // Second half to products
00256             template3[2 * i + 2] = num_units + 1;
00257         }
00258     }
00259     if (validity_check(template3.size(), template3.data()))
00260     {
00261         templates.push_back(template3);
00262     }
00263
00264     // Add templates directly to population
00265     for (const auto& tmpl : templates)
00266     {
00267         population.push_back(tmpl);
00268         unique_circuits.insert(tmpl);
00269     }
00270
00271     // Generate variations until we have enough unique circuits
00272     int max_attempts = population_size * 10;
00273     int attempts = 0;
00274
00275     std::cout << "Generating initial population of valid circuits..." << std::endl;
00276
00277     while (population.size() < population_size && attempts < max_attempts)
00278     {
00279         // Pick a random template
00280         const auto& tmpl = templates[std::uniform_int_distribution<int>(0, templates.size() -
1)(rng())];
00281
00282         // Create a variation
00283         auto candidate = create_varied_circuit(tmpl, num_units, validity_check);
00284
00285         // Check uniqueness
00286         if (unique_circuits.find(candidate) == unique_circuits.end())
00287         {
00288             population.push_back(candidate);
00289             unique_circuits.insert(candidate);
00290
00291             if (population.size() % 10 == 0)
00292             {
00293                 std::cout << "Generated " << population.size() << " valid circuits" << std::endl;
00294             }
00295         }
00296
00297         attempts++;
00298     }
00299
00300     std::cout << "Initial population: " << population.size() << " valid circuits" << std::endl;
00301
00302     return population;
00303 }
00304

```

```

00305 /**
00306 * @brief Check if all values in the vector are true
00307 *
00308 * This function checks if all values in the vector are true.
00309 *
00310 * @param iv Size of the vector
00311 * @param ivs Pointer to the vector
00312 * @param rv Size of the real vector
00313 * @param rvs Pointer to the real vector
00314 */
00315 bool all_true(int iv, int* ivs, int rv, double* rvs)
00316 {
00317     return true;
00318 }
00319 bool all_true_ints(int iv, int* ivs)
00320 {
00321     return true;
00322 }
00323 bool all_true_reals(int iv, double* rvs)
00324 {
00325     return true;
00326 }
00327
00328 static OptimizationResult last_result;
00329
00330 /**
00331 * @brief Get the last optimization result
00332 *
00333 * This function returns the last optimization result.
00334 * This structure holds the results of the optimization process,
00335 * including the best fitness, number of generations, average fitness,
00336 * standard deviation of fitness, time taken, and convergence status.
00337 *
00338 * @return The last optimization result
00339 */
00340 OptimizationResult get_last_optimization_result()
00341 {
00342     return last_result;
00343 }
00344
00345 // ****
00346 // 1) Discrete-only optimize with PARALLEL fitness evaluation
00347 // ****
00348
00349 /**
00350 * @brief Optimize a discrete vector using a genetic algorithm
00351 *
00352 * This function optimizes a discrete vector using a genetic algorithm.
00353 * It evaluates the fitness of the population in parallel and applies
00354 * selection, crossover, and mutation to generate new populations.
00355 *
00356 * @param int_vector_size Size of the integer vector
00357 * @param int_vector Pointer to the integer vector
00358 * @param func Function to evaluate the fitness of the circuit
00359 * @param validity Function to check the validity of the circuit
00360 * @param params Algorithm parameters for the optimization process
00361 *
00362 * @return The best fitness value found during optimization
00363 */
00364 int optimize(int int_vector_size, int* int_vector, std::function<double(int, int*)> func,
00365                 std::function<bool(int, int*)> validity, Algorithm_Parameters params)
00366 {
00367     using Clock = std::chrono::high_resolution_clock;
00368     auto t0 = Clock::now();
00369
00370     // Print OpenMP info
00371     std::cout << "OpenMP: Using " << omp_get_max_threads() << " threads for parallel fitness evaluation"
00372     << std::endl;
00373     // --- 1. Improved population initialization
00374     int n_units = (int_vector_size - 1) / 2;
00375     std::cout << "Initializing population for " << n_units << " units..." << std::endl;
00376
00377     // Generate valid initial population
00378     std::vector<std::vector<int>> population = generate_initial_population(params.population_size,
00379     n_units, validity);
00380
00381     // If we couldn't generate enough valid circuits, adjust population size
00382     if (population.size() < params.population_size)
00383     {
00384         std::cout << "Warning: Could only generate " << population.size()
00385         << " valid circuits, adjusting population size" << std::endl;
00386         params.population_size = population.size();
00387     }
00388
00389     double best_overall = -1e300;           // best seen so far
00390     int stall_count = 0;                   // gens since last improvement

```

```

00390     double eps = params.convergence_threshold; // "meaningful" fitness delta
00391     int max_stall = params.stall_generations; // allowed idle generations
00392
00393     // --- 2. Main GA loop
00394     for (int gen = 0; gen < params.max_iterations; ++gen)
00395     {
00396         // 2a) PARALLEL fitness evaluation - THIS IS THE KEY OPTIMIZATION!
00397         std::vector<double> fitnesses(population.size());
00398
00399     // Parallel fitness evaluation using OpenMP
00400 #pragma omp parallel for schedule(dynamic)
00401         for (size_t i = 0; i < population.size(); ++i)
00402         {
00403             int* gdata = population[i].data();
00404             if (!validity(int_vector_size, gdata))
00405             {
00406                 fitnesses[i] = -1e9; // heavy penalty
00407             }
00408             else
00409             {
00410                 fitnesses[i] = func(int_vector_size, gdata);
00411             }
00412         }
00413
00414         double gen_best = *std::max_element(fitnesses.begin(), fitnesses.end());
00415         if (gen_best > best_overall + eps)
00416         {
00417             best_overall = gen_best;
00418             stall_count = 0; // reset when we see new best
00419         }
00420         else
00421         {
00422             stall_count++;
00423         }
00424         if (stall_count >= max_stall)
00425         {
00426             if (params.verbose)
00427             {
00428                 std::cout << "[GA] No improvement for " << stall_count << " generations--stopping
early.\n";
00429             }
00430             break; // exit the generation loop
00431         }
00432
00433     // 2b) Elitism: copy best genome to next generation
00434     std::vector<std::vector<int>> next_gen;
00435     {
00436         auto best_it = std::max_element(fitnesses.begin(), fitnesses.end());
00437         size_t best_idx = std::distance(fitnesses.begin(), best_it);
00438         next_gen.push_back(population[best_idx]);
00439     }
00440
00441     // ----- TOURNAMENT SETUP -----
00442     int k = params.tournament_size > 0 ? params.tournament_size : 2;
00443     std::uniform_int_distribution<size_t> pop_dist(0, population.size() - 1);
00444     auto pick_parent = [&]()
00445     {
00446         size_t best = pop_dist(rng());
00447         double best_fit = fitnesses[best];
00448         for (int i = 1; i < k; ++i)
00449         {
00450             size_t idx = pop_dist(rng());
00451             if (fitnesses[idx] > best_fit)
00452             {
00453                 best = idx;
00454                 best_fit = fitnesses[idx];
00455             }
00456         }
00457         return population[best];
00458     };
00459
00460     // 2c) Fill rest via selection, crossover, mutation
00461     std::uniform_real_distribution<double> u01(0.0, 1.0);
00462     while (next_gen.size() < population.size())
00463     {
00464         // - Selection via k-way tournament
00465         auto p1 = pick_parent();
00466         auto p2 = pick_parent();
00467
00468         // - Crossover
00469         std::vector<int> cl = p1, c2 = p2;
00470         if (u01(rng()) < params.crossover_probability)
00471         {
00472             // Adaptive crossover points: more early on, fewer later
00473             double progress = static_cast<double>(gen) / params.max_iterations;
00474             int max_points = std::min(5, int_vector_size / 2); // limit excessive cuts
00475             int num_cuts = static_cast<int>((1.0 - progress) * max_points);
00476
00477         }
00478     }
00479
00480     // Mutation
00481     for (auto& genome : next_gen)
00482     {
00483         for (size_t i = 0; i < genome.size(); ++i)
00484         {
00485             if (u01(rng()) < params.mutation_probability)
00486             {
00487                 genome[i] = rng();
00488             }
00489         }
00490     }
00491
00492     // Swap parents with children
00493     for (size_t i = 0; i < population.size(); ++i)
00494     {
00495         population[i] = next_gen[i];
00496     }
00497
00498     // Print progress
00499     if (gen % 10 == 0)
00500     {
00501         std::cout << "[GA] Generation " << gen << " complete." << std::endl;
00502     }
00503
00504 }

```

```

00476             num_cuts = std::max(1, num_cuts); // always at least 1 point
00477
00478             std::vector<bool> crossover_mask(int_vector_size, false);
00479             for (int i = 0; i < num_cuts; ++i)
00480             {
00481                 int cut = std::uniform_int_distribution<int>(0, int_vector_size - 1)(rng());
00482                 crossover_mask[cut] = true;
00483             }
00484
00485             bool flip = false;
00486             for (int j = 0; j < int_vector_size; ++j)
00487             {
00488                 if (crossover_mask[j])
00489                     flip = !flip;
00490                 if (flip)
00491                     std::swap(c1[j], c2[j]);
00492             }
00493
00494
00495             // - Mutation (creep + optional inversion)
00496             {
00497                 // 1) Substitution ("creep") mutation on both children
00498                 int min_gene = 0;
00499                 int max_gene = n_units + 2;
00500                 int range = max_gene - min_gene + 1;
00501                 std::uniform_int_distribution<int> step_dist(-params.mutation_step_size,
00502                     params.mutation_step_size);
00502                 for (auto* child : {&c1, &c2})
00503                 {
00504                     for (int j = 0; j < int_vector_size; ++j)
00505                     {
00506                         if (u01(rng()) < params.mutation_probability)
00507                         {
00508                             int step = step_dist(rng());
00509                             int val = (*child)[j] + step;
00510                             (*child)[j] = min_gene + ((val - min_gene) % range + range) % range;
00511                         }
00512                     }
00513                 }
00514
00515                 // 2) Inversion mutation, if enabled
00516                 if (params.use_inversion)
00517                 {
00518                     // pick two indices a < b
00519                     std::uniform_int_distribution<int> a_dist(0, int_vector_size - 2);
00520                     int a = a_dist(rng());
00521                     std::uniform_int_distribution<int> b_dist(a + 1, int_vector_size - 1);
00522                     int b = b_dist(rng());
00523
00524                     // reverse that slice in each child with its own probability
00525                     if (u01(rng()) < params.inversion_probability)
00526                     {
00527                         std::reverse(c1.begin() + a, c1.begin() + b + 1);
00528                     }
00529                     if (u01(rng()) < params.inversion_probability)
00530                     {
00531                         std::reverse(c2.begin() + a, c2.begin() + b + 1);
00532                     }
00533                 }
00534
00535
00536                 // Check validity of children and add valid ones
00537                 if (validity(int_vector_size, c1.data()))
00538                 {
00539                     next_gen.push_back(std::move(c1));
00540                 }
00541
00542                 if (next_gen.size() < population.size() && validity(int_vector_size, c2.data()))
00543                 {
00544                     next_gen.push_back(std::move(c2));
00545                 }
00546             }
00547
00548             // 2d) Replace population
00549             population.swap(next_gen);
00550
00551             if (params.verbose && gen % 10 == 0)
00552             {
00553                 std::cout << "[GA] Gen " << gen << " best fitness " << *std::max_element(fitnesses.begin(),
00554                     fitnesses.end())
00555                     << " (thread utilization: " << omp_get_max_threads() << " cores)" << "\n";
00556             }
00557
00558             // --- 3. Write best genome back into int_vector[]
00559             // (Re-evaluate final fitness to find the winner) - Also parallel!
00560             double best_fit = -1e12;

```

```

00561     size_t best_idx = 0;
00562     std::vector<double> final_fitnesses(population.size());
00563
00564 #pragma omp parallel for schedule(dynamic)
00565     for (size_t i = 0; i < population.size(); ++i)
00566     {
00567         final_fitnesses[i] = func(int_vector_size, population[i].data());
00568     }
00569
00570     // Find best (sequential)
00571     for (size_t i = 0; i < population.size(); ++i)
00572     {
00573         if (final_fitnesses[i] > best_fit)
00574         {
00575             best_fit = final_fitnesses[i];
00576             best_idx = i;
00577         }
00578     }
00579
00580     // Copy best solution
00581     for (int i = 0; i < int_vector_size; ++i)
00582     {
00583         int_vector[i] = population[best_idx][i];
00584     }
00585
00586     // Store optimization results
00587     last_result.best_fitness = best_fit;
00588     last_result.generations = params.max_iterations;
00589
00590     auto t1 = Clock::now();
00591     if (params.verbose)
00592     {
00593         double secs = std::chrono::duration<double>(t1 - t0).count();
00594         std::cout << "[GA] Completed in " << secs << "s, best_fitness=" << best_fit << " (using "
00595                     << omp_get_max_threads() << " parallel threads)" << "\n";
00596     }
00597
00598     return 0;
00599 }
00600
00601 // ****
00602 // 2) Continuous-only optimize with PARALLEL fitness evaluation
00603 // ****
00604
00605 /**
00606 * @brief Optimize a continuous vector using a genetic algorithm
00607 *
00608 * This function optimizes a continuous vector using a genetic algorithm.
00609 * It evaluates the fitness of the population in parallel and applies
00610 * selection, crossover, and mutation to generate new populations.
00611 *
00612 * @param real_vector_size Size of the real vector
00613 * @param real_vector Pointer to the real vector
00614 * @param func Function to evaluate the fitness of the circuit
00615 * @param validity Function to check the validity of the circuit
00616 * @param params Algorithm parameters for the optimization process
00617 *
00618 * @return The best fitness value found during optimization
00619 */
00620 int optimize(int real_vector_size, double* real_vector, std::function<double(int, double*)> func,
00621             std::function<bool(int, double*)> validity, Algorithm_Parameters params)
00622 {
00623     using Clock = std::chrono::high_resolution_clock;
00624     auto t0 = Clock::now();
00625
00626     std::cout << "OpenMP: Using " << omp_get_max_threads() << " threads for continuous optimization" <<
00627     std::endl;
00628     std::uniform_real_distribution<double> dist01(0.0, 1.0);
00629     std::vector<std::vector<double>> population;
00630
00631     // --- 1. Initialise population
00632     population.reserve(params.population_size);
00633     while (population.size() < params.population_size)
00634     {
00635         std::vector<double> genome(real_vector_size);
00636         for (auto& g : genome)
00637             g = dist01(rng()); // all _i in [0,1]
00638         if (validity(real_vector_size, genome.data()))
00639             population.push_back(std::move(genome));
00640     }
00641
00642     double best_overall = -1e300;
00643     int stall_count = 0;
00644     double eps = params.convergence_threshold;
00645     int max_stall = params.stall_generations;
00646

```

```

00647     for (int gen = 0; gen < params.max_iterations; ++gen)
00648     {
00649         // PARALLEL fitness evaluation
00650         std::vector<double> fitnesses(population.size());
00651
00652 #pragma omp parallel for schedule(dynamic)
00653     for (size_t i = 0; i < population.size(); ++i)
00654     {
00655         fitnesses[i] =
00656             validity(real_vector_size, population[i].data()) ? func(real_vector_size,
00657             population[i].data()) : -1e9;
00658     }
00659
00660     double gen_best = *std::max_element(fitnesses.begin(), fitnesses.end());
00661     if (gen_best > best_overall + eps)
00662     {
00663         best_overall = gen_best;
00664         stall_count = 0;
00665     }
00666     else
00667     {
00668         stall_count++;
00669     }
00670
00671     if (stall_count >= max_stall)
00672     {
00673         if (params.verbose)
00674             std::cout << "[GA-Real] No improvement for " << stall_count << " generations --"
00675             stopping.\n";
00676         break;
00677     }
00678
00679     // Elitism
00680     std::vector<std::vector<double>> next_gen;
00681     {
00682         auto best_it = std::max_element(fitnesses.begin(), fitnesses.end());
00683         size_t best_idx = std::distance(fitnesses.begin(), best_it);
00684         next_gen.push_back(population[best_idx]);
00685     }
00686
00687     // Tournament selection
00688     int k = params.tournament_size > 0 ? params.tournament_size : 2;
00689     std::uniform_int_distribution<size_t> pop_dist(0, population.size() - 1);
00690     auto pick_parent = [&]()
00691     {
00692         size_t best = pop_dist(rng());
00693         double best_fit = fitnesses[best];
00694         for (int i = 1; i < k; ++i)
00695         {
00696             size_t idx = pop_dist(rng());
00697             if (fitnesses[idx] > best_fit)
00698             {
00699                 best = idx;
00700                 best_fit = fitnesses[idx];
00701             }
00702         }
00703         return population[best];
00704     };
00705
00706     // Crossover + Mutation
00707     while (next_gen.size() < population.size())
00708     {
00709         auto p1 = pick_parent();
00710         auto p2 = pick_parent();
00711         std::vector<double> c1 = p1, c2 = p2;
00712
00713         if (dist01(rng()) < params.crossover_probability)
00714         {
00715             for (int j = 0; j < real_vector_size; ++j)
00716             {
00717                 if (dist01(rng()) < 0.5)
00718                     std::swap(c1[j], c2[j]);
00719             }
00720
00721         // Mutation
00722         for (int j = 0; j < real_vector_size; ++j)
00723         {
00724             if (dist01(rng()) < params.mutation_probability)
00725             {
00726                 double step = dist01(rng()) * params.mutation_step_size;
00727                 c1[j] = std::clamp(c1[j] + step * (dist01(rng()) < 0.5 ? -1 : 1), 0.0, 1.0);
00728             }
00729             if (dist01(rng()) < params.mutation_probability)
00730             {
00731                 double step = dist01(rng()) * params.mutation_step_size;
00732                 c2[j] = std::clamp(c2[j] + step * (dist01(rng()) < 0.5 ? -1 : 1), 0.0, 1.0);
00733             }
00734         }
00735     }
00736 }
```

```

00732         }
00733     }
00734
00735     // Optional: scaling mutation
00736     if (params.use_scaling_mutation)
00737     {
00738         std::uniform_int_distribution<int> idx_dist(0, real_vector_size - 1);
00739         std::uniform_real_distribution<double> scale_dist(params.scaling_mutation_min,
00740                                         params.scaling_mutation_max);
00741
00742         // Child 1
00743         if (dist01(rng()) < params.scaling_mutation_prob)
00744         {
00745             int idx = idx_dist(rng());
00746             double factor = scale_dist(rng());
00747             c1[idx] = std::clamp(c1[idx] * factor, 0.0, 1.0);
00748         }
00749
00750         // Child 2
00751         if (dist01(rng()) < params.scaling_mutation_prob)
00752         {
00753             int idx = idx_dist(rng());
00754             double factor = scale_dist(rng());
00755             c2[idx] = std::clamp(c2[idx] * factor, 0.0, 1.0);
00756         }
00757     }
00758
00759     next_gen.push_back(std::move(c1));
00760     if (next_gen.size() < population.size())
00761         next_gen.push_back(std::move(c2));
00762 }
00763
00764 population.swap(next_gen);
00765
00766 if (params.verbose && gen % (params.max_iterations / 10) == 0)
00767 {
00768     std::cout << "[GA-Real] Gen " << gen << " best fitness " << gen_best
00769     << " (parallel threads: " << omp_get_max_threads() << ")" << "\n";
00770 }
00771
00772 // PARALLEL final evaluation
00773 double best_fit = -1e12;
00774 size_t best_idx = 0;
00775 std::vector<double> final_fitnesses(population.size());
00776
00777 #pragma omp parallel for schedule(dynamic)
00778 for (size_t i = 0; i < population.size(); ++i)
00779 {
00780     final_fitnesses[i] = func(real_vector_size, population[i].data());
00781 }
00782
00783
00784 // Find best (sequential)
00785 for (size_t i = 0; i < population.size(); ++i)
00786 {
00787     if (final_fitnesses[i] > best_fit)
00788     {
00789         best_fit = final_fitnesses[i];
00790         best_idx = i;
00791     }
00792 }
00793
00794 // Copy best solution
00795 for (int i = 0; i < real_vector_size; ++i)
00796 {
00797     real_vector[i] = population[best_idx][i];
00798 }
00799
00800 // Store optimization results
00801 last_result.best_fitness = best_fit;
00802 last_result.generations = params.max_iterations - stall_count;
00803
00804 auto t1 = Clock::now();
00805 if (params.verbose)
00806 {
00807     double secs = std::chrono::duration<double>(t1 - t0).count();
00808     std::cout << "[GA-Real] Completed in " << secs << "s, best_fitness=" << best_fit << " (using "
00809     << omp_get_max_threads() << " parallel threads)" << "\n";
00810 }
00811
00812 return 0;
00813 }
00814
00815 // ****
00816 // 3) Hybrid optimize with sequential approach but parallel evaluations
00817 // ****
00818

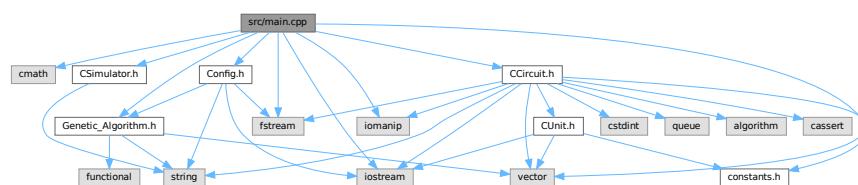
```

```

00819 /**
00820 * @brief Optimize a mixed discrete-continuous vector using a genetic algorithm
00821 *
00822 * This function optimizes a mixed discrete-continuous vector using a genetic
00823 * algorithm. It evaluates the fitness of the population in parallel and applies
00824 * selection, crossover, and mutation to generate new populations.
00825 *
00826 * @param int_vector_size Size of the integer vector
00827 * @param int_vector Pointer to the integer vector
00828 * @param real_vector_size Size of the real vector
00829 * @param real_vector Pointer to the real vector
00830 * @param hybrid_func Function to evaluate the fitness of the circuit
00831 * @param hybrid_validity Function to check the validity of the circuit
00832 * @param params Algorithm parameters for the optimization process
00833 *
00834 * @return The best fitness value found during optimization
00835 */
00836 int optimize(int int_vector_size, int* int_vector, int real_vector_size, double* real_vector,
00837                 std::function<double(int, int*, int, double*)> hybrid_func,
00838                 std::function<bool(int, int*, int, double*)> hybrid_validity, Algorithm_Parameters
00839     params)
00840 {
00841     std::cout << "OpenMP: Using " << omp_get_max_threads() << " threads for hybrid optimization" <<
00842     std::endl;
00843     // Discrete step: optimize only int vector
00844     auto wrapped_func_int = [&](int n, int* v)
00845     {
00846         return hybrid_func(n, v, real_vector_size,
00847                             real_vector); // current real_vector
00848     };
00849     auto wrapped_valid_int = [&](int n, int* v) { return hybrid_validity(n, v, real_vector_size,
00850     real_vector); };
00851     optimize(int_vector_size, int_vector, wrapped_func_int, wrapped_valid_int, params);
00852     // Continuous step: optimize only real vector
00853     auto wrapped_func_real = [&](int n, double* r)
00854     {
00855         return hybrid_func(int_vector_size, int_vector, n, r); // fixed int_vector
00856     };
00857     auto wrapped_valid_real = [&](int n, double* r) { return hybrid_validity(int_vector_size,
00858     int_vector, n, r); };
00859     optimize(real_vector_size, real_vector, wrapped_func_real, wrapped_valid_real, params);
00860
00861     return 0;
00862 }
00863 }
```

7.24 src/main.cpp File Reference

```
#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>
#include "CCircuit.h"
#include "CSimulator.h"
#include "Config.h"
#include "Config.h"
#include "Genetic_Algorithm.h"
Include dependency graph for main.cpp:
```



Functions

- int `main ()`

7.24.1 Function Documentation

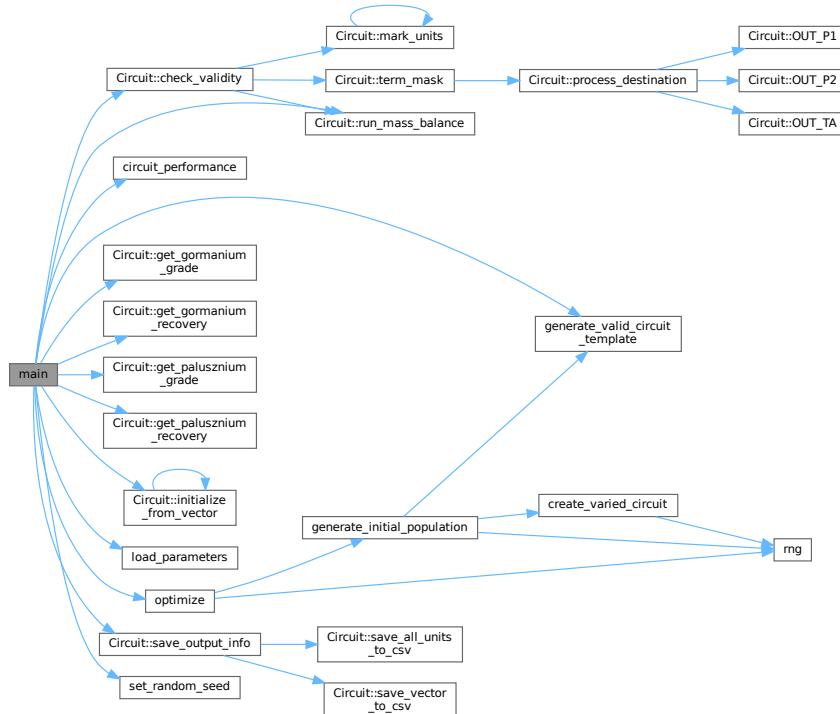
`main()`

```
int main ( )
```

Definition at line 16 of file `main.cpp`.

References `Algorithm_Parameters::allow_mutation_wrapping`, `Circuit::check_validity()`, `circuit_performance()`, `Algorithm_Parameters::convergence_threshold`, `Algorithm_Parameters::crossover_points`, `Algorithm_Parameters::crossover_probability`, `Algorithm_Parameters::elite_count`, `generate_valid_circuit_template()`, `Circuit::get_germanium_grade()`, `Circuit::get_germanium_recovery()`, `Circuit::get_palusznium_grade()`, `Circuit::get_palusznium_recovery()`, `Circuit::initialize_from_vector()`, `Algorithm_Parameters::inversion_load_parameters()`, `Algorithm_Parameters::log_file`, `Algorithm_Parameters::log_results`, `Algorithm_Parameters::max_iterations`, `Algorithm_Parameters::mode`, `Algorithm_Parameters::mutation_probability`, `Algorithm_Parameters::mutation_step_size`, `Algorithm_Parameters::num_units`, `optimize()`, `Algorithm_Parameters::population_size`, `Algorithm_Parameters::random_seed`, `Circuit::run_mass_balance()`, `Circuit::save_output_info()`, `Algorithm_Parameters::scaling_mutation_max`, `Algorithm_Parameters::scaling_mutation_prob`, `Algorithm_Parameters::selection_pressure`, `set_random_seed()`, `Algorithm_Parameters::stall_generations`, `Algorithm_Parameters::tournament_size`, `Algorithm_Parameters::use_inversion`, `Algorithm_Parameters::use_scaling_mutation`, and `Algorithm_Parameters::verbose`.

Here is the call graph for this function:



7.25 main.cpp

Go to the documentation of this file.

```
00001 #include <cmath>
00002 #include <fstream>
00003 #include <iomanip>
00004 #include <iostream>
00005 #include <vector>
00006
00007 #include "CCircuit.h"
00008 #include "CSimulator.h"
00009 #include "Config.h" // <-- your new loader
```

```

00010 #include "Genetic_Algorithm.h"
00011
00012 // static constexpr int DEFAULT_UNITS = 10;
00013 // static const int hard_circuit_10[2 * DEFAULT_UNITS + 1] = {1, 2, 4, 3, 5, 3, 0, 8, 11, 7, 12,
00014 //                                         7, 0, 7, 11, 8, 6, 9, 7, 10, 3};
00015
00016 int main()
00017 {
00018     // Save original cout buffer before we start
00019     std::streambuf* original_cout_buffer = std::cout.rdbuf();
00020
00021     // Create null stream to discard output
00022     std::ofstream null_stream("/dev/null");
00023
00024     std::cout << "==== Paluszniium Rush Circuit Optimizer ====\n\n";
00025
00026     // load GA & random-seed settings from parameters.txt
00027     Algorithm_Parameters params;
00028     load_parameters("parameters.txt", params);
00029
00030     // Optionally fix the RNG for reproducibility
00031     if (params.random_seed >= 0)
00032     {
00033         set_random_seed(params.random_seed);
00034         std::cout << "* Using fixed seed: " << params.random_seed << "\n";
00035     }
00036
00037     // Print to see
00038     std::cout << "GA parameters:\n"
00039             << "    mode                               = " << params.mode << "\n"
00040             << "    random_seed                         = " << params.random_seed << "\n\n"
00041             << "    num_units                           = " << params.num_units << "\n\n"
00042
00043             << "    population_size                    = " << params.population_size << "\n"
00044             << "    elite_count                        = " << params.elite_count << "\n"
00045             << "    max_iterations                     = " << params.max_iterations << "\n\n"
00046
00047             << "    tournament_size                   = " << params.tournament_size << "\n"
00048             << "    selection_pressure                = " << params.selection_pressure << "\n\n"
00049
00050             << "    crossover_probability              = " << params.crossover_probability << "\n"
00051             << "    crossover_points                 = " << params.crossover_points << "\n\n"
00052
00053             << "    mutation_probability              = " << params.mutation_probability << "\n"
00054             << "    mutation_step_size               = " << params.mutation_step_size << "\n"
00055             << "    allow_mutation_wrapping        = " << std::boolalpha << params.allow_mutation_wrapping <<
00056             "\n\n"
00057             << "    use_inversion                      = " << std::boolalpha << params.use_inversion << "\n"
00058             << "    inversion_probability            = " << params.inversion_probability << "\n\n"
00059
00060             << "    use_scaling_mutation             = " << std::boolalpha << params.use_scaling_mutation <<
00061             "\n"
00062             << "    scaling_mutation_prob           = " << params.scaling_mutation_prob << "\n"
00063             << "    scaling_mutation_min            = " << params.scaling_mutation_min << "\n"
00064             << "    scaling_mutation_max            = " << params.scaling_mutation_max << "\n\n"
00065
00066             << "    convergence_threshold          = " << params.convergence_threshold << "\n"
00067             << "    stall_generations              = " << params.stall_generations << "\n\n"
00068
00069             << "    verbose                           = " << std::boolalpha << params.verbose << "\n"
00070             << "    log_results                      = " << std::boolalpha << params.log_results << "\n"
00071             << "    log_file                          = " << params.log_file << "\n\n";
00072
00073     // Optimisation mode
00074     auto mode = params.mode; // "d", "c" or "h" from parameters.txt
00075     std::cout << "Mode: " << mode << "\n";
00076
00077     // Set number of units
00078     int num_units = params.num_units;
00079     int vector_size = 2 * num_units + 1;
00080
00081     // Create vectors to hold the optimization results
00082     std::vector<int> circuit_vector(vector_size, 0);
00083     std::vector<double> volume_params(num_units, 0.5);
00084
00085     if (mode == "d")
00086     {
00087         std::cout << "Running DISCRETE optimization...\n";
00088
00089         // std::cout.rdbuf(null_stream.rdbuf());
00090
00091         auto discrete_fitness = [] (int size, int* vec) -> double
00092         {
00093             // Discrete-only overload
00094             return circuit_performance(size, vec);
00095         };

```

```

00095     auto discrete_validity = [](int size, int* vec) -> bool
00096     {
00097         // Discrete-only constructor and validity
00098         Circuit c(size / 2); // (2n + 1 → n units)
00099         c.initialize_from_vector(size, vec);
00100         return c.check_validity(size, vec);
00101     };
00102
00103     optimize(vector_size, circuit_vector.data(), discrete_fitness, discrete_validity, params);
00104 }
00105
00106 else if (mode == "c")
00107 {
00108     std::cout << "Running CONTINUOUS optimization...\n";
00109
00110     // std::cout.rdbuf(null_stream.rdbuf());
00111
00112     // Build a simple *valid* circuit of the requested size
00113     auto base = generate_valid_circuit_template(num_units); // function from Genetic_Algorithm.cpp
00114     std::copy(base.begin(), base.end(), circuit_vector.begin());
00115     auto cont_fitness = [&](int r_size, double* rvec) -> double
00116     { return circuit_performance(vector_size, circuit_vector.data(), r_size, rvec); };
00117
00118     auto cont_validity = [&](int r_size, double* rvec) -> bool
00119     {
00120         Circuit c(num_units, rvec); // use volume constructor!
00121         c.initialize_from_vector(vector_size, circuit_vector.data(), rvec);
00122         return c.check_validity(vector_size, circuit_vector.data(), r_size, rvec);
00123     };
00124
00125     optimize(num_units, volume_params.data(), cont_fitness, cont_validity, params);
00126 }
00127
00128 else
00129 {
00130     std::cout << "Running hybrid optimization (connections + volumes)... \n";
00131
00132     // Redirect cout to null stream to silence debug output
00133     // std::cout.rdbuf(null_stream.rdbuf());
00134
00135     // Define hybrid fitness and validity functions
00136     auto hybrid_fitness = [](int i_size, int* i_vec, int r_size, double* r_vec) -> double
00137     { return circuit_performance(i_size, i_vec, r_size, r_vec); };
00138
00139     auto hybrid_validity = [num_units](int i_size, int* i_vec, int r_size, double* r_vec) -> bool
00140     {
00141         Circuit c(num_units);
00142         c.initialize_from_vector(i_size, i_vec);
00143         return c.check_validity(i_size, i_vec, r_size, r_vec);
00144     };
00145
00146
00147     // Run hybrid optimization (cout is redirected, so no debug output)
00148     optimize(vector_size, circuit_vector.data(), num_units, volume_params.data(), hybrid_fitness,
00149     hybrid_validity,
00150     params);
00151
00152     // Calculate performance with optimized values (still silent)
00153     double performance = circuit_performance(vector_size, circuit_vector.data(), num_units,
00154     volume_params.data());
00155
00156     // Create a circuit object for detailed analysis, still silent
00157     Circuit circuit(num_units, volume_params.data());
00158     circuit.initialize_from_vector(vector_size, circuit_vector.data(), volume_params.data());
00159     circuit.run_mass_balance();
00160
00161     // Extract important metrics before restoring cout
00162     double palusznium_recovery = circuit.get_palusznium_recovery() * 100;
00163     double palusznium_grade = circuit.get_palusznium_grade() * 100;
00164     double gormanium_recovery = circuit.get_gormanium_recovery() * 100;
00165     double gormanium_grade = circuit.get_gormanium_grade() * 100;
00166
00167     // Calculate volumes and costs while still silent
00168     double total_volume = 0.0;
00169     double unit_volumes[num_units];
00170     for (int i = 0; i < num_units; i++)
00171     {
00172         if (mode == "h" || mode == "c")
00173         {
00174             // Use scaled volumes
00175             double min_volume = 2.5;
00176             double max_volume = 20.0;
00177             unit_volumes[i] = min_volume + (max_volume - min_volume) * volume_params[i];
00178         }
00179     }
00180 }
```

```

00180         // Discrete mode: use fixed volume
00181         unit_volumes[i] = 10.0;
00182     }
00183     total_volume += unit_volumes[i];
00184 }
00185
00186 double operating_cost = 5.0 * std::pow(total_volume, 2.0 / 3.0);
00187 if (total_volume >= 150.0)
00188 {
00189     operating_cost += 1000.0 * std::pow(total_volume - 150.0, 2.0);
00190 }
00191
00192 // Now RESTORE cout to print results
00193 std::cout.rdbuf(original_cout_buffer);
00194
00195 // Print final results after optimization
00196 std::cout << "\nOptimization complete!\n";
00197 std::cout << "Final circuit economic value: £" << std::fixed << std::setprecision(2) << performance
00198 << " per second\n\n";
00199
00200 // Display the optimized circuit vector
00201 std::cout << "Optimized circuit vector: ";
00202 for (int i = 0; i < vector_size; ++i)
00203     std::cout << circuit_vector[i] << " ";
00204 std::cout << std::endl;
00205
00206 // Display the optimized volumes
00207 std::cout << "Optimized volume parameters: ";
00208 for (int i = 0; i < num_units; ++i)
00209     std::cout << std::fixed << std::setprecision(5) << volume_params[i] << " ";
00210 std::cout << std::endl;
00211
00212 // Display circuit performance metrics
00213 std::cout << "\nCircuit Performance:\n";
00214 std::cout << "-- Palusznium recovery: " << std::fixed << std::setprecision(2) << palusznium_recovery <
00215 "%\n";
00216 std::cout << "-- Palusznium grade: " << std::fixed << std::setprecision(2) << palusznium_grade << "%\n";
00217 std::cout << "-- Gormanium recovery: " << std::fixed << std::setprecision(2) << gormanium_recovery <
00218 "%\n";
00219 std::cout << "-- Gormanium grade: " << std::fixed << std::setprecision(2) << gormanium_grade << "%\n";
00220
00221 // Circuit configuration analysis
00222 std::cout << "\nCircuit Configuration Analysis:\n";
00223 int direct_to_p = 0, direct_to_g = 0, direct_to_t = 0, recycles = 0;
00224 for (int i = 0; i < num_units; i++)
00225 {
00226     // Check concentrate connections
00227     int conc_dest = circuit_vector[1 + 2 * i];
00228     if (conc_dest == num_units)
00229         direct_to_p++;
00230     else if (conc_dest == num_units + 1)
00231         direct_to_g++;
00232     else if (conc_dest == num_units + 2)
00233         direct_to_t++;
00234     else if (conc_dest < i)
00235         recycles++;
00236
00237     // Check tailing connections
00238     int tail_dest = circuit_vector[2 + 2 * i];
00239     if (tail_dest == num_units)
00240         direct_to_p++;
00241     else if (tail_dest == num_units + 1)
00242         direct_to_g++;
00243     else if (tail_dest == num_units + 2)
00244         direct_to_t++;
00245     else if (tail_dest < i)
00246         recycles++;
00247
00248     std::cout << "-- Units sending to Palusznium product: " << direct_to_p << "\n";
00249     std::cout << "-- Units sending to Gormanium product: " << direct_to_g << "\n";
00250     std::cout << "-- Units sending to Tailings: " << direct_to_t << "\n";
00251     std::cout << "-- Recycle connections: " << recycles << "\n";
00252
00253 // Unit volume analysis
00254 std::cout << "\nUnit Volumes (m³):\n";
00255 for (int i = 0; i < num_units; i++)
00256 {
00257     std::cout << "Unit " << i << ":" << std::fixed << std::setprecision(2) << unit_volumes[i] <<
00258 " m³\n";
00259 }
00260 std::cout << "Total volume: " << std::fixed << std::setprecision(2) << total_volume << " m³\n";
00261
00262 // Economic analysis
00263 std::cout << "\nEconomic Analysis:\n";
00264 double palusznium_value = circuit.get_palusznium_recovery() * 8 * 120;
00265 double gormanium_value = circuit.get_gormanium_recovery() * 12 * 80;

```

```
00264     std::cout << "-- Palusznium revenue: £" << std::fixed << std::setprecision(2) << palusznium_value <<
00265     "/s\n";
00266     std::cout << "-- Gormanium revenue: £" << std::fixed << std::setprecision(2) << gormanium_value <<
00267     "/s\n";
00268     std::cout << "-- Total revenue: £" << std::fixed << std::setprecision(2) << (palusznium_value +
00269     gormanium_value)
00270     << "/s\n";
00271 // Save raw circuit data into a CSV:
00272 const std::string out_csv = "plotting/circuit_results.csv";
00273 if (circuit.save_output_info(out_csv))
00274 {
00275     std::cout << "\n Saved detailed circuit info to " << out_csv << "\n";
00276 }
00277 else
00278 {
00279     std::cerr << "\n Failed to write circuit info to " << out_csv << "\n";
00280 }
00281
00282 return 0;
00283 }
```