# 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 *not* 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 |
|---|---|---|---|
| **num_units** | integer 2 | 6 | Number of separation units ∗(vector length=2·n+1)∗ |
| **mode** | `d` \| `c` \| `h` | `h` | GA operating mode: discrete, continuous (**dev-only**), or hybrid |
| **max_iterations** | integer | 100 | GA generations per optimisation call |
| **population_size** | integer | 600 | Individuals per generation |
| **elite_count** | integer | 2 | Best genomes copied unchanged each generation |
| **tournament_size** | integer | 3 | k-way tournament selection pressure |
| **crossover_probability** | 0–1 | 0.9 | Chance two parents cross |
| **mutation_probability** | 0–1 | 0.08 | Per-gene mutation chance (all modes) |
| **mutation_step_size** | integer 1 | 3 | Max ± step for discrete "creep" |
| **use_inversion** | bool | true | Enable contiguous slice reversal (discrete) |
| **inversion_probability** | 0–1 | 0.2 | Chance *per child* that inversion occurs |
| **use_scaling_mutation** | bool | true | Enable multiplicative tweak for genes |
| **scaling_mutation_prob** | 0–1 | 0.3 | Probability a child gets scaling mutation |
| **scaling_mutation_min** | >0 | 0.7 | Lower bound of scaling factor |
| **scaling_mutation_max** | >1 | 1.3 | Upper bound of scaling factor |
| **convergence_threshold** | real 0 | 0.1 | fitness below which a change is deemed “no improvement” |
| **stall_generations** | integer | 50 | Stop if no improvement for this many generations |
| **verbose** | bool | true | Print progress every 10 generations |
| **log_results** | bool | false | Append CSV copy of every generation to `log_file` |
| **log_file** | filename | `ga_run.log` | Only used if `log_results = true` |
| **random_seed** | 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:

# 3 Class Index

## 3.1 Class List

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



# 4 File Index

## 4.1 File List

Here is a list of all files with brief descriptions:

# 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 PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM = 120.0
- constexpr double GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM = -20.0
- constexpr double WASTE_PENALTY_IN_PALUSZNIUM_STREAM = -300.0
- constexpr double PALUSZNIUM_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_PALUSZNIUM_STREAM

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

#### PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM

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

#### PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM

`constexpr double Constants::Economic::PALUSZNIUM_VALUE_IN_PALUSZNIUM_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_PALUSZNIUM_STREAM

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

## 5.4 Constants::Feed Namespace Reference

#### Variables

- constexpr double DEFAULT_PALUSZNIUM_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_PALUSZNIUM_STREAM

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

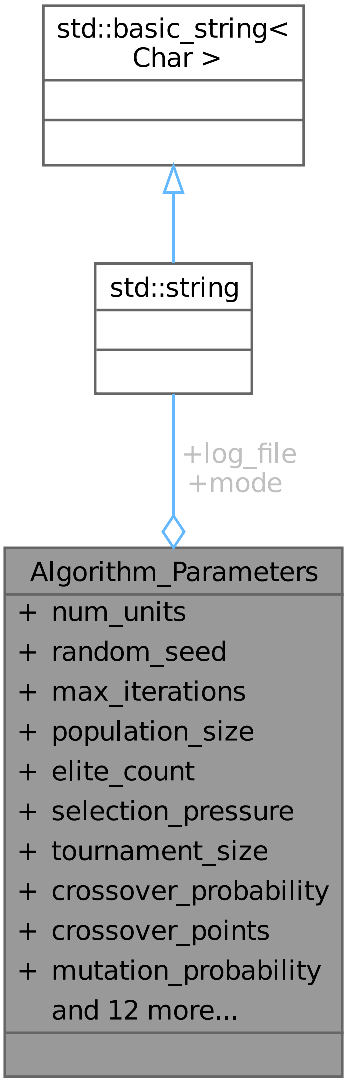
Definition at line 23 of file constants.h.
Referenced by Circuit::Circuit().

# 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

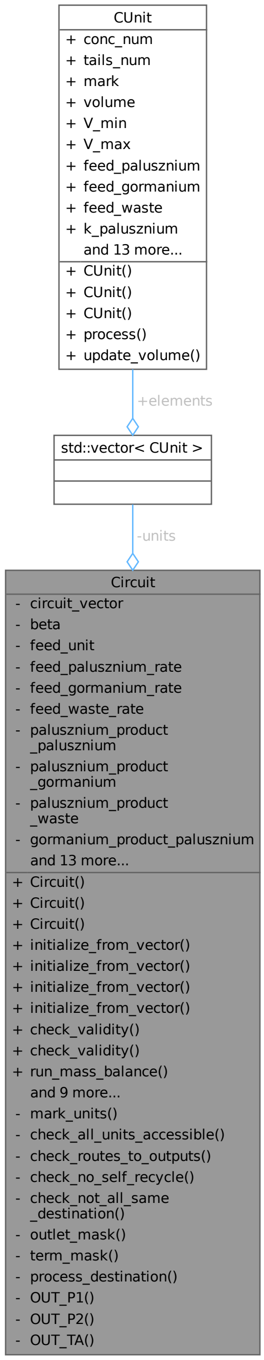
```
bool Algorithm_Parameters::verbose = false
```
Definition at line 60 of file Genetic_Algorithm.h.
Referenced by load_parameters(), main(), optimize(), and optimize().
The documentation for this struct was generated from the following file:

- 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_gormanium_recovery () const

    *Get the recovery of gormanium.*
- double get_palusznium_grade () const

    *Get the grade of palusznium.*
- double get_gormanium_grade () const

    *Get the grade of gormanium.*
- 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
- 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_gormanium_rate
- double feed_waste_rate
- double palusznium_product_palusznium
- double palusznium_product_gormanium
- double palusznium_product_waste
- double gormanium_product_palusznium
- double gormanium_product_gormanium
- double gormanium_product_waste
- double tailings_palusznium
- double tailings_gormanium
- double tailings_waste
- double palusznium_value
- double gormanium_value
- double gormanium_value_in_palusznium
- double palusznium_value_in_gormanium
- double waste_penalty_palusznium
- double waste_penalty_gormanium
- 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**

| | |
|---|---|
| *num_units* | Number of units in the circuit |
| *beta* | 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**

| | |
|---|---|
| *num_units* | Number of units in the circuit |
| *beta* | Pointer to the beta array |
| *testFlag* | 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_palusznium_rate, feed_waste_rate, gormanium_value, Constants::Test::GORMANIUM_VALUE_IN_GORMANIUM_STREAM, gormanium_value_in_palusznium, Constants::Test::GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM, palusznium_value, palusznium_value_in_gormanium, Constants::Test::PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM, Constants::Test::PALUSZNIUM_VALUE_IN_PALUSZNIUM_S waste_penalty_gormanium, Constants::Test::WASTE_PENALTY_IN_GORMANIUM_STREAM, Constants::Test::WASTE_PENALTY_ and waste_penalty_palusznium.

### 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**

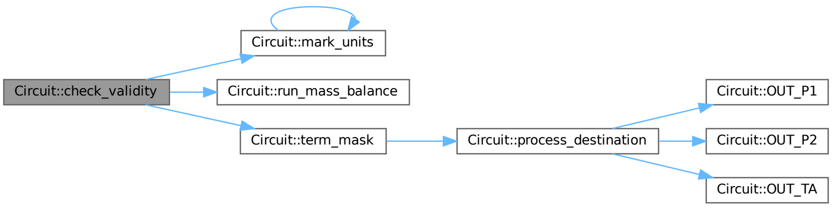
| | |
|---|---|
| *vector_size* | Size of the circuit vector |
| *vec* | Circuit vector |

**Returns**

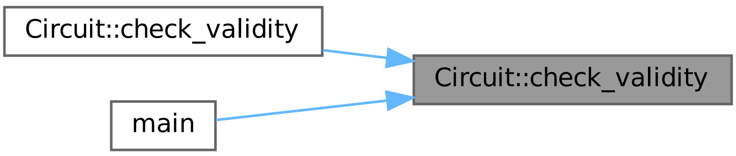
true if the circuit is valid, false otherwise

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.

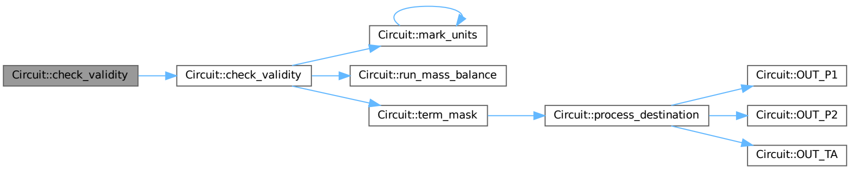
**Parameters**

| | |
|---|---|
| *vector_size* | Size of the circuit vector |
| *circuit_vector* | Circuit vector |
| *unit_parameters_size* | Size of the unit parameters |
| *unit_parameters* | 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**

| | |
|---|---|
| *filename* | 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, PALUSZNIUM_PRODUCT, TAILINGS_OUTPUT, and units.

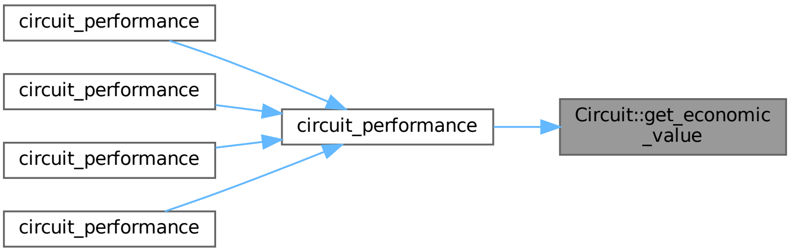
### get_economic_value()

```
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:



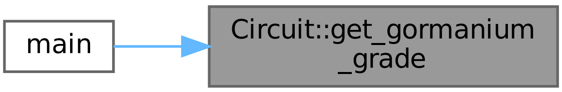

### get_gormanium_grade()

`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:



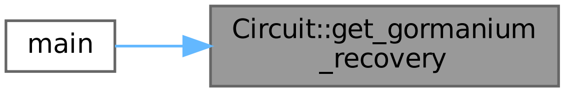

### get_gormanium_recovery()

`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()

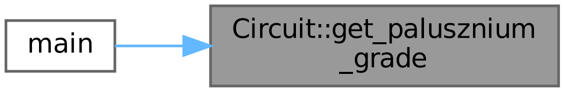
`double Circuit::get_palusznium_grade ( ) const`

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`

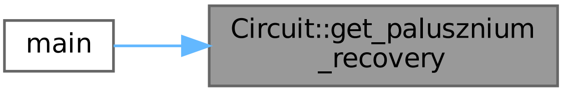
Get the recovery of valuable materials.
This function calculates the recovery of valuable materials in the circuit based on the product flow rates and the feed rates.

**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.

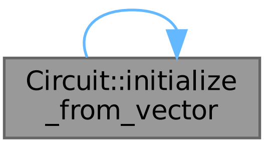
**Parameters**

| | |
|---|---|
| *vector_size* | Size of the circuit vector |
| *circuit_vector* | Circuit vector |

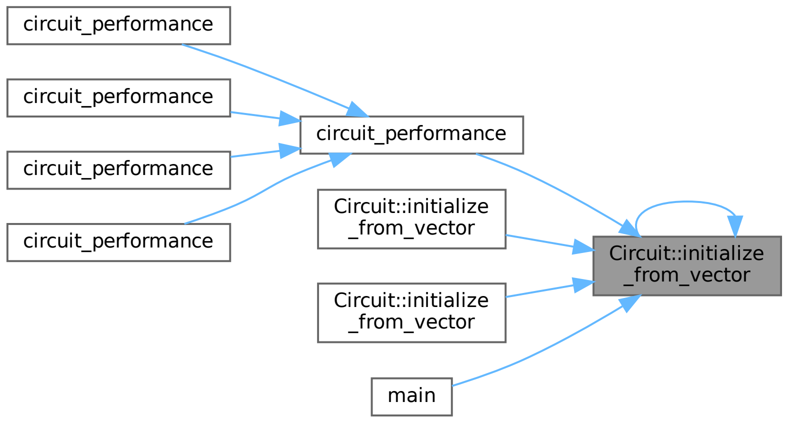
**Returns**

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.

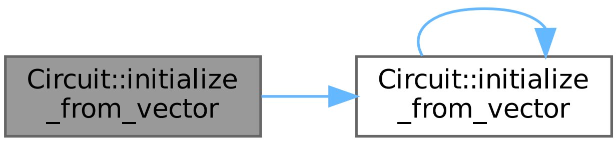
**Parameters**

| | |
|---|---|
| *vector_size* | Size of the circuit vector |
| *circuit_vector* | Circuit vector |
| *testFlag* | 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.

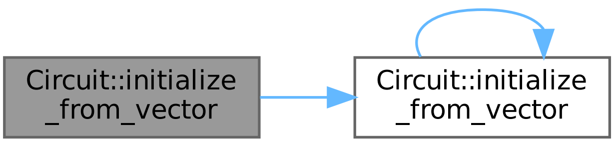
**Parameters**

| | |
|---|---|
| *vector_size* | Size of the circuit vector |
| *circuit_vector* | Circuit vector |
| *beta* | 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**

| | |
|---|---|
| *vector_size* | Size of the circuit vector |
| *circuit_vector* | Circuit vector |
| *beta* | Pointer to the beta array |
| *testFlag* | 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, PALUSZNIUM_PRODUCT, TAILINGS_OUTPUT, and units.

### mark_units()

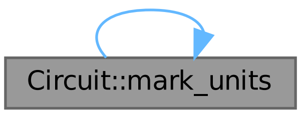
```
void Circuit::mark_units (
             int unit_num )  [private]
```

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.

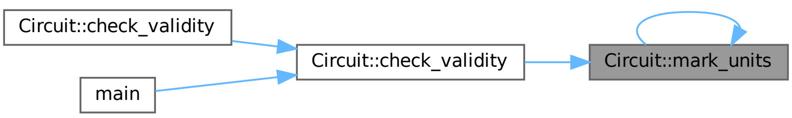
**Parameters**

| | |
|---|---|
| *unit_num* | 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:

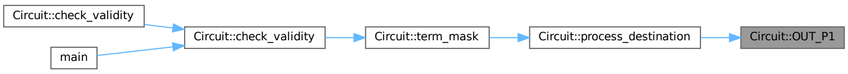



### OUT_P1()

```
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()

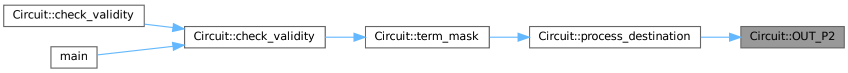
`int Circuit::OUT_P2 ( ) const [inline], [private]`

Definition at line 168 of file CCircuit.h.

References n.

Referenced by process_destination().

Here is the caller graph for this function:




### OUT_TA()

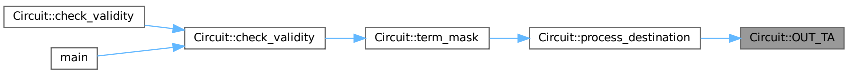
`int Circuit::OUT_TA ( ) const [inline], [private]`

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]
```

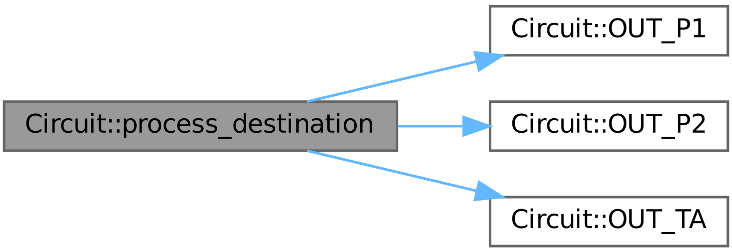
Process the destination unit.

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

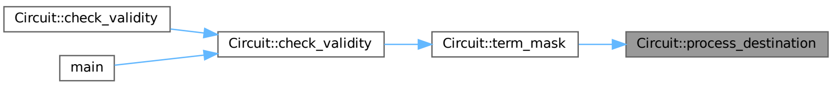
**Parameters**

| | |
|---|---|
| *dest* | The destination unit number |
| *mask* | The terminal mask |
| *visited* | Vector to keep track of visited units |
| *q* | 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**

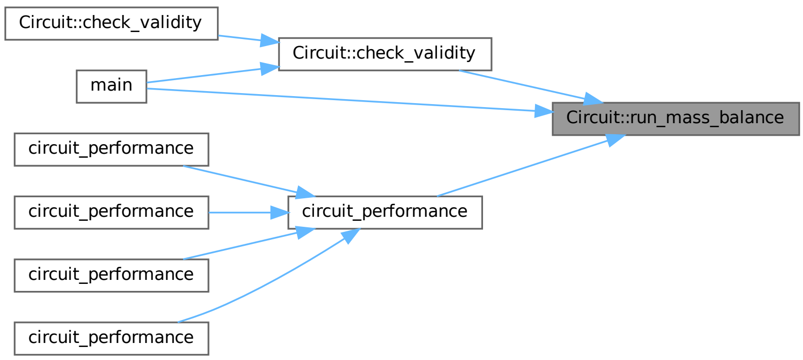
| | |
|---|---|
| *tolerance* | Tolerance for convergence |
| *max_iterations* | Maximum number of iterations |

**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, PALUSZNIUM_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**

| | |
|---|---|
| *filename* | 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.

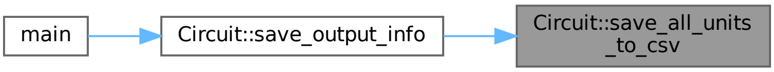
**Parameters**

| | |
|---|---|
| *filename* | 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**

| | |
|---|---|
| *filename* | 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.

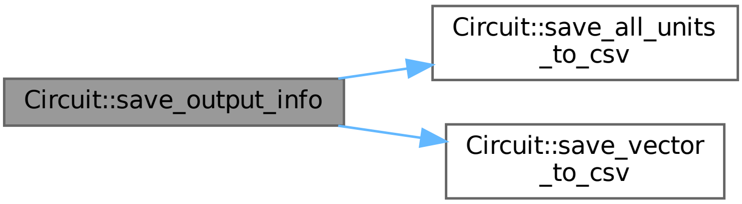
**Parameters**

| | |
|---|---|
| *filename* | 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**

| | |
|---|---|
| *filename* | 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.

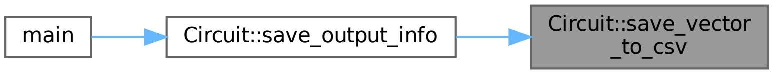
**Parameters**

| | |
|---|---|
| *filename* | 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**

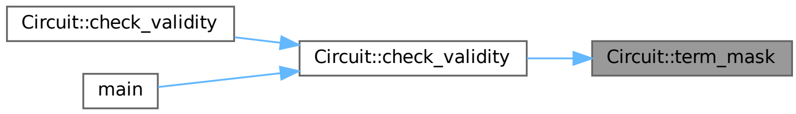
| | |
|---|---|
| *start* | The starting unit number |

**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_gormanium

`double Circuit::palusznium_value_in_gormanium [private]`
Definition at line 140 of file CCircuit.h.
Referenced by Circuit(), and get_economic_value().

#### tailings_gormanium

`double Circuit::tailings_gormanium [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<CUnit> 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_gormanium

`double Circuit::waste_penalty_gormanium [private]`
Definition at line 142 of file CCircuit.h.
Referenced by Circuit(), and get_economic_value().

#### waste_penalty_palusznium

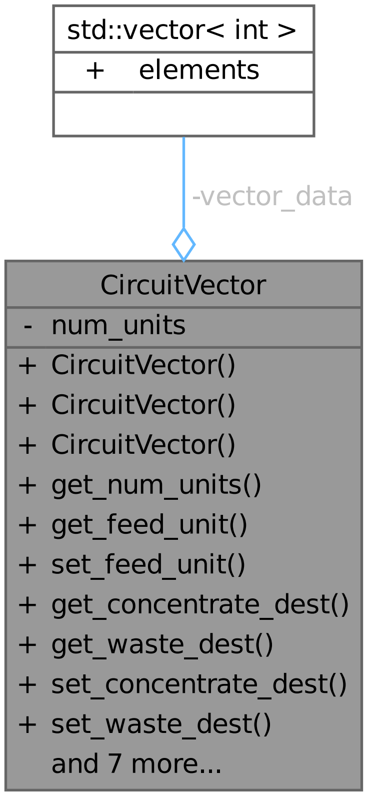
`double Circuit::waste_penalty_palusznium [private]`
Definition at line 141 of file CCircuit.h.
Referenced by Circuit(), and get_economic_value().
The documentation for this class was generated from the following files:

- 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.

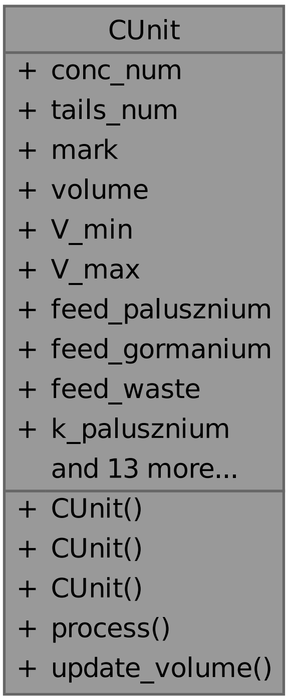
The documentation for this class was generated from the following file:

- include/circuit_vector.h

## 6.4 CUnit Class Reference

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

Collaboration diagram for CUnit:




### 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
- int tails_num
- bool mark
- double volume

  *Unit volume V ($m^3$) – default 10 $m^3$*
- double V_min

  *Minimum volume ($m^3$) – default 2.5 $m^3$*
- double V_max

  *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 (kg $s^{-1}$)*
- double conc_gormanium

  *C_G (kg $s^{-1}$)*
- double conc_waste

  *C_W (kg $s^{-1}$)*
- double tails_palusznium

  *T_P (kg $s^{-1}$)*
- double tails_gormanium

  *T_G (kg $s^{-1}$)*
- double tails_waste

  *T_W (kg $s^{-1}$)*
- 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**

| | |
|---|---|
| *conc* | Destination index for concentrate |
| *tails* | 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_palusznium, 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_palusznium, conc_waste, feed_gormanium, feed_palusznium, feed_waste, k_gormanium, k_palusznium, k_waste, phi, Rg, rho, Rp, Rw, tails_gormanium, tails_palusznium, 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**

| | |
|---|---|
| *beta* | The new volume of the unit. |

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

**Parameters**

| | |
|---|---|
| *beta* | 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 ($m^3$) – default 20 $m^3$

Definition at line 45 of file CUnit.h.

Referenced by CUnit(), and update_volume().

### V_min

`double CUnit::V_min`

Minimum volume ($m^3$) – default 2.5 $m^3$

Definition at line 44 of file CUnit.h.

Referenced by CUnit(), and update_volume().

### volume

`double CUnit::volume`

Unit volume V ($m^3$) – default 10 $m^3$

Definition at line 43 of file CUnit.h.

Referenced by CUnit(), process(), and update_volume().

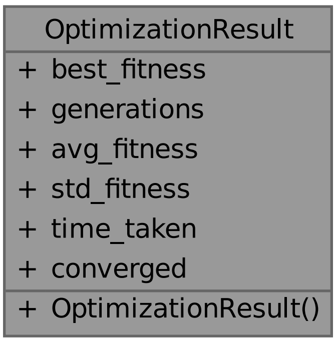
The documentation for this class was generated from the following files:

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

## 6.5 OptimizationResult Struct Reference

`#include <Genetic_Algorithm.h>`

Collaboration diagram for OptimizationResult:




### 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

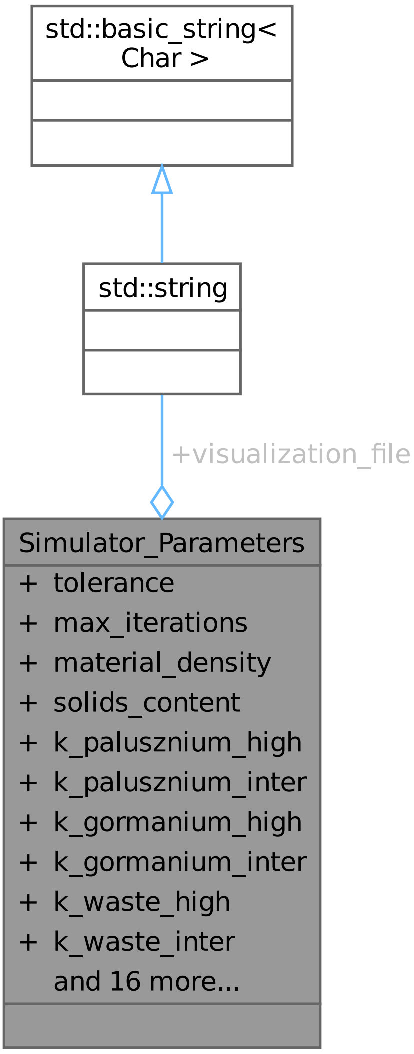
`double OptimizationResult::time_taken`
Definition at line 97 of file Genetic_Algorithm.h.
The documentation for this struct was generated from the following file:

- 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.

#### paluszniium_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

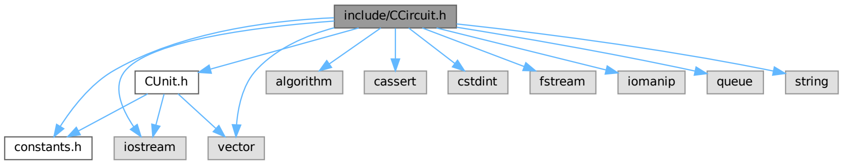
## 7.1 include/CCircuit.h File Reference

Declares the Circuit class – a mineral-processing circuit.

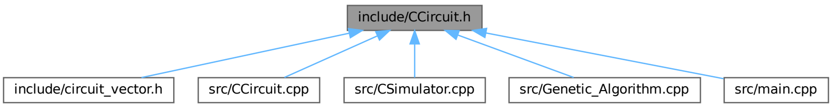
```
#include "CUnit.h"
#include "constants.h"
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <queue>
#include <string>
```

`#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 /**
```

```
 * @file CCircuit.h
 * @brief 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
 *
 */

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

// Constants for the circuit outlet destinations
// These values will be used in the circuit vector to indicate the final product
// streams
enum CircuitDestination
{
    PALUSZNIUM_PRODUCT = -1, // Final Palusznium concentrate product
    GORMANIUM_PRODUCT = -2,  // Final Gormanium concentrate product
    TAILINGS_OUTPUT = -3     // Final tailings output
};

/* ------------------------------------------------------------------ */
/*                        Circuit class                        */
/* ------------------------------------------------------------------ */

class Circuit
{
public:
    // Constructor that takes the number of units in the circuit
    Circuit(int num_units);

    // Constructor with beta values for unit volumes
    Circuit(int num_units, double* beta);

    // Test constructor with beta values and test flag
    Circuit(int num_units, double* beta, bool testFlag);

    // Initialize the circuit from a circuit vector
    bool initialize_from_vector(int vector_size, const int* circuit_vector);
    bool initialize_from_vector(int vector_size, const int* circuit_vector, const double* beta);

    bool initialize_from_vector(int vector_size, const int* circuit_vector, bool testFlag);
    bool initialize_from_vector(int vector_size, const int* circuit_vector, const double* beta, bool testFlag);

    // Check validity of a circuit vector
    bool check_validity(int vector_size, const int* circuit_vector);
    bool check_validity(int vector_size, const int* circuit_vector, int unit_parameters_size, double* unit_parameters);

    // Run a mass balance calculation on the circuit
    bool run_mass_balance(double tolerance = 1e-6, int max_iterations = 1000);

    // Get the economic value of the circuit
    double get_economic_value() const;

    // Get the recovery of valuable materials
    double get_palusznium_recovery() const;
    double get_gormanium_recovery() const;

    // Get the grade of valuable materials in products
    double get_palusznium_grade() const;
    double get_gormanium_grade() const;

    // Export the circuit to a dot file for visualization
    bool export_to_dot(const std::string& filename) const;

    /**
     * @brief Save all units to a CSV file.
     * @param filename The name of the output CSV file.
```

```
     * @return true if save is successful, false otherwise
     */
    bool save_all_units_to_csv(const std::string& filename);

    /**
     * @brief Save a vector to a CSV file.
     * @param filename The name of the output CSV file.
     * @return true if save is successful, false otherwise
     */
    bool save_vector_to_csv(const std::string& filename);

    /**
     * @brief Save the circuit data to a CSV file.
     * @param filename The name of the output CSV file.
     * @return true if save is successful, false otherwise
     */
    bool save_output_info(const std::string& filename);

private:
    // The array of units in the circuit
    std::vector<CUnit> units;

    // Circuit vector (for output)
    const int* circuit_vector;

    /* --------- volume information --------- */
    double* beta; // Array of beta values for unit volumes

    /* --------- flow information --------- */
    // Feed unit number and feed rates
    int feed_unit;
    double feed_palusznium_rate; // kg/s
    double feed_gormanium_rate;  // kg/s
    double feed_waste_rate;      // kg/s

    // Final product stream flow rates
    double palusznium_product_palusznium; // kg/s
    double palusznium_product_gormanium;  // kg/s
    double palusznium_product_waste;      // kg/s

    double gormanium_product_palusznium; // kg/s
    double gormanium_product_gormanium;  // kg/s
    double gormanium_product_waste;      // kg/s

    double tailings_palusznium; // kg/s
    double tailings_gormanium;  // kg/s
    double tailings_waste;      // kg/s

    /* --------- economic parameters --------- */
    // Economic parameters
    double palusznium_value;               // £/kg in Palusznium stream
    double gormanium_value;                // £/kg in Gormanium stream
    double gormanium_value_in_palusznium; // £/kg in Palusznium stream
    double palusznium_value_in_gormanium; // £/kg in Gormanium stream
    double waste_penalty_palusznium;       // £/kg waste in Palusznium stream
    double waste_penalty_gormanium;        // £/kg waste in Gormanium stream

    // Mark units that are accessible from a given unit (for validity checking)
    void mark_units(int unit_num);

    // Check if all units are accessible from the feed
    bool check_all_units_accessible() const;

    // Check if all units have routes to at least two output streams
    bool check_routes_to_outputs() const;

    // Check for self-recycle and other invalid configurations
    bool check_no_self_recycle() const;
    bool check_not_all_same_destination() const;

    /* ----------  Circuit validity checking ---------- */
    int n;
    int feed_dest = 0;
    uint8_t outlet_mask(int unit_idx, std::vector<int8_t>& cache) const;
    uint8_t term_mask(int start) const;
    void process_destination(int dest, uint8_t& mask, std::vector<bool>& visited, std::queue<int>& q)
const;

    inline int OUT_P1() const
    {
        return n;
    } // palusznium
    inline int OUT_P2() const
    {
        return n + 1;
    } // gormanium
    inline int OUT_TA() const
```

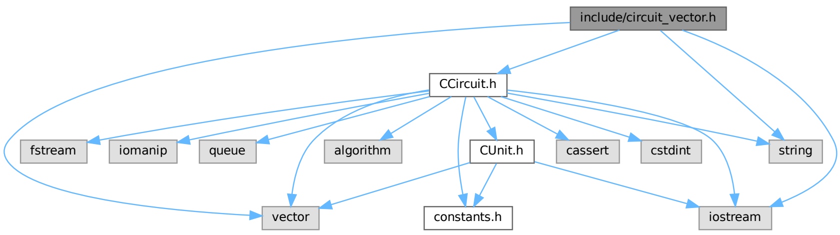
```
    {
        return n + 2;
    } // tails
};
```

## 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.

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

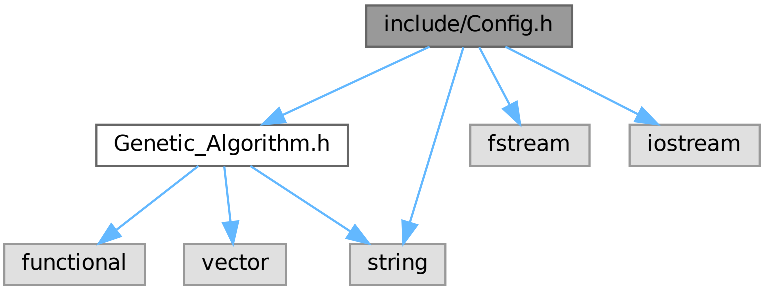
```
 *
 * A circuit vector has the following format:
 * [feed_unit, unit0_conc, unit0_waste, unit1_conc, unit1_waste, ...]
 *
 * where:
 * - feed_unit: Index of the unit receiving the circuit feed (0 to num_units-1)
 * - unitX_conc: Destination of the concentrate stream from unit X
 * - unitX_waste: 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
 */

#pragma once

#include "CCircuit.h" // Include for CircuitDestination enum
#include <iostream>
#include <string>
#include <vector>

class CircuitVector
{
public:
    // Constructor for empty circuit vector
    CircuitVector() = default;

    // Constructor for circuit vector with specified number of units
    CircuitVector(int num_units);

    // Constructor from existing circuit vector data
    CircuitVector(int vector_size, const int* data);

    // Get number of units in the circuit
    int get_num_units() const;

    // Get feed unit
    int get_feed_unit() const;

    // Set feed unit
    void set_feed_unit(int unit);

    // Get concentrate destination for unit
    int get_concentrate_dest(int unit) const;

    // Get waste destination for unit
    int get_waste_dest(int unit) const;

    // Set concentrate destination for unit
    void set_concentrate_dest(int unit, int dest);

    // Set waste destination for unit
    void set_waste_dest(int unit, int dest);

    // Get the raw vector data
    const std::vector<int>& get_data() const;

    // Convert to raw array (for compatibility with existing functions)
    const int* data() const;

    // Get the size of the vector
    int size() const;

    // Randomize the circuit vector with valid values
    void randomize();

    // Print the circuit vector
    void print(std::ostream& os = std::cout) const;

    // Save to file
    bool save_to_file(const std::string& filename) const;

    // Load from file
    bool load_from_file(const std::string& filename);

private:
    std::vector<int> vector_data;
    int num_units = 0;
};
```

## 7.5 include/Config.h File Reference

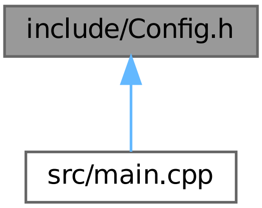
Configuration file for the Genetic Algorithm.

```
#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.

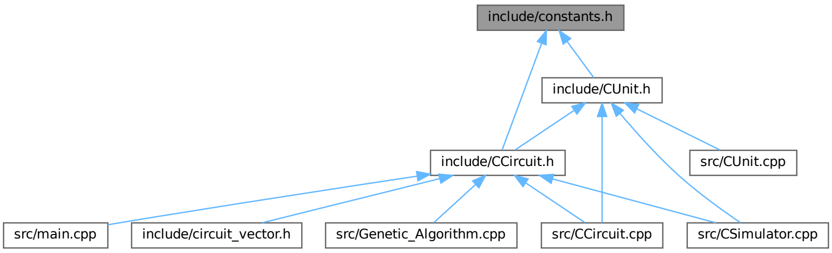
```
/**
 * @file Config.h
 * @brief Configuration file for the Genetic Algorithm
 *
 * This file contains the definition of the Algorithm_Parameters structure
 * and the function to load parameters from a configuration file.
 *
 * The Algorithm_Parameters structure holds various parameters for the
 * genetic algorithm.
 *
 */
#pragma once

#include "Genetic_Algorithm.h"
#include <fstream>
#include <iostream>
#include <string>

// Load GA parameters from a simple key=value text file
inline void load_parameters(const std::string& file, Algorithm_Parameters& p)
{
    std::ifstream in(file);
    if (!in)
    {
        std::cerr « "Warning: could not open " « file « " -- using default parameters.\n";
        return;
    }
    // Read the file line by line
    std::string line;
    auto trim = [&](std::string& s)
    {
        const char* ws = " \t\n\r";
        size_t start = s.find_first_not_of(ws);
        size_t end = s.find_last_not_of(ws);
        s = (start == std::string::npos) ? "" : s.substr(start, end - start + 1);
    };
    while (std::getline(in, line))
    {
        // remove comments
        if (auto pos = line.find('#'); pos != std::string::npos)
            line.resize(pos);
        trim(line);
        if (line.empty())
            continue;
        // split key and value
        auto pos = line.find('=');
        if (pos == std::string::npos)
            continue;
        std::string key = line.substr(0, pos);
        std::string val = line.substr(pos + 1);
```

```
        trim(key);
        trim(val);
        try
        {
            if (key == "random_seed") // Seed for the random number generator
                p.random_seed = std::stoi(val);
            else if (key == "num_units") // Number of units in the circuit
                p.num_units = std::stoi(val);
            else if (key == "mode")                   // Optimization mode: discrete, continuous, or hybrid
                p.mode = val;                         // d, c, or h
            else if (key == "max_iterations") // Maximum number of generations
                p.max_iterations = std::stoi(val);
            else if (key == "population_size") // Number of individuals in the population
                p.population_size = std::stoi(val);
            else if (key == "elite_count") // Number of best individuals to keep unchanged
                p.elite_count = std::stoi(val);
            else if (key == "tournament_size") // Number of contenders per tournament
                p.tournament_size = std::stoi(val);
            else if (key == "selection_pressure") // Linear rank selection pressure parameter
                p.selection_pressure = std::stod(val);
            else if (key == "crossover_probability") // Probability of crossover
                p.crossover_probability = std::stod(val);
            else if (key == "crossover_points") // Number of crossover points (1 or 2)
                p.crossover_points = std::stoi(val);
            else if (key == "mutation_probability") // Probability of mutation per gene
                p.mutation_probability = std::stod(val);
            else if (key == "mutation_step_size") // Maximum change in value during mutation
                p.mutation_step_size = std::stoi(val);
            else if (key == "allow_mutation_wrapping") // Allow mutations to wrap around
                p.allow_mutation_wrapping = (val == "true" || val == "1");
            else if (key == "use_inversion") // Use inversion mutation
                p.use_inversion = (val == "true" || val == "1");
            else if (key == "inversion_probability") // Probability of inversion mutation
                p.inversion_probability = std::stod(val);
            else if (key == "use_scaling_mutation") // Use scaling mutation
                p.use_scaling_mutation = (val == "true" || val == "1");
            else if (key == "scaling_mutation_prob") // Probability of scaling mutation
                p.scaling_mutation_prob = std::stod(val);
            else if (key == "scaling_mutation_min") // Minimum scaling factor
                p.scaling_mutation_min = std::stod(val);
            else if (key == "scaling_mutation_max") // Maximum scaling factor
                p.scaling_mutation_max = std::stod(val);
            else if (key == "convergence_threshold") // Convergence threshold
                p.convergence_threshold = std::stod(val);
            else if (key == "stall_generations") // Max generations with no improvement
                p.stall_generations = std::stoi(val);
            else if (key == "verbose") // Print progress information
                p.verbose = (val == "true" || val == "1");
            else if (key == "log_results") // Log results to file
                p.log_results = (val == "true" || val == "1");
            else if (key == "log_file") // Log file name
                p.log_file = val;
            else
            {
                std::cerr « "Warning: unknown parameter '" « key « "' in " « file « "\n";
            }
        }
        catch (...)
        {
            std::cerr « "Warning: could not parse '" « key « "='" « val « "'\n";
        }
    }
}
```

## 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_PALUSZNIUM_STREAM = -300.0
- constexpr double Constants::Economic::PALUSZNIUM_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_PALUSZNIUM_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.

```
/**
 * @file constants.h
 * @brief Header file for project-wide constants
 *
 * This file defines physical constants, economic parameters,
 * and other global values used throughout the project.
 */

#pragma once

namespace Constants
{
namespace Test
{
// Feed rates (kg/s)
constexpr double DEFAULT_PALUSZNIUM_FEED = 10.0;
constexpr double DEFAULT_GORMANIUM_FEED = 10.0;
constexpr double DEFAULT_WASTE_FEED = 10.0;

// Values in £/kg
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;

// Operating cost parameters
constexpr double COST_COEFFICIENT = 5.0;
constexpr double VOLUME_PENALTY_COEFFICIENT = 1000.0;

// Material properties
constexpr double MATERIAL_DENSITY = 3000.0; // kg/m^3, density of all solid materials
constexpr double SOLIDS_CONTENT = 0.1;      // Fraction of solids by volume

// Rate constants (s^-1)
constexpr double K_PALUSZNIUM = 0.008; // Rate constant for Palusznium
```

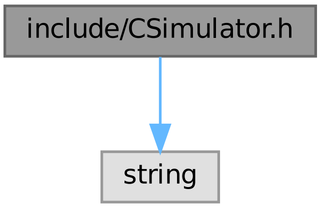
```
constexpr double K_GORMANIUM = 0.004;  // Rate constant for Gormanium
constexpr double K_WASTE = 0.0005;     // Rate constant for Waste

constexpr double DEFAULT_UNIT_VOLUME = 5.0;  // m^3
constexpr double MIN_UNIT_VOLUME = 2.5;      // m^3 (for variable case)
constexpr double MAX_UNIT_VOLUME = 20.0;     // m^3 (for variable case)
constexpr double MAX_CIRCUIT_VOLUME = 150.0; // m^3

constexpr int DEFAULT_NUM_UNITS = 10; // Default number of units in circuit

} // namespace Test

// Physical constants
namespace Physical
{
// Material properties
constexpr double MATERIAL_DENSITY = 3000.0; // kg/m^3, density of all solid materials
constexpr double SOLIDS_CONTENT = 0.1;      // Fraction of solids by volume

// Rate constants (s^-1)
constexpr double K_PALUSZNIUM = 0.008; // Rate constant for Palusznium
constexpr double K_GORMANIUM = 0.004;  // Rate constant for Gormanium
constexpr double K_WASTE = 0.0005;     // Rate constant for Waste
} // namespace Physical

// Economic parameters
namespace Economic
{
// Values in £/kg
constexpr double PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM = 120.0;
constexpr double GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM = -20.0;
constexpr double WASTE_PENALTY_IN_PALUSZNIUM_STREAM = -300.0;

constexpr double PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM = 0.0;
constexpr double GORMANIUM_VALUE_IN_GORMANIUM_STREAM = 80.0;
constexpr double WASTE_PENALTY_IN_GORMANIUM_STREAM = -25.0;

// Operating cost parameters
constexpr double COST_COEFFICIENT = 5.0;
constexpr double VOLUME_PENALTY_COEFFICIENT = 1000.0;
} // namespace Economic

// Default feed rates (kg/s)
namespace Feed
{
constexpr double DEFAULT_PALUSZNIUM_FEED = 8.0;
constexpr double DEFAULT_GORMANIUM_FEED = 12.0;
constexpr double DEFAULT_WASTE_FEED = 80.0;
} // namespace Feed

// Circuit parameters
namespace Circuit
{
constexpr double DEFAULT_UNIT_VOLUME = 10.0; // m^3
constexpr double MIN_UNIT_VOLUME = 2.5;      // m^3 (for variable case)
constexpr double MAX_UNIT_VOLUME = 20.0;     // m^3 (for variable case)
constexpr double MAX_CIRCUIT_VOLUME = 150.0; // m^3

constexpr int DEFAULT_NUM_UNITS = 10; // Default number of units in circuit
} // namespace Circuit

// Simulation parameters
namespace Simulation
{
constexpr double DEFAULT_TOLERANCE = 1e-6;
constexpr int DEFAULT_MAX_ITERATIONS = 1000;
constexpr double MIN_FLOW_RATE = 1e-6; // Minimum flow rate to prevent numerical issues
} // namespace Simulation

// Genetic algorithm parameters
namespace GA
{
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;
} // namespace GA
} // namespace Constants
```

## 7.9 include/CSimulator.h File Reference

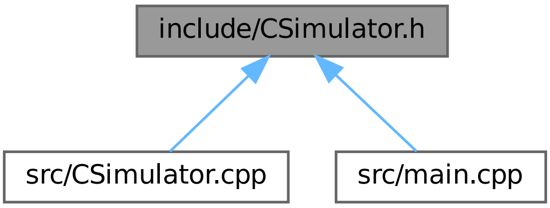
C++ header file for the circuit simulator.

```
#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

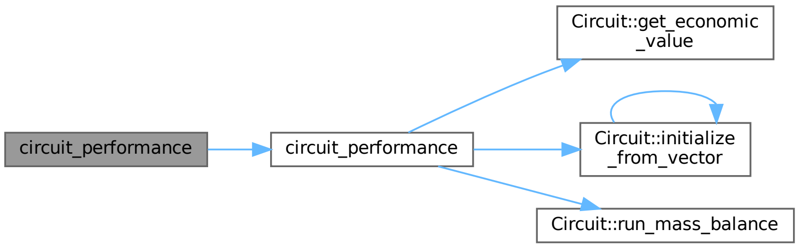
Definition in file CSimulator.h.

### 7.9.2 Function Documentation

#### circuit_performance() [1/6]

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

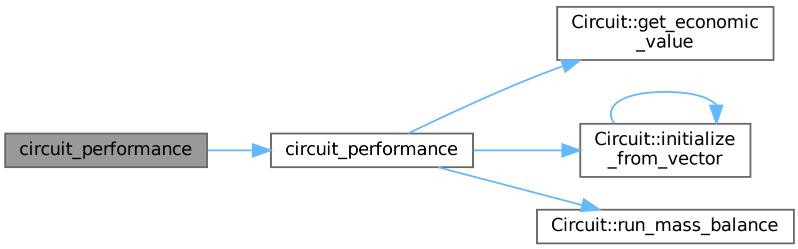
Definition at line 79 of file CSimulator.cpp.
References circuit_performance(), and default_simulator_parameters.
Here is the call graph for this function:




#### 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:

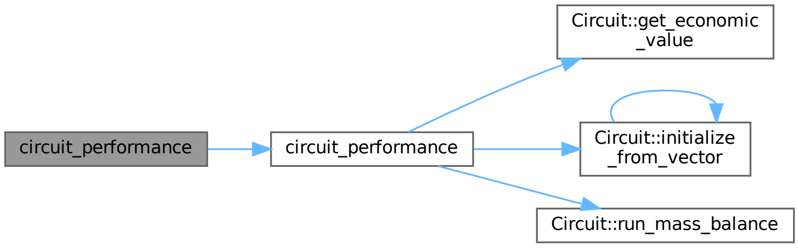



#### circuit_performance() [3/6]

```
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.

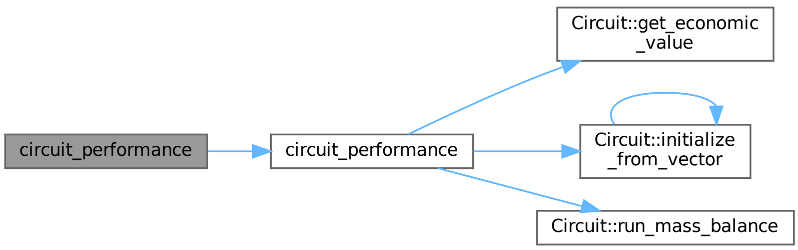
Here is the call graph for this function:




### 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.

```
/**
 * @file CSimulator.h
 * @brief 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
 */
#pragma once

#include <string>

// Structure to hold the simulation parameters
struct Simulator_Parameters
{
    // Convergence parameters
    double tolerance = 1e-6;
    int max_iterations = 1000;

    // Material properties
    double material_density = 3000.0; // kg/m^3, density of all solid materials
    double solids_content = 0.1;      // Fraction of solids by volume

    // Rate constants (s^-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;

    // Feed rates (kg/s)
    double feed_palusznium = 8.0;
    double feed_gormanium = 12.0;
    double feed_waste = 80.0;

    // Economic parameters (£/kg)
    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;
```

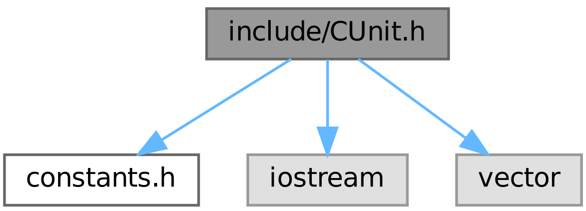
```
    double waste_penalty_in_gormanium_stream = -25.0;

    // Unit volume parameters
    double fixed_unit_volume = 10.0;   // m^3
    double min_unit_volume = 2.5;      // m^3 (for variable case)
    double max_unit_volume = 20.0;     // m^3 (for variable case)
    double max_circuit_volume = 150.0; // m^3

    // Circuit operating cost parameters
    double cost_coefficient = 5.0;
    double volume_penalty_coefficient = 1000.0;

    // Visualization options
    bool generate_visualization = false;
    std::string visualization_file = "circuit.dot";
};

// Default simulation parameters
extern Simulator_Parameters default_simulator_parameters;

// Main circuit performance evaluation function
double circuit_performance(int vector_size, int* circuit_vector, int unit_parameters_size, double* unit_parameters,
                           Simulator_Parameters simulator_parameters);

// Overloaded functions for convenience
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);
```

## 7.11 include/CUnit.h File Reference

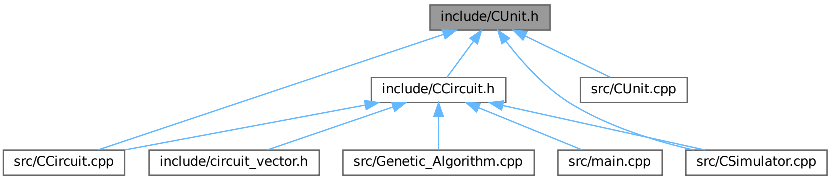
Declares the CUnit class – a single separation unit in the mineral-processing circuit (e.g.

```
#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.

```
/**
 * @file CUnit.h
 * @brief 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.
 */

#pragma once

#include "constants.h" // contains default k-values, density, , etc.
#include <iostream>
#include <vector>

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

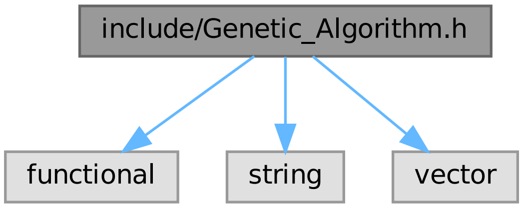
```
    int tails_num;
    // A Boolean that is changed to true if the unit has been seen during graph
    // traversal
    bool mark;

    /* ---------------- Physical / kinetic parameters ----------------------- */
    double volume; ///< Unit volume V  (m^3) - default 10 m^3
    double V_min;  ///< Minimum volume (m^3) - default 2.5 m^3
    double V_max;  ///< Maximum volume (m^3) - default 20 m^3

    // Material flow rates (kg/s)
    double feed_palusznium; // Palusznium in feed
    double feed_gormanium;  // Gormanium in feed
    double feed_waste;      // Waste material in feed

    // Rate constants for separation (s^-1)
    double k_palusznium; // Rate constant for Palusznium
    double k_gormanium;  // Rate constant for Gormanium
    double k_waste;      // Rate constant for Waste

    /* -------------------- Computed outlet mass flowrates ------------------ */
    // Concentrate stream
    double conc_palusznium; ///< C_P  (kg s^-1)
    double conc_gormanium;  ///< C_G  (kg s^-1)
    double conc_waste;      ///< C_W  (kg s^-1)
    // Tails stream
    double tails_palusznium; ///< T_P  (kg s^-1)
    double tails_gormanium;  ///< T_G  (kg s^-1)
    double tails_waste;      ///< T_W  (kg s^-1)

    double rho;
    double phi;

    /* -------------------- Computed recoveries for each component ----------- */
    double Rp, Rg, Rw; ///< Recoveries for each component

    /* --------------------------- Constructors ----------------------------- */
    /// Default constructor - initialises all numeric members to zero and
    /// routes to invalid destinations (e.g. -1) until set by GA vector.
    CUnit();

    /// Convenience constructor - sets outlet destinations; remaining
    /// parameters are pulled from constants.h defaults.
    /// @param conc  Destination index for concentrate
    /// @param tails Destination index for tails
    CUnit(int conc, int tails);

    /* ----------------------------- Methods ------------------------------- */
    /**
     * @brief 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.
     */

    CUnit(int conc, int tails, bool testFlag);

    void process();

    /**
     * @brief Check if the unit is valid.
     *
     * 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
     *
     * @return true if valid, false otherwise.
     */
    // double calculate_recovery(const string& component, double feed_rate) const;

    /**
     * @brief Update the volume of the unit.
     *
     * @param beta The new volume of the unit.
     */
    void update_volume(double beta);
};
```

## 7.13 include/Genetic_Algorithm.h File Reference

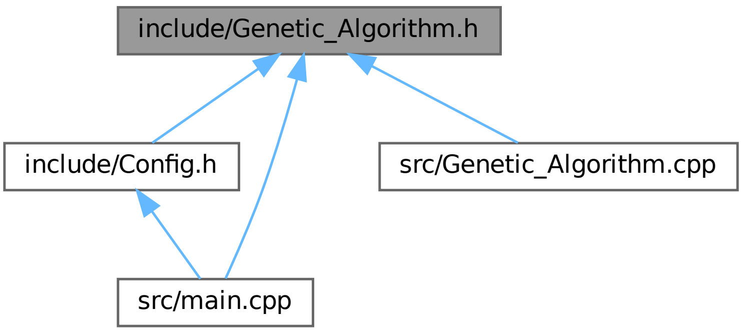
Genetic Algorithm Header.

`#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**

| | |
|---|---|
| *iv* | Size of the vector |
| *ivs* | Pointer to the vector |
| *rv* | Size of the real vector |
| *rvs* | 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**

| | |
|---|---|
| *num_units* | Number of units in the circuit |

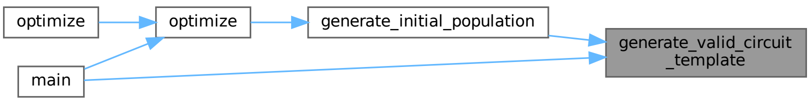
**Returns**

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**

| | |
|---|---|
| *int_vector_size* | Size of the integer vector |
| *int_vector* | Pointer to the integer vector |
| *real_vector_size* | Size of the real vector |
| *real_vector* | Pointer to the real vector |
| *hybrid_func* | Function to evaluate the fitness of the circuit |
| *hybrid_validity* | Function to check the validity of the circuit |
| *params* | Algorithm parameters for the optimization process |

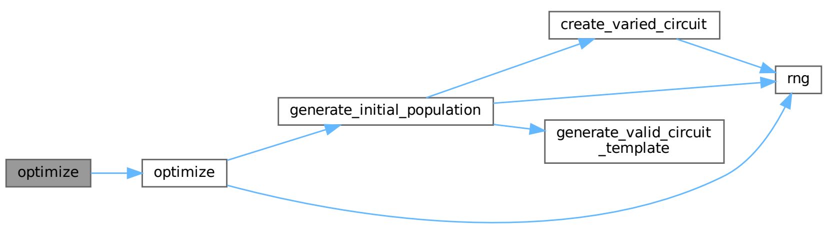
**Returns**

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.

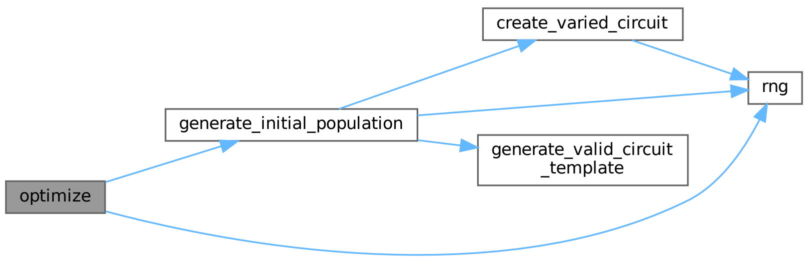
**Parameters**

| | |
|---|---|
| *int_vector_size* | Size of the integer vector |
| *int_vector* | Pointer to the integer vector |
| *func* | Function to evaluate the fitness of the circuit |
| *validity* | Function to check the validity of the circuit |
| *params* | Algorithm parameters for the optimization process |

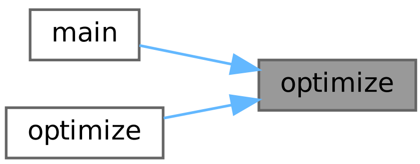
**Returns**

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_ste
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**

| | |
|---|---|
| *real_vector_size* | Size of the real vector |
| *real_vector* | Pointer to the real vector |
| *func* | Function to evaluate the fitness of the circuit |
| *validity* | Function to check the validity of the circuit |
| *params* | Algorithm parameters for the optimization process |

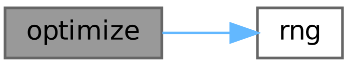
**Returns**

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_generatio 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.

```
/**
 * @file Genetic_Algorithm.h
 * @brief Genetic Algorithm Header
 * @author
 *
 * This header defines the interface for the genetic algorithm optimization
 */

#pragma once

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

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

// Structure to hold genetic algorithm parameters
struct Algorithm_Parameters
{
    // Number of units
    int num_units = 10;

    // Optimization mode: : "d", "c", or "h"
    std::string mode = "h";

    // General parameters
    int random_seed = -1;
    int max_iterations = 1000; // Maximum number of generations
    int population_size = 100; // Number of individuals in the population
    int elite_count = 1;       // Number of best individuals to keep unchanged

    // Selection parameters
    double selection_pressure = 1.5; // Linear rank selection pressure parameter
    int tournament_size = 2;         // Number of contenders per tournament

    // Crossover parameters
    double crossover_probability = 0.8; // Probability of crossover
    int crossover_points = 1;           // Number of crossover points (1 or 2)

    // Mutation parameters
    double mutation_probability = 0.01;  // Probability of mutation per gene
    int mutation_step_size = 2;          // Maximum change in value during mutation
    bool allow_mutation_wrapping = true; // Allow mutations to wrap around

    // Inversion-mutation parameters
    bool use_inversion = true;           // turn inversion on/off
    double inversion_probability = 0.05; // chance to invert per child

    // Scaling-mutation parameters
    bool use_scaling_mutation = true;
    double scaling_mutation_prob = 0.2; // how often to apply a scale mutation
    double scaling_mutation_min = 0.8;  // lower bound on the scale factor
    double scaling_mutation_max = 1.2;  // upper bound on the scale factor

    // Termination criteria
    double convergence_threshold = 1e-6; // Convergence threshold
    int stall_generations = 50;          // Max generations with no improvement

    // Debug options
    bool verbose = false;                // Print progress information
    bool log_results = false;            // Log results to file
    std::string log_file = "ga_log.txt"; // Log file name
};

// Default algorithm parameters
#define DEFAULT_ALGORITHM_PARAMETERS \
    Algorithm_Parameters {}

// Validity checking functions
bool all_true(int int_vector_size, int* int_vector, int real_vector_size, double* real_vector);
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);
// Optimization function for discrete vector
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 = DEFAULT_ALGORITHM_PARAMETERS);

// Optimization function for continuous vector
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 = DEFAULT_ALGORITHM_PARAMETERS);

```

```
// Optimization function for mixed discrete-continuous vector
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 = DEFAULT_ALGORITHM_PARAMETERS);

// Structure to hold statistics about the optimization process
struct OptimizationResult
{
    double best_fitness; // Best fitness value found
    int generations;     // Number of generations run
    double avg_fitness;  // Average fitness of final population
    double std_fitness;  // Standard deviation of final population fitness
    double time_taken;   // Time taken for optimization (seconds)
    bool converged;      // Whether algorithm converged

    // Default constructor
    OptimizationResult()
        : best_fitness(0), generations(0), avg_fitness(0), std_fitness(0), time_taken(0), converged(false)
    {
    }
};

// Get the last optimization result
OptimizationResult get_last_optimization_result();
```

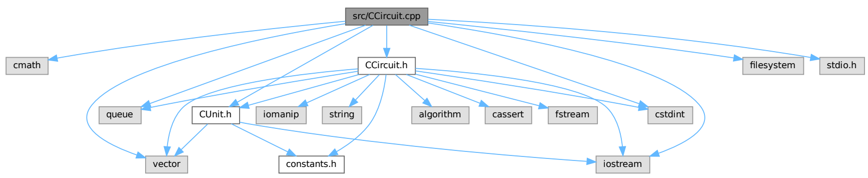
## 7.15 README.md File Reference

## 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.

```
/**
 * @file CCircuit.cpp
 * @brief 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.
 *
 */
#include <cmath>
#include <queue>
#include <vector>

#include <CCircuit.h>
#include <CUnit.h>
#include <cstdint>
#include <filesystem>
#include <iostream>
#include <stdio.h>

/**
 * @brief Constructor for the Circuit class
 *
 */
Circuit::Circuit(int num_units)
    : units(num_units), feed_unit(0), feed_palusznium_rate(Constants::Feed::DEFAULT_PALUSZNIUM_FEED),
      feed_gormanium_rate(Constants::Feed::DEFAULT_GORMANIUM_FEED),
      feed_waste_rate(Constants::Feed::DEFAULT_WASTE_FEED), palusznium_product_palusznium(0.0),
      palusznium_product_gormanium(0.0), palusznium_product_waste(0.0), gormanium_product_palusznium(0.0),
      gormanium_product_gormanium(0.0), gormanium_product_waste(0.0), tailings_palusznium(0.0), tailings_gormanium(0.0),
      tailings_waste(0.0), paluszium_value(Constants::Economic::PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM),
      gormanium_value(Constants::Economic::GORMANIUM_VALUE_IN_GORMANIUM_STREAM),
      waste_penalty_palusznium(Constants::Economic::WASTE_PENALTY_IN_PALUSZNIUM_STREAM),
      waste_penalty_gormanium(Constants::Economic::WASTE_PENALTY_IN_GORMANIUM_STREAM), beta(nullptr),
      circuit_vector(nullptr), n(num_units)
{
}

/**
 * @brief 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.
 *
 * @param vector_size Size of the circuit vector
 * @param vec Circuit vector
 *
 * @return true if the circuit is valid, false otherwise
 *
 */
bool Circuit::check_validity(int vector_size, const int* vec)
{
    // 1. length must be 2*n+1
    int expected = 2 * n + 1;
    if (vector_size != expected)
    {
        return false;
    }

    // 2. feed check: feed cannot directly feed to terminal
    feed_dest = vec[0]; // feed points to the unit
    if (feed_dest < 0 || feed_dest >= n)
    {
        return false;
    }

    // read each unit's concentrate and tailing and do static check
    struct Dest
    {
        int conc;
        int tail;
    };
    std::vector<Dest> dest(n);

    int max_idx = n + 2; // the last valid index

    for (int i = 0; i < n; ++i)
    {
        int conc = vec[1 + 2 * i]; // conc points to the unit
        int tail = vec[2 + 2 * i]; // tail points to the unit

        // 3. index check: conc must be in (0, n+2), tail must be in (0, n+2)
        if (conc < 0 || conc > max_idx)
        {
            return false;
        }
        if (tail < 0 || tail > max_idx)
```

```
        {
            return false;
        }

        // 4. no self-loop: conc cannot be equal to i, tail cannot be equal to i
        if (conc == i || tail == i)
        {
            return false;
        }

        // 5. same output: conc cannot be equal to tail
        if (conc == tail)
        {
            return false;
        }

        dest[i] = {conc, tail};

        units[i].conc_num = conc;
        units[i].tails_num = tail;
        units[i].mark = false;
    }

    // 6. reachability check: all units must be reachable from feed
    this->mark_units(feed_dest);

    for (int i = 0; i < n; ++i)
    {
        if (!units[i].mark)
        {
            return false;
        }
    }

    // 7. two terminals check: each unit must reach at least 2 different terminals
    std::vector<int8_t> cache(n, -1);
    uint8_t global_mask = 0;
    for (int i = 0; i < n; ++i)
    {
        uint8_t mask = this->term_mask(i);
        global_mask |= mask;
        int cnt = (mask & 1) + ((mask » 1) & 1) + ((mask » 2) & 1);
        if (cnt < 2)
        {
            return false;
        }
    }

    // 8. final terminal check: P1/P2, TA must be present
    if ((global_mask & (0b001 | 0b010)) == 0)
    {
        return false;
    }

    if ((global_mask & 0b100) == 0)
    {
        return false;
    }

    // 9. mass balance check: mass balance must converge
    if (!run_mass_balance(1e-6, 100))
    {
        return false;
    }

    return true;
}

/**
 * @brief 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.
 *
 * @param vector_size Size of the circuit vector
 * @param circuit_vector Circuit vector
 * @param unit_parameters_size Size of the unit parameters
 * @param unit_parameters Unit parameters
 *
 * @return true if the circuit vector and parameters are valid, false otherwise
 */
bool Circuit::check_validity(int vector_size, const int* circuit_vector, int unit_parameters_size,
                             double* unit_parameters)
{
    bool valid = check_validity(vector_size, circuit_vector);
    // check the validity of the circuit vector
```

```
    if (!valid)
    {
        return false;
    }
    // check the validity of the unit parameters
    if (unit_parameters == nullptr)
    {
        return valid;
    }

    // the length of the continuous parameters must be exactly n units
    if (unit_parameters_size != n)
    {
        return false;
    }

    // each parameter must be in [0,1] (or other physical range)
    for (int i = 0; i < unit_parameters_size; ++i)
    {
        double beta = unit_parameters[i];
        if (beta < 0.0 || beta > 1.0 || std::isnan(beta))
        {
            return false;
        }
    }

    return true;
}

/**
 * @brief 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.
 *
 * @param unit_num The unit number to start marking from
 */
void Circuit::mark_units(int unit_num)
{

    if (this->units[unit_num].mark)
        return;

    this->units[unit_num].mark = true;

    // If we have seen this unit already exit
    // Mark that we have now seen the unit

    // If conc_num does not point at a circuit outlet recursively call the
    // function
    if (this->units[unit_num].conc_num < this->units.size())
    {
        mark_units(this->units[unit_num].conc_num);
    }

    // If tails_num does not point at a circuit outletrecursively call the
    // function

    if (this->units[unit_num].tails_num < this->units.size())
    {
        mark_units(this->units[unit_num].tails_num);
    }
}

/**
 * @brief Constructor for the Circuit class
 *
 * This constructor initializes the circuit with the given number of units
 * and a pointer to the beta array.
 *
 * @param num_units Number of units in the circuit
 * @param beta Pointer to the beta array
 */
Circuit::Circuit(int num_units, double* beta)
    : units(num_units), feed_unit(0), n(num_units),

      feed_palusznium_rate(Constants::Feed::DEFAULT_PALUSZNIUM_FEED),
      feed_gormanium_rate(Constants::Feed::DEFAULT_GORMANIUM_FEED),
      feed_waste_rate(Constants::Feed::DEFAULT_WASTE_FEED), palusznium_product_palusznium(0.0),
      palusznium_product_gormanium(0.0), palusznium_product_waste(0.0),
gormanium_product_palusznium(0.0),
      gormanium_product_gormanium(0.0), gormanium_product_waste(0.0), tailings_palusznium(0.0),
tailings_gormanium(0.0),
      tailings_waste(0.0), beta(beta),
palusznium_value(Constants::Economic::PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM),
      gormanium_value(Constants::Economic::GORMANIUM_VALUE_IN_GORMANIUM_STREAM),
```

```
      waste_penalty_palusznium(Constants::Economic::WASTE_PENALTY_IN_PALUSZNIUM_STREAM),
      waste_penalty_gormanium(Constants::Economic::WASTE_PENALTY_IN_GORMANIUM_STREAM),
      palusznium_value_in_gormanium(Constants::Economic::PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM),
      gormanium_value_in_palusznium(Constants::Economic::GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM)
{
}

/**
 * @brief 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.
 *
 * @param num_units Number of units in the circuit
 * @param beta Pointer to the beta array
 * @param testFlag Test flag to indicate whether to use test parameters
 */
Circuit::Circuit(int num_units, double* beta, bool testFlag)
    : units(num_units), feed_unit(0), n(num_units),

      feed_palusznium_rate(Constants::Feed::DEFAULT_PALUSZNIUM_FEED),
      feed_gormanium_rate(Constants::Feed::DEFAULT_GORMANIUM_FEED),
      feed_waste_rate(Constants::Feed::DEFAULT_WASTE_FEED), palusznium_product_palusznium(0.0),
      palusznium_product_gormanium(0.0), palusznium_product_waste(0.0), gormanium_product_palusznium(0.0),
      gormanium_product_gormanium(0.0), gormanium_product_waste(0.0), tailings_palusznium(0.0), tailings_gormanium(0.0),
      tailings_waste(0.0), beta(beta), palusznium_value(Constants::Economic::PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM),
      gormanium_value(Constants::Economic::GORMANIUM_VALUE_IN_GORMANIUM_STREAM),
      waste_penalty_palusznium(Constants::Economic::WASTE_PENALTY_IN_PALUSZNIUM_STREAM),
      waste_penalty_gormanium(Constants::Economic::WASTE_PENALTY_IN_GORMANIUM_STREAM),
      palusznium_value_in_gormanium(Constants::Economic::PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM),
      gormanium_value_in_palusznium(Constants::Economic::GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM)
{
    if (testFlag)
    {
        this->feed_palusznium_rate = Constants::Test::DEFAULT_PALUSZNIUM_FEED;
        this->feed_gormanium_rate = Constants::Test::DEFAULT_GORMANIUM_FEED;
        this->feed_waste_rate = Constants::Test::DEFAULT_WASTE_FEED;

        this->palusznium_value = Constants::Test::PALUSZNIUM_VALUE_IN_PALUSZNIUM_STREAM;
        this->gormanium_value = Constants::Test::GORMANIUM_VALUE_IN_GORMANIUM_STREAM;
        this->waste_penalty_palusznium = Constants::Test::WASTE_PENALTY_IN_PALUSZNIUM_STREAM;
        this->waste_penalty_gormanium = Constants::Test::WASTE_PENALTY_IN_GORMANIUM_STREAM;
        this->palusznium_value_in_gormanium = Constants::Test::PALUSZNIUM_VALUE_IN_GORMANIUM_STREAM;
        this->gormanium_value_in_palusznium = Constants::Test::GORMANIUM_VALUE_IN_PALUSZNIUM_STREAM;
    }
}

/**
 * @brief 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.
 *
 * @param vector_size Size of the circuit vector
 * @param circuit_vector Circuit vector
 *
 * @return true if initialization is successful, false otherwise
 */
bool Circuit::initialize_from_vector(int vector_size, const int* circuit_vector)
{
    return initialize_from_vector(vector_size, circuit_vector, nullptr, false);
}

/**
 * @brief 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.
 *
 * @param vector_size Size of the circuit vector
 * @param circuit_vector Circuit vector
 * @param beta Pointer to the beta array
 *
 * @return true if initialization is successful, false otherwise
 */
bool Circuit::initialize_from_vector(int vector_size, const int* circuit_vector, const double* beta)
{
    return initialize_from_vector(vector_size, circuit_vector, beta, false);
}

/**
 * @brief 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.
 *
 * @param vector_size Size of the circuit vector
 * @param circuit_vector Circuit vector
 * @param testFlag Test flag to indicate whether to use test parameters
 *
 * @return true if initialization is successful, false otherwise
 *
 */
bool Circuit::initialize_from_vector(int vector_size, const int* circuit_vector, bool testFlag)
{

    // Initialize the circuit from the circuit vector
    return initialize_from_vector(vector_size, circuit_vector, nullptr, testFlag);
}

/**
 * @brief 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.
 *
 * @param vector_size Size of the circuit vector
 * @param circuit_vector Circuit vector
 * @param beta Pointer to the beta array
 * @param testFlag Test flag to indicate whether to use test parameters
 *
 * @return true if initialization is successful, false otherwise
 */
bool Circuit::initialize_from_vector(int vector_size, const int* circuit_vector, const double* beta,
      bool testFlag)
{
    // num_units = n
    int num_units = (vector_size - 1) / 2;
    if (vector_size != 2 * num_units + 1)
        return false;
    units.resize(num_units);
    this->circuit_vector = circuit_vector;

    // feed_unit is the first element of the circuit vector
    feed_unit = circuit_vector[0];

    // Map the target units to corresponding unit numbers
    for (int i = 0; i < num_units; ++i)
    {
        int conc = circuit_vector[1 + 2 * i];
        int tails = circuit_vector[1 + 2 * i + 1];

        // transform the unit numbers from n, n+1, n+2 to -1, -2, -3
        if (conc == num_units)
            conc = PALUSZNIUM_PRODUCT;
        else if (conc == num_units + 1)
            conc = GORMANIUM_PRODUCT;
        else if (conc == num_units + 2)
            conc = TAILINGS_OUTPUT;

        if (tails == num_units)
            tails = PALUSZNIUM_PRODUCT;
        else if (tails == num_units + 1)
            tails = GORMANIUM_PRODUCT;
        else if (tails == num_units + 2)
            tails = TAILINGS_OUTPUT;

        units[i] = CUnit(conc, tails, testFlag);
        if (beta != nullptr)
        {

            units[i].update_volume(beta[i]);
        }
    }
    return true;
}

/**
 * @brief 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.
 *
 * @param tolerance Tolerance for convergence
 * @param max_iterations Maximum number of iterations
 *
 * @return true if mass balance converges, false otherwise
 */
```

```cpp
bool Circuit::run_mass_balance(double tolerance, int max_iterations)
{
    // Initialize feed for all the units
    for (auto& u : units)
    {
        u.feed_palusznium = 0.0;
        u.feed_gormanium = 0.0;
        u.feed_waste = 0.0;
    }
    // Initialize feed for the first unit
    units[feed_unit].feed_palusznium = feed_palusznium_rate;
    units[feed_unit].feed_gormanium = feed_gormanium_rate;
    units[feed_unit].feed_waste = feed_waste_rate;

    // Record the last feed for each unit for convergence check
    std::vector<double> last_feed_p(units.size(), 0.0);
    std::vector<double> last_feed_g(units.size(), 0.0);
    std::vector<double> last_feed_w(units.size(), 0.0);
    // std::cout « "Unit number: " « units.size() « std::endl;

    for (int iter = 0; iter < max_iterations; ++iter)
    {

        // Record the current feed
        // Record the current feed to last_feed and clear the current feed
        if (iter == 0)
        {
            for (size_t i = 0; i < units.size(); ++i)
            {
                last_feed_p[i] = units[i].feed_palusznium;
                last_feed_g[i] = units[i].feed_gormanium;
                last_feed_w[i] = units[i].feed_waste;
                // clear the current feed
                units[i].feed_palusznium = 0.0;
                units[i].feed_gormanium = 0.0;
                units[i].feed_waste = 0.0;
            }
        }
        else
        {
            for (size_t i = 0; i < units.size(); ++i)
            {
                last_feed_p[i] = units[i].feed_palusznium;
                last_feed_g[i] = units[i].feed_gormanium;
                last_feed_w[i] = units[i].feed_waste;
            }
        }
        // Initialize feed for the first unit
        units[feed_unit].feed_palusznium = feed_palusznium_rate;
        units[feed_unit].feed_gormanium = feed_gormanium_rate;
        units[feed_unit].feed_waste = feed_waste_rate;

        // Process all units
        for (size_t i = 0; i < units.size(); ++i)
        {
            units[i].process();
        }

        // This vector is used to mark whether the feed for each unit has been
        // cleared We need to make sure that the feed for each unit is cleared only
        // once
        std::vector<bool> feedCleared(units.size(), false);

        // Initialize the product flow rates
        palusznium_product_palusznium = palusznium_product_gormanium = palusznium_product_waste = 0.0;
        gormanium_product_palusznium = gormanium_product_gormanium = gormanium_product_waste = 0.0;
        tailings_palusznium = tailings_gormanium = tailings_waste = 0.0;

        // "=====Distributing downstream data====="«std::endl;
        for (size_t i = 0; i < units.size(); ++i)
        {
            // concentrate flow
            int concDest = units[i].conc_num;
            if (concDest == PALUSZNIUM_PRODUCT)
            {
                palusznium_product_palusznium += units[i].conc_palusznium;
                palusznium_product_gormanium += units[i].conc_gormanium;
                palusznium_product_waste += units[i].conc_waste;
            }
            else if (concDest == GORMANIUM_PRODUCT)
            {
                gormanium_product_palusznium += units[i].conc_palusznium;
                gormanium_product_gormanium += units[i].conc_gormanium;
                gormanium_product_waste += units[i].conc_waste;
            }
            else if (concDest == TAILINGS_OUTPUT)
            {
```

```
                tailings_palusznium += units[i].conc_palusznium;
                tailings_gormanium += units[i].conc_gormanium;
                tailings_waste += units[i].conc_waste;
            }
            else if (concDest >= 0 && concDest < (int)units.size())
            {
                if (!feedCleared[concDest])
                {
                    feedCleared[concDest] = true;
                    units[concDest].feed_palusznium = 0.0;
                    units[concDest].feed_gormanium = 0.0;
                    units[concDest].feed_waste = 0.0;
                }
                units[concDest].feed_palusznium += units[i].conc_palusznium;
                units[concDest].feed_gormanium += units[i].conc_gormanium;
                units[concDest].feed_waste += units[i].conc_waste;
            }

            // tailings flow
            int tailsDest = units[i].tails_num;
            if (tailsDest == PALUSZNIUM_PRODUCT)
            {
                palusznium_product_palusznium += units[i].tails_palusznium;
                palusznium_product_gormanium += units[i].tails_gormanium;
                palusznium_product_waste += units[i].tails_waste;
            }
            else if (tailsDest == GORMANIUM_PRODUCT)
            {
                gormanium_product_palusznium += units[i].tails_palusznium;
                gormanium_product_gormanium += units[i].tails_gormanium;
                gormanium_product_waste += units[i].tails_waste;
            }
            else if (tailsDest == TAILINGS_OUTPUT)
            {
                tailings_palusznium += units[i].tails_palusznium;
                tailings_gormanium += units[i].tails_gormanium;
                tailings_waste += units[i].tails_waste;
            }
            else if (tailsDest >= 0 && tailsDest < (int)units.size())
            {
                if (!feedCleared[tailsDest])
                {
                    feedCleared[tailsDest] = true;
                    units[tailsDest].feed_palusznium = 0.0;
                    units[tailsDest].feed_gormanium = 0.0;
                    units[tailsDest].feed_waste = 0.0;
                }

                units[tailsDest].feed_palusznium += units[i].tails_palusznium;
                units[tailsDest].feed_gormanium += units[i].tails_gormanium;
                units[tailsDest].feed_waste += units[i].tails_waste;
            }
        }

        // convergence check
        double max_rel_change = 0.0;
        for (size_t i = 0; i < units.size(); ++i)
        {
            double rel_p = std::abs(units[i].feed_palusznium - last_feed_p[i]) / std::max(last_feed_p[i], 1e-12);
            double rel_g = std::abs(units[i].feed_gormanium - last_feed_g[i]) / std::max(last_feed_g[i], 1e-12);
            double rel_w = std::abs(units[i].feed_waste - last_feed_w[i]) / std::max(last_feed_w[i], 1e-12);
            max_rel_change = std::max({max_rel_change, rel_p, rel_g, rel_w});
        }

        if (max_rel_change < tolerance)
            return true;
    }
    return false; // not converged
}

/**
 * @brief 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.
 *
 * @return The economic value of the circuit
 *
 */
double Circuit::get_economic_value() const
{
    double value = 0.0;

    // Palusznium product
```

```
    value += palusznium_product_palusznium * palusznium_value;
    value += palusznium_product_gormanium * gormanium_value_in_palusznium;
    value += palusznium_product_waste * waste_penalty_palusznium;

    // Gormanium product
    value += gormanium_product_gormanium * gormanium_value;
    value += gormanium_product_palusznium * palusznium_value_in_gormanium;
    value += gormanium_product_waste * waste_penalty_gormanium;

    double total_volume = 0.0;
    for (const auto& u : units)
        total_volume += u.volume;
    double cost = 5.0 * std::pow(total_volume, 2.0 / 3.0);
    if (total_volume >= 150.0)
    {
        cost += 1000.0 * std::pow(total_volume - 150.0, 2.0);
    }
    value -= cost; // cost of the circuit
    return value;
}

/**
 * @brief 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.
 *
 * @return The recovery of valuable materials
 *
 */
double Circuit::get_palusznium_recovery() const
{

    double total_feed = feed_palusznium_rate;
    double recovered = palusznium_product_palusznium;
    if (total_feed < 1e-12)
        return 0.0;
    return recovered / total_feed;
}

/**
 * @brief 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.
 *
 * @return The recovery of gormanium
 *
 */
double Circuit::get_gormanium_recovery() const
{
    double total_feed = feed_gormanium_rate;
    double recovered = gormanium_product_gormanium;
    if (total_feed < 1e-12)
        return 0.0;
    return recovered / total_feed;
}

/**
 * @brief Get the grade of palusznium
 *
 * This function calculates the grade of palusznium in the circuit
 * based on the product flow rates.
 *
 * @return The grade of palusznium
 *
 */
double Circuit::get_palusznium_grade() const
{
    double total = palusznium_product_palusznium + palusznium_product_gormanium + palusznium_product_waste;
    return (total > 0) ? (palusznium_product_palusznium / total) : 0.0;
}

/**
 * @brief Get the grade of gormanium
 *
 * This function calculates the grade of gormanium in the circuit
 * based on the product flow rates.
 *
 * @return The grade of gormanium
 *
 */
double Circuit::get_gormanium_grade() const
{
    double total = gormanium_product_palusznium + gormanium_product_gormanium + gormanium_product_waste;
```

```
    return (total > 0) ? (gormanium_product_gormanium / total) : 0.0;
}

/**
 * @brief Export the circuit to a DOT file
 *
 * This function exports the circuit to a DOT file for visualization.
 *
 * @param filename The name of the output DOT file
 *
 * @return true if export is successful, false otherwise
 */
bool Circuit::export_to_dot(const std::string& filename) const
{
    std::ofstream ofs(filename);
    if (!ofs)
        return false;
    ofs « "digraph Circuit {\n";
    for (size_t i = 0; i < units.size(); ++i)
    {
        ofs « "  unit" « i « " [label=\"Unit " « i « "\"];\n";
        // concentrate flow
        if (units[i].conc_num >= 0)
            ofs « "  unit" « i « " -> unit" « units[i].conc_num « " [label=\"conc\"];\n";
        else if (units[i].conc_num == PALUSZNIUM_PRODUCT)
            ofs « "  unit" « i « " -> palusznium_product [label=\"conc\"];\n";
        else if (units[i].conc_num == GORMANIUM_PRODUCT)
            ofs « "  unit" « i « " -> gormanium_product [label=\"conc\"];\n";
        else if (units[i].conc_num == TAILINGS_OUTPUT)
            ofs « "  unit" « i « " -> tailings [label=\"conc\"];\n";
        // tailings flow
        if (units[i].tails_num >= 0)
            ofs « "  unit" « i « " -> unit" « units[i].tails_num « " [label=\"tails\"];\n";
        else if (units[i].tails_num == PALUSZNIUM_PRODUCT)
            ofs « "  unit" « i « " -> palusznium_product [label=\"tails\"];\n";
        else if (units[i].tails_num == GORMANIUM_PRODUCT)
            ofs « "  unit" « i « " -> gormanium_product [label=\"tails\"];\n";
        else if (units[i].tails_num == TAILINGS_OUTPUT)
            ofs « "  unit" « i « " -> tailings [label=\"tails\"];\n";
    }
    ofs « "  palusznium_product [shape=box, label=\"Palusznium Product\"];\n";
    ofs « "  gormanium_product [shape=box, label=\"Gormanium Product\"];\n";
    ofs « "  tailings [shape=box, label=\"Tailings\"];\n";
    ofs « "}\n";
    return true;
}

/**
 * @brief 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.
 *
 * @param start The starting unit number
 *
 * @return The terminal mask
 */
uint8_t Circuit::term_mask(int start) const
{
    uint8_t mask = 0;
    std::vector<bool> visited(n, false);

    // Use queue for breadth-first search
    std::queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty())
    {
        int current = q.front();
        q.pop();

        const int conc_dest = units[current].conc_num;
        const int tail_dest = units[current].tails_num;

        process_destination(conc_dest, mask, visited, q);
        process_destination(tail_dest, mask, visited, q);

        if ((mask & (mask - 1)) >= 3)
            break;
    }

    return mask;
}

/**
 * @brief 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.
 *
 * @param dest The destination unit number
 * @param mask The terminal mask
 * @param visited Vector to keep track of visited units
 * @param q Queue for breadth-first search
 */
void Circuit::process_destination(int dest, uint8_t& mask, std::vector<bool>& visited,
std::queue<int>& q) const
{
    if (dest >= n)
    {
        if (dest == OUT_P1())
            mask |= 0b001;
        else if (dest == OUT_P2())
            mask |= 0b010;
        else if (dest == OUT_TA())
            mask |= 0b100;
    }
    else
    {
        if (!visited[dest])
        {
            visited[dest] = true;
            q.push(dest);
        }
    }
}

/**
 * @brief Save the circuit output information to a CSV file
 *
 * This function saves the circuit output information to a CSV file.
 * It appends the data to the file if it already exists.
 *
 * @param filename The name of the output CSV file
 *
 * @return true if saving is successful, false otherwise
 */
bool Circuit::save_all_units_to_csv(const std::string& filename)
{
    std::ofstream ofs(filename, std::ios::app);
    if (!ofs.is_open())
    {
        std::cerr « "Error: Unable to open file " « filename « std::endl;
        return false;
    }

    // output in a single line
    ofs « std::fixed « std::setprecision(2);

    for (size_t i = 0; i < units.size(); ++i)
    {
        const CUnit& unit = units[i];

        ofs « unit.conc_palusznium + unit.conc_gormanium + unit.conc_waste « ","
            « unit.tails_palusznium + unit.tails_gormanium + unit.tails_waste;

        if (i < units.size() - 1)
        {
            ofs « ",";
        }
    }
    ofs « "\n";

    ofs.close();
    return true;
}

/**
 * @brief Save the circuit output information to a CSV file
 *
 * This function saves the circuit output information to a CSV file.
 * It appends the data to the file if it already exists.
 *
 * @param filename The name of the output CSV file
 *
 * @return true if saving is successful, false otherwise
 */
bool Circuit::save_vector_to_csv(const std::string& filename)
{
    std::ofstream ofs(filename, std::ios::app);
    if (!ofs.is_open())
    {
```

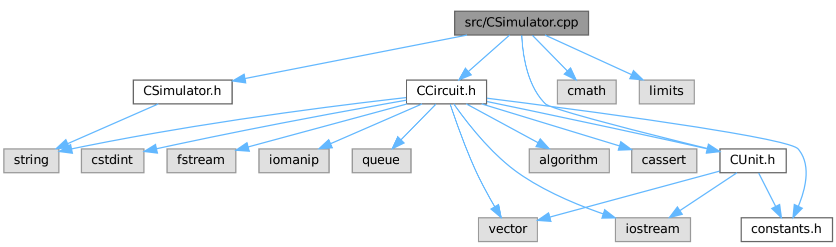
```
        std::cerr « "Error: Unable to open file " « filename « std::endl;
        return false;
    }

    int length = static_cast<int>(units.size()) * 2 + 1;

    for (int i = 0; i < length; ++i)
    {
        ofs « circuit_vector[i];
        if (i < length - 1)
        {
            ofs « ",";
        }
    }
    ofs « "\n";

    ofs.close();
    return true;
}

/**
 * @brief Save the circuit output information to a CSV file
 *
 * This function saves the circuit output information to a CSV file.
 * It appends the data to the file if it already exists.
 *
 * @param filename The name of the output CSV file
 *
 * @return true if saving is successful, false otherwise
 */
bool Circuit::save_output_info(const std::string& filename)
{
    namespace fs = std::filesystem; // If you see an error here it is because of the C++ version, it will compile fine
    fs::path p(filename);

    if (p.has_parent_path())
    {
        std::error_code ec;
        fs::create_directories(p.parent_path(), ec); // ok if it already exists
        if (ec)
        {
            std::cerr « "Cannot create directory " « p.parent_path() « " : " « ec.message() « '\n';
            return false;
        }
    }

    // truncate the file, then append the two blocks of data
    std::ofstream{filename, std::ios::trunc};
    return save_vector_to_csv(filename) && save_all_units_to_csv(filename);
}
```

## 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**

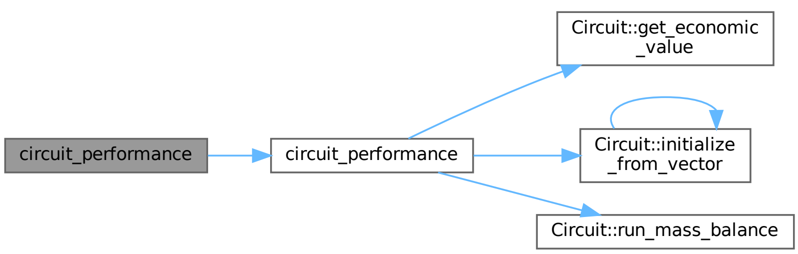
This source file contains the implementation of the function that will be used to evaluate the circuit and the parameters for the simulation.
Definition in file CSimulator.cpp.

### 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:

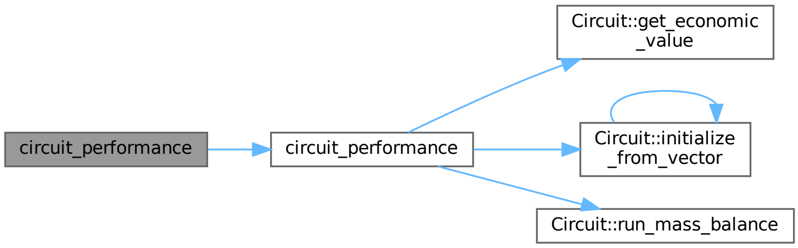



#### circuit_performance() [2/5]

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

Definition at line 95 of file CSimulator.cpp.

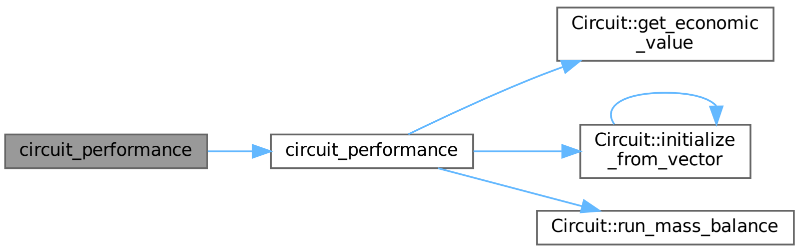
References circuit_performance(), and default_simulator_parameters.
Here is the call graph for this function:




### 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:

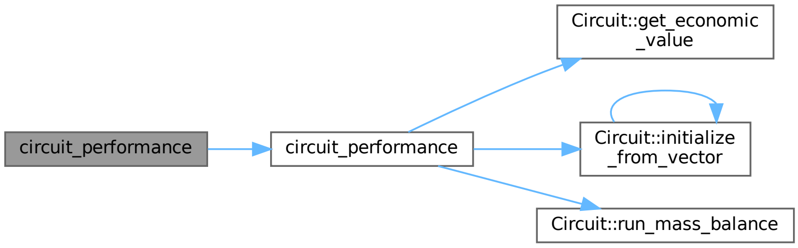



### circuit_performance() [4/5]

```
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.

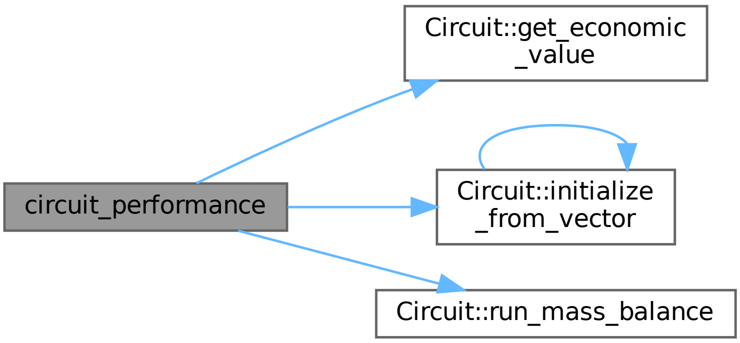
**Parameters**

| | |
|---|---|
| *vector_size* | Size of the circuit vector |
| *circuit_vector* | Circuit vector |
| *unit_parameters_size* | Size of the unit parameters |
| *unit_parameters* | Unit parameters |
| *simulator_parameters* | Simulation parameters |
| *testFlag* | Test flag to indicate whether to use test parameters |

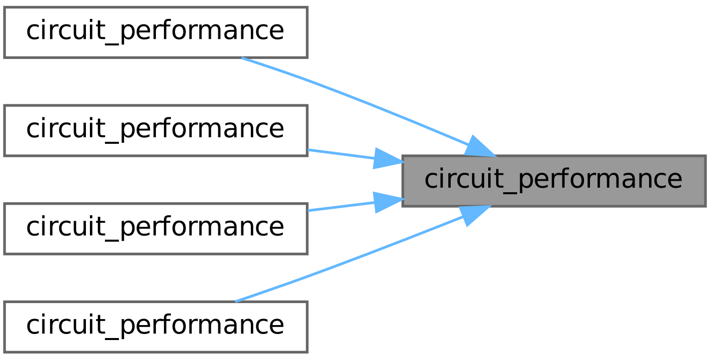
**Returns**

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.

```
/**
 * @file CSimulator.cpp
 * @brief 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.
 *
 */
#include "CSimulator.h"
#include "CCircuit.h"
#include "CUnit.h"
#include <cmath>
#include <limits>

// Default simulation parameters
struct Simulator_Parameters default_simulator_parameters = {1e-6, 100};

/**
 * @brief 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.
 *
 * @param vector_size Size of the circuit vector
 * @param circuit_vector Circuit vector
 * @param unit_parameters_size Size of the unit parameters
 * @param unit_parameters Unit parameters
 * @param simulator_parameters Simulation parameters
 * @param testFlag Test flag to indicate whether to use test parameters
 *
 * @return Economic value of the circuit
 *
 */
double circuit_performance(int vector_size, int* circuit_vector,

                           int unit_parameters_size, double* unit_parameters,
                           struct Simulator_Parameters simulator_parameters, bool testFlag)
{

    // Calculate the number of units
    int num_units = (vector_size - 1) / 2;
    // Check if the vector size is valid
    if (vector_size != 2 * num_units + 1 || num_units <= 0)
    {
        // Invalid vector size
        return -1e12;
    }

    // Initialize the circuit
    Circuit circuit(num_units, unit_parameters, testFlag);
    if (!circuit.initialize_from_vector(vector_size, circuit_vector, unit_parameters, testFlag))
    {
        // Invalid structure
        return -1e12;
    }

    // Run the mass balance
    bool converged = circuit.run_mass_balance(simulator_parameters.tolerance,
simulator_parameters.max_iterations);
    if (!converged)
    {
        // Not converged, consider invalid
        return -1e12;
    }

    // Return performance
    return circuit.get_economic_value();
}

// Overloads for other input

double circuit_performance(int vector_size, int* circuit_vector, int unit_parameters_size, double*
unit_parameters)
{
    return circuit_performance(vector_size, circuit_vector, unit_parameters_size, unit_parameters,
                               default_simulator_parameters, false);
}

double circuit_performance(int vector_size, int* circuit_vector)
{
    int num_parameters = (vector_size - 1) / 2;
    double result =
```

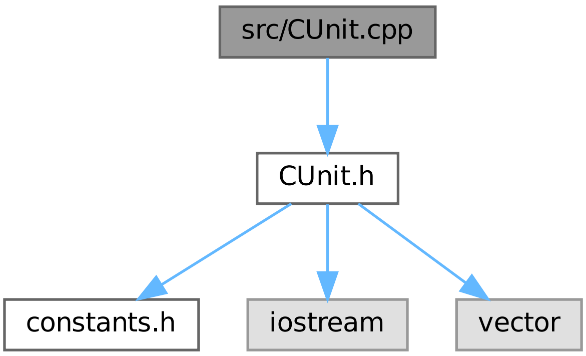
```
        circuit_performance(vector_size, circuit_vector, num_parameters, nullptr,
default_simulator_parameters, false);

    return result;
}

double circuit_performance(int vector_size, int* circuit_vector, int unit_parameters_size, double*
unit_parameters,
                            bool testFlag)
{
    return circuit_performance(vector_size, circuit_vector, unit_parameters_size, unit_parameters,
                                default_simulator_parameters, testFlag);
}

double circuit_performance(int vector_size, int* circuit_vector, bool testFlag)
{
    int num_parameters = (vector_size - 1) / 2;
    double* parameters = new double[num_parameters];
    for (int i = 0; i < num_parameters; i++)
    {
        parameters[i] = 1.0;
    }
    double result = circuit_performance(vector_size, circuit_vector, num_parameters, nullptr,
                                        default_simulator_parameters, testFlag);

    // Clean up
    delete[] parameters;
    return result;
}
```

## 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.

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

```
 * @brief 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.
 *
 */
#include "CUnit.h"

/**
 * @brief Constructors for the CUnit class
 *
 */
CUnit::CUnit()
    : conc_num(0), tails_num(0), mark(false), volume(10.0), feed_palusznium(0.0), feed_gormanium(0.0), feed_waste(0.0),
      k_palusznium(0.008), k_gormanium(0.004), k_waste(0.0005), conc_palusznium(0.0), conc_gormanium(0.0),
      conc_waste(0.0), rho(0.0), phi(0.0), tails_palusznium(0.0), tails_gormanium(0.0), tails_waste(0.0), V_min(2.5),
      V_max(20.0)
{
}

CUnit::CUnit(int conc, int tails)
    : conc_num(conc), tails_num(tails), mark(false), volume(Constants::Circuit::DEFAULT_UNIT_VOLUME),
      feed_palusznium(0.0), feed_gormanium(0.0), feed_waste(0.0), k_palusznium(Constants::Physical::K_PALUSZNIUM),
      k_gormanium(Constants::Physical::K_GORMANIUM), k_waste(Constants::Physical::K_WASTE),
      rho(Constants::Physical::MATERIAL_DENSITY), phi(Constants::Physical::SOLIDS_CONTENT), conc_palusznium(0.0),
      conc_gormanium(0.0), conc_waste(0.0), tails_palusznium(0.0), tails_gormanium(0.0), tails_waste(0.0),
      V_min(Constants::Circuit::MIN_UNIT_VOLUME), V_max(Constants::Circuit::MAX_UNIT_VOLUME)
{
}

CUnit::CUnit(int conc, int tails, bool testFlag)
    : conc_num(conc), tails_num(tails), mark(false), volume(Constants::Circuit::DEFAULT_UNIT_VOLUME),
      feed_palusznium(0.0), feed_gormanium(0.0), feed_waste(0.0), k_palusznium(Constants::Physical::K_PALUSZNIUM),
      k_gormanium(Constants::Physical::K_GORMANIUM), k_waste(Constants::Physical::K_WASTE),
      rho(Constants::Physical::MATERIAL_DENSITY), phi(Constants::Physical::SOLIDS_CONTENT), conc_palusznium(0.0),
      conc_gormanium(0.0), conc_waste(0.0), tails_palusznium(0.0), tails_gormanium(0.0), tails_waste(0.0),
      V_min(Constants::Circuit::MIN_UNIT_VOLUME), V_max(Constants::Circuit::MAX_UNIT_VOLUME)
{
    if (testFlag)
    {

        this->k_palusznium = Constants::Test::K_PALUSZNIUM;
        this->k_gormanium = Constants::Test::K_GORMANIUM;
        this->k_waste = Constants::Test::K_WASTE;
        this->rho = Constants::Test::MATERIAL_DENSITY;
        this->phi = Constants::Test::SOLIDS_CONTENT;
        this->V_min = Constants::Test::MIN_UNIT_VOLUME;
        this->V_max = Constants::Test::MAX_UNIT_VOLUME;
        this->volume = Constants::Test::DEFAULT_UNIT_VOLUME;
    }
}

/**
 * @brief Process the unit
 *
 * This function processes the unit by calculating the residence time,
 * recoveries, and splitting the feed into products.
 *
 */
void CUnit::process()
{
    /* ----------- 1. Residence time  ----------- */

    // total solids feed (kg/s)
    const double Ftot = feed_palusznium + feed_gormanium + feed_waste;
    // guard against division-by-zero / vanishing flow
    const double minFlow = 1e-10;
    const double tau = phi * this->volume / (std::max(Ftot, minFlow) / rho);

    /* ----------- 2. Recoveries R_i^C ----------- */
    Rp = k_palusznium * tau / (1.0 + k_palusznium * tau);
    Rg = k_gormanium * tau / (1.0 + k_gormanium * tau);
    Rw = k_waste * tau / (1.0 + k_waste * tau);

    /* ----------- 3. Split feed into products --- */
    // Palusznium
```

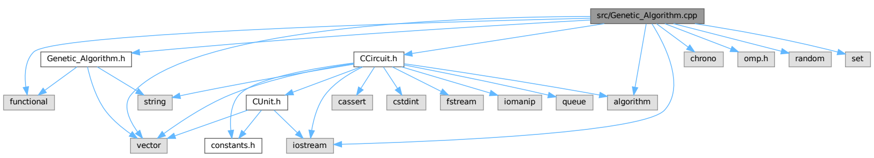
```
    conc_palusznium = feed_palusznium * Rp;
    tails_palusznium = feed_palusznium - conc_palusznium;

    // Gormanium
    conc_gormanium = feed_gormanium * Rg;
    tails_gormanium = feed_gormanium - conc_gormanium;

    // Waste
    conc_waste = feed_waste * Rw;
    tails_waste = feed_waste - conc_waste;
}

/**
 * @brief Update the volume of the unit
 *
 * This function updates the volume of the unit based on the given beta
 * value.
 *
 * @param beta The beta value to update the volume
 *
 */
void CUnit::update_volume(double beta)
{
    this->volume = this->V_min + (this->V_max - this->V_min) * beta;
}
```

## 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**

| | |
|---|---|
| *iv* | Size of the vector |
| *ivs* | Pointer to the vector |
| *rv* | Size of the real vector |
| *rvs* | 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**

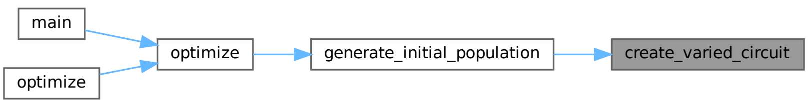
| | |
|---|---|
| *template_vec* | The template vector to modify |
| *num_units* | Number of units in the circuit |
| *validity_check* | Function to check the validity of the circuit |

**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.

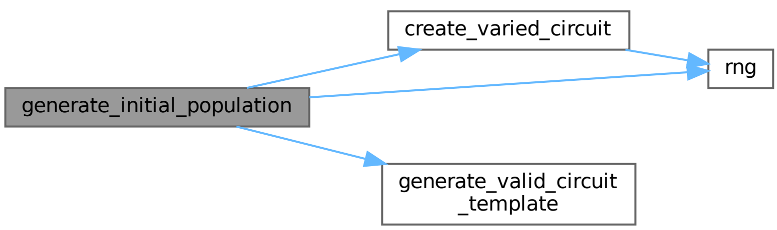
**Parameters**

| | |
|---|---|
| *population_size* | Size of the population to generate |
| *num_units* | Number of units in the circuit |
| *validity_check* | Function to check the validity of the circuit |

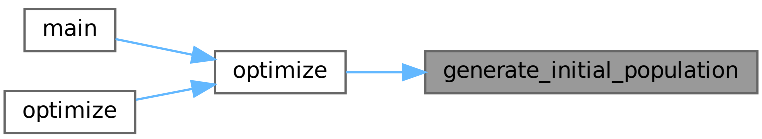
**Returns**

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.

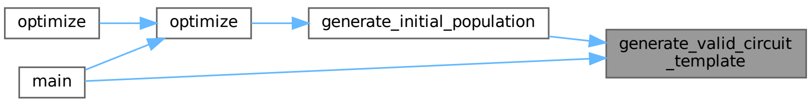
**Parameters**

| | |
|---|---|
| *num_units* | 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.

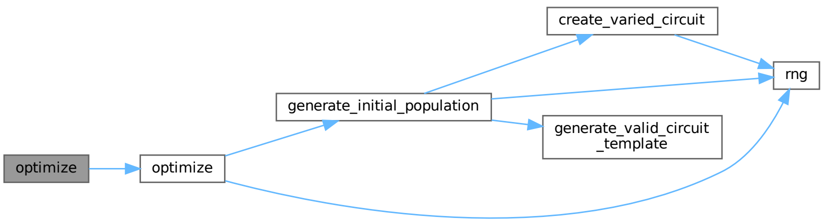
**Parameters**

| | |
|---|---|
| *int_vector_size* | Size of the integer vector |
| *int_vector* | Pointer to the integer vector |
| *real_vector_size* | Size of the real vector |
| *real_vector* | Pointer to the real vector |
| *hybrid_func* | Function to evaluate the fitness of the circuit |
| *hybrid_validity* | Function to check the validity of the circuit |
| *params* | 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.

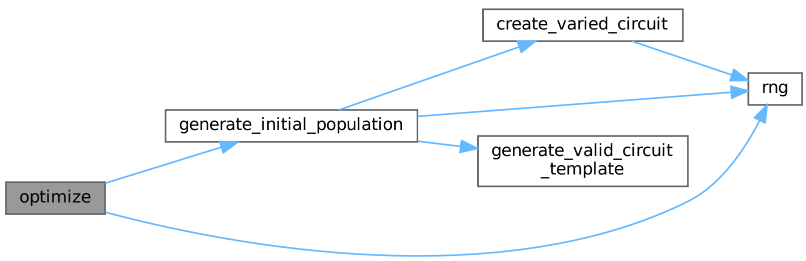
**Parameters**

| | |
|---|---|
| *int_vector_size* | Size of the integer vector |
| *int_vector* | Pointer to the integer vector |
| *func* | Function to evaluate the fitness of the circuit |
| *validity* | Function to check the validity of the circuit |
| *params* | Algorithm parameters for the optimization process |

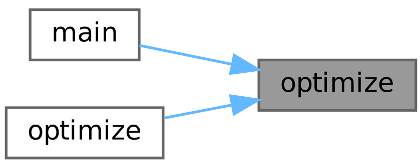
**Returns**

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_ste 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**

| | |
|---|---|
| *real_vector_size* | Size of the real vector |
| *real_vector* | Pointer to the real vector |

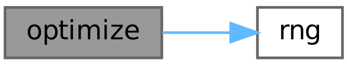
**Parameters**

| | |
|---|---|
| *func* | Function to evaluate the fitness of the circuit |
| *validity* | Function to check the validity of the circuit |
| *params* | Algorithm parameters for the optimization process |

**Returns**

The best fitness value found during optimization

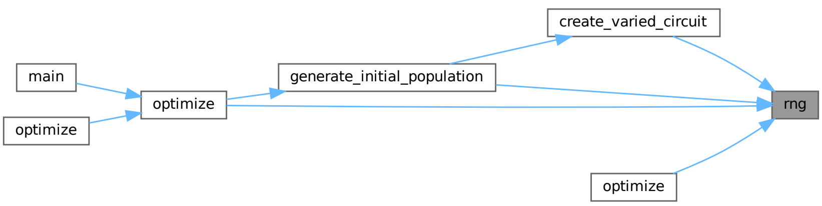
Definition at line 620 of file Genetic_Algorithm.cpp.
References OptimizationResult::best_fitness, Algorithm_Parameters::convergence_threshold, Algorithm_Parameters::crossover_prob
OptimizationResult::generations, last_result, Algorithm_Parameters::max_iterations, Algorithm_Parameters::mutation_probability, 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_generatio
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;
```

```
    }
}

/**
 * @brief 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.
 *
 * @param num_units Number of units in the circuit
 *
 * @return A vector of integers representing the circuit connections
 */
std::vector<int> generate_valid_circuit_template(int num_units)
{
    const int n = num_units;
    const int vec_size = 2 * n + 1;
    std::vector<int> vec(vec_size);

    // Set feed to unit 0 (most common valid configuration)
    vec[0] = 0;

    // Basic linear flow pattern with some recycling
    for (int i = 0; i < n; i++)
    {
        // For the concentrate stream (high-grade)
        if (i < n - 1)
        {
            // Forward flow to next unit for most units
            vec[2 * i + 1] = i + 1;
        }
        else
        {
            // Last unit sends concentrate to Palusznium product (n)
            vec[2 * i + 1] = n;
        }

        // For the tailings stream
        if (i % 3 == 0)
        {
            // Every 3rd unit sends tailings to Gormanium product
            vec[2 * i + 2] = n + 1;
        }
        else if (i % 3 == 1)
        {
            // Every 3rd+1 unit sends tailings to final tailings
            vec[2 * i + 2] = n + 2;
        }
        else
        {
            // Other units recycle tailings back to unit 0
            vec[2 * i + 2] = 0;
        }
    }

    return vec;
}

/**
 *
 * @brief 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.
 *
 * @param template_vec The template vector to modify
 * @param num_units Number of units in the circuit
 * @param validity_check Function to check the validity of the circuit
 *
 * @return A vector representing the varied circuit
 *
 */
std::vector<int> create_varied_circuit(const std::vector<int>& template_vec, int num_units,
                                       std::function<bool(int, int*)> validity_check)
{
    const int n = num_units;
    std::vector<int> result = template_vec;

    // Apply a few random changes
    int num_changes = std::uniform_int_distribution<int>(1, n)(rng());

    // Try multiple times to create a valid variation
    for (int attempt = 0; attempt < 20; attempt++)
    {
        std::vector<int> candidate = template_vec;

```

```
        for (int i = 0; i < num_changes; i++)
        {
            // Pick a random position to modify (excluding feed position)
            int pos = std::uniform_int_distribution<int>(1, 2 * n)(rng());

            // Determine which unit this connection belongs to
            int unit_idx = (pos - 1) / 2;

            // Pick a valid destination (any unit or terminal except self)
            std::vector<int> valid_dests;
            for (int dest = 0; dest < n + 3; dest++)
            {
                if (dest != unit_idx)
                { // Avoid self-loop
                    valid_dests.push_back(dest);
                }
            }

            // Shuffle possible destinations
            std::shuffle(valid_dests.begin(), valid_dests.end(), rng());

            // Try each possible destination until we find one that works
            bool found_valid = false;
            for (int dest : valid_dests)
            {
                int old_val = candidate[pos];
                candidate[pos] = dest;

                // Check if both connections from this unit point to the same place
                int other_conn = (pos % 2 == 1) ? pos + 1 : pos - 1;
                if (other_conn < candidate.size() && candidate[other_conn] == dest)
                {
                    candidate[pos] = old_val; // Restore old value
                    continue;                 // Can’t have both connections to same place
                }

                // Check if the circuit is valid with this change
                if (validity_check(candidate.size(), candidate.data()))
                {
                    found_valid = true;
                    break;
                }
                else
                {
                    candidate[pos] = old_val; // Restore old value
                }
            }

            if (!found_valid)
            {
                // If we couldn’t find a valid change for this position, try another
                continue;
            }
        }

        // Check if the final candidate is valid
        if (validity_check(candidate.size(), candidate.data()))
        {
            return candidate;
        }
    }

    // If all attempts failed, return the original template
    return template_vec;
}

/**
 * @brief Generate an initial population of valid circuits
 *
 * This function generates an initial population of valid circuits
 * based on a set of templates.
 *
 * @param population_size Size of the population to generate
 * @param num_units Number of units in the circuit
 * @param validity_check Function to check the validity of the circuit
 *
 * @return A vector of vectors representing the initial population
 *
 */
std::vector<std::vector<int» generate_initial_population(int population_size, int num_units,
                                                         std::function<bool(int, int*)>
validity_check)
{
    std::vector<std::vector<int» population;
    std::set<std::vector<int» unique_circuits; // To ensure uniqueness

    // Create base templates
```

```
    std::vector<std::vector<int» templates;

    // Template 1: Linear flow with recycling
    templates.push_back(generate_valid_circuit_template(num_units));

    // Template 2: Alternating product outputs
    auto template2 = generate_valid_circuit_template(num_units);
    for (int i = 0; i < num_units; i++)
    {
        if (i % 2 == 0)
        {
            template2[2 * i + 1] = num_units;     // Even units send concentrate to Palusznium
            template2[2 * i + 2] = num_units + 2; // and tailings to final tailings
        }
        else
        {
            template2[2 * i + 1] = num_units + 1; // Odd units send concentrate to Gormanium
            template2[2 * i + 2] = 0;             // and tailings back to first unit
        }
    }
    if (validity_check(template2.size(), template2.data()))
    {
        templates.push_back(template2);
    }

    // Template 3: Butterfly pattern
    auto template3 = generate_valid_circuit_template(num_units);
    for (int i = 0; i < num_units; i++)
    {
        if (i < num_units / 2)
        {
            template3[2 * i + 1] = i + num_units / 2; // First half feed to second half
            template3[2 * i + 2] = num_units + 2;     // Tailings to final tailings
        }
        else
        {
            template3[2 * i + 1] = num_units; // Second half to products
            template3[2 * i + 2] = num_units + 1;
        }
    }
    if (validity_check(template3.size(), template3.data()))
    {
        templates.push_back(template3);
    }

    // Add templates directly to population
    for (const auto& tmpl : templates)
    {
        population.push_back(tmpl);
        unique_circuits.insert(tmpl);
    }

    // Generate variations until we have enough unique circuits
    int max_attempts = population_size * 10;
    int attempts = 0;

    std::cout « "Generating initial population of valid circuits..." « std::endl;

    while (population.size() < population_size && attempts < max_attempts)
    {
        // Pick a random template
        const auto& tmpl = templates[std::uniform_int_distribution<int>(0, templates.size() - 1)(rng())];

        // Create a variation
        auto candidate = create_varied_circuit(tmpl, num_units, validity_check);

        // Check uniqueness
        if (unique_circuits.find(candidate) == unique_circuits.end())
        {
            population.push_back(candidate);
            unique_circuits.insert(candidate);

            if (population.size() % 10 == 0)
            {
                std::cout « "Generated " « population.size() « " valid circuits" « std::endl;
            }
        }

        attempts++;
    }

    std::cout « "Initial population: " « population.size() « " valid circuits" « std::endl;

    return population;
}
```

```
/**
 * @brief Check if all values in the vector are true
 *
 * This function checks if all values in the vector are true.
 *
 * @param iv Size of the vector
 * @param ivs Pointer to the vector
 * @param rv Size of the real vector
 * @param rvs Pointer to the real vector
 */
bool all_true(int iv, int* ivs, int rv, double* rvs)
{
    return true;
}
bool all_true_ints(int iv, int* ivs)
{
    return true;
}
bool all_true_reals(int iv, double* rvs)
{
    return true;
}

static OptimizationResult last_result;

/**
 * @brief 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.
 *
 * @return The last optimization result
 */
OptimizationResult get_last_optimization_result()
{
    return last_result;
}

// **********************************************************************
// 1) Discrete-only optimize with PARALLEL fitness evaluation
// **********************************************************************

/**
 * @brief 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.
 *
 * @param int_vector_size Size of the integer vector
 * @param int_vector Pointer to the integer vector
 * @param func Function to evaluate the fitness of the circuit
 * @param validity Function to check the validity of the circuit
 * @param params Algorithm parameters for the optimization process
 *
 * @return The best fitness value found during optimization
 */
int optimize(int int_vector_size, int* int_vector, std::function<double(int, int*)> func,
             std::function<bool(int, int*)> validity, Algorithm_Parameters params)
{
    using Clock = std::chrono::high_resolution_clock;
    auto t0 = Clock::now();

    // Print OpenMP info
    std::cout « "OpenMP: Using " « omp_get_max_threads() « " threads for parallel fitness evaluation"
« std::endl;

    // --- 1. Improved population initialization
    int n_units = (int_vector_size - 1) / 2;
    std::cout « "Initializing population for " « n_units « " units..." « std::endl;

    // Generate valid initial population
    std::vector<std::vector<int» population = generate_initial_population(params.population_size,
n_units, validity);

    // If we couldn't generate enough valid circuits, adjust population size
    if (population.size() < params.population_size)
    {
        std::cout « "Warning: Could only generate " « population.size()
                  « " valid circuits, adjusting population size" « std::endl;
        params.population_size = population.size();
    }

    double best_overall = -1e300;              // best seen so far
    int stall_count = 0;                       // gens since last improvement
```

```
    double eps = params.convergence_threshold; // "meaningful" fitness delta
    int max_stall = params.stall_generations;  // allowed idle generations

    // --- 2. Main GA loop
    for (int gen = 0; gen < params.max_iterations; ++gen)
    {
        // 2a) PARALLEL fitness evaluation - THIS IS THE KEY OPTIMIZATION!
        std::vector<double> fitnesses(population.size());

// Parallel fitness evaluation using OpenMP
#pragma omp parallel for schedule(dynamic)
        for (size_t i = 0; i < population.size(); ++i)
        {
            int* gdata = population[i].data();
            if (!validity(int_vector_size, gdata))
            {
                fitnesses[i] = -1e9; // heavy penalty
            }
            else
            {
                fitnesses[i] = func(int_vector_size, gdata);
            }
        }

        double gen_best = *std::max_element(fitnesses.begin(), fitnesses.end());
        if (gen_best > best_overall + eps)
        {
            best_overall = gen_best;
            stall_count = 0; // reset when we see new best
        }
        else
        {
            stall_count++;
        }
        if (stall_count >= max_stall)
        {
            if (params.verbose)
            {
                std::cout « "[GA] No improvement for " « stall_count « " generations--stopping
early.\n";
            }
            break; // exit the generation loop
        }

        // 2b) Elitism: copy best genome to next generation
        std::vector<std::vector<int» next_gen;
        {
            auto best_it = std::max_element(fitnesses.begin(), fitnesses.end());
            size_t best_idx = std::distance(fitnesses.begin(), best_it);
            next_gen.push_back(population[best_idx]);
        }

        // ----- TOURNAMENT SETUP -----
        int k = params.tournament_size > 0 ? params.tournament_size : 2;
        std::uniform_int_distribution<size_t> pop_dist(0, population.size() - 1);
        auto pick_parent = [&]()
        {
            size_t best = pop_dist(rng());
            double best_fit = fitnesses[best];
            for (int i = 1; i < k; ++i)
            {
                size_t idx = pop_dist(rng());
                if (fitnesses[idx] > best_fit)
                {
                    best = idx;
                    best_fit = fitnesses[idx];
                }
            }
            return population[best];
        };

        // 2c) Fill rest via selection, crossover, mutation
        std::uniform_real_distribution<double> u01(0.0, 1.0);
        while (next_gen.size() < population.size())
        {
            // - Selection via k-way tournament
            auto p1 = pick_parent();
            auto p2 = pick_parent();

            // - Crossover
            std::vector<int> c1 = p1, c2 = p2;
            if (u01(rng()) < params.crossover_probability)
            {
                // Adaptive crossover points: more early on, fewer later
                double progress = static_cast<double>(gen) / params.max_iterations;
                int max_points = std::min(5, int_vector_size / 2); // limit excessive cuts
                int num_cuts = static_cast<int>((1.0 - progress) * max_points);
```

```
                num_cuts = std::max(1, num_cuts); // always at least 1 point

                std::vector<bool> crossover_mask(int_vector_size, false);
                for (int i = 0; i < num_cuts; ++i)
                {
                    int cut = std::uniform_int_distribution<int>(0, int_vector_size - 1)(rng());
                    crossover_mask[cut] = true;
                }

                bool flip = false;
                for (int j = 0; j < int_vector_size; ++j)
                {
                    if (crossover_mask[j])
                        flip = !flip;
                    if (flip)
                        std::swap(c1[j], c2[j]);
                }
            }

            // - Mutation (creep + optional inversion)
            {
                // 1) Substitution ("creep") mutation on both children
                int min_gene = 0;
                int max_gene = n_units + 2;
                int range = max_gene - min_gene + 1;
                std::uniform_int_distribution<int> step_dist(-params.mutation_step_size, params.mutation_step_size);
                for (auto* child : {&c1, &c2})
                {
                    for (int j = 0; j < int_vector_size; ++j)
                    {
                        if (u01(rng()) < params.mutation_probability)
                        {
                            int step = step_dist(rng());
                            int val = (*child)[j] + step;
                            (*child)[j] = min_gene + ((val - min_gene) % range + range) % range;
                        }
                    }
                }

                // 2) Inversion mutation, if enabled
                if (params.use_inversion)
                {
                    // pick two indices a < b
                    std::uniform_int_distribution<int> a_dist(0, int_vector_size - 2);
                    int a = a_dist(rng());
                    std::uniform_int_distribution<int> b_dist(a + 1, int_vector_size - 1);
                    int b = b_dist(rng());

                    // reverse that slice in each child with its own probability
                    if (u01(rng()) < params.inversion_probability)
                    {
                        std::reverse(c1.begin() + a, c1.begin() + b + 1);
                    }
                    if (u01(rng()) < params.inversion_probability)
                    {
                        std::reverse(c2.begin() + a, c2.begin() + b + 1);
                    }
                }
            }

            // Check validity of children and add valid ones
            if (validity(int_vector_size, c1.data()))
            {
                next_gen.push_back(std::move(c1));
            }

            if (next_gen.size() < population.size() && validity(int_vector_size, c2.data()))
            {
                next_gen.push_back(std::move(c2));
            }
        }

        // 2d) Replace population
        population.swap(next_gen);

        if (params.verbose && gen % 10 == 0)
        {
            std::cout « "[GA] Gen " « gen « " best fitness " « *std::max_element(fitnesses.begin(), fitnesses.end())
                      « " (thread utilization: " « omp_get_max_threads() « " cores)" « "\n";
        }
    }

    // --- 3. Write best genome back into int_vector[]
    // (Re-evaluate final fitness to find the winner) - Also parallel!
    double best_fit = -1e12;
```

```
    size_t best_idx = 0;
    std::vector<double> final_fitnesses(population.size());

#pragma omp parallel for schedule(dynamic)
    for (size_t i = 0; i < population.size(); ++i)
    {
        final_fitnesses[i] = func(int_vector_size, population[i].data());
    }

    // Find best (sequential)
    for (size_t i = 0; i < population.size(); ++i)
    {
        if (final_fitnesses[i] > best_fit)
        {
            best_fit = final_fitnesses[i];
            best_idx = i;
        }
    }

    // Copy best solution
    for (int i = 0; i < int_vector_size; ++i)
    {
        int_vector[i] = population[best_idx][i];
    }

    // Store optimization results
    last_result.best_fitness = best_fit;
    last_result.generations = params.max_iterations;

    auto t1 = Clock::now();
    if (params.verbose)
    {
        double secs = std::chrono::duration<double>(t1 - t0).count();
        std::cout « "[GA] Completed in " « secs « "s, best_fitness=" « best_fit « " (using "
                  « omp_get_max_threads() « " parallel threads)" « "\n";
    }

    return 0;
}

// ***********************************************************************
// 2) Continuous-only optimize with PARALLEL fitness evaluation
// ***********************************************************************

/**
 * @brief 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.
 *
 * @param real_vector_size Size of the real vector
 * @param real_vector Pointer to the real vector
 * @param func Function to evaluate the fitness of the circuit
 * @param validity Function to check the validity of the circuit
 * @param params Algorithm parameters for the optimization process
 *
 * @return The best fitness value found during optimization
 */
int optimize(int real_vector_size, double* real_vector, std::function<double(int, double*)> func,
             std::function<bool(int, double*)> validity, Algorithm_Parameters params)
{
    using Clock = std::chrono::high_resolution_clock;
    auto t0 = Clock::now();

    std::cout « "OpenMP: Using " « omp_get_max_threads() « " threads for continuous optimization" «
std::endl;

    std::uniform_real_distribution<double> dist01(0.0, 1.0);
    std::vector<std::vector<double» population;

    // --- 1. Initialise population
    population.reserve(params.population_size);
    while (population.size() < params.population_size)
    {
        std::vector<double> genome(real_vector_size);
        for (auto& g : genome)
            g = dist01(rng()); // all _i in [0,1]
        if (validity(real_vector_size, genome.data()))
            population.push_back(std::move(genome));
    }

    double best_overall = -1e300;
    int stall_count = 0;
    double eps = params.convergence_threshold;
    int max_stall = params.stall_generations;

```

```
    for (int gen = 0; gen < params.max_iterations; ++gen)
    {
        // PARALLEL fitness evaluation
        std::vector<double> fitnesses(population.size());

#pragma omp parallel for schedule(dynamic)
        for (size_t i = 0; i < population.size(); ++i)
        {
            fitnesses[i] =
                validity(real_vector_size, population[i].data()) ? func(real_vector_size, population[i].data()) : -1e9;
        }

        double gen_best = *std::max_element(fitnesses.begin(), fitnesses.end());
        if (gen_best > best_overall + eps)
        {
            best_overall = gen_best;
            stall_count = 0;
        }
        else
        {
            stall_count++;
        }

        if (stall_count >= max_stall)
        {
            if (params.verbose)
                std::cout « "[GA-Real] No improvement for " « stall_count « " generations -- stopping.\n";
            break;
        }

        // Elitism
        std::vector<std::vector<double» next_gen;
        {
            auto best_it = std::max_element(fitnesses.begin(), fitnesses.end());
            size_t best_idx = std::distance(fitnesses.begin(), best_it);
            next_gen.push_back(population[best_idx]);
        }

        // Tournament selection
        int k = params.tournament_size > 0 ? params.tournament_size : 2;
        std::uniform_int_distribution<size_t> pop_dist(0, population.size() - 1);
        auto pick_parent = [&]()
        {
            size_t best = pop_dist(rng());
            double best_fit = fitnesses[best];
            for (int i = 1; i < k; ++i)
            {
                size_t idx = pop_dist(rng());
                if (fitnesses[idx] > best_fit)
                {
                    best = idx;
                    best_fit = fitnesses[idx];
                }
            }
            return population[best];
        };

        // Crossover + Mutation
        while (next_gen.size() < population.size())
        {
            auto p1 = pick_parent();
            auto p2 = pick_parent();
            std::vector<double> c1 = p1, c2 = p2;

            if (dist01(rng()) < params.crossover_probability)
            {
                for (int j = 0; j < real_vector_size; ++j)
                {
                    if (dist01(rng()) < 0.5)
                        std::swap(c1[j], c2[j]);
                }
            }

            // Mutation
            for (int j = 0; j < real_vector_size; ++j)
            {
                if (dist01(rng()) < params.mutation_probability)
                {
                    double step = dist01(rng()) * params.mutation_step_size;
                    c1[j] = std::clamp(c1[j] + step * (dist01(rng()) < 0.5 ? -1 : 1), 0.0, 1.0);
                }
                if (dist01(rng()) < params.mutation_probability)
                {
                    double step = dist01(rng()) * params.mutation_step_size;
                    c2[j] = std::clamp(c2[j] + step * (dist01(rng()) < 0.5 ? -1 : 1), 0.0, 1.0);
```

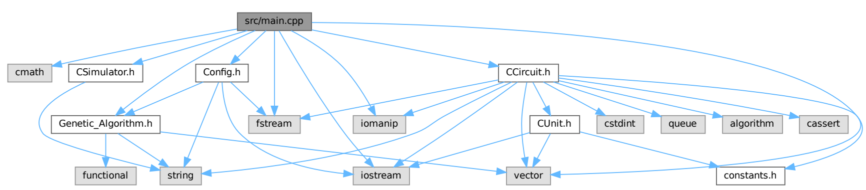
```
                }
            }

            // Optional: scaling mutation
            if (params.use_scaling_mutation)
            {
                std::uniform_int_distribution<int> idx_dist(0, real_vector_size - 1);
                std::uniform_real_distribution<double> scale_dist(params.scaling_mutation_min,
                                                                  params.scaling_mutation_max);

                // Child 1
                if (dist01(rng()) < params.scaling_mutation_prob)
                {
                    int idx = idx_dist(rng());
                    double factor = scale_dist(rng());
                    c1[idx] = std::clamp(c1[idx] * factor, 0.0, 1.0);
                }

                // Child 2
                if (dist01(rng()) < params.scaling_mutation_prob)
                {
                    int idx = idx_dist(rng());
                    double factor = scale_dist(rng());
                    c2[idx] = std::clamp(c2[idx] * factor, 0.0, 1.0);
                }
            }

            next_gen.push_back(std::move(c1));
            if (next_gen.size() < population.size())
                next_gen.push_back(std::move(c2));
        }

        population.swap(next_gen);

        if (params.verbose && gen % (params.max_iterations / 10) == 0)
        {
            std::cout « "[GA-Real] Gen " « gen « " best fitness " « gen_best
                      « " (parallel threads: " « omp_get_max_threads() « ")" « "\n";
        }
    }

    // PARALLEL final evaluation
    double best_fit = -1e12;
    size_t best_idx = 0;
    std::vector<double> final_fitnesses(population.size());

#pragma omp parallel for schedule(dynamic)
    for (size_t i = 0; i < population.size(); ++i)
    {
        final_fitnesses[i] = func(real_vector_size, population[i].data());
    }

    // Find best (sequential)
    for (size_t i = 0; i < population.size(); ++i)
    {
        if (final_fitnesses[i] > best_fit)
        {
            best_fit = final_fitnesses[i];
            best_idx = i;
        }
    }

    // Copy best solution
    for (int i = 0; i < real_vector_size; ++i)
    {
        real_vector[i] = population[best_idx][i];
    }

    // Store optimization results
    last_result.best_fitness = best_fit;
    last_result.generations = params.max_iterations - stall_count;

    auto t1 = Clock::now();
    if (params.verbose)
    {
        double secs = std::chrono::duration<double>(t1 - t0).count();
        std::cout « "[GA-Real] Completed in " « secs « "s, best_fitness=" « best_fit « " (using "
                  « omp_get_max_threads() « " parallel threads)" « "\n";
    }

    return 0;
}

// ********************************************************************
// 3) Hybrid optimize with sequential approach but parallel evaluations
// ********************************************************************

```

```cpp
/**
 * @brief 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.
 *
 * @param int_vector_size Size of the integer vector
 * @param int_vector Pointer to the integer vector
 * @param real_vector_size Size of the real vector
 * @param real_vector Pointer to the real vector
 * @param hybrid_func Function to evaluate the fitness of the circuit
 * @param hybrid_validity Function to check the validity of the circuit
 * @param params Algorithm parameters for the optimization process
 *
 * @return The best fitness value found during optimization
 */
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)
{

    std::cout « "OpenMP: Using " « omp_get_max_threads() « " threads for hybrid optimization" « std::endl;

    // Discrete step: optimize only int vector
    auto wrapped_func_int = [&](int n, int* v)
    {
        return hybrid_func(n, v, real_vector_size,
                           real_vector); // current real_vector
    };
    auto wrapped_valid_int = [&](int n, int* v) { return hybrid_validity(n, v, real_vector_size, real_vector); };

    optimize(int_vector_size, int_vector, wrapped_func_int, wrapped_valid_int, params);

    // Continuous step: optimize only real vector
    auto wrapped_func_real = [&](int n, double* r)
    {
        return hybrid_func(int_vector_size, int_vector, n, r); // fixed int_vector
    };
    auto wrapped_valid_real = [&](int n, double* r) { return hybrid_validity(int_vector_size, int_vector, n, r); };

    optimize(real_vector_size, real_vector, wrapped_func_real, wrapped_valid_real, params);

    return 0;
}
```

## 7.24 src/main.cpp File Reference

```cpp
#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>
#include "CCircuit.h"
#include "CSimulator.h"
#include "Config.h"
#include "Genetic_Algorithm.h"
```

Include dependency graph for main.cpp:



### Functions

- int main ()

## 7.24.1 Function Documentation

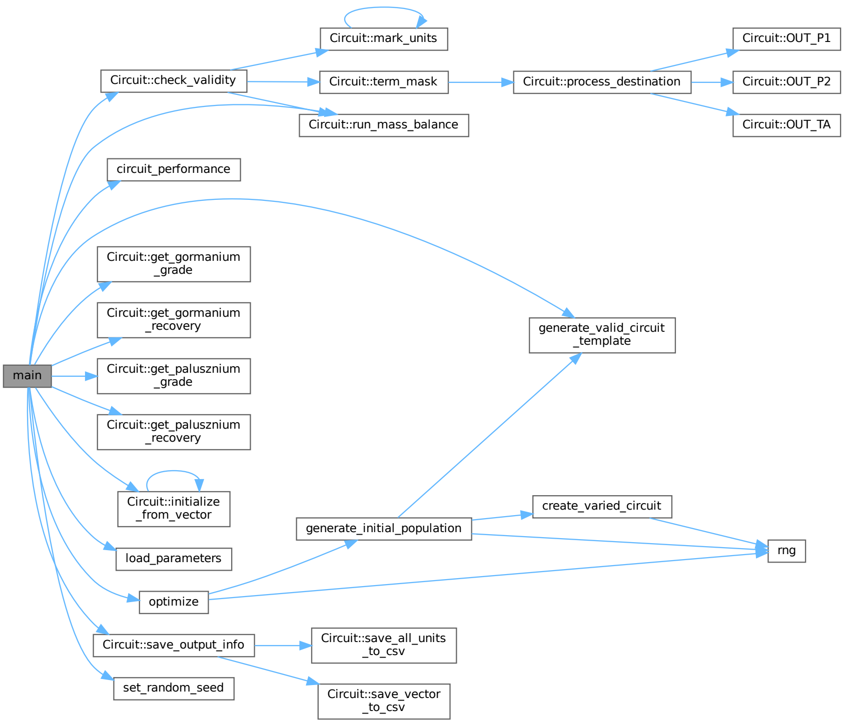
### main()

`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_probabi Algorithm_Parameters::elite_count, generate_valid_circuit_template(), Circuit::get_gormanium_grade(), Circuit::get_gormanium_reco Circuit::get_palusznium_grade(), Circuit::get_palusznium_recovery(), Circuit::initialize_from_vector(), Algorithm_Parameters::inversio 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::scali 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
```

```
#include "Genetic_Algorithm.h"

// static constexpr int DEFAULT_UNITS = 10;
// static const int hard_circuit_10[2 * DEFAULT_UNITS + 1] = {1, 2, 4, 3,  5, 3, 0, 8, 11, 7, 12,
//                                                             7, 0, 7, 11, 8, 6, 9, 7, 10, 3};

int main()
{
    // Save original cout buffer before we start
    std::streambuf* original_cout_buffer = std::cout.rdbuf();

    // Create null stream to discard output
    std::ofstream null_stream("/dev/null");

    std::cout « "=== Palusznium Rush Circuit Optimizer ===\n\n";

    // load GA & random-seed settings from parameters.txt
    Algorithm_Parameters params;
    load_parameters("parameters.txt", params);

    // Optionally fix the RNG for reproducibility
    if (params.random_seed >= 0)
    {
        set_random_seed(params.random_seed);
        std::cout « "* Using fixed seed: " « params.random_seed « "\n";
    }

    // Print to see
    std::cout « "GA parameters:\n"
              « "  mode                   = " « params.mode « "\n"
              « "  random_seed            = " « params.random_seed « "\n\n"
              « "  num_units              = " « params.num_units « "\n\n"

              « "  population_size        = " « params.population_size « "\n"
              « "  elite_count            = " « params.elite_count « "\n"
              « "  max_iterations         = " « params.max_iterations « "\n\n"

              « "  tournament_size        = " « params.tournament_size « "\n"
              « "  selection_pressure     = " « params.selection_pressure « "\n\n"

              « "  crossover_probability  = " « params.crossover_probability « "\n"
              « "  crossover_points       = " « params.crossover_points « "\n\n"

              « "  mutation_probability   = " « params.mutation_probability « "\n"
              « "  mutation_step_size     = " « params.mutation_step_size « "\n"
              « "  allow_mutation_wrapping = " « std::boolalpha « params.allow_mutation_wrapping « "\n\n"

              « "  use_inversion          = " « std::boolalpha « params.use_inversion « "\n"
              « "  inversion_probability  = " « params.inversion_probability « "\n\n"

              « "  use_scaling_mutation   = " « std::boolalpha « params.use_scaling_mutation « "\n"
              « "  scaling_mutation_prob  = " « params.scaling_mutation_prob « "\n"
              « "  scaling_mutation_min   = " « params.scaling_mutation_min « "\n"
              « "  scaling_mutation_max   = " « params.scaling_mutation_max « "\n\n"

              « "  convergence_threshold  = " « params.convergence_threshold « "\n"
              « "  stall_generations      = " « params.stall_generations « "\n\n"

              « "  verbose                = " « std::boolalpha « params.verbose « "\n"
              « "  log_results            = " « std::boolalpha « params.log_results « "\n"
              « "  log_file               = " « params.log_file « "\n\n";

    // Optimisation mode
    auto mode = params.mode; // "d", "c" or "h" from parameters.txt
    std::cout « "Mode: " « mode « "\n";

    // Set number of units
    int num_units = params.num_units;
    int vector_size = 2 * num_units + 1;

    // Create vectors to hold the optimization results
    std::vector<int> circuit_vector(vector_size, 0);
    std::vector<double> volume_params(num_units, 0.5);

    if (mode == "d")
    {
        std::cout « "Running DISCRETE optimization...\n";

        // std::cout.rdbuf(null_stream.rdbuf());

        auto discrete_fitness = [](int size, int* vec) -> double
        {
            // Discrete-only overload
            return circuit_performance(size, vec);
        };
```

```

        auto discrete_validity = [](int size, int* vec) -> bool
        {
            // Discrete-only constructor and validity
            Circuit c(size / 2); // (2n + 1 → n units)
            c.initialize_from_vector(size, vec);
            return c.check_validity(size, vec);
        };

        optimize(vector_size, circuit_vector.data(), discrete_fitness, discrete_validity, params);
    }

    else if (mode == "c")
    {
        std::cout « "Running CONTINUOUS optimization...\n";

        // std::cout.rdbuf(null_stream.rdbuf());

        // Build a simple *valid* circuit of the requested size
        auto base = generate_valid_circuit_template(num_units); // function from Genetic_Algorithm.cpp
        std::copy(base.begin(), base.end(), circuit_vector.begin());
        auto cont_fitness = [&](int r_size, double* rvec) -> double
        { return circuit_performance(vector_size, circuit_vector.data(), r_size, rvec); };

        auto cont_validity = [&](int r_size, double* rvec) -> bool
        {
            Circuit c(num_units, rvec); // use volume constructor!
            c.initialize_from_vector(vector_size, circuit_vector.data(), rvec);
            return c.check_validity(vector_size, circuit_vector.data(), r_size, rvec);
        };

        optimize(num_units, volume_params.data(), cont_fitness, cont_validity, params);
    }

    else
    {
        std::cout « "Running hybrid optimization (connections + volumes)...\n";

        // Redirect cout to null stream to silence debug output
        // std::cout.rdbuf(null_stream.rdbuf());

        // Define hybrid fitness and validity functions
        auto hybrid_fitness = [](int i_size, int* i_vec, int r_size, double* r_vec) -> double
        { return circuit_performance(i_size, i_vec, r_size, r_vec); };

        auto hybrid_validity = [num_units](int i_size, int* i_vec, int r_size, double* r_vec) -> bool
        {
            Circuit c(num_units);
            c.initialize_from_vector(i_size, i_vec);
            return c.check_validity(i_size, i_vec, r_size, r_vec);
        };

        // Run hybrid optimization (cout is redirected, so no debug output)
        optimize(vector_size, circuit_vector.data(), num_units, volume_params.data(), hybrid_fitness, hybrid_validity,
                 params);
    }

    // Calculate performance with optimized values (still silent)
    double performance = circuit_performance(vector_size, circuit_vector.data(), num_units, volume_params.data());

    // Create a circuit object for detailed analysis, still silent
    Circuit circuit(num_units, volume_params.data());
    circuit.initialize_from_vector(vector_size, circuit_vector.data(), volume_params.data());
    circuit.run_mass_balance();

    // Extract important metrics before restoring cout
    double palusznium_recovery = circuit.get_palusznium_recovery() * 100;
    double palusznium_grade = circuit.get_palusznium_grade() * 100;
    double gormanium_recovery = circuit.get_gormanium_recovery() * 100;
    double gormanium_grade = circuit.get_gormanium_grade() * 100;

    // Calculate volumes and costs while still silent
    double total_volume = 0.0;
    double unit_volumes[num_units];
    for (int i = 0; i < num_units; i++)
    {
        if (mode == "h" || mode == "c")
        {
            // Use scaled volumes
            double min_volume = 2.5;
            double max_volume = 20.0;
            unit_volumes[i] = min_volume + (max_volume - min_volume) * volume_params[i];
        }
        else
        {
```

```
            // Discrete mode: use fixed volume
            unit_volumes[i] = 10.0;
        }
        total_volume += unit_volumes[i];
    }

    double operating_cost = 5.0 * std::pow(total_volume, 2.0 / 3.0);
    if (total_volume >= 150.0)
    {
        operating_cost += 1000.0 * std::pow(total_volume - 150.0, 2.0);
    }

    // Now RESTORE cout to print results
    std::cout.rdbuf(original_cout_buffer);

    // Print final results after optimization
    std::cout « "\nOptimization complete!\n";
    std::cout « "Final circuit economic value: £" « std::fixed « std::setprecision(2) « performance
              « " per second\n\n";

    // Display the optimized circuit vector
    std::cout « "Optimized circuit vector: ";
    for (int i = 0; i < vector_size; ++i)
        std::cout « circuit_vector[i] « " ";
    std::cout « std::endl;

    // Display the optimized volumes
    std::cout « "Optimized volume parameters: ";
    for (int i = 0; i < num_units; ++i)
        std::cout « std::fixed « std::setprecision(5) « volume_params[i] « " ";
    std::cout « std::endl;

    // Display circuit performance metrics
    std::cout « "\nCircuit Performance:\n";
    std::cout « "- Palusznium recovery: " « std::fixed « std::setprecision(2) « palusznium_recovery « "%\n";
    std::cout « "- Palusznium grade: " « std::fixed « std::setprecision(2) « palusznium_grade « "%\n";
    std::cout « "- Gormanium recovery: " « std::fixed « std::setprecision(2) « gormanium_recovery « "%\n";
    std::cout « "- Gormanium grade: " « std::fixed « std::setprecision(2) « gormanium_grade « "%\n";

    // Circuit configuration analysis
    std::cout « "\nCircuit Configuration Analysis:\n";
    int direct_to_p = 0, direct_to_g = 0, direct_to_t = 0, recycles = 0;
    for (int i = 0; i < num_units; i++)
    {
        // Check concentrate connections
        int conc_dest = circuit_vector[1 + 2 * i];
        if (conc_dest == num_units)
            direct_to_p++;
        else if (conc_dest == num_units + 1)
            direct_to_g++;
        else if (conc_dest == num_units + 2)
            direct_to_t++;
        else if (conc_dest < i)
            recycles++;

        // Check tailing connections
        int tail_dest = circuit_vector[2 + 2 * i];
        if (tail_dest == num_units)
            direct_to_p++;
        else if (tail_dest == num_units + 1)
            direct_to_g++;
        else if (tail_dest == num_units + 2)
            direct_to_t++;
        else if (tail_dest < i)
            recycles++;
    }

    std::cout « "- Units sending to Palusznium product: " « direct_to_p « "\n";
    std::cout « "- Units sending to Gormanium product: " « direct_to_g « "\n";
    std::cout « "- Units sending to Tailings: " « direct_to_t « "\n";
    std::cout « "- Recycle connections: " « recycles « "\n";

    // Unit volume analysis
    std::cout « "\nUnit Volumes (m³):\n";
    for (int i = 0; i < num_units; i++)
    {
        std::cout « "Unit " « i « ": " « std::fixed « std::setprecision(2) « unit_volumes[i] « " m³\n";
    }
    std::cout « "Total volume: " « std::fixed « std::setprecision(2) « total_volume « " m³\n";

    // Economic analysis
    std::cout « "\nEconomic Analysis:\n";
    double palusznium_value = circuit.get_palusznium_recovery() * 8 * 120;
    double gormanium_value = circuit.get_gormanium_recovery() * 12 * 80;
```

```
    std::cout « "- Palusznium revenue: £" « std::fixed « std::setprecision(2) « palusznium_value « "/s\n";
    std::cout « "- Gormanium revenue: £" « std::fixed « std::setprecision(2) « gormanium_value « "/s\n";
    std::cout « "- Total revenue: £" « std::fixed « std::setprecision(2) « (palusznium_value + gormanium_value)
              « "/s\n";
    std::cout « "- Operating cost: £" « std::fixed « std::setprecision(2) « operating_cost « "/s\n";
    std::cout « "- Net profit: £" « std::fixed « std::setprecision(2) « performance « "/s\n";

    // Save raw circuit data into a CSV:
    const std::string out_csv = "plotting/circuit_results.csv";
    if (circuit.save_output_info(out_csv))
    {
        std::cout « "\n Saved detailed circuit info to " « out_csv « "\n";
    }
    else
    {
        std::cerr « "\n  Failed to write circuit info to " « out_csv « "\n";
    }

    return 0;
}
```