Quilë

v1.0.0

Generated by Doxygen 1.9.1

Chapter 1

Introduction

Quilë is a C++20 header-only general purpose genetic algorithms library with no external dependencies supporting floating-point, integer, binary and permutation representations. It is released under the terms of the MIT License. Source code is available at `https://github.com/ttarkowski/quile`.

The name of this library origins from fictional language Neo-Quenya and means *color*.

This work is a result of the project funded by National Science Centre Poland (Twardowskiego 16, PL-30312 Kraków, Poland, `http://www.ncn.gov.pl/`) under the grant number UMO-2016/23/B/ST3/03575.

Example compilation command: `g++ -std=c++20 -DNDEBUG -O3 -Wall -Wextra -pedantic -I/home/user/repos/quile -pthread program.cc`. Clang compilation flags are identical. Please remove `-DNDEBUG` to enable assertions. Please add `-DQUILE_ENABLE_LOGGING` to enable logging.

Please note that examples from `doc/examples` directory were compiled with following command `g++ -std=c++20 -DQUILE_ENABLE_LOGGING -O3 -Wall -Wextra -pedantic -I../../ -pthread` and their output consisting of `std::cout` and `std::cerr` streams were wrapped to fit 80 colums lines.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 quile::detail Namespace Reference

Functions

- template<std::floating_point T>
  T angle (T x, T y)
- template<typename It >
  It advance_cpy (It it, std::size_t n)
- template<typename It , typename Compare = std::less<>>
  std::vector< std::size_t > rank (It first, It last, Compare comp={})
- template<typename T , typename U >
  T id (U u)
- template<typename G >
  requires chromosome< G > population< G > generate (std::size_t lambda, const std::function< G()> &f)

6.1.1 Detailed Description

`detail` contains library implementation details and is not inteded for use by library end-user.

6.1.2 Function Documentation

6.1.2.1 advance_cpy()

```
template<typename It >
It quile::detail::advance_cpy (
            It it,
            std::size_t n )
```

`detail::advance_cpy` wraps `std::advance` and returns iterator.

Template Parameters

It	Iterator type.

Parameters

it	Iterator.
n	Distance to advance.

Returns

Advanced iterator.

6.1.2.2 angle()

```
template<std::floating_point T>
T quile::detail::angle (
            T x,
            T y )
```

`detail::angle` returns angle ϕ in polar coordinate system.

Template Parameters

T	Argument and return type (floating-point).

Parameters

x	x coordinate in Cartesian coordinate system.
y	y coordinate in Cartesian coordinate system.

Returns

ϕ coordinate in polar coordinate system corresponding to (x, y) point.

6.1.2.3 generate()

```
template<typename G >
requires chromosome<G> population<G> quile::detail::generate (
            std::size_t lambda,
            const std::function< G()> & f )
```

`detail::generate` creates population of size `lambda` filled with result of function `f`.

Template Parameters

G	Some `genotype` specialization.

Parameters

lambda	Result population size.
f	Function returning genotype.

Returns

Population of size `lambda` where each member is result of `f`.

6.1.2.4 id()

```
template<typename T , typename U >
T quile::detail::id (
             U u )
```

`detail::id` performs identity operation between types.

Template Parameters

T	Destination type.
U	Source type.

Parameters

u	Argument.

Returns

Argument after transformation to destination type.

6.1.2.5 rank()

```
template<typename It , typename Compare = std::less<>>
std::vector<std::size_t> quile::detail::rank (
             It first,
             It last,
             Compare comp = {} )
```

`detail::rank` returns ranking position of element in range after stable sort.

Template Parameters

It	Iterator type.
Compare	Type of comparison mechanism.

Parameters

first	Range begin.
last	Range end.
comp	Comparison function object.

Returns

Sequence container with ranking positions.

6.2 quile::test_functions Namespace Reference

Classes

- class test_function

Typedefs

- template<std::floating_point T, std::size_t N>
  using point = std::array< T, N >

Functions

- template<std::floating_point T, std::size_t N>
  T distance (const point< T, N > &p0, const point< T, N > &p1)
- template<std::floating_point T>
  std::tuple< T, T > coordinates (const point< T, 2 > &p)
- template<std::floating_point T>
  std::tuple< T, T, T > coordinates (const point< T, 3 > &p)
- template<std::floating_point T, std::size_t N>
  point< T, N > uniform_point (T v)

Variables

- template<std::floating_point T, std::size_t N>
  const test_function< T, N > Ackley
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > Alpine
- template<std::floating_point T>
  const test_function< T, 2 > Aluffi_Pentini
- template<std::floating_point T>
  const test_function< T, 2 > Booth

- template<std::floating_point T>
  const test_function< T, 4 > Colville
- template<std::floating_point T>
  const test_function< T, 2 > Easom
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > exponential
- template<std::floating_point T>
  const test_function< T, 2 > Goldstein_Price
- template<std::floating_point T>
  const test_function< T, 2 > Hosaki
- template<std::floating_point T>
  const test_function< T, 2 > Leon
- template<std::floating_point T>
  const test_function< T, 2 > Matyas
- template<std::floating_point T>
  const test_function< T, 2 > Mexican_hat
- template<std::floating_point T>
  const test_function< T, 4 > Miele_Cantrell
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > Rosenbrock
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > Schwefel
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > sphere

6.2.1 Detailed Description

`test_functions` contains mechanisms for floating-point test functions.

6.2.2 Typedef Documentation

6.2.2.1 point

```
template<std::floating_point T, std::size_t N>
using quile::test_functions::point = typedef std::array<T, N>
```

`test_functions::point` is point in N-dimensional space.

Template Parameters

T	Floating-point type.
N	Space dimension.

6.2.3 Function Documentation

6.2.3.1 coordinates() [1/2]

```
template<std::floating_point T>
std::tuple<T, T> quile::test_functions::coordinates (
            const point< T, 2 > & p )
```

test_functions::coordinates converts 2D-point to std::tuple containing point coordinates.

Template Parameters

T	Floating-point type.

Parameters

p	2D-point.

Returns

Corresponding tuple.

6.2.3.2 coordinates() [2/2]

```
template<std::floating_point T>
std::tuple<T, T, T> quile::test_functions::coordinates (
            const point< T, 3 > & p )
```

test_functions::coordinates converts 3D-point to std::tuple containing point coordinates.

Template Parameters

T	Floating-point type.

Parameters

p	3D-point.

Returns

Corresponding tuple.

6.2.3.3 distance()

```
template<std::floating_point T, std::size_t N>
T quile::test_functions::distance (
```

```
            const point< T, N > & p0,
            const point< T, N > & p1 )
```

test_functions::distance returns distance between two points.

Template Parameters

T	Floating-point type.
N	Space dimension.

Parameters

p0	First point.
p1	Second point.

Returns

Distance between p0 and p1.

6.2.3.4 uniform_point()

```
template<std::floating_point T, std::size_t N>
point<T, N> quile::test_functions::uniform_point (
            T v )
```

test_functions::uniform_point returns point in N-dimensional space, where each coordinate has the same value v.

Template Parameters

T	Floating-point type.
Space	dimension.

Parameters

v	Coordinate value.

Returns

Point in N-dimensional space.

6.2.4 Variable Documentation

6.2.4.1 Ackley

```
template<std::floating_point T, std::size_t N>
const test_function<T, N> quile::test_functions::Ackley
```

Initial value:

```
{
  "Ackley",
  [](const point<T, N>& p) {
    T s0 = 0.;
    T s1 = 0.;
    for (auto x :  p) {
      s0 += square(x);
      s1 += std::cos(2 * pi<T> * x);
    }
    return -20.  * std::exp(-.02 * std::sqrt(s0) / std::sqrt(N)) -
           std::exp(s1 / N) + 20.  + e<T>;
  },
  [] () { return uniform_domain<T, N>(-35., 35.); },
  [] () { return uniform_point<T, N>(0.); }
}
```

test_functions::Ackley is Ackley test function.

$$f^*(\vec{x}) = -20\exp\left(\frac{-0.02}{\sqrt{n}}\sqrt{\sum_{i=0}^{n-1} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=0}^{n-1}\cos(2\pi x_i)\right) + 20 + e$$

6.2.4.2 Alpine

```
template<std::floating_point T, std::size_t N>
const test_function<T, N> quile::test_functions::Alpine
```

Initial value:

```
{
  "Alpine",
  [](const point<T, N>& p) {
    return std::transform_reduce(
      std::begin(p), std::end(p), T{ 0.  }, std::plus<T>{}, [](auto x) {
        return std::fabs(x * std::sin(x) + .1 * x);
      });
  },
  [] () { return uniform_domain<T, N>(-10., 10.); },
  [] () { return uniform_point<T, N>(0.); }
}
```

test_functions::Alpine is Alpine test function.

$$f^*(\vec{x}) = \sum_{i=0}^{n-1} |x_i \sin x_i + 0.1 x_i|$$

6.2.4.3 Aluffi_Pentini

```
template<std::floating_point T>
const test_function<T, 2> quile::test_functions::Aluffi_Pentini
```

Initial value:

```
{
  "Aluffi-Pentini",
  [](const point<T, 2>& p) {
    const auto [x, y] = coordinates(p);
    return ((.25 * x * x - .5) * x + .1) * x + 0.5 * y * y;
  },
  [] () { return uniform_domain<T, 2>(-10., 10.); },
  [] () {
    return point<T, 2>{ [q = 0.1](int k) -> T {
                         return 2.  * std::sqrt(3.)  *
                                std::cos(
                                  std::acos(-3.  * std::sqrt(3.)  * q / 2.)  / 3.  -
                                  2.  * std::numbers::pi_v<T> * k / 3.)  /
                                3.;
                       }(2),
                        0.  };
  }
}
```

`test_functions::Aluffi_Pentini` is Aluffi-Pentini test function.

$$f^*(\vec{x}) = \frac{1}{4}x_0^4 - \frac{1}{2}x_0^2 + \frac{1}{10}x_0 + \frac{1}{2}x_1^2$$

6.2.4.4 Booth

```
template<std::floating_point T>
const test_function<T, 2> quile::test_functions::Booth
```

Initial value:

```
{
  "Booth",
  [](const point<T, 2>& p) {
    const auto [x, y] = coordinates(p);
    return square(x + 2.  * y - 7.)  + square(2.  * x + y - 5.);
  },
  [] () { return uniform_domain<T, 2>(-10., 10.); },
  [] () {
    return point<T, 2>{ 1., 3.  };
  }
}
```

`test_functions::Booth` is Booth test function.

$$f^*(\vec{x}) = (x_0 + 2x_1 - 7)^2 + (2x_0 + x_1 - 5)^2$$

6.2.4.5 Colville

```
template<std::floating_point T>
const test_function<T, 4> quile::test_functions::Colville
```

Initial value:

```
{
  "Colville",
  [](const point<T, 4>& p) {
    return 100.  * square(p[0] - square(p[1])) + square(1.  - p[0]) +
           90.  * square(p[3] - p[2] * p[2]) + square(1.  - p[2]) +
           10.1 * square(p[1] - 1.)  + square(p[3] - 1.)  +
           19.8 * (p[1] - 1.)  * (p[3] - 1.);
  },
  []() { return uniform_domain<T, 4>(-10., 10.); },
  []() { return uniform_point<T, 4>(1.); }
}
```

`test_functions::Colville` is Colville test function.

$$\begin{aligned} f^*(\vec{x}) &= 100\left(x_0 - x_1^2\right)^2 + (1 - x_0)^2 + 90\left(x_3 - x_2^2\right)^2 + (1 - x_2)^2 \\ &\quad + 10.1\left(x_1 - 1\right)^2 + \left(x_3 - 1\right)^2 + 19.8\left(x_1 - 1\right)\left(x_3 - 1\right) \end{aligned}$$

6.2.4.6 Easom

```
template<std::floating_point T>
const test_function<T, 2> quile::test_functions::Easom
```

Initial value:

```
{
  "Easom",
  [](const point<T, 2>& p) {
    const auto [x, y] = coordinates(p);
    return -std::cos(x) * std::cos(y) *
           std::exp(-square(x - pi<T>) - square(y - pi<T>));
  },
  []() { return uniform_domain<T, 2>(-100., 100); },
  []() { return uniform_point<T, 2>(pi<T>); }
}
```

`test_functions::Easom` is Easom test function.

$$f^*(\vec{x}) = -\cos x_0 \cdot \cos x_1 \cdot \exp\left(-(x_0 - \pi)^2 - (x_1 - \pi)^2\right)$$

6.2.4.7 exponential

```
template<std::floating_point T, std::size_t N>
const test_function<T, N> quile::test_functions::exponential
```

Initial value:

```
{
  "exponential",
  [](const point<T, N>& p) {
    return -std::exp(
      -.5 * std::transform_reduce(
              std::begin(p), std::end(p), T{ 0.  }, std::plus<T>{}, square<T>));
  },
  []() { return uniform_domain<T, N>(-1., 1.); },
  []() { return uniform_point<T, N>(0.); }
}
```

`test_functions::exponential` is exponential test function.

$$f^*(\vec{x}) = -\exp\left(-\frac{1}{2}\sum_{i=0}^{n-1} x_i^2\right)$$

6.2.4.8 Goldstein_Price

```
template<std::floating_point T>
const test_function<T, 2> quile::test_functions::Goldstein_Price
```

Initial value:

```
{
  "Goldstein-Price",
  [](const point<T, 2>& p) {
    const auto [x, y] = coordinates(p);
    const auto [x2, y2] = std::tuple<T, T>{ x * x, y * y };
    const auto xy = x * y;
    return (1.  + square(x + y + 1.)  *
                  (19.  - 14.  * x + 3.  * x2 - 14.  * y + 6.  * xy + 3.  * y2)) *
           (30.  + square(2.  * x - 3.  * y) *
                   (18.  - 32.  * x + 12.  * x2 + 48.  * y - 36.  * xy + 27.  * y2));
  },
  [] () { return uniform_domain<T, 2>(-2., 2.); },
  [] () {
    return point<T, 2>{ 0., -1.  };
  }
}
```

`test_functions::Goldstein_Price` is Goldstein-Price test function.

$$f^*(\vec{x}) = \left(1 + (x_0 + x_1 + 1)^2 \left(19 - 14x_0 + 3x_0^2 - 14x_1 + 6x_0x_1 + 3x_1^2\right)\right) \cdot \left(30 + (2x_0 - 3x_1)^2 \left(18 - 32x_0 + 12x_0^2 + 48x_1 - 36x_0x_1 + 27x_1^2\right)\right)$$

6.2.4.9 Hosaki

```
template<std::floating_point T>
const test_function<T, 2> quile::test_functions::Hosaki
```

Initial value:

```
{
  "Hosaki",
  [](const point<T, 2>& p) {
    const auto [x, y] = coordinates(p);
    return (1.  + x * (-8.  + x * (7.  + x * (-7.  / 3.  + x / 4.))))  * y * y *
           std::exp(-y);
  },
  [] () { return uniform_domain<T, 2>(-10., 10.); },
  [] () {
    return point<T, 2>{ 4., 2.  };
  }
}
```

`test_functions::Hosaki` is Hosaki test function.

$$f^*(\vec{x}) = \left(1 - 8x_0 + 7x_0^2 - \frac{7}{3}x_0^3 + \frac{1}{4}x_0^4\right) x_1^2 \exp(-x_1)$$

6.2.4.10 Leon

```
template<std::floating_point T>
const test_function<T, 2> quile::test_functions::Leon
```

Initial value:

```
{
  "Leon",
  [](const point<T, 2>& p) {
    const auto [x, y] = coordinates(p);
    return 100.  * square(y - x * x) + square(1.  - x);
  },
  [] () { return uniform_domain<T, 2>(-1.2, 1.2); },
  [] () {
    return point<T, 2>{ 1., 1.  };
  }
}
```

`test_functions::Leon` is Leon test function.

$$f^*(\vec{x}) = 100\left(x_1 - x_0^2\right)^2 + (1 - x_0)^2$$

6.2.4.11 Matyas

```
template<std::floating_point T>
const test_function<T, 2> quile::test_functions::Matyas
```

Initial value:

```
{ "Matyas",
                                  [](const point<T, 2>& p) {
                                    const auto [x, y] = coordinates(p);
                                    return .26 * (x * x + y * y) - .48 * x * y;
                                  },
                                  [] () {
                                    return uniform_domain<T, 2>(-10., 10.);
                                  },
                                  [] () { return uniform_point<T, 2>(0.); } }
```

`test_functions::Matyas` is Matyas test function.

$$f^*(\vec{x}) = 0.26\left(x_0^2 + x_1^2\right) - 0.48 x_0 x_1$$

6.2.4.12 Mexican_hat

```
template<std::floating_point T>
const test_function<T, 2> quile::test_functions::Mexican_hat
```

Initial value:

```
{
  "Mexican hat",
  [](const point<T, 2>& p) {
    const auto [x, y] = coordinates(p);
    const auto f = [&, x = x, y = y]() {
      return .1 + std::sqrt(square(x - 4.)  + square(y - 4.));
    };
    return -20.  * std::sin(f()) / f();
  },
  [] () { return uniform_domain<T, 2>(-10., 10.); },
  [] () { return uniform_point<T, 2>(4.); }
}
```

`test_functions::Mexican_hat` is Mexican hat test function.

$$f^*(\vec{x}) = -20\frac{\sin g(x_0, x_1)}{g(x_0, x_1)},\ g(x_0, x_1) = 0.1 + \sqrt{(x_0 - 4)^2 + (x_1 - 4)^2}$$

6.2.4.13 Miele_Cantrell

```
template<std::floating_point T>
const test_function<T, 4> quile::test_functions::Miele_Cantrell
```

Initial value:

```
{
  "Miele-Cantrell",
  [](const point<T, 4>& p) {
    return std::pow(std::exp(-p[0]) - p[1], 4.)  +
           100.  * std::pow(p[1] - p[2], 6.)  +
           std::pow(std::tan(p[2] - p[3]), 4.)  + std::pow(p[0], 8.);
  },
  []() { return uniform_domain<T, 4>(-1., 1); },
  []() {
    return point<T, 4>{ 0., 1., 1., 1.  };
  }
}
```

`test_functions::Miele_Cantrell` is Miele-Cantrell test function.

$$f^*(\vec{x}) = (\exp(-x_0) - x_1)^4 + 100\,(x_1 - x_2)^6 + \tan^4(x_2 - x_3) + x_0^8$$

6.2.4.14 Rosenbrock

```
template<std::floating_point T, std::size_t N>
const test_function<T, N> quile::test_functions::Rosenbrock
```

Initial value:

```
{
  "Rosenbrock",
  [](const point<T, N>& p) {
    T res = 0.;
    for (std::size_t i = 0; i < N - 1; ++i) {
      res += 100.  * square(p[i + 1] - square(p[i])) + square(p[i] - 1.);
    }
    return res;
  },
  []() { return uniform_domain<T, N>(-30., 30.); },
  []() { return uniform_point<T, N>(1.); }
}
```

`test_functions::Rosenbrock` is Rosenbrock test function.

$$f^*(\vec{x}) = \sum_{i=0}^{n-2} \left(100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2\right)$$

6.2.4.15 Schwefel

```
template<std::floating_point T, std::size_t N>
const test_function<T, N> quile::test_functions::Schwefel
```

Initial value:

```
{ "Schwefel",
                                    [](const point<T, N>& p) {
                                      T res = 0.;
                                      for (T sum = 0.; auto x :  p) {
                                        res += square(sum += x);
                                      }
                                      return res;
```

```
},
[]() {
  return uniform_domain<T, N>(-100., 100.);
},
[]() { return uniform_point<T, N>(0.); } }
```

test_functions::Schwefel is Schwefel test function.

$$f^*(\vec{x}) = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{i} x_i \right)^2$$

6.2.4.16 sphere

```
template<std::floating_point T, std::size_t N>
const test_function<T, N> quile::test_functions::sphere
```

Initial value:

```
{
  "sphere",
  [](const point<T, N>& p) {
    return std::transform_reduce(
      std::begin(p), std::end(p), T{ 0.  }, std::plus<T>{}, square<T>);
  },
  []() { return uniform_domain<T, N>(0., 10.); },
  []() { return uniform_point<T, N>(0.); }
}
```

test_functions::sphere is sphere test function.

$$f^*(\vec{x}) = \sum_{i=0}^{n-1} x_i^2$$

6.3 std Namespace Reference

Classes

- struct hash< G >

6.3.1 Detailed Description

Namespace `std` is opened only for the purpose of injecting `hash` into it.

Chapter 7

Class Documentation

7.1 quile::fitness_db< G > Class Template Reference

`#include <quile/quile.h>`

Public Types

- using const_iterator = typename database::const_iterator

Public Member Functions

- fitness_db (const fitness_function< G > &f, const genotype_constraints< G > auto &gc, unsigned int thread_sz=std::thread::hardware_concurrency())
- fitness_db (const fitness_db &)=default
- fitness_db & operator= (const fitness_db &)=default
- fitness operator() (const G &g) const
- fitnesses operator() (const population< G > &p) const
- std::size_t size () const
- const_iterator begin () const
- const_iterator end () const
- population< G > rank_order () const

7.1.1 Detailed Description

template<typename G>
class quile::fitness_db< G >

`fitness_db` is an intermediary object to fitness function values database.

Template Parameters

G	Some `genotype` specialization.

Note

Intermediary objects own database through the `std::shared_ptr`.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d»;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout « "Fitness function value for " « g « " is " « fd(g) « '.'
             « std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout « "fitness function value for " « p[i] « " is " « fs[i] « '.'
             « std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout « "Database size is equal to " « fd.size() « '.'  « std::endl;
  std::cout « "Database content:" « std::endl;
  for (auto x :  fd) {
    std::cout « ' ' « x.first « " " « x.second « std::endl;
  }
  std::cout « "Database content (once again):" « std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout « ' ' « it->first « " " « it->second « std::endl;
  }
  std::cout « "The best genotype is " « fd.rank_order()[0] « '.'
           « std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
```

```
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

7.1.2 Member Typedef Documentation

7.1.2.1 const_iterator

```
template<typename G >
using quile::fitness_db< G >::const_iterator = typename database::const_iterator
```

`fitness_db::const_iterator` is a constant iterator to the underlying `database` container.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d>>;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout << "Fitness function value for " << g << " is " << fd(g) << '.'
              << std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout << "fitness function value for " << p[i] << " is " << fs[i] << '.'
              << std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout << "Database size is equal to " << fd.size() << '.'  << std::endl;
  std::cout << "Database content:" << std::endl;
  for (auto x :  fd) {
    std::cout << ' ' << x.first << " " << x.second << std::endl;
  }
  std::cout << "Database content (once again):" << std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout << ' ' << it->first << " " << it->second << std::endl;
  }
  std::cout << "The best genotype is " << fd.rank_order()[0] << '.'
            << std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

7.1.3 Constructor & Destructor Documentation

7.1.3.1 fitness_db() [1/2]

```
template<typename G >
quile::fitness_db< G >::fitness_db (
            const fitness_function< G > & f,
            const genotype_constraints< G > auto & gc,
            unsigned int thread_sz = std::thread::hardware_concurrency() )  [inline], [explicit]
```

fitness_db::fitness_db constructor creates intermediary object to fitness function values database.

Parameters

f	Fitness function.
gc	Predicate defining proper genotypes.
thread_sz	Number of threads for concurrent fitness function values calculations. Default value is equal to `std::thread::hardware_concurrency()`.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d>>;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout << "Fitness function value for " << g << " is " << fd(g) << '.'
              << std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout << "fitness function value for " << p[i] << " is " << fs[i] << '.'
              << std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout << "Database size is equal to " << fd.size() << '.'  << std::endl;
  std::cout << "Database content:" << std::endl;
  for (auto x :  fd) {
    std::cout << ' ' << x.first << " " << x.second << std::endl;
  }
  std::cout << "Database content (once again):" << std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout << ' ' << it->first << " " << it->second << std::endl;
  }
  std::cout << "The best genotype is " << fd.rank_order()[0] << '.'
            << std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
```

```
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

7.1.3.2 fitness_db() [2/2]

```
template<typename G >
quile::fitness_db< G >::fitness_db (
            const fitness_db< G > &  )  [default]
```

Default copy constructor fitness_db::fitness_db.

7.1.4 Member Function Documentation

7.1.4.1 begin()

```
template<typename G >
const_iterator quile::fitness_db< G >::begin ( ) const  [inline]
```

fitness_db::begin returns constant iterator to the begin of database.

Returns

Constant iterator to the begin of database.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d»;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout « "Fitness function value for " « g « " is " « fd(g) « '.'
             « std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout « "fitness function value for " « p[i] « " is " « fs[i] « '.'
             « std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout « "Database size is equal to " « fd.size() « '.'  « std::endl;
  std::cout « "Database content:" « std::endl;
  for (auto x :  fd) {
    std::cout « ' ' « x.first « " " « x.second « std::endl;
  }
  std::cout « "Database content (once again):" « std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout « ' ' « it->first « " " « it->second « std::endl;
  }
  std::cout « "The best genotype is " « fd.rank_order()[0] « '.'
           « std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
```

```
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

7.1.4.2 end()

```
template<typename G >
const_iterator quile::fitness_db< G >::end ( ) const  [inline]
```

`fitness_db::end` returns constant iterator to the end of database.

Returns

Constant iterator to the end of database.

Note

The word *end* means past-the-last element.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d»;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout « "Fitness function value for " « g « " is " « fd(g) « '.'
              « std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout « "fitness function value for " « p[i] « " is " « fs[i] « '.'
              « std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout « "Database size is equal to " « fd.size() « '.'  « std::endl;
```

```
  std::cout « "Database content:" « std::endl;
  for (auto x :  fd) {
    std::cout « ' ' « x.first « " " « x.second « std::endl;
  }
  std::cout « "Database content (once again):" « std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout « ' ' « it->first « " " « it->second « std::endl;
  }
  std::cout « "The best genotype is " « fd.rank_order()[0] « '.'
            « std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

7.1.4.3 operator()() [1/2]

```
template<typename G >
fitness quile::fitness_db< G >::operator() (
            const G & g ) const  [inline]
```

`fitness_db::operator()` returns fitness function value for genotype `g` from database. If the value is not yet available, it is calculated, inserted to the database and then returned to the caller.

Parameters

g	Genotype for which fitness function value is needed.

Returns

Fitness function value for genotype `g`.

Note

This method is non-concurrent.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d>>;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout << "Fitness function value for " << g << " is " << fd(g) << '.'
              << std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout << "fitness function value for " << p[i] << " is " << fs[i] << '.'
              << std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout << "Database size is equal to " << fd.size() << '.'  << std::endl;
  std::cout << "Database content:" << std::endl;
  for (auto x :  fd) {
    std::cout << ' ' << x.first << " " << x.second << std::endl;
  }
  std::cout << "Database content (once again):" << std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout << ' ' << it->first << " " << it->second << std::endl;
  }
  std::cout << "The best genotype is " << fd.rank_order()[0] << '.'
            << std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
```

```
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

7.1.4.4 operator()() [2/2]

```
template<typename G >
fitnesses quile::fitness_db< G >::operator() (
            const population< G > & p ) const  [inline]
```

`fitness_db::operator()` returns fitness function values for genotypes from population from database. If some values are not yet available, they are calculated, inserted to the database and then all values are returned to the caller.

Parameters

p	Population for which fitness function values are needed.

Returns

Fitness function values for genotypes from population `p` in order corresponding to the order of genotypes in population itself.

Note

This method is potentially concurrent.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d»;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout « "Fitness function value for " « g « " is " « fd(g) « '.'
             « std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout « "fitness function value for " « p[i] « " is " « fs[i] « '.'
             « std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout « "Database size is equal to " « fd.size() « '.'  « std::endl;
  std::cout « "Database content:" « std::endl;
  for (auto x :  fd) {
    std::cout « ' ' « x.first « " " « x.second « std::endl;
  }
  std::cout « "Database content (once again):" « std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout « ' ' « it->first « " " « it->second « std::endl;
  }
  std::cout « "The best genotype is " « fd.rank_order()[0] « '.'
           « std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
```

```
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

7.1.4.5 operator=()

```
template<typename G >
fitness_db& quile::fitness_db< G >::operator= (
             const fitness_db< G > &  )  [default]
```

Default assignment operator `fitness_db::operator=`.

7.1.4.6 rank_order()

```
template<typename G >
population<G> quile::fitness_db< G >::rank_order ( ) const  [inline]
```

`fitness_db::rank_order` returns all genotypes from database in descending order of fitness function value.

Returns

Population consisting of all genotypes from database.

Note

`rank_order()[0]` gives the best genotype for non-empty database.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d>>;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
```

```
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout « "Fitness function value for " « g « " is " « fd(g) « '.'
             « std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout « "fitness function value for " « p[i] « " is " « fs[i] « '.'
             « std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout « "Database size is equal to " « fd.size() « '.'  « std::endl;
  std::cout « "Database content:" « std::endl;
  for (auto x :  fd) {
    std::cout « ' ' « x.first « " " « x.second « std::endl;
  }
  std::cout « "Database content (once again):" « std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout « ' ' « it->first « " " « it->second « std::endl;
  }
  std::cout « "The best genotype is " « fd.rank_order()[0] « '.'
           « std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
```

```
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

7.1.4.7 size()

```
template<typename G >
std::size_t quile::fitness_db< G >::size ( ) const  [inline]
```

fitness_db::size returns number of keys (genotypes) in database.

Returns

Number of database keys.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d>>;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout << "Fitness function value for " << g << " is " << fd(g) << '.'
              << std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout << "fitness function value for " << p[i] << " is " << fs[i] << '.'
              << std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout << "Database size is equal to " << fd.size() << '.'  << std::endl;
  std::cout << "Database content:" << std::endl;
  for (auto x :  fd) {
    std::cout << ' ' << x.first << " " << x.second << std::endl;
  }
  std::cout << "Database content (once again):" << std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout << ' ' << it->first << " " << it->second << std::endl;
  }
  std::cout << "The best genotype is " << fd.rank_order()[0] << '.'
            << std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
```

```
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

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

- quile/quile.h

7.2 quile::fitness_proportional_selection< G > Class Template Reference

`#include <quile/quile.h>`

Public Member Functions

- fitness_proportional_selection (const fitness_db< G > &ff)
- selection_probabilities operator() (const population< G > &p) const

7.2.1 Detailed Description

template<typename G>
class quile::fitness_proportional_selection< G >

`fitness_proportional_selection` is fitness proportional selection (a.k.a. fitness *proportionate* selection) with windowing procedure (FPS).

Template Parameters

G	Some `genotype` specialization.

Note

This implementation has workarounds for population of equally fit genotypes and populations containing genotypes which fitnesses cannot be calculated. Please note that there should be at least one genotype, which fitness can be calculated.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout « p[i] « ":  " « sp[i] « ", " « cp[i] « '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
```

```
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

7.2.2 Constructor & Destructor Documentation

7.2.2.1 fitness_proportional_selection()

```
template<typename G >
quile::fitness_proportional_selection< G >::fitness_proportional_selection (
            const fitness_db< G > & ff )  [inline], [explicit]
```

fitness_proportional_selection::fitness_proportional_selection constructor creates FPS mechanism for database represented by intermediary object `ff`.

Parameters

ff	Fitness database intermediary object.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
```

```
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout « p[i] « ":  " « sp[i] « ", " « cp[i] « '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

7.2.3 Member Function Documentation

7.2.3.1 operator()()

```
template<typename G >
selection_probabilities quile::fitness_proportional_selection< G >::operator() (
            const population< G > & p ) const  [inline]
```

`fitness_proportional_selection::operator()` returns selection probabilities for population `p`.

Parameters

p	Population.

Returns

FPS selection probabilities for population `p`.

Exceptions

std::runtime_error	Exception is raised if fitness function evaluates to `incalculable` for all genotypes from `p`.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout << p[i] << ":  " << sp[i] << ", " << cp[i] << '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
```

```
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

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

- quile/quile.h

7.3 quile::g_binary< N > Struct Template Reference

`#include <quile/quile.h>`

Public Types

- using type = bool
- using chain_t = chain< type, size()>

Static Public Member Functions

- static constexpr std::size_t size ()
- static constexpr const domain< type, size()> constraints ()
- static bool valid (const chain< type, size()> &)
- static chain_t default_chain ()

7.3.1 Detailed Description

template<std::size_t N>
struct quile::g_binary< N >

`g_binary` specifies that `genotype` has binary representation.

Template Parameters

N	Genotype length.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
requires quile::binary_representation<T>
struct test
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

7.3.2 Member Typedef Documentation

7.3.2.1 chain_t

```
template<std::size_t N>
using quile::g_binary< N >::chain_t = chain<type, size()>
```

`g_binary::chain_t` is genetic chain type used as underlying representation in `genotype`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
requires quile::binary_representation<T>
struct test
```

```
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

7.3.2.2 type

```
template<std::size_t N>
using quile::g_binary< N >::type = bool
```

g_binary::type is Boolean type used for representing gene values.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
requires quile::binary_representation<T>
struct test
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

7.3.3 Member Function Documentation

7.3.3.1 constraints()

```
template<std::size_t N>
static constexpr const domain<type, size()> quile::g_binary< N >::constraints ( )  [inline],
[static], [constexpr]
```

`constraints` returns domain, i.e. $\{0,1\}^N$.

Returns

Domain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
requires quile::binary_representation<T>
struct test
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

7.3.3.2 default_chain()

```
template<std::size_t N>
static chain_t quile::g_binary< N >::default_chain ( )  [inline], [static]
```

`default_chain` returns chain filled in default way.

Returns

Default chain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
```

```
requires quile::binary_representation<T>
struct test
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

7.3.3.3 size()

```
template<std::size_t N>
static constexpr std::size_t quile::g_binary< N >::size ( )  [inline], [static], [constexpr]
```

`size` returns domain size, i.e. `N`.

Returns

Genotype length (domain size).

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
requires quile::binary_representation<T>
struct test
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

7.3.3.4 valid()

```
template<std::size_t N>
static bool quile::g_binary< N >::valid (
            const chain< type, size()> &  )  [inline], [static]
```

`valid` checks whether its argument belongs to the domain.

Returns

Boolean value of check result.

Note

Result is equal to `true` by definition.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
requires quile::binary_representation<T>
struct test
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

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

- quile/quile.h

7.4 quile::g_floating_point< T, N, D > Struct Template Reference

`#include <quile/quile.h>`

Public Types

- using type = T
- using chain_t = chain< type, size()>

Static Public Member Functions

- static constexpr std::size_t size ()
- static constexpr const domain< type, size()> & constraints ()
- static bool valid (const chain< type, size()> &c)
- static chain_t default_chain ()

7.4.1 Detailed Description

template<typename T, std::size_t N, const domain< T, N > * D>
struct quile::g_floating_point< T, N, D >

`g_floating_point` specifies that `genotype` has floating-point representation.

Template Parameters

T	Floating-point type of representation.
N	Genotype length.
D	Pointer to the genotype domain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

7.4.2 Member Typedef Documentation

7.4.2.1 chain_t

```
template<typename T , std::size_t N, const domain< T, N > * D>
using quile::g_floating_point< T, N, D >::chain_t = chain<type, size()>
```

g_floating_point::chain_t is genetic chain type used as underlying representation in genotype.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

7.4.2.2 type

```
template<typename T , std::size_t N, const domain< T, N > * D>
using quile::g_floating_point< T, N, D >::type = T
```

g_floating_point::type is floating-point type used for representing gene values.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

7.4.3 Member Function Documentation

7.4.3.1 constraints()

```
template<typename T , std::size_t N, const domain< T, N > * D>
static constexpr const domain<type, size()>& quile::g_floating_point< T, N, D >::constraints
( )  [inline], [static], [constexpr]
```

`constraints` returns domain, i.e. `*D`.

Returns

Domain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

7.4.3.2 default_chain()

```
template<typename T , std::size_t N, const domain< T, N > * D>
static chain_t quile::g_floating_point< T, N, D >::default_chain ( )  [inline], [static]
```

`default_chain` returns chain filled in default way.

Returns

Default chain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
```

```
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

7.4.3.3 size()

```
template<typename T , std::size_t N, const domain< T, N > * D>
static constexpr std::size_t quile::g_floating_point< T, N, D >::size ( )  [inline], [static],
[constexpr]
```

`size` returns domain size, i.e. `N`.

Returns

Genotype length (domain size).

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

7.4.3.4 valid()

```
template<typename T , std::size_t N, const domain< T, N > * D>
static bool quile::g_floating_point< T, N, D >::valid (
            const chain< type, size()> & c )  [inline], [static]
```

`valid` checks whether `c` belongs to the domain and returns `true` in that case. Otherwise returns `false`.

Parameters

c	Chain to be checked.

Returns

Boolean value of check result.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

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

- quile/quile.h

7.5 quile::g_integer< T, N, D > Struct Template Reference

`#include <quile/quile.h>`

Public Types

- using type = T
- using chain_t = chain< type, size()>

Static Public Member Functions

- static constexpr std::size_t size ()
- static constexpr const domain< type, size()> & constraints ()
- static bool valid (const chain< type, size()> &c)
- static chain_t default_chain ()

7.5.1 Detailed Description

template<typename T, std::size_t N, const domain< T, N > * D>
struct quile::g_integer< T, N, D >

`g_integer` specifies that `genotype` has integer representation.

Template Parameters

T	Integer type of representation (excluding Boolean).
N	Genotype length.
D	Pointer to the genotype domain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

7.5.2 Member Typedef Documentation

7.5.2.1 chain_t

```
template<typename T , std::size_t N, const domain< T, N > * D>
using quile::g_integer< T, N, D >::chain_t = chain<type, size()>
```

`g_integer::chain_t` is genetic chain type used as underlying representation in `genotype`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

7.5.2.2 type

```
template<typename T , std::size_t N, const domain< T, N > * D>
using quile::g_integer< T, N, D >::type = T
```

`g_integer::type` is integer non-Boolean type used for representing gene values.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

7.5.3 Member Function Documentation

7.5.3.1 constraints()

```
template<typename T , std::size_t N, const domain< T, N > * D>
static constexpr const domain<type, size()>& quile::g_integer< T, N, D >::constraints ( )
[inline], [static], [constexpr]
```

`constraints` returns domain, i.e. `*D`.

Returns

Domain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
```

```
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

7.5.3.2 default_chain()

```
template<typename T , std::size_t N, const domain< T, N > * D>
static chain_t quile::g_integer< T, N, D >::default_chain ( )  [inline], [static]
```

`default_chain` returns chain filled in default way.

Returns

Default chain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

7.5.3.3 size()

```
template<typename T , std::size_t N, const domain< T, N > * D>
static constexpr std::size_t quile::g_integer< T, N, D >::size ( )  [inline], [static], [constexpr]
```

`size` returns domain size, i.e. `N`.

Returns

Genotype length (domain size).

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

7.5.3.4 valid()

```
template<typename T , std::size_t N, const domain< T, N > * D>
static bool quile::g_integer< T, N, D >::valid (
             const chain< type, size()> & c )  [inline], [static]
```

`valid` checks whether `c` belongs to the domain and returns `true` in that case. Otherwise returns `false`.

Parameters

c	Chain to be checked.

Returns

Boolean value of check result.

Example:

```
#include <cassert>
```

```
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

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

- quile/quile.h

7.6 quile::g_permutation< T, N, M > Struct Template Reference

`#include <quile/quile.h>`

Public Types

- using type = T
- using chain_t = chain< type, size()>

Static Public Member Functions

- static constexpr std::size_t size ()
- static constexpr const domain< type, size()> constraints ()
- static bool valid (const chain< type, size()> &c)
- static chain_t default_chain ()

7.6.1 Detailed Description

template<typename T, std::size_t N, T M>
struct quile::g_permutation< T, N, M >

`g_permutation` specifies that `genotype` has permutation representation.

Template Parameters

T	Integer type of representation (excluding Boolean).
N	Genotype length.
M	The lowest permuted number.

Note

Numbers from $\{M, \ldots, M + N - 1\}$ set are permuted.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
requires quile::permutation_representation<T>
struct test
{};
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

7.6.2 Member Typedef Documentation

7.6.2.1 chain_t

```
template<typename T , std::size_t N, T M>
using quile::g_permutation< T, N, M >::chain_t = chain<type, size()>
```

`g_permutation::chain_t` is genetic chain type used as underlying representation in `genotype`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
requires quile::permutation_representation<T>
struct test
{};
```

```
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

7.6.2.2 type

```
template<typename T , std::size_t N, T M>
using quile::g_permutation< T, N, M >::type = T
```

g_permutation::type is integer non-Boolean type used for representing gene values.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
requires quile::permutation_representation<T>
struct test
{};
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

7.6.3 Member Function Documentation

7.6.3.1 constraints()

```
template<typename T , std::size_t N, T M>
static constexpr const domain<type, size()> quile::g_permutation< T, N, M >::constraints ( )
[inline], [static], [constexpr]
```

`constraints` returns domain, i.e. $\{M, \ldots, M + N - 1\}^N$.

Returns

Domain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
requires quile::permutation_representation<T>
struct test
{};
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

7.6.3.2 default_chain()

```
template<typename T , std::size_t N, T M>
static chain_t quile::g_permutation< T, N, M >::default_chain ( )  [inline], [static]
```

`default_chain` returns chain filled in default way.

Returns

Default chain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
```

```
requires quile::permutation_representation<T>
struct test
{};
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

7.6.3.3 size()

```
template<typename T , std::size_t N, T M>
static constexpr std::size_t quile::g_permutation< T, N, M >::size ( )  [inline], [static],
[constexpr]
```

`size` returns domain size, i.e. `N`.

Returns

Genotype length (domain size).

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
requires quile::permutation_representation<T>
struct test
{};
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

7.6.3.4 valid()

```
template<typename T , std::size_t N, T M>
static bool quile::g_permutation< T, N, M >::valid (
            const chain< type, size()> & c )  [inline], [static]
```

`valid` checks whether `c` belongs to the domain (incl. check of the permutation condition) and returns `true` in that case. Otherwise returns `false`.

Parameters

c	Chain to be checked.

Returns

Boolean value of check result.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
requires quile::permutation_representation<T>
struct test
{};
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

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

- quile/quile.h

7.7 quile::genotype< R > Class Template Reference

`#include <quile/quile.h>`

Public Types

- using chain_t = chain< typename R::type, R::size()>
- using const_iterator = typename chain_t::const_iterator
- using gene_t = typename R::type
- using genotype_t = R

Public Member Functions

- genotype ()
- genotype (const chain_t &c)
- genotype (const genotype &)=default
- genotype (genotype &&)=default
- genotype & operator= (const genotype &)=default
- genotype & operator= (genotype &&)=default
- gene_t value (std::size_t i) const
- template<typename S = R, typename = std::enable_if_t<!permutation_representation<S>>> genotype & value (std::size_t i, gene_t v)
- genotype & random_reset ()
- template<typename S = R, typename = std::enable_if_t<!permutation_representation<S>>> genotype & random_reset (std::size_t i)
- auto operator<=> (const genotype &g) const
- bool operator== (const genotype &g) const
- const chain_t & data () const
- const_iterator begin () const
- const_iterator end () const

Static Public Member Functions

- static constexpr std::size_t size ()
- static constexpr const domain< gene_t, size()> constraints ()
- static bool valid (const chain_t &c)
- static genotype random ()

Static Public Attributes

- static constexpr bool uniform_domain = uniform(constraints())

7.7.1 Detailed Description

template<typename R>
class quile::genotype< R >

`genotype` is central type of the library—it allows genotype creation and manipulation.

Template Parameters

R	Type satisfying `chromosome_representation` concept.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
```

```
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

7.7.2 Member Typedef Documentation

7.7.2.1 chain_t

```
template<typename R >
using quile::genotype< R >::chain_t = chain<typename R::type, R::size()>
```

genotype::chain_t is genetic chain type (underlying genotype representation).

Example:

```
#include <exception>
#include <iostream>
#include <quile/quile.h>
#include <type_traits>
using p_type = quile::g_permutation<int, 42, 0>;
using c_type = quile::chain<int, 42>;
static_assert(std::is_same_v<c_type, p_type::chain_t>);
int
main()
{
  using G = quile::genotype<p_type>;
  const G g0{};
  c_type c{ g0.data() };
  for (auto& x :  c) {
    x = 0;
  }
  try {
    const G g1{ c };
  } catch (std::exception& e) {
    std::cout « e.what() « '\n';
  }
}
```

Result:

```
invalid chain
```

7.7.2.2 const_iterator

```
template<typename R >
using quile::genotype< R >::const_iterator = typename chain_t::const_iterator
```

genotype::const_iterator is constant iterator to access underlying representation.

Example:

```
#include <iostream>
#include <quile/quile.h>
using p_type = quile::g_permutation<int, 42, 0>;
int
main()
{
  using G = quile::genotype<p_type>;
  const G g{};
  for (G::const_iterator it = g.begin(); it != g.end(); ++it) {
    std::cout « *it « ' ';
  }
  std::cout « '\n';
  for (auto x :  g) {
    std::cout « x « ' ';
  }
  std::cout « '\n';
}
```

Result:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41
```

7.7.2.3 gene_t

```
template<typename R >
using quile::genotype< R >::gene_t = typename R::type
```

genotype::gene_t is type of gene (e.g. floating-point).

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
```

```
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

7.7.2.4 genotype_t

```
template<typename R >
using quile::genotype< R >::genotype_t = R
```

`genotype::genotype_t` is type of genotype representation dispatch tag.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

7.7.3 Constructor & Destructor Documentation

7.7.3.1 genotype() [1/4]

```
template<typename R >
quile::genotype< R >::genotype ( )  [inline]
```

genotype::genotype constructor creates object initialized with default genetic chain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

7.7.3.2 genotype() [2/4]

```
template<typename R >
quile::genotype< R >::genotype (
            const chain_t & c )  [inline], [explicit]
```

genotype::genotype constructor creates object initialized with genetic chain passed as its argument.

Parameters

c	Genetic chain to be used for initialization.

Exceptions

std::invalid_argument	Exception is raised if c does not belong to the domain or it does not fulfill condition of given representation (cf. permutation condition).

7.7.3.3 genotype() [3/4]

```
template<typename R >
quile::genotype< R >::genotype (
            const genotype< R > &  )  [default]
```

Default copy constructor genotype::genotype.

7.7.3.4 genotype() [4/4]

```
template<typename R >
quile::genotype< R >::genotype (
            genotype< R > &&  )  [default]
```

Default move constructor genotype::genotype.

7.7.4 Member Function Documentation

7.7.4.1 begin()

```
template<typename R >
const_iterator quile::genotype< R >::begin ( ) const  [inline]
```

genotype::begin returns constant iterator to the begin of genetic chain.

Returns

Constant iterator to the begin of genetic chain.

Example:

```
#include <iostream>
#include <quile/quile.h>
using p_type = quile::g_permutation<int, 42, 0>;
int
main()
{
  using G = quile::genotype<p_type>;
  const G g{};
  for (G::const_iterator it = g.begin(); it != g.end(); ++it) {
    std::cout « *it « ' ';
  }
  std::cout « '\n';
  for (auto x :  g) {
    std::cout « x « ' ';
  }
  std::cout « '\n';
}
```

Result:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41
```

7.7.4.2 constraints()

```
template<typename R >
static constexpr const domain<gene_t, size()> quile::genotype< R >::constraints ( )  [inline],
[static], [constexpr]
```

`genotype::constraints` returns domain.

Returns

Domain.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

7.7.4.3 data()

```
template<typename R >
const chain_t& quile::genotype< R >::data ( ) const  [inline]
```

`genotype::data` returns constant reference to the underlying genetic chain.

Returns

Constant reference to the genetic chain.

Example:

```
#include <exception>
#include <iostream>
#include <quile/quile.h>
#include <type_traits>
using p_type = quile::g_permutation<int, 42, 0>;
using c_type = quile::chain<int, 42>;
static_assert(std::is_same_v<c_type, p_type::chain_t>);
int
main()
{
  using G = quile::genotype<p_type>;
  const G g0{};
  c_type c{ g0.data() };
  for (auto& x :  c) {
    x = 0;
  }
  try {
    const G g1{ c };
  } catch (std::exception& e) {
    std::cout « e.what() « '\n';
  }
}
```

Result:

```
invalid chain
```

7.7.4.4 end()

```
template<typename R >
const_iterator quile::genotype< R >::end ( ) const  [inline]
```

`genotype::end` returns constant iterator to the end of genetic chain.

Returns

Constant iterator to the end of genetic chain.

Note

The word *end* means past-the-last element.

Example:

```
#include <iostream>
#include <quile/quile.h>
using p_type = quile::g_permutation<int, 42, 0>;
int
main()
{
  using G = quile::genotype<p_type>;
  const G g{};
  for (G::const_iterator it = g.begin(); it != g.end(); ++it) {
    std::cout « *it « ' ';
  }
  std::cout « '\n';
  for (auto x :  g) {
    std::cout « x « ' ';
  }
  std::cout « '\n';
}
```

Result:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41
```

7.7.4.5 operator<=>()

```
template<typename R >
auto quile::genotype< R >::operator<=> (
             const genotype< R > & g ) const  [inline]
```

`genotype::operator<=>` performs default lexicographical comparison with use of genotypes' genetic chain.

Parameters

g	Genotype to be compared with `*this`.

Returns

Ordering (cf. `std::strong_ordering`, `std::weak_ordering`, `std::partial_ordering`).

Note

Comparisons of genotypes with floating-point representation does not include tolerance.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  static const auto d{ quile::uniform_domain<int, 2>(0, 2) };
  using G = quile::genotype<quile::g_integer<int, 2, &d>>;
  const G g0{ { 0, 2 } };
  const G g1{ { 1, 2 } };
  assert(g0 == g0);
  assert(g0 != g1);
  assert(g0 < g1);
  assert(g0 <= g1);
  assert(g1 > g0);
  assert(g1 >= g0);
}
```

Result (might be empty):

```
#include <cassert>
#include <quile/quile.h>

int
main()
{
  static const auto d{ quile::uniform_domain<int, 2>(0, 2) };
  using G = quile::genotype<quile::g_integer<int, 2, &d>>;
  const G g0{ { 0, 2 } };
  const G g1{ { 1, 2 } };
  assert(g0 == g0);
  assert(g0 != g1);
  assert(g0 < g1);
  assert(g0 <= g1);
  assert(g1 > g0);
  assert(g1 >= g0);
}
```

7.7.4.6 operator=() [1/2]

```
template<typename R >
genotype& quile::genotype< R >::operator= (
            const genotype< R > &  )  [default]
```

Default assignment operator genotype::operator=.

7.7.4.7 operator=() [2/2]

```
template<typename R >
genotype& quile::genotype< R >::operator= (
            genotype< R > &&  )  [default]
```

Default move assignment operator genotype::operator=.

7.7.4.8 operator==()

```
template<typename R >
bool quile::genotype< R >::operator== (
            const genotype< R > & g ) const  [inline]
```

genotype::operator== performs comparison with use of genotypes' genetic chain.

Parameters

g	Genotype to be compared with *this.

Returns

Boolean value true if chains are equal and false, otherwise.

Note

Comparisons of genotypes with floating-point representation does not include tolerance.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  static const auto d{ quile::uniform_domain<int, 2>(0, 2) };
  using G = quile::genotype<quile::g_integer<int, 2, &d>>;
  const G g0{ { 0, 2 } };
  const G g1{ { 1, 2 } };
  assert(g0 == g0);
  assert(g0 != g1);
  assert(g0 < g1);
  assert(g0 <= g1);
  assert(g1 > g0);
  assert(g1 >= g0);
}
```

Result (might be empty):

```
#include <cassert>
#include <quile/quile.h>

int
main()
{
  static const auto d{ quile::uniform_domain<int, 2>(0, 2) };
  using G = quile::genotype<quile::g_integer<int, 2, &d>>;
  const G g0{ { 0, 2 } };
  const G g1{ { 1, 2 } };
  assert(g0 == g0);
  assert(g0 != g1);
  assert(g0 < g1);
  assert(g0 <= g1);
  assert(g1 > g0);
  assert(g1 >= g0);
}
```

7.7.4.9 random()

```
template<typename R >
static genotype quile::genotype< R >::random ( )  [inline], [static]
```

genotype::random returns random genotype.

Returns

Random genotype.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  static const auto d{ quile::uniform_domain<int, 5>(0, 9) };
  std::cout « quile::genotype<quile::g_integer<int, 5, &d»::random() « '\n';
}
```

Result (might be different due to randomness):

```
7 3 6 5 3
```

7.7.4.10 random_reset() [1/2]

```
template<typename R >
genotype& quile::genotype< R >::random_reset ( )  [inline]
```

genotype::random_reset changes each gene value randomly using uniform random distribution with intervals defined by domain.

Returns

Reference to *this.

Note

This overload is also available for permutation representation and draws new permutation in that case.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  static const auto d{ quile::uniform_domain<int, 5>(0, 9) };
  using G = quile::genotype<quile::g_integer<int, 5, &d>>;
  G g{};
  std::cout « g « '\n';
  std::cout « g.random_reset(2) « '\n';
  std::cout « g.random_reset() « '\n';
}
```

Result (might be different due to randomness):

```
0 0 0 0 0
0 0 6 0 0
4 2 9 8 8
```

7.7.4.11 random_reset() [2/2]

```
template<typename R >
template<typename S = R, typename = std::enable_if_t<!permutation_representation<S>>>
genotype& quile::genotype< R >::random_reset (
            std::size_t i )  [inline]
```

`genotype::random_reset` changes gene at *locus* `i` value randomly using uniform random distribution with interval defined by domain.

Parameters

i	Gene *locus*.

Returns

Reference to `*this`.

Note

This overload is not available for permutation representation.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  static const auto d{ quile::uniform_domain<int, 5>(0, 9) };
  using G = quile::genotype<quile::g_integer<int, 5, &d>>;
  G g{};
  std::cout « g « '\n';
  std::cout « g.random_reset(2) « '\n';
  std::cout « g.random_reset() « '\n';
}
```

Result (might be different due to randomness):

```
0 0 0 0 0
0 0 6 0 0
4 2 9 8 8
```

7.7.4.12 size()

```
template<typename R >
static constexpr std::size_t quile::genotype< R >::size ( )  [inline], [static], [constexpr]
```

genotype::size returns domain size.

Returns

Genotype length (domain size).

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

7.7.4.13 valid()

```
template<typename R >
static bool quile::genotype< R >::valid (
            const chain_t & c )  [inline], [static]
```

genotype::valid checks whether c belongs to the domain and returns true in that case. Otherwise returns false.

Parameters

c	Chain to be checked.

Returns

Boolean value of check result.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

7.7.4.14 value() [1/2]

```
template<typename R >
gene_t quile::genotype< R >::value (
            std::size_t i ) const  [inline]
```

`genotype::value` returns gene value at *locus* i.

Parameters

i	Gene *locus*.

Returns

Gene value.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  static const auto d{ quile::uniform_domain<int, 5>(0, 9) };
  using G = quile::genotype<quile::g_integer<int, 5, &d>>;
  G g{};
  std::cout « g « '\n';
  std::cout « g.value(2, 5) « '\n';
  std::cout « g.value(2) « '\n';
  std::cout « g « '\n';
}
```

Result:

```
0 0 0 0 0
0 0 5 0 0
5
0 0 5 0 0
```

7.7.4.15 value() [2/2]

```
template<typename R >
template<typename S = R, typename = std::enable_if_t<!permutation_representation<S>>>
genotype& quile::genotype< R >::value (
            std::size_t i,
            gene_t v )  [inline]
```

`genotype::value` changes gene value to `v` at *locus* `i`.

Parameters

i	Gene *locus*.
v	New gene value.

Returns

Reference to `*this`.

Exceptions

std::invalid_argument	Exception is raised if new gene value is outside permitted interval for given *locus* `i`.

Note

This method is not available for permutation representation.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  static const auto d{ quile::uniform_domain<int, 5>(0, 9) };
  using G = quile::genotype<quile::g_integer<int, 5, &d>>;
  G g{};
  std::cout « g « '\n';
  std::cout « g.value(2, 5) « '\n';
  std::cout « g.value(2) « '\n';
  std::cout « g « '\n';
}
```

Result:

```
0 0 0 0 0
0 0 5 0 0
5
0 0 5 0 0
```

7.7.5 Member Data Documentation

7.7.5.1 uniform_domain

```
template<typename R >
constexpr bool quile::genotype< R >::uniform_domain = uniform(constraints())  [static], [constexpr]
```

`genotype::uniform_domain` states whether `genotype` domain is uniform, i.e. its domain is of form X_0^N.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

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

- quile/quile.h

7.8 std::hash< G > Struct Template Reference

`#include <quile/quile.h>`

Public Member Functions

- std::size_t operator() (const G &g) const noexcept

7.8.1 Detailed Description

template<typename G>
struct std::hash< G >

`std::hash` for `genotype`.

Note

This `std::hash` specialization allows interoperability with STL unordered associative containers like `std::unordered_map`.

`std::hash` is injected into `std` namespace of programs using this library.

7.8.2 Member Function Documentation

7.8.2.1 operator()()

```
template<typename G >
std::size_t std::hash< G >::operator() (
            const G & g ) const  [inline], [noexcept]
```

`std::hash::operator()` calculates hash function value for genotype `g`.

Parameters

g	Genotype.

Returns

Hash function value.

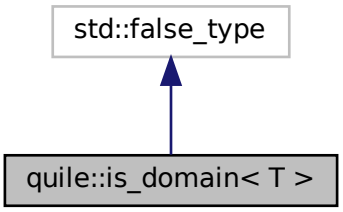
The documentation for this struct was generated from the following file:

- quile/quile.h

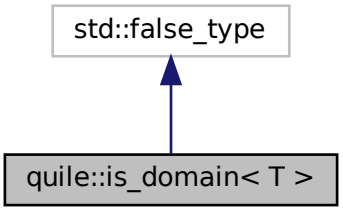
7.9 quile::is_domain< T > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_domain< T >:



Collaboration diagram for quile::is_domain< T >:



7.9.1 Detailed Description

template<typename T>
struct quile::is_domain< T >

If `T` is some specialization of `domain` then `is_domain` provides member constant `value` equal to `true`. Otherwise `value` is `false`.

Example:

```
#include <quile/quile.h>
#include <vector>
using type = quile::domain<double, 42>;
static_assert(quile::is_domain<type>::value);
static_assert(!quile::is_domain_v<std::vector<double»);
template<typename T>
requires quile::set_of_departure<T>
struct test
{};
int
main()
{
  [[maybe_unused]] test<type> t{};
}
```

Result (might be empty):

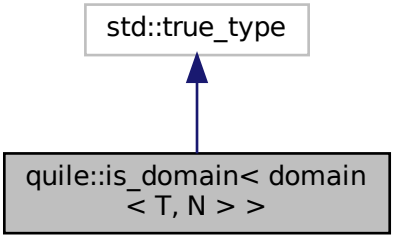
The documentation for this struct was generated from the following file:

- quile/quile.h

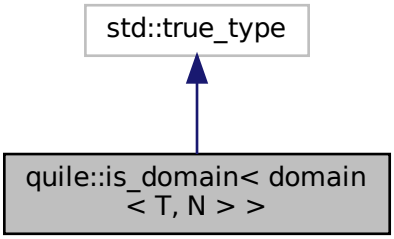
7.10 quile::is_domain< domain< T, N > > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_domain< domain< T, N > >:



Collaboration diagram for quile::is_domain< domain< T, N > >:



7.10.1 Detailed Description

template<typename T, std::size_t N>
struct quile::is_domain< domain< T, N > >

Please see documentation for `is_domain<T>`.

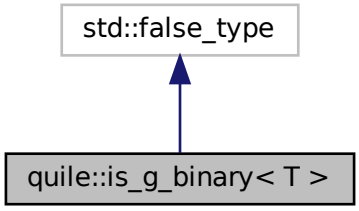
The documentation for this struct was generated from the following file:

- quile/quile.h

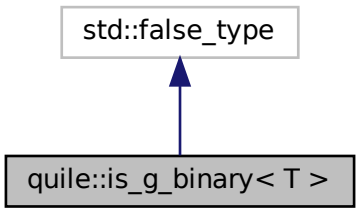
7.11 quile::is_g_binary< T > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_g_binary< T >:



Collaboration diagram for quile::is_g_binary< T >:



7.11.1 Detailed Description

template<typename T>
struct quile::is_g_binary< T >

If `T` is some specialization of `g_binary` then `is_g_binary` provides member constant value equal to true. Otherwise value is false.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
```

```
requires quile::binary_representation<T>
struct test
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

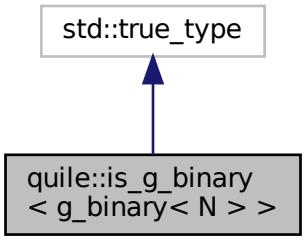
The documentation for this struct was generated from the following file:

- quile/quile.h

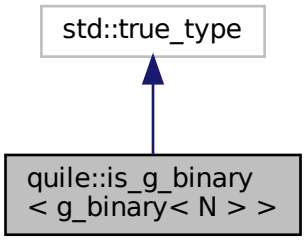
7.12 quile::is_g_binary< g_binary< N > > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_g_binary< g_binary< N > >:



Collaboration diagram for quile::is_g_binary< g_binary< N > >:



7.12.1 Detailed Description

template<std::size_t N>
struct quile::is_g_binary< g_binary< N > >

Please see documentation for is_g_binary<T>.

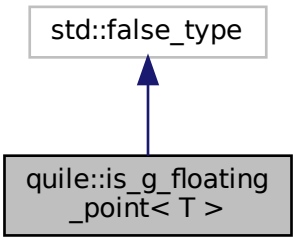
The documentation for this struct was generated from the following file:

- quile/quile.h

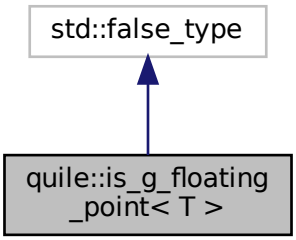
7.13 quile::is_g_floating_point< T > Struct Template Reference

#include <quile/quile.h>

Inheritance diagram for quile::is_g_floating_point< T >:



Collaboration diagram for quile::is_g_floating_point< T >:



7.13.1 Detailed Description

template<typename T>
struct quile::is_g_floating_point< T >

If `T` is some specialization of `g_floating_point` then `is_g_floating_point` provides member constant value equal to true. Otherwise value is false.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

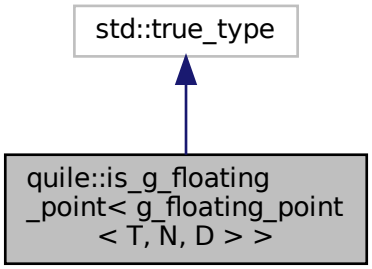
The documentation for this struct was generated from the following file:

- quile/quile.h

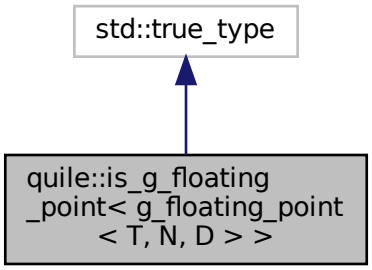
7.14 quile::is_g_floating_point< g_floating_point< T, N, D > > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_g_floating_point< g_floating_point< T, N, D > >:



Collaboration diagram for quile::is_g_floating_point< g_floating_point< T, N, D > >:



7.14.1 Detailed Description

template<typename T, std::size_t N, const domain< T, N > * D>
struct quile::is_g_floating_point< g_floating_point< T, N, D > >

Please see documentation for `is_g_floating_point<T>`.

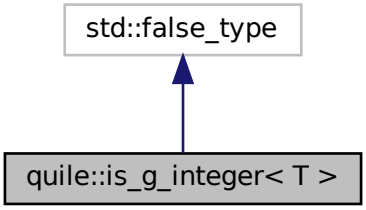
The documentation for this struct was generated from the following file:

- quile/quile.h

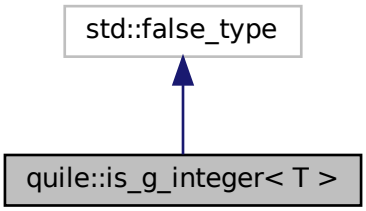
7.15 quile::is_g_integer< T > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_g_integer< T >:



Collaboration diagram for quile::is_g_integer< T >:



7.15.1 Detailed Description

template<typename T>
struct quile::is_g_integer< T >

If `T` is some specialization of `g_integer` then `is_g_integer` provides member constant value equal to true. Otherwise value is false.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

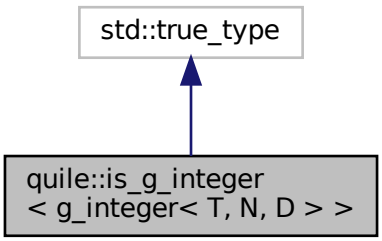
The documentation for this struct was generated from the following file:

- quile/quile.h

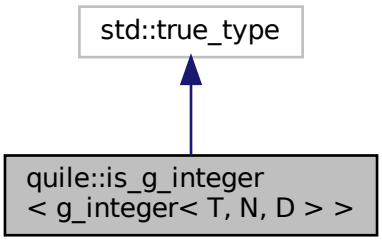
7.16 quile::is_g_integer< g_integer< T, N, D > > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_g_integer< g_integer< T, N, D > >:



Collaboration diagram for quile::is_g_integer< g_integer< T, N, D > >:



7.16.1 Detailed Description

template<typename T, std::size_t N, const domain< T, N > * D>
struct quile::is_g_integer< g_integer< T, N, D > >

Please see documentation for `is_g_integer<T>`.

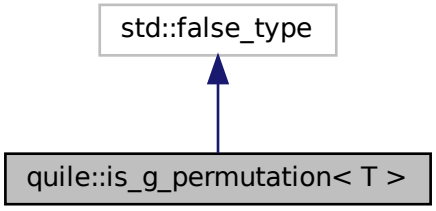
The documentation for this struct was generated from the following file:

- quile/quile.h

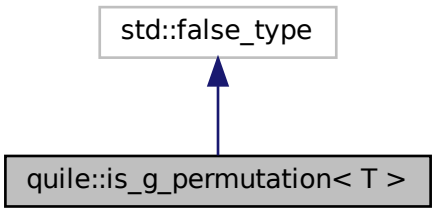
7.17 quile::is_g_permutation< T > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_g_permutation< T >:



Collaboration diagram for quile::is_g_permutation< T >:



7.17.1 Detailed Description

template<typename T>
struct quile::is_g_permutation< T >

If `T` is some specialization of `g_permutation` then `is_g_permutation` provides member constant value equal to true. Otherwise value is false.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
```

```
requires quile::permutation_representation<T>
struct test
{};
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

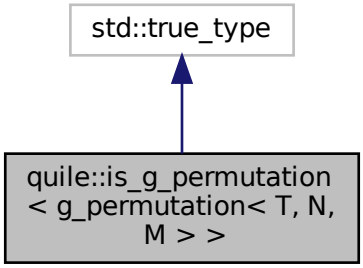
The documentation for this struct was generated from the following file:

- quile/quile.h

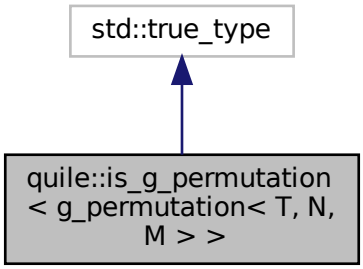
7.18 quile::is_g_permutation< g_permutation< T, N, M > > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_g_permutation< g_permutation< T, N, M > >:



Collaboration diagram for quile::is_g_permutation< g_permutation< T, N, M > >:



7.18.1 Detailed Description

template<typename T, std::size_t N, T M>
struct quile::is_g_permutation< g_permutation< T, N, M > >

Please see documentation for `is_g_permutation<T>`.

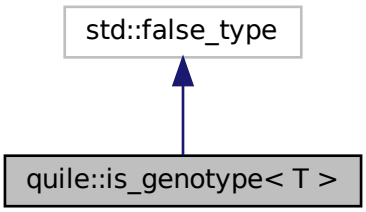
The documentation for this struct was generated from the following file:

- quile/quile.h

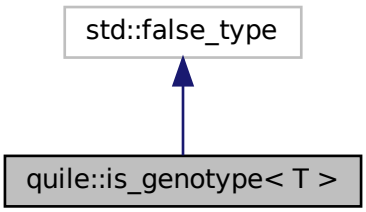
7.19 quile::is_genotype< T > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_genotype< T >:



Collaboration diagram for quile::is_genotype< T >:



7.19.1 Detailed Description

template<typename T>
struct quile::is_genotype< T >

If `T` is some specialization of `genotype` then `is_genotype` provides member constant value equal to true. Otherwise value is false.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

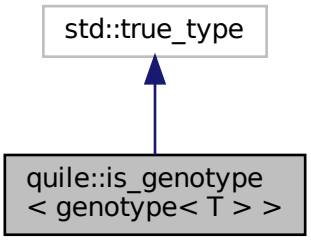
The documentation for this struct was generated from the following file:

- quile/quile.h

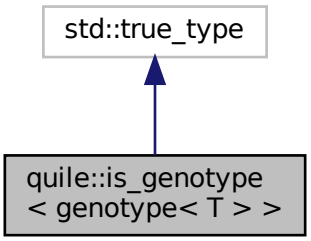
7.20 quile::is_genotype< genotype< T > > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_genotype< genotype< T > >:



Collaboration diagram for quile::is_genotype< genotype< T > >:



7.20.1 Detailed Description

template<typename T>
struct quile::is_genotype< genotype< T > >

Please see documentation for `is_genotype<T>`.

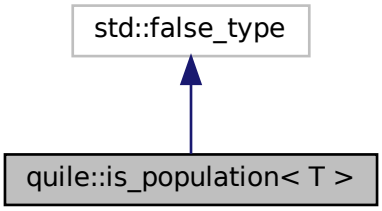
The documentation for this struct was generated from the following file:

- quile/quile.h

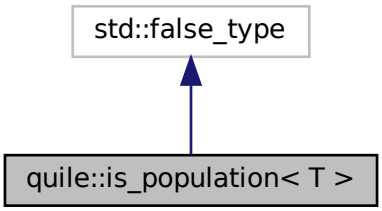
7.21 quile::is_population< T > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_population< T >:



Collaboration diagram for quile::is_population< T >:



7.21.1 Detailed Description

template<typename T>
struct quile::is_population< T >

If `T` is some specialization of `population` then `is_population` provides member constant value equal to true. Otherwise value is false.

Example:

```
#include <iostream>
#include <quile/quile.h>
const std::size_t n = 32;
using genotype_t = quile::genotype<quile::g_binary<n>>;
using population_t = quile::population<genotype_t>;
static_assert(quile::is_population<population_t>::value);
static_assert(!quile::is_population_v<genotype_t>);
template<typename T>
requires quile::genetic_pool<T>
struct test
{};
int
main()
{
  [[maybe_unused]] test<population_t> t{};
  population_t p{};
  for (int i = 0; i < 8; ++i) {
    p.push_back(genotype_t{});
```

```
  }
  for (auto& g :  p) {
    g.random_reset();
  }
  for (auto& g :  p) {
    std::cout « g « '\n';
  }
}
```

Result (might be different due to randomness):

```
1 1 1 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 1 0 0 1 1 0 1 0
0 0 1 0 1 0 1 1 0 0 0 1 1 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 0 0 0 1
0 0 0 0 1 0 1 0 1 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 1
1 0 0 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 1
0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 1 1 0 0 1 0 1 0 1 0 0 1 1
1 1 1 1 0 0 0 0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 1 0
1 1 1 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1
0 0 1 0 1 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1 0 0 0 0 1 1 1
```

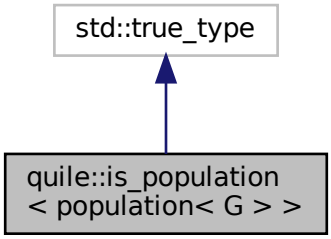
The documentation for this struct was generated from the following file:

- quile/quile.h

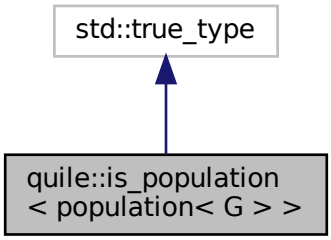
7.22 quile::is_population< population< G > > Struct Template Reference

`#include <quile/quile.h>`

Inheritance diagram for quile::is_population< population< G > >:



Collaboration diagram for quile::is_population< population< G > >:



7.22.1 Detailed Description

template<typename G>
struct quile::is_population< population< G > >

Please see documentation for `is_population<T>`.

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

- quile/quile.h

7.23 quile::range< T > Class Template Reference

`#include <quile/quile.h>`

Public Member Functions

- constexpr range (T min, T max)
- template<typename U = T, typename = std::enable_if_t<std::numeric_limits<U>::is_specialized>>
  constexpr range ()
- constexpr range (const range &)=default
- constexpr range (range &&)=default
- range & operator= (const range &)=default
- range & operator= (range &&)=default
- T min () const
- T max () const
- template<typename U = T, typename = std::enable_if_t<!std::is_same_v<U, bool>>>
  T midpoint () const
- template<typename U = T, typename = std::enable_if_t<!std::is_same_v<U, bool>>>
  T clamp (T t) const
- bool contains (T t) const
- auto operator<=> (const range< T > &r) const =default

7.23.1 Detailed Description

template<typename T>
class quile::range< T >

`range` represents subrange (closed interval) of type `T`.

Template Parameters

T	Base type.

7.23.2 Constructor & Destructor Documentation

7.23.2.1 range() [1/4]

```
template<typename T >
constexpr quile::range< T >::range (
            T min,
            T max )  [inline], [constexpr]
```

`range` constructor creates object representing closed interval $[\min, \max]_{\mathrm{T}}$.

Parameters

min	Range infimum.
max	Range supremum.

Exceptions

std::invalid_argument	Exception is raised if `min` is greater than `max`.

7.23.2.2 range() [2/4]

```
template<typename T >
template<typename U = T, typename = std::enable_if_t<std::numeric_limits<U>::is_specialized>>
constexpr quile::range< T >::range ( )  [inline], [constexpr]
```

`range` constructor creates object representing full (closed) range for type `T`.

Note

This constructor is available for types possesing `std::numeric_limits` specialization.

7.23.2.3 range() [3/4]

```
template<typename T >
constexpr quile::range< T >::range (
            const range< T > &  )  [constexpr], [default]
```

Default copy constructor `range::range`.

7.23.2.4 range() [4/4]

```
template<typename T >
constexpr quile::range< T >::range (
            range< T > &&  )  [constexpr], [default]
```

Default move constructor range::range.

7.23.3 Member Function Documentation

7.23.3.1 clamp()

```
template<typename T >
template<typename U = T, typename = std::enable_if_t<!std::is_same_v<U, bool>>>
T quile::range< T >::clamp (
            T t ) const  [inline]
```

range::clamp returns left endpoint if t compares less than left endpoint; otherwise returns right endpoint if right endpoint compares less than t; otherwise returns t.

Parameters

t	Value to be clamped.

Returns

Value clamped to the range.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  quile::range<int> r{ 0, 42 };
  assert(r.clamp(-10) == 0);
  assert(r.clamp(0) == 0);
  assert(r.clamp(7) == 7);
  assert(r.clamp(42) == 42);
  assert(r.clamp(100) == 42);
}
```

Result (might be empty):

7.23.3.2 contains()

```
template<typename T >
bool quile::range< T >::contains (
            T t ) const  [inline]
```

range::contains checks if its argument is contained within interval $[\min, \max]_{\mathrm{T}}$ represented by the range.

Parameters

t	Argument to be checked.

Returns

Boolean value of check: `true` if `t` is in $[\min, \max]_{\mathrm{T}}$ interval and `false` otherwise.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  quile::range r{ 0., .42 };
  assert(r.contains(.2));
  assert(!r.contains(-42.));
}
```

Result (might be empty):

7.23.3.3 max()

```
template<typename T >
T quile::range< T >::max ( ) const  [inline]
```

`range::max` returns range supremum.

Returns

Range supremum, i.e. right endpoint of interval $[\min, \max]_{\mathrm{T}}$ (`max` value) corresponding to the range.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  quile::range r{ 0, 42 };
  assert(r.max() == 42);
}
```

Result (might be empty):

7.23.3.4 midpoint()

```
template<typename T >
template<typename U = T, typename = std::enable_if_t<!std::is_same_v<U, bool>>>
T quile::range< T >::midpoint ( ) const  [inline]
```

range::midpoint returns midpoint of interval represented by range.

Returns

Range midpoint, i.e. center of interval $[\min, \max]_{\mathrm{T}}$ (arithmetic mean of `min` and `max` values) corresponding to the range. If `T` is of integer type and the sum of `min` and `max` is odd, the result is rounded towards `min`.

Note

This method is disabled for ranges of type `bool`.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  quile::range r{ 0, 42 };
  assert(r.midpoint() == 21);
}
```

Result (might be empty):

7.23.3.5 min()

```
template<typename T >
T quile::range< T >::min ( ) const  [inline]
```

range::min returns range infimum.

Returns

Range infimum, i.e. left endpoint of interval $[\min, \max]_{\mathrm{T}}$ (`min` value) corresponding to the range.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  quile::range r{ 0, 42 };
  assert(r.min() == 0);
}
```

Result (might be empty):

7.23.3.6 operator<=>()

```
template<typename T >
auto quile::range< T >::operator<=> (
            const range< T > & r ) const  [default]
```

range::operator<=> performs default lexicographical comparison with use of left and right endpoints.

Parameters

r	Range to be compared with `*this`.

Returns

Ordering (cf. `std::strong_ordering`, `std::weak_ordering`, `std::partial_ordering`).

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  quile::range r0{ 0, 2 };
  quile::range r1{ 1, 2 };
  assert(r0 < r1);
  assert(r0 <= r1);
  assert(r1 > r0);
  assert(r1 >= r0);
}
```

Result (might be empty):

7.23.3.7 operator=() [1/2]

```
template<typename T >
range& quile::range< T >::operator= (
            const range< T > &  )  [default]
```

Default assignment operator `range::operator=`.

7.23.3.8 operator=() [2/2]

```
template<typename T >
range& quile::range< T >::operator= (
            range< T > &&  )  [default]
```

Default move assignment operator `range::operator=`.

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

- quile/quile.h

7.24 quile::ranking_selection< G > Class Template Reference

`#include <quile/quile.h>`

Public Member Functions

- ranking_selection (const fitness_db< G > &ff, const probability_fn &pf)
- selection_probabilities operator() (const population< G > &p) const

7.24.1 Detailed Description

template<typename G>
class quile::ranking_selection< G >

`ranking_selection` is ranking selection (RS) mechanism.

Note

This implementation has workarounds for populations containing genotypes which fitnesses cannot be calculated. Please note that there should be at least one genotype, which fitness can be calculated.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout << p[i] << ":  " << sp[i] << ", " << cp[i] << '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
```

```
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

7.24.2 Constructor & Destructor Documentation

7.24.2.1 ranking_selection()

```
template<typename G >
quile::ranking_selection< G >::ranking_selection (
            const fitness_db< G > & ff,
            const probability_fn & pf )  [inline]
```

ranking_selection::ranking_selection constructor creates RS.

Parameters

ff	Fitness function database intermediary object.
pf	Selection pressure mechanism (linear or exponential).

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
```

```
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d»;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout « p[i] « ":  " « sp[i] « ", " « cp[i] « '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
```

```
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

7.24.3 Member Function Documentation

7.24.3.1 operator()()

```
template<typename G >
selection_probabilities quile::ranking_selection< G >::operator() (
            const population< G > & p ) const  [inline]
```

`ranking_selection::operator()` returns selection probabilities for population `p`.

Parameters

p	Population.

Returns

RS selection probabilities for population `p`.

Exceptions

std::runtime_error	Exception is raised if fitness function evaluates to `incalculable` for all genotypes from `p`.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
```

```
      std::cout « p[i] « ":  " « sp[i] « ", " « cp[i] « '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

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

- quile/quile.h

7.25 quile::roulette_wheel_selection< G > Class Template Reference

```
#include <quile/quile.h>
```

Public Member Functions

- roulette_wheel_selection (const selection_probabilities_fn< G > &spf)
- population< G > operator() (std::size_t lambda, const population< G > &p) const

7.25.1 Detailed Description

template<typename G>
class quile::roulette_wheel_selection< G >

`roulette_wheel_selection` is roulette wheel selection (a.k.a. roulette wheel *algorithm*, RWA).

Template Parameters

G	Some `genotype` specialization.

7.25.2 Constructor & Destructor Documentation

7.25.2.1 roulette_wheel_selection()

```
template<typename G >
quile::roulette_wheel_selection< G >::roulette_wheel_selection (
            const selection_probabilities_fn< G > & spf )  [inline], [explicit]
```

`roulette_wheel_selection::roulette_wheel_selection` constructor creates RWA with selection probability function `spf`.

Parameters

spf	Selection probability function.

7.25.3 Member Function Documentation

7.25.3.1 operator()()

```
template<typename G >
population<G> quile::roulette_wheel_selection< G >::operator() (
            std::size_t lambda,
            const population< G > & p ) const  [inline]
```

`roulette_wheel_selection::operator()` draws `lambda` genotypes from population `p` according to the RWA.

Parameters

lambda	Size of the returned population.
p	Source population.

Returns

Population consisting of genotypes drawn from `p`.

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

- quile/quile.h

7.26 quile::static_loop< T, I, N > Struct Template Reference

`#include <quile/quile.h>`

Static Public Member Functions

- static void body ([[maybe_unused]] auto &&f)

7.26.1 Detailed Description

template<std::integral T, T I, T N>
struct quile::static_loop< T, I, N >

`static_loop` is a `for`-loop replacement with loop index usable at compile time.

Template Parameters

T	Loop index type.
I	First index value (inclusive).
N	Last index value (exclusive).

7.26.2 Member Function Documentation

7.26.2.1 body()

```
template<std::integral T, T I, T N>
static void quile::static_loop< T, I, N >::body (
            [[maybe_unused] ] auto && f )  [inline], [static]
```

`static_loop::body` performs `f` in a loop iterating from `I` (inclusively) to `N` (exclusively), i.e. it replaces the loop of form `for (T i = I; i < N; ++i) { f(i); }`.

Parameters

f	Callable object to be invoked in loop. It must accept argument of type convertible from `T`, being the value of current loop index.

Example:

```
#include <iostream>
#include <quile/quile.h>
template<int N>
struct test
{
  constexpr static int value = N;
};
int
main()
{
  quile::static_loop<int, 0, 3>::body([=](auto I) {
    using T = test<I>;
    std::cout « T::value « std::endl;
  });
}
```

Result:

```
0
1
2
```

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

- quile/quile.h

7.27 quile::stochastic_universal_sampling< G > Class Template Reference

`#include <quile/quile.h>`

Public Member Functions

- stochastic_universal_sampling (const selection_probabilities_fn< G > &spf)
- population< G > operator() (std::size_t lambda, const population< G > &p) const

7.27.1 Detailed Description

template<typename G>
class quile::stochastic_universal_sampling< G >

`stochastic_universal_sampling` is stochastic universal sampling (SUS).

Template Parameters

G	Some `genotype` specialization.

Example:

```
#include <cassert>
#include <cmath>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc + std::abs(x); });
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const populate_0_fn<G> generator =
    random_population<constraints_satisfied<G>, G>;
  const populate_1_fn<G> parents_selection =
    stochastic_universal_sampling<G>{ fps };
  const populate_2_fn<G> selection_to_the_next_generation =
    adapter<G>(stochastic_universal_sampling<G>{ fps });
  const population<G> p0 = generator(42);
  const population<G> p1 = generator(42);
  const population<G> q0 = parents_selection(10, p0);
  assert(q0.size() == 10);
  const population<G> q1 = selection_to_the_next_generation(42, p0, p1);
  assert(q1.size() == 42);
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Fitness value for [8 2 5 7 3 1 1 1 5 3 8 9 5 9 5 8 1 9 4 3 0 0 8 1
8 2 1 0 7 8 3 1]: 136 (taken from database)
# Quile log: Fitness value for [5 7 9 0 5 3 6 0 6 8 4 4 0 7 1 1 8 3 8 6 9 7 8 1
9 8 8 1 9 8 5 5]: 169 (taken from database)
# Quile log: Fitness value for [9 3 2 1 9 2 4 2 3 3 9 2 7 1 8 5 0 6 2 1 9 7 9 7
2 7 1 2 5 7 9 5]: 149 (taken from database)
# Quile log: Fitness value for [8 5 8 3 6 9 7 1 7 3 8 7 6 2 1 8 3 8 6 3 5 2 2 6
7 9 8 2 6 7 3 5]: 171 (taken from database)
# Quile log: Fitness value for [2 2 6 9 6 7 3 5 5 0 6 6 0 3 3 6 2 8 0 1 2 5 7 9
1 5 7 9 8 6 2 5]: 146 (taken from database)
# Quile log: Fitness value for [7 9 1 8 7 7 5 2 2 6 1 1 9 2 0 2 5 7 1 9 7 0 7 4
0 3 7 8 0 4 3 1]: 135 (taken from database)
# Quile log: Fitness value for [2 0 1 0 1 6 4 4 6 9 1 6 3 8 3 7 6 8 9 8 4 5 6 6
0 6 1 0 0 4 9 7]: 140 (taken from database)
# Quile log: Fitness value for [0 5 6 7 7 1 2 8 8 4 5 0 4 2 3 1 3 1 6 1 5 3 8 9
2 2 7 4 5 4 4 9]: 136 (taken from database)
# Quile log: Fitness value for [6 2 3 0 2 7 8 6 8 8 3 8 1 7 3 8 2 1 6 4 8 3 1 2
8 3 8 2 1 9 1 6]: 145 (taken from database)
# Quile log: Fitness value for [3 8 8 3 0 4 1 3 8 7 2 0 7 3 9 1 8 8 2 4 4 5 7 0
4 3 3 5 6 5 0 5]: 136 (taken from database)
# Quile log: Fitness value for [0 9 3 9 4 0 2 8 2 0 5 9 3 7 2 8 7 0 9 6 6 0 5 6
2 8 8 9 3 4 8 1]: 153 (taken from database)
# Quile log: Fitness value for [1 7 2 7 3 7 4 9 5 5 8 3 4 3 6 4 6 6 8 8 4 1 9 9
0 4 5 2 5 3 1 1]: 150 (taken from database)
# Quile log: Fitness value for [7 7 4 8 2 3 9 2 2 3 7 6 1 6 0 5 7 8 7 5 9 5 7 6
0 5 9 6 6 6 7 1]: 166 (taken from database)
# Quile log: Fitness value for [9 4 6 2 4 0 9 7 6 1 3 9 5 7 2 3 1 7 0 9 1 3 6 8
6 3 8 9 8 0 4 6]: 156 (taken from database)
# Quile log: Fitness value for [3 1 6 7 2 7 4 1 6 9 6 3 9 5 9 7 2 6 2 0 8 7 1 9
7 9 2 5 6 8 9 9]: 175 (taken from database)
# Quile log: Fitness value for [0 1 4 9 2 2 8 7 6 9 7 9 9 5 2 4 0 7 9 7 1 2 7 9
2 4 0 5 8 2 7 0]: 154 (taken from database)
# Quile log: Fitness value for [6 8 9 7 3 9 4 5 3 5 0 1 7 9 6 9 6 3 1 6 1 6 4 8
0 0 1 2 7 0 4 8]: 148 (taken from database)
# Quile log: Fitness value for [6 0 1 9 5 6 8 4 2 5 1 8 9 6 8 3 2 7 1 3 3 4 1 4
2 3 2 6 3 5 1 2]: 130 (taken from database)
# Quile log: Fitness value for [9 7 2 3 1 3 9 9 0 2 7 6 2 0 8 2 3 9 6 7 4 5 2 6
8 2 0 9 0 2 7 1]: 141 (taken from database)
```

```
# Quile log: Fitness value for [3 4 0 7 3 0 8 0 3 5 8 1 7 6 0 4 3 8 5 0 0 0 1 1
6 8 3 2 4 2 3 0]: 105 (taken from database)
# Quile log: Fitness value for [7 8 1 4 1 8 2 1 4 7 6 9 5 3 7 3 7 1 4 3 3 3 8 2
7 8 6 1 3 5 4 1]: 142 (taken from database)
# Quile log: Fitness value for [8 5 2 6 1 2 8 6 8 9 2 9 7 5 7 1 4 9 3 5 8 5 8 1
1 9 8 2 4 1 1 2]: 157 (taken from database)
# Quile log: Fitness value for [7 4 8 5 9 7 5 4 5 2 2 3 5 4 6 4 0 2 6 2 2 5 4 1
5 1 3 7 4 2 5 4]: 133 (taken from database)
# Quile log: Fitness value for [6 3 1 8 2 3 3 4 1 4 4 7 0 3 6 1 0 0 7 0 7 7 2 2
4 1 1 4 4 3 2 4]: 104 (taken from database)
# Quile log: Fitness value for [8 7 8 5 8 1 9 4 8 5 0 4 0 1 8 1 8 1 5 1 1 3 3 8
3 7 9 7 5 6 6 7]: 157 (taken from database)
```

7.27.2 Constructor & Destructor Documentation

7.27.2.1 stochastic_universal_sampling()

```
template<typename G >
quile::stochastic_universal_sampling< G >::stochastic_universal_sampling (
            const selection_probabilities_fn< G > & spf )  [inline], [explicit]
```

stochastic_universal_sampling::stochastic_universal_sampling constructor creates SUS with selection probability function `spf`.

Parameters

spf	Selection probability function.

Example:

```
#include <cassert>
#include <cmath>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc + std::abs(x); });
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const populate_0_fn<G> generator =
    random_population<constraints_satisfied<G>, G>;
  const populate_1_fn<G> parents_selection =
    stochastic_universal_sampling<G>{ fps };
  const populate_2_fn<G> selection_to_the_next_generation =
    adapter<G>(stochastic_universal_sampling<G>{ fps });
  const population<G> p0 = generator(42);
  const population<G> p1 = generator(42);
  const population<G> q0 = parents_selection(10, p0);
  assert(q0.size() == 10);
  const population<G> q1 = selection_to_the_next_generation(42, p0, p1);
  assert(q1.size() == 42);
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Fitness value for [8 2 5 7 3 1 1 1 5 3 8 9 5 9 5 8 1 9 4 3 0 0 8 1
8 2 1 0 7 8 3 1]: 136 (taken from database)
# Quile log: Fitness value for [5 7 9 0 5 3 6 0 6 8 4 4 0 7 1 1 8 3 8 6 9 7 8 1
9 8 8 1 9 8 5 5]: 169 (taken from database)
# Quile log: Fitness value for [9 3 2 1 9 2 4 2 3 3 9 2 7 1 8 5 0 6 2 1 9 7 9 7
2 7 1 2 5 7 9 5]: 149 (taken from database)
# Quile log: Fitness value for [8 5 8 3 6 9 7 1 7 3 8 7 6 2 1 8 3 8 6 3 5 2 2 6
7 9 8 2 6 7 3 5]: 171 (taken from database)
# Quile log: Fitness value for [2 2 6 9 6 7 3 5 5 0 6 6 0 3 3 6 2 8 0 1 2 5 7 9
1 5 7 9 8 6 2 5]: 146 (taken from database)
# Quile log: Fitness value for [7 9 1 8 7 7 5 2 2 6 1 1 9 2 0 2 5 7 1 9 7 0 7 4
0 3 7 8 0 4 3 1]: 135 (taken from database)
# Quile log: Fitness value for [2 0 1 0 1 6 4 4 6 9 1 6 3 8 3 7 6 8 9 8 4 5 6 6
0 6 1 0 0 4 9 7]: 140 (taken from database)
# Quile log: Fitness value for [0 5 6 7 7 1 2 8 8 4 5 0 4 2 3 1 3 1 6 1 5 3 8 9
2 2 7 4 5 4 4 9]: 136 (taken from database)
# Quile log: Fitness value for [6 2 3 0 2 7 8 6 8 8 3 8 1 7 3 8 2 1 6 4 8 3 1 2
8 3 8 2 1 9 1 6]: 145 (taken from database)
# Quile log: Fitness value for [3 8 8 3 0 4 1 3 8 7 2 0 7 3 9 1 8 8 2 4 4 5 7 0
4 3 3 5 6 5 0 5]: 136 (taken from database)
# Quile log: Fitness value for [0 9 3 9 4 0 2 8 2 0 5 9 3 7 2 8 7 0 9 6 6 0 5 6
2 8 8 9 3 4 8 1]: 153 (taken from database)
# Quile log: Fitness value for [1 7 2 7 3 7 4 9 5 5 8 3 4 3 6 4 6 6 8 8 4 1 9 9
0 4 5 2 5 3 1 1]: 150 (taken from database)
# Quile log: Fitness value for [7 7 4 8 2 3 9 2 2 3 7 6 1 6 0 5 7 8 7 5 9 5 7 6
0 5 9 6 6 6 7 1]: 166 (taken from database)
# Quile log: Fitness value for [9 4 6 2 4 0 9 7 6 1 3 9 5 7 2 3 1 7 0 9 1 3 6 8
6 3 8 9 8 0 4 6]: 156 (taken from database)
# Quile log: Fitness value for [3 1 6 7 2 7 4 1 6 9 6 3 9 5 9 7 2 6 2 0 8 7 1 9
7 9 2 5 6 8 9 9]: 175 (taken from database)
# Quile log: Fitness value for [0 1 4 9 2 2 8 7 6 9 7 9 9 5 2 4 0 7 9 7 1 2 7 9
2 4 0 5 8 2 7 0]: 154 (taken from database)
# Quile log: Fitness value for [6 8 9 7 3 9 4 5 3 5 0 1 7 9 6 9 6 3 1 6 1 6 4 8
0 0 1 2 7 0 4 8]: 148 (taken from database)
# Quile log: Fitness value for [6 0 1 9 5 6 8 4 2 5 1 8 9 6 8 3 2 7 1 3 3 4 1 4
2 3 2 6 3 5 1 2]: 130 (taken from database)
# Quile log: Fitness value for [9 7 2 3 1 3 9 9 0 2 7 6 2 0 8 2 3 9 6 7 4 5 2 6
8 2 0 9 0 2 7 1]: 141 (taken from database)
# Quile log: Fitness value for [3 4 0 7 3 0 8 0 3 5 8 1 7 6 0 4 3 8 5 0 0 0 1 1
6 8 3 2 4 2 3 0]: 105 (taken from database)
# Quile log: Fitness value for [7 8 1 4 1 8 2 1 4 7 6 9 5 3 7 3 7 1 4 3 3 3 8 2
7 8 6 1 3 5 4 1]: 142 (taken from database)
# Quile log: Fitness value for [8 5 2 6 1 2 8 6 8 9 2 9 7 5 7 1 4 9 3 5 8 5 8 1
1 9 8 2 4 1 1 2]: 157 (taken from database)
# Quile log: Fitness value for [7 4 8 5 9 7 5 4 5 2 2 3 5 4 6 4 0 2 6 2 2 5 4 1
5 1 3 7 4 2 5 4]: 133 (taken from database)
# Quile log: Fitness value for [6 3 1 8 2 3 3 4 1 4 4 7 0 3 6 1 0 0 7 0 7 7 2 2
4 1 1 4 4 3 2 4]: 104 (taken from database)
# Quile log: Fitness value for [8 7 8 5 8 1 9 4 8 5 0 4 0 1 8 1 8 1 5 1 1 3 3 8
3 7 9 7 5 6 6 7]: 157 (taken from database)
```

7.27.3 Member Function Documentation

7.27.3.1 operator()()

```
template<typename G >
population<G> quile::stochastic_universal_sampling< G >::operator() (
            std::size_t lambda,
            const population< G > & p ) const  [inline]
```

`stochastic_universal_sampling::operator()` draws `lambda` genotypes from population `p` according to the SUS.

Parameters

lambda	Size of the returned population.
p	Source population.

Returns

Population consisting of genotypes drawn from p.

Example:

```
#include <cassert>
#include <cmath>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc + std::abs(x); });
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const populate_0_fn<G> generator =
    random_population<constraints_satisfied<G>, G>;
  const populate_1_fn<G> parents_selection =
    stochastic_universal_sampling<G>{ fps };
  const populate_2_fn<G> selection_to_the_next_generation =
    adapter<G>(stochastic_universal_sampling<G>{ fps });
  const population<G> p0 = generator(42);
  const population<G> p1 = generator(42);
  const population<G> q0 = parents_selection(10, p0);
  assert(q0.size() == 10);
  const population<G> q1 = selection_to_the_next_generation(42, p0, p1);
  assert(q1.size() == 42);
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Fitness value for [8 2 5 7 3 1 1 1 5 3 8 9 5 9 5 8 1 9 4 3 0 0 8 1
8 2 1 0 7 8 3 1]: 136 (taken from database)
# Quile log: Fitness value for [5 7 9 0 5 3 6 0 6 8 4 4 0 7 1 1 8 3 8 6 9 7 8 1
9 8 8 1 9 8 5 5]: 169 (taken from database)
# Quile log: Fitness value for [9 3 2 1 9 2 4 2 3 3 9 2 7 1 8 5 0 6 2 1 9 7 9 7
2 7 1 2 5 7 9 5]: 149 (taken from database)
# Quile log: Fitness value for [8 5 8 3 6 9 7 1 7 3 8 7 6 2 1 8 3 8 6 3 5 2 2 6
7 9 8 2 6 7 3 5]: 171 (taken from database)
# Quile log: Fitness value for [2 2 6 9 6 7 3 5 5 0 6 6 0 3 3 6 2 8 0 1 2 5 7 9
1 5 7 9 8 6 2 5]: 146 (taken from database)
# Quile log: Fitness value for [7 9 1 8 7 7 5 2 2 6 1 1 9 2 0 2 5 7 1 9 7 0 7 4
0 3 7 8 0 4 3 1]: 135 (taken from database)
# Quile log: Fitness value for [2 0 1 0 1 6 4 4 6 9 1 6 3 8 3 7 6 8 9 8 4 5 6 6
0 6 1 0 0 4 9 7]: 140 (taken from database)
# Quile log: Fitness value for [0 5 6 7 7 1 2 8 8 4 5 0 4 2 3 1 3 1 6 1 5 3 8 9
2 2 7 4 5 4 4 9]: 136 (taken from database)
# Quile log: Fitness value for [6 2 3 0 2 7 8 6 8 8 3 8 1 7 3 8 2 1 6 4 8 3 1 2
8 3 8 2 1 9 1 6]: 145 (taken from database)
# Quile log: Fitness value for [3 8 8 3 0 4 1 3 8 7 2 0 7 3 9 1 8 8 2 4 4 5 7 0
4 3 3 5 6 5 0 5]: 136 (taken from database)
# Quile log: Fitness value for [0 9 3 9 4 0 2 8 2 0 5 9 3 7 2 8 7 0 9 6 6 0 5 6
2 8 8 9 3 4 8 1]: 153 (taken from database)
# Quile log: Fitness value for [1 7 2 7 3 7 4 9 5 5 8 3 4 3 6 4 6 6 8 8 4 1 9 9
0 4 5 2 5 3 1 1]: 150 (taken from database)
```

```
# Quile log: Fitness value for [7 7 4 8 2 3 9 2 2 3 7 6 1 6 0 5 7 8 7 5 9 5 7 6
0 5 9 6 6 6 7 1]: 166 (taken from database)
# Quile log: Fitness value for [9 4 6 2 4 0 9 7 6 1 3 9 5 7 2 3 1 7 0 9 1 3 6 8
6 3 8 9 8 0 4 6]: 156 (taken from database)
# Quile log: Fitness value for [3 1 6 7 2 7 4 1 6 9 6 3 9 5 9 7 2 6 2 0 8 7 1 9
7 9 2 5 6 8 9 9]: 175 (taken from database)
# Quile log: Fitness value for [0 1 4 9 2 2 8 7 6 9 7 9 9 5 2 4 0 7 9 7 1 2 7 9
2 4 0 5 8 2 7 0]: 154 (taken from database)
# Quile log: Fitness value for [6 8 9 7 3 9 4 5 3 5 0 1 7 9 6 9 6 3 1 6 1 6 4 8
0 0 1 2 7 0 4 8]: 148 (taken from database)
# Quile log: Fitness value for [6 0 1 9 5 6 8 4 2 5 1 8 9 6 8 3 2 7 1 3 3 4 1 4
2 3 2 6 3 5 1 2]: 130 (taken from database)
# Quile log: Fitness value for [9 7 2 3 1 3 9 9 0 2 7 6 2 0 8 2 3 9 6 7 4 5 2 6
8 2 0 9 0 2 7 1]: 141 (taken from database)
# Quile log: Fitness value for [3 4 0 7 3 0 8 0 3 5 8 1 7 6 0 4 3 8 5 0 0 0 1 1
6 8 3 2 4 2 3 0]: 105 (taken from database)
# Quile log: Fitness value for [7 8 1 4 1 8 2 1 4 7 6 9 5 3 7 3 7 1 4 3 3 3 8 2
7 8 6 1 3 5 4 1]: 142 (taken from database)
# Quile log: Fitness value for [8 5 2 6 1 2 8 6 8 9 2 9 7 5 7 1 4 9 3 5 8 5 8 1
1 9 8 2 4 1 1 2]: 157 (taken from database)
# Quile log: Fitness value for [7 4 8 5 9 7 5 4 5 2 2 3 5 4 6 4 0 2 6 2 2 5 4 1
5 1 3 7 4 2 5 4]: 133 (taken from database)
# Quile log: Fitness value for [6 3 1 8 2 3 3 4 1 4 4 7 0 3 6 1 0 0 7 0 7 7 2 2
4 1 1 4 4 3 2 4]: 104 (taken from database)
# Quile log: Fitness value for [8 7 8 5 8 1 9 4 8 5 0 4 0 1 8 1 8 1 5 1 1 3 3 8
3 7 9 7 5 6 6 7]: 157 (taken from database)
```

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

- quile/quile.h

7.28 quile::test_functions::test_function< T, N > Class Template Reference

`#include <quile/quile.h>`

Public Types

- using function = std::function< T(const point< T, N > &)>
- using domain_fn = std::function< domain< T, N >()>
- using point_fn = std::function< point< T, N >()>

Public Member Functions

- test_function (const std::string &name, const function &fn, const domain_fn &d, const point_fn &p_min)
- std::string name () const
- T operator() (const point< T, N > &p) const
- domain< T, N > function_domain () const
- point< T, N > p_min () const

7.28.1 Detailed Description

template<std::floating_point T, std::size_t N>
class quile::test_functions::test_function< T, N >

`test_functions::test_function` is floating-point test function.

Template Parameters

T	Floating-point type.
N	Space dimension.

7.28.2 Member Typedef Documentation

7.28.2.1 domain_fn

```
template<std::floating_point T, std::size_t N>
using quile::test_functions::test_function< T, N >::domain_fn = std::function<domain<T, N>()>
```

test_functions::test_function::domain_fn is test function domain.

7.28.2.2 function

```
template<std::floating_point T, std::size_t N>
using quile::test_functions::test_function< T, N >::function = std::function<T(const point<T,
N>&)>
```

test_functions::test_function::function is underlying test function type.

7.28.2.3 point_fn

```
template<std::floating_point T, std::size_t N>
using quile::test_functions::test_function< T, N >::point_fn = std::function<point<T, N>()>
```

test_functions::test_function::point_fn is solution generating function type.

7.28.3 Constructor & Destructor Documentation

7.28.3.1 test_function()

```
template<std::floating_point T, std::size_t N>
quile::test_functions::test_function< T, N >::test_function (
            const std::string & name,
            const function & fn,
            const domain_fn & d,
            const point_fn & p_min )  [inline]
```

test_functions::test_function::test_function creates test function.

Parameters

name	Test function name.
fn	The test function representation.
d	The test function domain.
p_min	Function generating solution minimizing the test function.

7.28.4 Member Function Documentation

7.28.4.1 function_domain()

```
template<std::floating_point T, std::size_t N>
domain<T, N> quile::test_functions::test_function< T, N >::function_domain ( ) const  [inline]
```

`test_functions::test_function::function_domain` returns test function domain.

Returns

Test function domain.

7.28.4.2 name()

```
template<std::floating_point T, std::size_t N>
std::string quile::test_functions::test_function< T, N >::name ( ) const  [inline]
```

`test_functions::test_function::name` returns test function name.

Returns

Name of the test function.

7.28.4.3 operator()()

```
template<std::floating_point T, std::size_t N>
T quile::test_functions::test_function< T, N >::operator() (
            const point< T, N > & p ) const  [inline]
```

`test_functions::test_function::operator()` returns test function value at point `p`.

Parameters

p	Point.

Returns

Test function value at point p.

7.28.4.4 p_min()

```
template<std::floating_point T, std::size_t N>
point<T, N> quile::test_functions::test_function< T, N >::p_min ( ) const  [inline]
```

test_functions::test_function::p_min returns point minimizing test function over its domain.

Returns

Point minimizing test function over its domain.

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

- quile/quile.h

7.29 quile::thread_pool Class Reference

```
#include <quile/quile.h>
```

Public Member Functions

- thread_pool (std::size_t sz)
- template<typename T >
  std::future< T > async (std::launch policy, const std::function< T()> &f)

7.29.1 Detailed Description

thread_pool implements modified thread pool design pattern.

7.29.2 Constructor & Destructor Documentation

7.29.2.1 thread_pool()

```
quile::thread_pool::thread_pool (
            std::size_t sz )  [inline], [explicit]
```

thread_pool constructor.

Parameters

sz	Number of threads for concurrent calculations.

7.29.3 Member Function Documentation

7.29.3.1 async()

```
template<typename T >
std::future<T> quile::thread_pool::async (
            std::launch policy,
            const std::function< T()> & f )  [inline]
```

`thread_pool::async` asynchronically executes callable object `f` postponing start of `f` until number of concurrently executing threads in pool drops below number `sz`, described in constructor.

Parameters

policy	Lauch policy (see `std::launch` documentation).
f	Callable object to be concurrently executed.

Example:

```
#include <chrono>
#include <iostream>
#include <quile/quile.h>
#include <thread>
#include <vector>
int
main()
{
  using namespace std::chrono_literals;
  quile::thread_pool tp{ 4 };
  std::vector<std::future<void» v0{};
  for (int i = 0; i < 10; ++i) {
    v0.push_back(tp.async<void>(std::launch::async, [i]() {
      std::cout « i « '\n';
      std::this_thread::sleep_for((i % 2 + 1) * 1s);
    }));
  }
  for (auto& x :  v0) {
    x.get();
  }
}
```

Result (might be different due to concurrent execution):

```
2
3
4
5
7
6
8
1
9
0
```

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

- quile/quile.h

7.30 quile::variation< G > Class Template Reference

`#include <quile/quile.h>`

Public Member Functions

- variation (const mutation_fn< G > &m, const recombination_fn< G > &r)
- variation ()
- variation (const mutation_fn< G > &m)
- variation (const recombination_fn< G > &r)
- population< G > operator() (const G &g0, const G &g1) const
- population< G > operator() (const population< G > &p) const

7.30.1 Detailed Description

template<typename G>
class quile::variation< G >

`variation` represents variation operator.

Template Parameters

G	Some `genotype` specialization.

Note

At the moment library supports only canonical forms of variations (unary and binary).

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const variation<G> v0{};
  const auto g0 = G::random();
  const auto g1 = G::random();
  const population<G> p0 = v0(g0, g1);
  assert(p0[0] == g0 && p0[1] == g1);
  const variation<G> v1{ swap_mutation<G> };
  const auto g2 = G::random();
  const auto g3 = G::random();
  const auto p1 = v1(population<G>{ g0, g1, g2, g3 });
  assert(p1.size() == 4);
  std::cout « p1[0] « '\n';
  std::cout « p1[1] « '\n';
  std::cout « p1[2] « '\n';
  std::cout « p1[3] « '\n';
  const variation<G> v2{ cut_n_crossfill<G> };
  try {
    [[maybe_unused]] const auto p2 = v2(population<G>{ g0, g1, g2 });
  } catch (...)  {
    std::cout « "Even number of genotypes is required.\n";
  }
  const variation<G> v3{ swap_mutation<G>, cut_n_crossfill<G> };
  const auto p3 = v3(g0, g1);
  std::cout « p3[0] « '\n';
  std::cout « p3[1] « '\n';
}
```

Result (might be different due to randomness):

```
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 5 0 1 6 3 4 2, 3 1 6 0 5 2 4
4 5 6 3 0 2 1
1 0 4 5 2 6 3
5 0 1 4 3 6 2
4 1 6 0 5 2 3
Even number of genotypes is required.
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
1 5 0 3 4 2 6
0 1 4 5 3 6 2
```

7.30.2 Constructor & Destructor Documentation

7.30.2.1 variation() [1/4]

```
template<typename G >
quile::variation< G >::variation (
            const mutation_fn< G > & m,
            const recombination_fn< G > & r )  [inline]
```

variation::variation constructor creates object representing variation consisting of recombination r with mutation m applied separately to each child coming from r.

Parameters

m	Mutation.
r	Recombination.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const variation<G> v0{};
  const auto g0 = G::random();
  const auto g1 = G::random();
  const population<G> p0 = v0(g0, g1);
  assert(p0[0] == g0 && p0[1] == g1);
  const variation<G> v1{ swap_mutation<G> };
  const auto g2 = G::random();
  const auto g3 = G::random();
  const auto p1 = v1(population<G>{ g0, g1, g2, g3 });
  assert(p1.size() == 4);
  std::cout « p1[0] « '\n';
  std::cout « p1[1] « '\n';
  std::cout « p1[2] « '\n';
  std::cout « p1[3] « '\n';
  const variation<G> v2{ cut_n_crossfill<G> };
  try {
    [[maybe_unused]] const auto p2 = v2(population<G>{ g0, g1, g2 });
  } catch (...)  {
    std::cout « "Even number of genotypes is required.\n";
  }
  const variation<G> v3{ swap_mutation<G>, cut_n_crossfill<G> };
  const auto p3 = v3(g0, g1);
  std::cout « p3[0] « '\n';
  std::cout « p3[1] « '\n';
}
```

Result (might be different due to randomness):

```
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 5 0 1 6 3 4 2, 3 1 6 0 5 2 4
4 5 6 3 0 2 1
1 0 4 5 2 6 3
5 0 1 4 3 6 2
4 1 6 0 5 2 3
Even number of genotypes is required.
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
1 5 0 3 4 2 6
0 1 4 5 3 6 2
```

7.30.2.2 variation() [2/4]

```
template<typename G >
quile::variation< G >::variation ( )  [inline]
```

variation::variation constructor creates identity variation.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const variation<G> v0{};
  const auto g0 = G::random();
  const auto g1 = G::random();
  const population<G> p0 = v0(g0, g1);
  assert(p0[0] == g0 && p0[1] == g1);
  const variation<G> v1{ swap_mutation<G> };
  const auto g2 = G::random();
  const auto g3 = G::random();
  const auto p1 = v1(population<G>{ g0, g1, g2, g3 });
  assert(p1.size() == 4);
  std::cout « p1[0] « '\n';
  std::cout « p1[1] « '\n';
  std::cout « p1[2] « '\n';
  std::cout « p1[3] « '\n';
  const variation<G> v2{ cut_n_crossfill<G> };
  try {
    [[maybe_unused]] const auto p2 = v2(population<G>{ g0, g1, g2 });
  } catch (...)  {
    std::cout « "Even number of genotypes is required.\n";
  }
  const variation<G> v3{ swap_mutation<G>, cut_n_crossfill<G> };
  const auto p3 = v3(g0, g1);
  std::cout « p3[0] « '\n';
  std::cout « p3[1] « '\n';
}
```

Result (might be different due to randomness):

```
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 5 0 1 6 3 4 2, 3 1 6 0 5 2 4
4 5 6 3 0 2 1
1 0 4 5 2 6 3
5 0 1 4 3 6 2
4 1 6 0 5 2 3
Even number of genotypes is required.
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
1 5 0 3 4 2 6
0 1 4 5 3 6 2
```

7.30.2.3 variation() [3/4]

```
template<typename G >
quile::variation< G >::variation (
            const mutation_fn< G > & m )  [inline], [explicit]
```

variation::variation constructor creates variation equal to mutation m.

Parameters

m	Mutation.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const variation<G> v0{};
  const auto g0 = G::random();
  const auto g1 = G::random();
  const population<G> p0 = v0(g0, g1);
  assert(p0[0] == g0 && p0[1] == g1);
  const variation<G> v1{ swap_mutation<G> };
  const auto g2 = G::random();
  const auto g3 = G::random();
  const auto p1 = v1(population<G>{ g0, g1, g2, g3 });
  assert(p1.size() == 4);
  std::cout « p1[0] « '\n';
  std::cout « p1[1] « '\n';
  std::cout « p1[2] « '\n';
  std::cout « p1[3] « '\n';
  const variation<G> v2{ cut_n_crossfill<G> };
  try {
    [[maybe_unused]] const auto p2 = v2(population<G>{ g0, g1, g2 });
  } catch (...)  {
    std::cout « "Even number of genotypes is required.\n";
  }
  const variation<G> v3{ swap_mutation<G>, cut_n_crossfill<G> };
  const auto p3 = v3(g0, g1);
  std::cout « p3[0] « '\n';
  std::cout « p3[1] « '\n';
}
```

Result (might be different due to randomness):

```
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 5 0 1 6 3 4 2, 3 1 6 0 5 2 4
4 5 6 3 0 2 1
1 0 4 5 2 6 3
5 0 1 4 3 6 2
4 1 6 0 5 2 3
Even number of genotypes is required.
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
1 5 0 3 4 2 6
0 1 4 5 3 6 2
```

7.30.2.4 variation() [4/4]

```
template<typename G >
quile::variation< G >::variation (
            const recombination_fn< G > & r )  [inline], [explicit]
```

variation::variation constructor creates variation equal to recombination r.

Parameters

r	Recombination.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const variation<G> v0{};
  const auto g0 = G::random();
  const auto g1 = G::random();
  const population<G> p0 = v0(g0, g1);
  assert(p0[0] == g0 && p0[1] == g1);
  const variation<G> v1{ swap_mutation<G> };
  const auto g2 = G::random();
  const auto g3 = G::random();
  const auto p1 = v1(population<G>{ g0, g1, g2, g3 });
  assert(p1.size() == 4);
  std::cout « p1[0] « '\n';
  std::cout « p1[1] « '\n';
  std::cout « p1[2] « '\n';
  std::cout « p1[3] « '\n';
  const variation<G> v2{ cut_n_crossfill<G> };
  try {
    [[maybe_unused]] const auto p2 = v2(population<G>{ g0, g1, g2 });
  } catch (...)  {
    std::cout « "Even number of genotypes is required.\n";
  }
  const variation<G> v3{ swap_mutation<G>, cut_n_crossfill<G> };
  const auto p3 = v3(g0, g1);
  std::cout « p3[0] « '\n';
  std::cout « p3[1] « '\n';
}
```

Result (might be different due to randomness):

```
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 5 0 1 6 3 4 2, 3 1 6 0 5 2 4
4 5 6 3 0 2 1
1 0 4 5 2 6 3
5 0 1 4 3 6 2
4 1 6 0 5 2 3
Even number of genotypes is required.
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
1 5 0 3 4 2 6
0 1 4 5 3 6 2
```

7.30.3 Member Function Documentation

7.30.3.1 operator()() [1/2]

```
template<typename G >
population<G> quile::variation< G >::operator() (
            const G & g0,
            const G & g1 ) const  [inline]
```

`variation::operator()` applies variation to genotypes `g0` and `g1`.

Parameters

g0	Genotype.
g1	Genotype.

Returns

Population resulting from application of variation to genotypes.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const variation<G> v0{};
  const auto g0 = G::random();
  const auto g1 = G::random();
  const population<G> p0 = v0(g0, g1);
  assert(p0[0] == g0 && p0[1] == g1);
  const variation<G> v1{ swap_mutation<G> };
  const auto g2 = G::random();
  const auto g3 = G::random();
  const auto p1 = v1(population<G>{ g0, g1, g2, g3 });
  assert(p1.size() == 4);
  std::cout « p1[0] « '\n';
  std::cout « p1[1] « '\n';
  std::cout « p1[2] « '\n';
  std::cout « p1[3] « '\n';
  const variation<G> v2{ cut_n_crossfill<G> };
  try {
    [[maybe_unused]] const auto p2 = v2(population<G>{ g0, g1, g2 });
  } catch (...)  {
    std::cout « "Even number of genotypes is required.\n";
  }
  const variation<G> v3{ swap_mutation<G>, cut_n_crossfill<G> };
  const auto p3 = v3(g0, g1);
  std::cout « p3[0] « '\n';
  std::cout « p3[1] « '\n';
}
```

Result (might be different due to randomness):

```
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 5 0 1 6 3 4 2, 3 1 6 0 5 2 4
4 5 6 3 0 2 1
1 0 4 5 2 6 3
5 0 1 4 3 6 2
4 1 6 0 5 2 3
Even number of genotypes is required.
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
1 5 0 3 4 2 6
0 1 4 5 3 6 2
```

7.30.3.2 operator()() [2/2]

```
template<typename G >
population<G> quile::variation< G >::operator() (
            const population< G > & p ) const  [inline]
```

`variation::operator()` applies variation to consecutive pairs of genotypes in population `p`.

Parameters

p	Population consisting of pairs of parents.

Returns

Populative consisting of cumulative offspring.

Exceptions

std::invalid_argument	Exception is raised if population size is odd.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const variation<G> v0{};
  const auto g0 = G::random();
  const auto g1 = G::random();
  const population<G> p0 = v0(g0, g1);
  assert(p0[0] == g0 && p0[1] == g1);
  const variation<G> v1{ swap_mutation<G> };
  const auto g2 = G::random();
  const auto g3 = G::random();
  const auto p1 = v1(population<G>{ g0, g1, g2, g3 });
  assert(p1.size() == 4);
  std::cout « p1[0] « '\n';
  std::cout « p1[1] « '\n';
  std::cout « p1[2] « '\n';
  std::cout « p1[3] « '\n';
  const variation<G> v2{ cut_n_crossfill<G> };
  try {
    [[maybe_unused]] const auto p2 = v2(population<G>{ g0, g1, g2 });
  } catch (...)  {
    std::cout « "Even number of genotypes is required.\n";
  }
  const variation<G> v3{ swap_mutation<G>, cut_n_crossfill<G> };
  const auto p3 = v3(g0, g1);
  std::cout « p3[0] « '\n';
  std::cout « p3[1] « '\n';
}
```

Result (might be different due to randomness):

```
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 5 0 1 6 3 4 2, 3 1 6 0 5 2 4
4 5 6 3 0 2 1
1 0 4 5 2 6 3
5 0 1 4 3 6 2
4 1 6 0 5 2 3
Even number of genotypes is required.
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
1 5 0 3 4 2 6
0 1 4 5 3 6 2
```

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

- quile/quile.h

Chapter 8

File Documentation

8.1 quile/quile.h File Reference

```
#include <algorithm>
#include <array>
#include <cassert>
#include <climits>
#include <cmath>
#include <concepts>
#include <condition_variable>
#include <cstddef>
#include <deque>
#include <functional>
#include <future>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <limits>
#include <map>
#include <memory>
#include <mutex>
#include <numbers>
#include <numeric>
#include <random>
#include <ranges>
#include <stdexcept>
#include <thread>
#include <tuple>
#include <type_traits>
#include <unordered_set>
#include <utility>
#include <vector>
```

Include dependency graph for quile.h:

Classes

- struct quile::static_loop< T, I, N >
- class quile::thread_pool
- class quile::range< T >
- struct quile::is_domain< T >
- struct quile::is_domain< domain< T, N > >
- struct quile::g_floating_point< T, N, D >
- struct quile::is_g_floating_point< T >
- struct quile::is_g_floating_point< g_floating_point< T, N, D > >
- struct quile::g_integer< T, N, D >
- struct quile::is_g_integer< T >
- struct quile::is_g_integer< g_integer< T, N, D > >
- struct quile::g_binary< N >
- struct quile::is_g_binary< T >
- struct quile::is_g_binary< g_binary< N > >
- struct quile::g_permutation< T, N, M >
- struct quile::is_g_permutation< T >
- struct quile::is_g_permutation< g_permutation< T, N, M > >
- class quile::genotype< R >
- struct quile::is_genotype< T >
- struct quile::is_genotype< genotype< T > >
- struct std::hash< G >
- struct quile::is_population< T >
- struct quile::is_population< population< G > >
- class quile::variation< G >
- class quile::fitness_db< G >
- class quile::fitness_proportional_selection< G >
- class quile::ranking_selection< G >
- class quile::roulette_wheel_selection< G >
- class quile::stochastic_universal_sampling< G >
- class quile::test_functions::test_function< T, N >

Namespaces

- quile::detail
- std
- quile::test_functions

Macros

- #define QUILE_LOG(x)

Typedefs

- using quile::probability = double
- template<typename T , std::size_t N>
  using quile::domain = std::array< range< T >, N >
- template<typename T , std::size_t N>
  using quile::chain = std::array< T, N >
- template<typename G >
  using quile::population = std::vector< G >
- template<typename G >
  using quile::populate_0_fn = std::function< population< G >(std::size_t)>
- template<typename G >
  using quile::populate_1_fn = std::function< population< G >(std::size_t, const population< G > &)>
- template<typename G >
  using quile::populate_2_fn = std::function< population< G >(std::size_t, const population< G > &, const population< G > &)>
- template<typename G >
  using quile::generations = std::deque< population< G > >
- template<typename G >
  using quile::mutation_fn = std::function< population< G >(const G &)>
- template<typename G >
  using quile::recombination_fn = std::function< population< G >(const G &, const G &)>
- template<typename G >
  using quile::termination_condition_fn = std::function< bool(std::size_t, const generations< G > &)>
- using quile::fitness = double
- using quile::fitnesses = std::vector< fitness >
- template<typename G >
  using quile::fitness_function = std::function< fitness(const G &)>
- using quile::selection_probabilities = std::vector< probability >
- template<typename G >
  using quile::selection_probabilities_fn = std::function< selection_probabilities(const population< G > &)>
- template<std::floating_point T, std::size_t N>
  using quile::test_functions::point = std::array< T, N >

Functions

- template<std::integral T, T I, T N>
  quile::requires (I< N) struct static_loop< T
- auto quile::fn_and (const auto &... fs)
- auto quile::fn_or (const auto &... fs)
- template<typename T >
  std::ostream & quile::operator<< (std::ostream &os, const range< T > &r)
- template<typename T , std::size_t N>
  std::array< T, N > quile::iota (T t)
- std::mt19937 & quile::random_engine ()
- bool quile::success (probability success_probability)
- template<std::floating_point T>
  T quile::random_N (T mean, T standard_deviation)
- template<typename T >
  T quile::random_U (T a, T b)
- template<typename T >
  requires std::floating_point< T > std::integral< T > T quile::square (T x)
- template<typename T >
  requires std::floating_point< T > std::integral< T > T quile::cube (T x)

- template<std::floating_point T>
  T quile::detail::angle (T x, T y)
- template<std::floating_point T>
  std::tuple< T, T, T > quile::cart2spher (T x, T y, T z)
- template<std::floating_point T>
  std::tuple< T, T, T > quile::spher2cart (T r, T theta, T phi)
- template<std::floating_point T>
  std::tuple< T, T > quile::cart2polar (T x, T y)
- template<std::floating_point T>
  std::tuple< T, T > quile::polar2cart (T r, T phi)
- template<typename T , std::size_t N>
  bool quile::contains (const domain< T, N > &d, const std::array< T, N > &p)
- template<typename T , std::size_t N>
  constexpr domain< T, N > quile::uniform_domain (const range< T > &r)
- template<typename T , std::size_t N>
  constexpr domain< T, N > quile::uniform_domain (T lo, T hi)
- template<typename T , std::size_t N>
  constexpr bool quile::uniform (const domain< T, N > &d)
- template<typename T , std::size_t N>
  requires constexpr std::floating_point< T > domain< T, 2 *N > quile::self_adaptive_variation_domain (const domain< T, N > &d, T lo)
- template<typename T , std::size_t N>
  chain< T, N > quile::chain_min (const domain< T, N > &d)
- template<typename G >
  requires chromosome< G > std::ostream & quile::operator<< (std::ostream &os, const G &g)
- template<typename G >
  requires floating_point_chromosome< G > std::ostream & quile::operator<< (std::ostream &os, const G &g)
- template<typename G >
  requires chromosome< G > population< G > quile::unary_identity (const G &g)
- template<typename G >
  requires chromosome< G > population< G > quile::binary_identity (const G &g0, const G &g1)
- template<typename G >
  requires chromosome< G > auto quile::stochastic_mutation (const mutation_fn< G > &m, probability p)
- template<typename G >
  requires chromosome< G > auto quile::stochastic_recombination (const recombination_fn< G > &r, probability p)
- template<typename G >
  requires chromosome< G > generations< G > quile::evolution (const variation< G > v, const population< G > &first_generation, const populate_1_fn< G > &p1, const populate_2_fn< G > &p2, const termination_condition_fn< G > &tc, std::size_t parents_sz, std::size_t max_history=0)
- template<typename G >
  requires chromosome< G > generations< G > quile::evolution (const variation< G > &v, const populate_0_fn< G > &p0, const populate_1_fn< G > &p1, const populate_2_fn< G > &p2, const termination_condition_fn< G > &tc, std::size_t generation_sz, std::size_t parents_sz, std::size_t max_history=0)
- template<typename G >
  requires chromosome< G > void quile::print (std::ostream &os, const generations< G > &gs, const fitness_db< G > *fd=nullptr)
- template<typename G >
  requires chromosome< G > selection_probabilities quile::cumulative_probabilities (const selection_probabilities_fn< G > &spf, const population< G > &p)
- template<typename C , typename T >
  C quile::select_different_than (const C &c, T t, bool require_nonempty_result)
- fitnesses quile::select_calculable (const fitnesses &fs, bool require_nonempty_result=false)
- template<typename It >
  It quile::detail::advance_cpy (It it, std::size_t n)
- template<typename It , typename Compare = std::less<>>
  std::vector< std::size_t > quile::detail::rank (It first, It last, Compare comp={})

- template<typename T , typename U >
  T quile::detail::id (U u)
- auto quile::linear_ranking_selection (double s)
- probability quile::exponential_ranking_selection (std::size_t mu, std::size_t j)
- template<typename G >
  requires chromosome< G > population< G > quile::detail::generate (std::size_t lambda, const std::function< G()> &f)
- template<typename G >
  requires chromosome< G > populate_2_fn< G > quile::adapter (const populate_1_fn< G > &fn)
- template<auto C, typename G >
  requires genotype_constraints< decltype(C), G > &&chromosome< G > population< G > quile::random_population (std::size_t lambda)
- template<typename G >
  requires chromosome< G > population< G > quile::generational_survivor_selection (std::size_t sz, const population< G > &generation, const population< G > &offspring)
- fitness quile::max (const fitnesses &fs)
- template<typename G >
  requires chromosome< G > fitness quile::max (const population< G > &p, const fitness_db< G > &ff)
- template<typename G >
  requires chromosome< G > fitnesses quile::max (const generations< G > &gs, const fitness_db< G > &ff)
- fitness quile::min (const fitnesses &fs)
- template<typename G >
  requires chromosome< G > fitness quile::min (const population< G > &p, const fitness_db< G > &ff)
- template<typename G >
  requires chromosome< G > fitnesses quile::min (const generations< G > &gs, const fitness_db< G > &ff)
- template<typename G >
  termination_condition_fn< G > quile::max_iterations_termination (std::size_t max)
- template<typename G >
  termination_condition_fn< G > quile::max_fitness_improvement_termination (const fitness_db< G > &ff, std::size_t n, double frac)
- template<typename G >
  termination_condition_fn< G > quile::max_fitness_improvement_termination_2 (const fitness_db< G > &ff, std::size_t n, fitness delta)
- template<typename G , typename F >
  requires chromosome< G > &&std::predicate< F, G > termination_condition_fn< G > quile::threshold_termination (const F &thr)
- template<typename G >
  termination_condition_fn< G > quile::fitness_threshold_termination (const fitness_db< G > &fd, fitness thr, fitness eps)
- template<typename G >
  requires floating_point_chromosome< G > auto quile::Gaussian_mutation (typename G::gene_t sigma, probability p)
- template<typename G >
  requires floating_point_chromosome< G > auto quile::self_adaptive_mutation (typename G::gene_t a0, typename G::gene_t a1)
- template<typename G >
  requires uniform_chromosome< G > population< G > quile::swap_mutation (const G &g)
- template<typename G >
  requires floating_point_chromosome< G > integer_chromosome< G > binary_chromosome< G > auto quile::random_reset (probability p)
- template<typename G >
  requires binary_chromosome< G > auto quile::bit_flipping (probability p)
- template<typename G >
  requires floating_point_chromosome< G > population< G > quile::arithmetic_recombination (const G &g0, const G &g1)

- template<typename G >
  requires floating_point_chromosome< G > population< G > quile::single_arithmetic_recombination (const G &g0, const G &g1)
- template<typename G >
  requires floating_point_chromosome< G > integer_chromosome< G > binary_chromosome< G > population< G > quile::one_point_xover (const G &g0, const G &g1)
- template<typename G >
  requires permutation_chromosome< G > population< G > quile::cut_n_crossfill (const G &g0, const G &g1)
- template<std::floating_point T, std::size_t N>
  T quile::test_functions::distance (const point< T, N > &p0, const point< T, N > &p1)
- template<std::floating_point T>
  std::tuple< T, T > quile::test_functions::coordinates (const point< T, 2 > &p)
- template<std::floating_point T>
  std::tuple< T, T, T > quile::test_functions::coordinates (const point< T, 3 > &p)
- template<std::floating_point T, std::size_t N>
  point< T, N > quile::test_functions::uniform_point (T v)

Variables

- **quile::I**
- **quile::N**
- template<typename F , typename R , typename... Args>
  concept quile::callable = std::convertible_to<std::invoke_result_t<F, Args...>, R>
- template<std::floating_point T>
  const T quile::pi = std::numbers::pi_v<T>
- template<std::floating_point T>
  const T quile::e = std::numbers::e_v<T>
- template<std::floating_point T>
  const T quile::ln2 = std::numbers::ln2_v<T>
- template<typename T >
  constexpr bool quile::is_domain_v = is_domain<T>::value
- template<typename T >
  concept quile::set_of_departure = is_domain_v<T>
- template<typename T >
  constexpr bool quile::is_g_floating_point_v = is_g_floating_point<T>::value
- template<typename T >
  concept quile::floating_point_representation = is_g_floating_point_v<T>
- template<typename T >
  constexpr bool quile::is_g_integer_v = is_g_integer<T>::value
- template<typename T >
  concept quile::integer_representation = is_g_integer_v<T>
- template<typename T >
  constexpr bool quile::is_g_binary_v = is_g_binary<T>::value
- template<typename T >
  concept quile::binary_representation = is_g_binary_v<T>
- template<typename T >
  constexpr bool quile::is_g_permutation_v = is_g_permutation<T>::value
- template<typename T >
  concept quile::permutation_representation = is_g_permutation_v<T>
- template<typename T >
  concept quile::chromosome_representation
- template<typename T >
  constexpr bool quile::is_genotype_v = is_genotype<T>::value
- template<typename G >
  concept quile::chromosome = is_genotype_v<G>

- template<typename G >
  concept quile::floating_point_chromosome
- template<typename G >
  concept quile::integer_chromosome
- template<typename G >
  concept quile::binary_chromosome
- template<typename G >
  concept quile::permutation_chromosome
- template<typename G >
  concept quile::uniform_chromosome = chromosome<G> && G::uniform_domain
- template<typename F , typename G >
  concept quile::genotype_constraints = std::predicate<F, G> && chromosome<G>
- template<typename G >
  requires chromosome< G > const auto quile::constraints_satisfied = [ ](const G&) { return true; }
- template<typename G >
  constexpr bool quile::is_population_v = is_population<G>::value
- template<typename G >
  concept quile::genetic_pool = is_population_v<G>
- template<typename M , typename G >
  concept quile::mutation
- template<typename R , typename G >
  concept quile::recombination
- template<typename F , typename G >
  concept quile::termination_condition = std::predicate<F, std::size_t, generations<G>>
- const fitness quile::incalculable = -std::numeric_limits<fitness>::infinity()
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > quile::test_functions::Ackley
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > quile::test_functions::Alpine
- template<std::floating_point T>
  const test_function< T, 2 > quile::test_functions::Aluffi_Pentini
- template<std::floating_point T>
  const test_function< T, 2 > quile::test_functions::Booth
- template<std::floating_point T>
  const test_function< T, 4 > quile::test_functions::Colville
- template<std::floating_point T>
  const test_function< T, 2 > quile::test_functions::Easom
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > quile::test_functions::exponential
- template<std::floating_point T>
  const test_function< T, 2 > quile::test_functions::Goldstein_Price
- template<std::floating_point T>
  const test_function< T, 2 > quile::test_functions::Hosaki
- template<std::floating_point T>
  const test_function< T, 2 > quile::test_functions::Leon
- template<std::floating_point T>
  const test_function< T, 2 > quile::test_functions::Matyas
- template<std::floating_point T>
  const test_function< T, 2 > quile::test_functions::Mexican_hat
- template<std::floating_point T>
  const test_function< T, 4 > quile::test_functions::Miele_Cantrell
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > quile::test_functions::Rosenbrock
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > quile::test_functions::Schwefel
- template<std::floating_point T, std::size_t N>
  const test_function< T, N > quile::test_functions::sphere

8.1.1 Detailed Description

quile/quile.h is one and only file of the library implementation.

Please include quile/quile.h header using the code #include <quile/quile.h> and compiling your program with apropriate include flag, e.g. -I/home/user/repos/quile for GCC and Clang compilers.

8.1.2 Macro Definition Documentation

8.1.2.1 QUILE_LOG

```
#define QUILE_LOG(
                x )
```

QUILE_LOG(x) macro prints diagnostic information.

QUILE_LOG(x) macro prints diagnostic information to the standard error stream std::cerr provided that QUILE_ENABLE_LOGGING token is defined. Otherwise it has no effect.

Example:

```
#include <quile/quile.h>
int
main()
{
  QUILE_LOG("main");
}
```

Result:

```
# Quile log: main
```

8.1.3 Typedef Documentation

8.1.3.1 chain

```
template<typename T , std::size_t N>
using quile::chain = typedef std::array<T, N>
```

chain represents genetic chain.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  const quile::domain<int, 3> d{ quile::range{ 0, 5 },
                                 quile::range{ 1, 6 },
                                 quile::range{ 2, 7 } };
  const quile::chain<int, 3> c0{ 0, 1, 2 };
  const quile::chain<int, 3> c1{ quile::chain_min(d) };
  assert(c0 == c1);
}
```

Result (might be empty):

8.1.3.2 domain

```
template<typename T , std::size_t N>
using quile::domain = typedef std::array<range<T>, N>
```

`domain` is a type representing domain in form of N-dimensional parallelepiped.

Template Parameters

T	Base type.
N	Domain dimension.

Example:

```
#include <quile/quile.h>
#include <vector>
using type = quile::domain<double, 42>;
static_assert(quile::is_domain<type>::value);
static_assert(!quile::is_domain_v<std::vector<double»);
template<typename T>
requires quile::set_of_departure<T>
struct test
{};
int
main()
{
  [[maybe_unused]] test<type> t{};
}
```

Result (might be empty):

8.1.3.3 fitness

```
using quile::fitness = typedef double
```

`fitness` is a fitness function value type.

Example:

```
#include <cassert>
#include <cmath>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc + std::abs(x); });
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d»;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const populate_0_fn<G> generator =
    random_population<constraints_satisfied<G>, G>;
  const populate_1_fn<G> parents_selection =
    stochastic_universal_sampling<G>{ fps };
```

```
  const populate_2_fn<G> selection_to_the_next_generation =
    adapter<G>(stochastic_universal_sampling<G>{ fps });
  const population<G> p0 = generator(42);
  const population<G> p1 = generator(42);
  const population<G> q0 = parents_selection(10, p0);
  assert(q0.size() == 10);
  const population<G> q1 = selection_to_the_next_generation(42, p0, p1);
  assert(q1.size() == 42);
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Fitness value for [8 2 5 7 3 1 1 1 5 3 8 9 5 9 5 8 1 9 4 3 0 0 8 1
8 2 1 0 7 8 3 1]: 136 (taken from database)
# Quile log: Fitness value for [5 7 9 0 5 3 6 0 6 8 4 4 0 7 1 1 8 3 8 6 9 7 8 1
9 8 8 1 9 8 5 5]: 169 (taken from database)
# Quile log: Fitness value for [9 3 2 1 9 2 4 2 3 3 9 2 7 1 8 5 0 6 2 1 9 7 9 7
2 7 1 2 5 7 9 5]: 149 (taken from database)
# Quile log: Fitness value for [8 5 8 3 6 9 7 1 7 3 8 7 6 2 1 8 3 8 6 3 5 2 2 6
7 9 8 2 6 7 3 5]: 171 (taken from database)
# Quile log: Fitness value for [2 2 6 9 6 7 3 5 5 0 6 6 0 3 3 6 2 8 0 1 2 5 7 9
1 5 7 9 8 6 2 5]: 146 (taken from database)
# Quile log: Fitness value for [7 9 1 8 7 7 5 2 2 6 1 1 9 2 0 2 5 7 1 9 7 0 7 4
0 3 7 8 0 4 3 1]: 135 (taken from database)
# Quile log: Fitness value for [2 0 1 0 1 6 4 4 6 9 1 6 3 8 3 7 6 8 9 8 4 5 6 6
0 6 1 0 0 4 9 7]: 140 (taken from database)
# Quile log: Fitness value for [0 5 6 7 7 1 2 8 8 4 5 0 4 2 3 1 3 1 6 1 5 3 8 9
2 2 7 4 5 4 4 9]: 136 (taken from database)
# Quile log: Fitness value for [6 2 3 0 2 7 8 6 8 8 3 8 1 7 3 8 2 1 6 4 8 3 1 2
8 3 8 2 1 9 1 6]: 145 (taken from database)
# Quile log: Fitness value for [3 8 8 3 0 4 1 3 8 7 2 0 7 3 9 1 8 8 2 4 4 5 7 0
4 3 3 5 6 5 0 5]: 136 (taken from database)
# Quile log: Fitness value for [0 9 3 9 4 0 2 8 2 0 5 9 3 7 2 8 7 0 9 6 6 0 5 6
2 8 8 9 3 4 8 1]: 153 (taken from database)
# Quile log: Fitness value for [1 7 2 7 3 7 4 9 5 5 8 3 4 3 6 4 6 6 8 8 4 1 9 9
0 4 5 2 5 3 1 1]: 150 (taken from database)
# Quile log: Fitness value for [7 7 4 8 2 3 9 2 2 3 7 6 1 6 0 5 7 8 7 5 9 5 7 6
0 5 9 6 6 6 7 1]: 166 (taken from database)
# Quile log: Fitness value for [9 4 6 2 4 0 9 7 6 1 3 9 5 7 2 3 1 7 0 9 1 3 6 8
6 3 8 9 8 0 4 6]: 156 (taken from database)
# Quile log: Fitness value for [3 1 6 7 2 7 4 1 6 9 6 3 9 5 9 7 2 6 2 0 8 7 1 9
7 9 2 5 6 8 9 9]: 175 (taken from database)
# Quile log: Fitness value for [0 1 4 9 2 2 8 7 6 9 7 9 9 5 2 4 0 7 9 7 1 2 7 9
2 4 0 5 8 2 7 0]: 154 (taken from database)
# Quile log: Fitness value for [6 8 9 7 3 9 4 5 3 5 0 1 7 9 6 9 6 3 1 6 1 6 4 8
0 0 1 2 7 0 4 8]: 148 (taken from database)
# Quile log: Fitness value for [6 0 1 9 5 6 8 4 2 5 1 8 9 6 8 3 2 7 1 3 3 4 1 4
2 3 2 6 3 5 1 2]: 130 (taken from database)
# Quile log: Fitness value for [9 7 2 3 1 3 9 9 0 2 7 6 2 0 8 2 3 9 6 7 4 5 2 6
8 2 0 9 0 2 7 1]: 141 (taken from database)
# Quile log: Fitness value for [3 4 0 7 3 0 8 0 3 5 8 1 7 6 0 4 3 8 5 0 0 0 1 1
6 8 3 2 4 2 3 0]: 105 (taken from database)
# Quile log: Fitness value for [7 8 1 4 1 8 2 1 4 7 6 9 5 3 7 3 7 1 4 3 3 3 8 2
7 8 6 1 3 5 4 1]: 142 (taken from database)
# Quile log: Fitness value for [8 5 2 6 1 2 8 6 8 9 2 9 7 5 7 1 4 9 3 5 8 5 8 1
1 9 8 2 4 1 1 2]: 157 (taken from database)
# Quile log: Fitness value for [7 4 8 5 9 7 5 4 5 2 2 3 5 4 6 4 0 2 6 2 2 5 4 1
5 1 3 7 4 2 5 4]: 133 (taken from database)
# Quile log: Fitness value for [6 3 1 8 2 3 3 4 1 4 4 7 0 3 6 1 0 0 7 0 7 7 2 2
4 1 1 4 4 3 2 4]: 104 (taken from database)
# Quile log: Fitness value for [8 7 8 5 8 1 9 4 8 5 0 4 0 1 8 1 8 1 5 1 1 3 3 8
3 7 9 7 5 6 6 7]: 157 (taken from database)
```

8.1.3.4 fitness_function

```
template<typename G >
using quile::fitness_function = typedef std::function<fitness(const G&)>
```

`fitness_function` describes how fit genotype is.

Note

From implementation point of view the `fitness_function` should be thread-safe.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d»;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout « "Fitness function value for " « g « " is " « fd(g) « '.'
             « std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout « "fitness function value for " « p[i] « " is " « fs[i] « '.'
             « std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout « "Database size is equal to " « fd.size() « '.'  « std::endl;
  std::cout « "Database content:" « std::endl;
  for (auto x :  fd) {
    std::cout « ' ' « x.first « " " « x.second « std::endl;
  }
  std::cout « "Database content (once again):" « std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout « ' ' « it->first « " " « it->second « std::endl;
  }
  std::cout « "The best genotype is " « fd.rank_order()[0] « '.'
           « std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
```

```
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

8.1.3.5 fitnesses

```
using quile::fitnesses = typedef std::vector<fitness>
```

`fitnesses` is a sequential container of fitness values.

Example:

```
#include <cassert>
#include <cmath>
#include <cstddef>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using type = double;
  const std::size_t dim = 2;
  static const auto d = uniform_domain<type, dim>(-10., +10.);
  using G = genotype<g_floating_point<type, dim, &d>>;
  const fitness_function<G> ff = [](const G& g) {
    return -(std::fabs(g.value(0)) + std::fabs(g.value(1)));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  for (int i = 0; i < 2; ++i) {
    const auto g = G::random();
    std::cout << "Fitness function value for " << g << " is " << fd(g) << '.'
              << std::endl;
  }
  const population<G> p{ G::random(), G::random() };
  const fitnesses fs = fd(p);
  for (std::size_t i = 0; i < fs.size(); ++i) {
    std::cout << "fitness function value for " << p[i] << " is " << fs[i] << '.'
              << std::endl;
    assert(fs[i] == fd(p[i]));
  }
  std::cout << "Database size is equal to " << fd.size() << '.'  << std::endl;
  std::cout << "Database content:" << std::endl;
  for (auto x :  fd) {
    std::cout << ' ' << x.first << " " << x.second << std::endl;
  }
  std::cout << "Database content (once again):" << std::endl;
  for (fitness_db<G>::const_iterator it = fd.begin(); it != fd.end(); ++it) {
    std::cout << ' ' << it->first << " " << it->second << std::endl;
  }
  std::cout << "The best genotype is " << fd.rank_order()[0] << '.'
            << std::endl;
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Asynchronous fitness value calculations (multithreaded)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (calculated asynchronously on demand)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (calculated asynchronously on demand)
# Quile log: Fitness values for population of size 2
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
fitness function value for 3.612192941474973e+00 -4.847543710851241e+00 is -8.45
9736652326214e+00.
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
fitness function value for -8.986745863807746e+00 2.729955013942309e-02 is -9.01
4045413947169e+00.
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
Database size is equal to 4.
Database content:
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
Database content (once again):
 3.612192941474973e+00 -4.847543710851241e+00 -8.459736652326214e+00
 -8.986745863807746e+00 2.729955013942309e-02 -9.014045413947169e+00
 -1.029802596258641e+00 -6.981969708737376e+00 -8.011772304996017e+00
 -3.212461318882854e+00 -7.280898017904082e+00 -1.049335933678694e+01
The best genotype is # Quile log: Fitness value for [-8.986745863807746e+00 2.72
9955013942309e-02]: -9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [3.612192941474973e+00 -4.847543710851241e+00]: -
8.459736652326214e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-1.029802596258641e+00 -6.981969708737376e+00]:
-8.011772304996017e+00 (taken from database)
# Quile log: Fitness value for [-3.212461318882854e+00 -7.280898017904082e+00]:
-1.049335933678694e+01 (taken from database)
# Quile log: Fitness value for [-8.986745863807746e+00 2.729955013942309e-02]: -
9.014045413947169e+00 (taken from database)
-1.029802596258641e+00 -6.981969708737376e+00.
```

8.1.3.6 generations

```
template<typename G >
using quile::generations = typedef std::deque<population<G> >
```

`generations` is a sequence of populations.

Note

For memory optimization purpose the `std::deque` was chosen instead of `std::vector`—this implementation allows for erasing of the oldest generation.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  const std::size_t n = 4;
  static constexpr const auto d{ quile::uniform_domain<int, n>(0, 9) };
  using G = quile::genotype<quile::g_integer<int, n, &d>>;
  const quile::populate_0_fn<G> generator =
    quile::random_population<quile::constraints_satisfied<G>, G>;
  quile::generations<G> gs{};
  for (int i = 0; i < 10; ++i) {
    gs.push_back(generator(5));
  }
  assert(gs.size() == 10);
  for (int i = 0; auto& p :  gs) {
    std::cout « '#' « i++ « ":  ";
    for (auto& g :  p) {
      std::cout « g « "  ";
    }
    std::cout « '\n';
  }
}
```

Result (might be different due to randomness):

```
#0: 2 0 8 3  6 9 5 1  8 5 0 9  4 8 9 1  0 5 3 9
#1: 1 9 5 8  1 3 1 6  3 8 3 1  6 0 0 5  1 8 1 6
#2: 1 0 2 9  7 7 4 9  0 8 7 4  7 3 7 1  9 7 1 4
#3: 0 9 0 5  5 3 7 6  9 5 3 0  0 3 0 9  2 8 9 0
#4: 9 7 2 5  0 2 6 0  7 0 8 2  6 9 2 9  6 1 0 5
#5: 5 6 5 6  2 5 4 6  9 2 2 9  2 1 4 1  0 6 3 4
#6: 2 8 7 5  1 1 8 9  1 5 6 9  0 1 1 3  2 4 4 6
#7: 6 2 1 8  5 1 2 6  1 5 9 5  6 2 3 5  5 6 1 2
#8: 0 9 1 3  0 7 1 7  7 1 0 6  9 9 7 5  0 9 9 6
#9: 0 3 2 9  7 2 4 5  3 8 1 1  1 5 9 1  8 7 3 6
```

8.1.3.7 mutation_fn

```
template<typename G >
using quile::mutation_fn = typedef std::function<population<G>(const G&)>
```

`mutation_fn` is a callable object which can be invoked on `genotype` and returns `population`.

Example:

```
#include <iostream>
#include <quile/quile.h>
using namespace quile;
namespace {
const auto id = [](auto x) { return x; };
auto
compose()
{
  return id;
}
template<typename F, typename...  Fs>
auto
compose(F&& f, Fs&&...  fs)
{
  return [=](auto x) { return f(compose(fs...)(x)); };
}
template<chromosome G, typename...  Ms>
requires(mutation<Ms, G>&&...)  auto mutation_composition(Ms&&...  ms)
{
  return [=](const G& g) -> population<G> {
    return { compose([=](const G& g) -> G { return ms(g)[0]; }...)(g) };
  };
}
template<typename G>
requires integer_chromosome<G>
```

```
auto
deterministic_mutation(std::size_t pos, typename G::gene_t val)
{
  return [=](G g) -> population<G> {
    g.value(pos, val);
    return population<G>{ g };
  };
}
}
int
main()
{
  const std::size_t n = 5;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<g_integer<int, n, &d»;
  const G g{};
  mutation_fn<G> m0 = mutation_composition<G>();
  std::cout « m0(g)[0] « '\n';
  mutation_fn<G> m1 = mutation_composition<G>(deterministic_mutation<G>(1, 1));
  std::cout « m1(g)[0] « '\n';
  mutation_fn<G> m2 = mutation_composition<G>(deterministic_mutation<G>(2, 2));
  std::cout « m2(g)[0] « '\n';
  mutation_fn<G> m12 = mutation_composition<G>(deterministic_mutation<G>(1, 1),
                                               deterministic_mutation<G>(2, 2));
  std::cout « m12(g)[0] « '\n';
  mutation_fn<G> m134 =
    mutation_composition<G>(deterministic_mutation<G>(1, 1),
                            deterministic_mutation<G>(3, 3),
                            deterministic_mutation<G>(4, 4));
  std::cout « m134(g)[0] « '\n';
}
```

Result:

```
0 0 0 0 0
0 1 0 0 0
0 0 2 0 0
0 1 2 0 0
0 1 0 3 4
```

8.1.3.8 populate_0_fn

```
template<typename G >
using quile::populate_0_fn = typedef std::function<population<G>(std::size_t)>
```

`populate_0_fn` can be used for first generation creation.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ quile::uniform_domain<int, n>(0, 9) };
  using G = quile::genotype<quile::g_integer<int, n, &d>>;
  const quile::populate_0_fn<G> generator =
    quile::random_population<quile::constraints_satisfied<G>, G>;
  const quile::population<G> p = generator(10);
  assert(p.size() == 10);
  for (auto& g :  p) {
    std::cout « g « '\n';
  }
}
```

Result (might be different due to randomness):

```
7 7 0 5 2 6 2 6 3 0 9 6 2 1 8 2 6 8 2 1 6 2 5 5 5 0 7 1 4 9 8 6
7 0 4 1 0 3 4 2 1 0 9 9 8 7 9 0 3 2 8 2 9 5 9 8 2 2 1 0 4 3 4 9
5 6 5 8 0 5 3 1 9 0 0 9 2 5 5 9 6 6 1 6 7 3 2 7 6 4 2 5 3 7 8 6
8 2 5 0 2 4 2 4 5 0 8 1 6 6 4 7 8 5 6 5 6 3 1 1 5 8 3 6 6 1 0 9
9 3 5 9 1 5 6 2 7 3 6 1 8 6 2 9 0 6 3 1 8 3 0 7 9 6 3 9 6 2 0 4
5 1 4 6 1 6 4 6 4 0 1 6 0 5 3 0 7 3 1 9 2 6 6 8 5 9 0 7 0 3 4 0
9 4 0 2 6 8 3 1 0 4 4 9 3 1 4 9 8 8 6 7 8 0 2 6 8 7 0 3 9 2 2 8
4 0 1 1 5 9 4 9 6 4 3 5 2 6 5 0 0 5 8 9 0 7 3 6 9 9 6 2 3 3 1 8
5 8 0 4 2 1 5 8 3 6 1 1 0 0 0 3 6 2 4 1 0 4 0 5 5 6 5 7 1 5 5 5
6 8 6 6 3 1 1 7 5 0 7 9 9 4 4 0 7 4 0 4 6 7 9 2 7 9 7 7 3 2 2 7
```

8.1.3.9 populate_1_fn

```
template<typename G >
using quile::populate_1_fn = typedef std::function<population<G>(std::size_t, const population<G>&)>
```

`populate_1_fn` can be used for parents selection.

Example:

```
#include <cassert>
#include <cmath>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc + std::abs(x); });
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const populate_0_fn<G> generator =
    random_population<constraints_satisfied<G>, G>;
  const populate_1_fn<G> parents_selection =
    stochastic_universal_sampling<G>{ fps };
  const populate_2_fn<G> selection_to_the_next_generation =
    adapter<G>(stochastic_universal_sampling<G>{ fps });
  const population<G> p0 = generator(42);
  const population<G> p1 = generator(42);
  const population<G> q0 = parents_selection(10, p0);
  assert(q0.size() == 10);
  const population<G> q1 = selection_to_the_next_generation(42, p0, p1);
  assert(q1.size() == 42);
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Fitness value for [8 2 5 7 3 1 1 1 5 3 8 9 5 9 5 8 1 9 4 3 0 0 8 1
8 2 1 0 7 8 3 1]: 136 (taken from database)
# Quile log: Fitness value for [5 7 9 0 5 3 6 0 6 8 4 4 0 7 1 1 8 3 8 6 9 7 8 1
9 8 8 1 9 8 5 5]: 169 (taken from database)
# Quile log: Fitness value for [9 3 2 1 9 2 4 2 3 3 9 2 7 1 8 5 0 6 2 1 9 7 9 7
2 7 1 2 5 7 9 5]: 149 (taken from database)
# Quile log: Fitness value for [8 5 8 3 6 9 7 1 7 3 8 7 6 2 1 8 3 8 6 3 5 2 2 6
7 9 8 2 6 7 3 5]: 171 (taken from database)
# Quile log: Fitness value for [2 2 6 9 6 7 3 5 5 0 6 6 0 3 3 6 2 8 0 1 2 5 7 9
1 5 7 9 8 6 2 5]: 146 (taken from database)
# Quile log: Fitness value for [7 9 1 8 7 7 5 2 2 6 1 1 9 2 0 2 5 7 1 9 7 0 7 4
0 3 7 8 0 4 3 1]: 135 (taken from database)
```

```
# Quile log: Fitness value for [2 0 1 0 1 6 4 4 6 9 1 6 3 8 3 7 6 8 9 8 4 5 6 6
0 6 1 0 0 4 9 7]: 140 (taken from database)
# Quile log: Fitness value for [0 5 6 7 7 1 2 8 8 4 5 0 4 2 3 1 3 1 6 1 5 3 8 9
2 2 7 4 5 4 4 9]: 136 (taken from database)
# Quile log: Fitness value for [6 2 3 0 2 7 8 6 8 8 3 8 1 7 3 8 2 1 6 4 8 3 1 2
8 3 8 2 1 9 1 6]: 145 (taken from database)
# Quile log: Fitness value for [3 8 8 3 0 4 1 3 8 7 2 0 7 3 9 1 8 8 2 4 4 5 7 0
4 3 3 5 6 5 0 5]: 136 (taken from database)
# Quile log: Fitness value for [0 9 3 9 4 0 2 8 2 0 5 9 3 7 2 8 7 0 9 6 6 0 5 6
2 8 8 9 3 4 8 1]: 153 (taken from database)
# Quile log: Fitness value for [1 7 2 7 3 7 4 9 5 5 8 3 4 3 6 4 6 6 8 8 4 1 9 9
0 4 5 2 5 3 1 1]: 150 (taken from database)
# Quile log: Fitness value for [7 7 4 8 2 3 9 2 2 3 7 6 1 6 0 5 7 8 7 5 9 5 7 6
0 5 9 6 6 6 7 1]: 166 (taken from database)
# Quile log: Fitness value for [9 4 6 2 4 0 9 7 6 1 3 9 5 7 2 3 1 7 0 9 1 3 6 8
6 3 8 9 8 0 4 6]: 156 (taken from database)
# Quile log: Fitness value for [3 1 6 7 2 7 4 1 6 9 6 3 9 5 9 7 2 6 2 0 8 7 1 9
7 9 2 5 6 8 9 9]: 175 (taken from database)
# Quile log: Fitness value for [0 1 4 9 2 2 8 7 6 9 7 9 9 5 2 4 0 7 9 7 1 2 7 9
2 4 0 5 8 2 7 0]: 154 (taken from database)
# Quile log: Fitness value for [6 8 9 7 3 9 4 5 3 5 0 1 7 9 6 9 6 3 1 6 1 6 4 8
0 0 1 2 7 0 4 8]: 148 (taken from database)
# Quile log: Fitness value for [6 0 1 9 5 6 8 4 2 5 1 8 9 6 8 3 2 7 1 3 3 4 1 4
2 3 2 6 3 5 1 2]: 130 (taken from database)
# Quile log: Fitness value for [9 7 2 3 1 3 9 9 0 2 7 6 2 0 8 2 3 9 6 7 4 5 2 6
8 2 0 9 0 2 7 1]: 141 (taken from database)
# Quile log: Fitness value for [3 4 0 7 3 0 8 0 3 5 8 1 7 6 0 4 3 8 5 0 0 0 1 1
6 8 3 2 4 2 3 0]: 105 (taken from database)
# Quile log: Fitness value for [7 8 1 4 1 8 2 1 4 7 6 9 5 3 7 3 7 1 4 3 3 3 8 2
7 8 6 1 3 5 4 1]: 142 (taken from database)
# Quile log: Fitness value for [8 5 2 6 1 2 8 6 8 9 2 9 7 5 7 1 4 9 3 5 8 5 8 1
1 9 8 2 4 1 1 2]: 157 (taken from database)
# Quile log: Fitness value for [7 4 8 5 9 7 5 4 5 2 2 3 5 4 6 4 0 2 6 2 2 5 4 1
5 1 3 7 4 2 5 4]: 133 (taken from database)
# Quile log: Fitness value for [6 3 1 8 2 3 3 4 1 4 4 7 0 3 6 1 0 0 7 0 7 7 2 2
4 1 1 4 4 3 2 4]: 104 (taken from database)
# Quile log: Fitness value for [8 7 8 5 8 1 9 4 8 5 0 4 0 1 8 1 8 1 5 1 1 3 3 8
3 7 9 7 5 6 6 7]: 157 (taken from database)
```

8.1.3.10 populate_2_fn

```
template<typename G >
using quile::populate_2_fn = typedef std::function< population<G>(std::size_t, const population<G>&,
const population<G>&)>
```

`populate_2_fn` can be used for selection to the next generation.

Example:

```
#include <cassert>
#include <cmath>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc + std::abs(x); });
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
```

```
  const fitness_proportional_selection<G> fps{ fd };
  const populate_0_fn<G> generator =
    random_population<constraints_satisfied<G>, G>;
  const populate_1_fn<G> parents_selection =
    stochastic_universal_sampling<G>{ fps };
  const populate_2_fn<G> selection_to_the_next_generation =
    adapter<G>(stochastic_universal_sampling<G>{ fps });
  const population<G> p0 = generator(42);
  const population<G> p1 = generator(42);
  const population<G> q0 = parents_selection(10, p0);
  assert(q0.size() == 10);
  const population<G> q1 = selection_to_the_next_generation(42, p0, p1);
  assert(q1.size() == 42);
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Fitness value for [8 2 5 7 3 1 1 1 5 3 8 9 5 9 5 8 1 9 4 3 0 0 8 1
8 2 1 0 7 8 3 1]: 136 (taken from database)
# Quile log: Fitness value for [5 7 9 0 5 3 6 0 6 8 4 4 0 7 1 1 8 3 8 6 9 7 8 1
9 8 8 1 9 8 5 5]: 169 (taken from database)
# Quile log: Fitness value for [9 3 2 1 9 2 4 2 3 3 9 2 7 1 8 5 0 6 2 1 9 7 9 7
2 7 1 2 5 7 9 5]: 149 (taken from database)
# Quile log: Fitness value for [8 5 8 3 6 9 7 1 7 3 8 7 6 2 1 8 3 8 6 3 5 2 2 6
7 9 8 2 6 7 3 5]: 171 (taken from database)
# Quile log: Fitness value for [2 2 6 9 6 7 3 5 5 0 6 6 0 3 3 6 2 8 0 1 2 5 7 9
1 5 7 9 8 6 2 5]: 146 (taken from database)
# Quile log: Fitness value for [7 9 1 8 7 7 5 2 2 6 1 1 9 2 0 2 5 7 1 9 7 0 7 4
0 3 7 8 0 4 3 1]: 135 (taken from database)
# Quile log: Fitness value for [2 0 1 0 1 6 4 4 6 9 1 6 3 8 3 7 6 8 9 8 4 5 6 6
0 6 1 0 0 4 9 7]: 140 (taken from database)
# Quile log: Fitness value for [0 5 6 7 7 1 2 8 8 4 5 0 4 2 3 1 3 1 6 1 5 3 8 9
2 2 7 4 5 4 4 9]: 136 (taken from database)
# Quile log: Fitness value for [6 2 3 0 2 7 8 6 8 8 3 8 1 7 3 8 2 1 6 4 8 3 1 2
8 3 8 2 1 9 1 6]: 145 (taken from database)
# Quile log: Fitness value for [3 8 8 3 0 4 1 3 8 7 2 0 7 3 9 1 8 8 2 4 4 5 7 0
4 3 3 5 6 5 0 5]: 136 (taken from database)
# Quile log: Fitness value for [0 9 3 9 4 0 2 8 2 0 5 9 3 7 2 8 7 0 9 6 6 0 5 6
2 8 8 9 3 4 8 1]: 153 (taken from database)
# Quile log: Fitness value for [1 7 2 7 3 7 4 9 5 5 8 3 4 3 6 4 6 6 8 8 4 1 9 9
0 4 5 2 5 3 1 1]: 150 (taken from database)
# Quile log: Fitness value for [7 7 4 8 2 3 9 2 2 3 7 6 1 6 0 5 7 8 7 5 9 5 7 6
0 5 9 6 6 6 7 1]: 166 (taken from database)
# Quile log: Fitness value for [9 4 6 2 4 0 9 7 6 1 3 9 5 7 2 3 1 7 0 9 1 3 6 8
6 3 8 9 8 0 4 6]: 156 (taken from database)
# Quile log: Fitness value for [3 1 6 7 2 7 4 1 6 9 6 3 9 5 9 7 2 6 2 0 8 7 1 9
7 9 2 5 6 8 9 9]: 175 (taken from database)
# Quile log: Fitness value for [0 1 4 9 2 2 8 7 6 9 7 9 9 5 2 4 0 7 9 7 1 2 7 9
2 4 0 5 8 2 7 0]: 154 (taken from database)
# Quile log: Fitness value for [6 8 9 7 3 9 4 5 3 5 0 1 7 9 6 9 6 3 1 6 1 6 4 8
0 0 1 2 7 0 4 8]: 148 (taken from database)
# Quile log: Fitness value for [6 0 1 9 5 6 8 4 2 5 1 8 9 6 8 3 2 7 1 3 3 4 1 4
2 3 2 6 3 5 1 2]: 130 (taken from database)
# Quile log: Fitness value for [9 7 2 3 1 3 9 9 0 2 7 6 2 0 8 2 3 9 6 7 4 5 2 6
8 2 0 9 0 2 7 1]: 141 (taken from database)
# Quile log: Fitness value for [3 4 0 7 3 0 8 0 3 5 8 1 7 6 0 4 3 8 5 0 0 0 1 1
6 8 3 2 4 2 3 0]: 105 (taken from database)
# Quile log: Fitness value for [7 8 1 4 1 8 2 1 4 7 6 9 5 3 7 3 7 1 4 3 3 3 8 2
7 8 6 1 3 5 4 1]: 142 (taken from database)
# Quile log: Fitness value for [8 5 2 6 1 2 8 6 8 9 2 9 7 5 7 1 4 9 3 5 8 5 8 1
1 9 8 2 4 1 1 2]: 157 (taken from database)
# Quile log: Fitness value for [7 4 8 5 9 7 5 4 5 2 2 3 5 4 6 4 0 2 6 2 2 5 4 1
5 1 3 7 4 2 5 4]: 133 (taken from database)
# Quile log: Fitness value for [6 3 1 8 2 3 3 4 1 4 4 7 0 3 6 1 0 0 7 0 7 7 2 2
4 1 1 4 4 3 2 4]: 104 (taken from database)
# Quile log: Fitness value for [8 7 8 5 8 1 9 4 8 5 0 4 0 1 8 1 8 1 5 1 1 3 3 8
3 7 9 7 5 6 6 7]: 157 (taken from database)
```

8.1.3.11 population

```
template<typename G >
using quile::population = typedef std::vector<G>
```

`population` is a sequence of genotypes implemented as a `std::vector` sequence container.

Example:

```
#include <iostream>
#include <quile/quile.h>
const std::size_t n = 32;
using genotype_t = quile::genotype<quile::g_binary<n>>;
using population_t = quile::population<genotype_t>;
static_assert(quile::is_population<population_t>::value);
static_assert(!quile::is_population_v<genotype_t>);
template<typename T>
requires quile::genetic_pool<T>
struct test
{};
int
main()
{
  [[maybe_unused]] test<population_t> t{};
  population_t p{};
  for (int i = 0; i < 8; ++i) {
    p.push_back(genotype_t{});
  }
  for (auto& g :  p) {
    g.random_reset();
  }
  for (auto& g :  p) {
    std::cout « g « '\n';
  }
}
```

Result (might be different due to randomness):

```
1 1 1 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 1 0 0 1 1 0 1 0
0 0 1 0 1 0 1 1 0 0 0 1 1 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 0 0 0 1
0 0 0 0 1 0 1 0 1 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 1
1 0 0 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 1
0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 1 1 0 0 1 0 1 0 1 0 0 1 1
1 1 1 1 0 0 0 0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 1 0
1 1 1 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1
0 0 1 0 1 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1 0 0 0 0 1 1 1
```

8.1.3.12 probability

```
using quile::probability = typedef double
```

`probability` type represents probability values, i.e. numbers from $[0, 1]_{\mathbb{R}}$ interval.

8.1.3.13 recombination_fn

```
template<typename G >
using quile::recombination_fn = typedef std::function<population<G>(const G&, const G&)>
```

`recombination_fn` is a callable object which can be invoked on two objects of type `genotype` and returns `population`.

Example:

```
#include <iostream>
#include <quile/quile.h>
```

```
#include <vector>
using namespace quile;
namespace {
template<chromosome G, typename...  Rs>
requires(recombination<Rs, G>&&...)  auto random_recombination(Rs&&...  rs)
{
  return
    [r = std::vector<recombination_fn<G>{ rs...  }](const G& g0, const G& g1) {
      return r[random_U<std::size_t>(0, r.size() - 1)](g0, g1);
    };
}
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<double, n>(0., 1.)  };
  using G = genotype<g_floating_point<double, n, &d>;
  const auto g0 = G::random();
  const auto g1 = G::random();
  std::cout « "Parents:\n";
  std::cout « g0 « '\n';
  std::cout « g1 « '\n';
  const recombination_fn<G> r = random_recombination<G>(
    single_arithmetic_recombination<G>, one_point_xover<G>);
  const population<G> p = r(g0, g1);
  std::cout « "Offspring:\n";
  std::cout « p[0] « '\n';
  std::cout « p[1] « '\n';
}
```

Result (might be different due to randomness):

```
Parents:
7.632938301095844e-01 5.380850537336410e-01 8.662196756244862e-01 2.385804670907
881e-01 3.992661030128286e-01 9.637379325762896e-01 8.015222496416907e-01 4.0781
02496612352e-01 6.495688681819360e-02 3.773104410660121e-01 4.470655570561451e-0
1 9.051977045719400e-01 3.754480928190284e-01 5.852131314403002e-01 2.7607692016
30901e-01 6.901601014454185e-01 1.876479488560436e-02 9.691397176956201e-01 2.92
6243459260146e-01 7.015068115063313e-01 5.096234113058886e-01 6.169321355472475e
-02 9.275202166839894e-01 3.908546545463924e-01 8.545031327328174e-01 2.57658215
2773367e-01 6.437327114578221e-01 9.651463910919517e-01 3.518883850377567e-01 9.
158175963353863e-01 5.548562472262517e-01 4.290178207550817e-01
3.567643357649279e-01 1.391264328060377e-01 6.003527872330220e-01 2.862740240145
400e-01 4.883058640665820e-01 9.146947899566816e-01 3.620078563266221e-01 8.7475
99088295999e-01 9.608142268763895e-01 9.611901113709199e-01 8.793766878269520e-0
1 4.577419434232913e-01 8.500066867665092e-02 2.611466104241356e-01 2.0453898352
02442e-02 7.230066340434387e-01 8.718111857170628e-01 5.250218644745700e-01 1.16
1269539668478e-02 1.131626122567013e-01 6.667633275191966e-01 6.620535971330396e
-01 4.047631606157035e-01 3.884035499872193e-01 3.895695861220623e-01 9.19629564
4505946e-01 2.274506717811672e-02 2.862834266068857e-01 9.566141505366095e-02 8.
618512812097814e-01 3.744678094228941e-01 5.132685497467671e-01
Offspring:
7.632938301095844e-01 5.380850537336410e-01 8.662196756244862e-01 2.385804670907
881e-01 3.992661030128286e-01 9.637379325762896e-01 8.015222496416907e-01 4.0781
02496612352e-01 6.495688681819360e-02 3.773104410660121e-01 4.470655570561451e-0
1 9.051977045719400e-01 3.754480928190284e-01 5.852131314403002e-01 2.7607692016
30901e-01 6.901601014454185e-01 1.876479488560436e-02 9.691397176956201e-01 2.92
6243459260146e-01 7.015068115063313e-01 5.096234113058886e-01 6.620535971330396e
-01 4.047631606157035e-01 3.884035499872193e-01 3.895695861220623e-01 9.19629564
4505946e-01 2.274506717811672e-02 2.862834266068857e-01 9.566141505366095e-02 8.
618512812097814e-01 3.744678094228941e-01 5.132685497467671e-01
3.567643357649279e-01 1.391264328060377e-01 6.003527872330220e-01 2.862740240145
400e-01 4.883058640665820e-01 9.146947899566816e-01 3.620078563266221e-01 8.7475
99088295999e-01 9.608142268763895e-01 9.611901113709199e-01 8.793766878269520e-0
1 4.577419434232913e-01 8.500066867665092e-02 2.611466104241356e-01 2.0453898352
02442e-02 7.230066340434387e-01 8.718111857170628e-01 5.250218644745700e-01 1.16
1269539668478e-02 1.131626122567013e-01 6.667633275191966e-01 6.169321355472475e
-02 9.275202166839894e-01 3.908546545463924e-01 8.545031327328174e-01 2.57658215
2773367e-01 6.437327114578221e-01 9.651463910919517e-01 3.518883850377567e-01 9.
158175963353863e-01 5.548562472262517e-01 4.290178207550817e-01
```

8.1.3.14 selection_probabilities

```
using quile::selection_probabilities = typedef std::vector<probability>
```

`selection_probabilities` is a sequential container intended for keeping selection probabilities values.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d»;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout « p[i] « ":  " « sp[i] « ", " « cp[i] « '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
```

```
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

8.1.3.15 selection_probabilities_fn

```
template<typename G >
using quile::selection_probabilities_fn = typedef std::function<selection_probabilities(const
population<G>&)>
```

`selection_probabilities_fn` is a callable object which can be invoked on population and returns corresponding selection probabilities for each genotype from population.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout << p[i] << ":  " << sp[i] << ", " << cp[i] << '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

8.1.3.16 termination_condition_fn

```
template<typename G >
using quile::termination_condition_fn = typedef std::function<bool(std::size_t, const generations<G>&)>
```

`termination_condition_fn` is a callable object which states when evolution should be finished.

8.1.4 Function Documentation

8.1.4.1 adapter()

```
template<typename G >
requires chromosome<G> populate_2_fn<G> quile::adapter (
             const populate_1_fn< G > & fn )
```

`adapter` implements *flatten* function between `populate_2_fn` mechanism and `populate_1_fn` mechanism.

Template Parameters

G	Some `genotype` specialization.

Parameters

fn	Mechanism of `populate_1_fn` type.

Returns

Mechanism of `populate_2_fn` type, which applies `fn` to two flattened populations.

Note

`adapter` can be useful when used with roulette wheel selection or stochastic universal sampling as a method of selection to the next generation.

Example:

```
#include <cassert>
#include <cmath>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc + std::abs(x); });
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const populate_0_fn<G> generator =
    random_population<constraints_satisfied<G>, G>;
  const populate_1_fn<G> parents_selection =
    stochastic_universal_sampling<G>{ fps };
  const populate_2_fn<G> selection_to_the_next_generation =
    adapter<G>(stochastic_universal_sampling<G>{ fps });
  const population<G> p0 = generator(42);
  const population<G> p1 = generator(42);
  const population<G> q0 = parents_selection(10, p0);
  assert(q0.size() == 10);
  const population<G> q1 = selection_to_the_next_generation(42, p0, p1);
  assert(q1.size() == 42);
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Fitness value for [8 2 5 7 3 1 1 1 5 3 8 9 5 9 5 8 1 9 4 3 0 0 8 1
8 2 1 0 7 8 3 1]: 136 (taken from database)
# Quile log: Fitness value for [5 7 9 0 5 3 6 0 6 8 4 4 0 7 1 1 8 3 8 6 9 7 8 1
9 8 8 1 9 8 5 5]: 169 (taken from database)
# Quile log: Fitness value for [9 3 2 1 9 2 4 2 3 3 9 2 7 1 8 5 0 6 2 1 9 7 9 7
2 7 1 2 5 7 9 5]: 149 (taken from database)
# Quile log: Fitness value for [8 5 8 3 6 9 7 1 7 3 8 7 6 2 1 8 3 8 6 3 5 2 2 6
7 9 8 2 6 7 3 5]: 171 (taken from database)
# Quile log: Fitness value for [2 2 6 9 6 7 3 5 5 0 6 6 0 3 3 6 2 8 0 1 2 5 7 9
1 5 7 9 8 6 2 5]: 146 (taken from database)
# Quile log: Fitness value for [7 9 1 8 7 7 5 2 2 6 1 1 9 2 0 2 5 7 1 9 7 0 7 4
0 3 7 8 0 4 3 1]: 135 (taken from database)
# Quile log: Fitness value for [2 0 1 0 1 6 4 4 6 9 1 6 3 8 3 7 6 8 9 8 4 5 6 6
0 6 1 0 0 4 9 7]: 140 (taken from database)
# Quile log: Fitness value for [0 5 6 7 7 1 2 8 8 4 5 0 4 2 3 1 3 1 6 1 5 3 8 9
2 2 7 4 5 4 4 9]: 136 (taken from database)
# Quile log: Fitness value for [6 2 3 0 2 7 8 6 8 8 3 8 1 7 3 8 2 1 6 4 8 3 1 2
8 3 8 2 1 9 1 6]: 145 (taken from database)
# Quile log: Fitness value for [3 8 8 3 0 4 1 3 8 7 2 0 7 3 9 1 8 8 2 4 4 5 7 0
4 3 3 5 6 5 0 5]: 136 (taken from database)
# Quile log: Fitness value for [0 9 3 9 4 0 2 8 2 0 5 9 3 7 2 8 7 0 9 6 6 0 5 6
2 8 8 9 3 4 8 1]: 153 (taken from database)
# Quile log: Fitness value for [1 7 2 7 3 7 4 9 5 5 8 3 4 3 6 4 6 6 8 8 4 1 9 9
0 4 5 2 5 3 1 1]: 150 (taken from database)
# Quile log: Fitness value for [7 7 4 8 2 3 9 2 2 3 7 6 1 6 0 5 7 8 7 5 9 5 7 6
0 5 9 6 6 6 7 1]: 166 (taken from database)
# Quile log: Fitness value for [9 4 6 2 4 0 9 7 6 1 3 9 5 7 2 3 1 7 0 9 1 3 6 8
6 3 8 9 8 0 4 6]: 156 (taken from database)
# Quile log: Fitness value for [3 1 6 7 2 7 4 1 6 9 6 3 9 5 9 7 2 6 2 0 8 7 1 9
7 9 2 5 6 8 9 9]: 175 (taken from database)
# Quile log: Fitness value for [0 1 4 9 2 2 8 7 6 9 7 9 9 5 2 4 0 7 9 7 1 2 7 9
2 4 0 5 8 2 7 0]: 154 (taken from database)
# Quile log: Fitness value for [6 8 9 7 3 9 4 5 3 5 0 1 7 9 6 9 6 3 1 6 1 6 4 8
0 0 1 2 7 0 4 8]: 148 (taken from database)
# Quile log: Fitness value for [6 0 1 9 5 6 8 4 2 5 1 8 9 6 8 3 2 7 1 3 3 4 1 4
2 3 2 6 3 5 1 2]: 130 (taken from database)
# Quile log: Fitness value for [9 7 2 3 1 3 9 9 0 2 7 6 2 0 8 2 3 9 6 7 4 5 2 6
8 2 0 9 0 2 7 1]: 141 (taken from database)
# Quile log: Fitness value for [3 4 0 7 3 0 8 0 3 5 8 1 7 6 0 4 3 8 5 0 0 0 1 1
6 8 3 2 4 2 3 0]: 105 (taken from database)
# Quile log: Fitness value for [7 8 1 4 1 8 2 1 4 7 6 9 5 3 7 3 7 1 4 3 3 3 8 2
7 8 6 1 3 5 4 1]: 142 (taken from database)
# Quile log: Fitness value for [8 5 2 6 1 2 8 6 8 9 2 9 7 5 7 1 4 9 3 5 8 5 8 1
1 9 8 2 4 1 1 2]: 157 (taken from database)
# Quile log: Fitness value for [7 4 8 5 9 7 5 4 5 2 2 3 5 4 6 4 0 2 6 2 2 5 4 1
5 1 3 7 4 2 5 4]: 133 (taken from database)
# Quile log: Fitness value for [6 3 1 8 2 3 3 4 1 4 4 7 0 3 6 1 0 0 7 0 7 7 2 2
4 1 1 4 4 3 2 4]: 104 (taken from database)
# Quile log: Fitness value for [8 7 8 5 8 1 9 4 8 5 0 4 0 1 8 1 8 1 5 1 1 3 3 8
3 7 9 7 5 6 6 7]: 157 (taken from database)
```

8.1.4.2 arithmetic_recombination()

```
template<typename G >
requires floating_point_chromosome<G> population<G> quile::arithmetic_recombination (
            const G & g0,
            const G & g1 )
```

`arithmetic_recombination` is arithmetic recombination.

Template Parameters

Some	`G` specialization.

Parameters

g0	First parent.
g1	Secong parent.

Returns

Population containing one offspring genotype.

Example:

```
#include <cmath>
#include <cstddef>
#include <fstream>
#include <quile/quile.h>
using namespace quile;
using type = double;
const std::size_t dim = 2;
fitness
f(type x, type y)
{
  return -(x * x + y * y);
}
int
main()
{
  static const domain<type, dim> d0{ range{ -35., +35.  }, range{ -35., +35.  } };
  static const domain<type, 2 * dim> d{ self_adaptive_variation_domain(d0,
                                                                       .001) };
  using G = genotype<g_floating_point<type, 2 * dim, &d>;
  const fitness_function<G> ff = [](const G& g) {
    return f(g.value(0), g.value(1));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const auto p0 = random_population<constraints_satisfied<G>, G>;
  const auto p1 = stochastic_universal_sampling<G>{ fps };
  const auto p2 = adapter<G>(stochastic_universal_sampling<G>{ fps });
  const std::size_t generation_sz{ 1000 };
  const std::size_t parents_sz{ 42 };
  const auto tc = max_fitness_improvement_termination<G>(fd, 10, 0.05);
  const variation<G> v{ self_adaptive_mutation<G>(.002, .002),
                        arithmetic_recombination<G> };
  std::ofstream file{ "evolution.dat" };
  print(file, evolution<G>(v, p0, p1, p2, tc, generation_sz, parents_sz));
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
135605076150335e+01 1.087888184480514e+01]: -1.347580197669043e+01 (taken from d
atabase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.269892419837285e+00 5.600610920462099e-01 1.5
74992500337886e+01 5.375036543150264e+00]: -1.926295184784189e+00 (taken from da
tabase)
# Quile log: Fitness value for [-1.054741505834507e+00 6.756811722622036e-01 6.5
02789087426748e+00 5.428130674046989e+00]: -1.569024690679668e+00 (taken from da
tabase)
# Quile log: Fitness value for [4.518610900865383e-01 2.731480419554952e+00 1.29
4046196412222e+01 6.323685183509894e+00]: -7.665163727146291e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.895858669305523e+00 1.184854220819247e-01 7.4
78855481721993e+00 1.360918242461105e+01]: -3.608318889226842e+00 (taken from da
tabase)
# Quile log: Fitness value for [1.547716625980463e+00 1.824177922655826e+00 6.18
0341656735098e+00 1.119516183733311e+01]: -5.723051847841274e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.337581995874235e+00 2.944374431065633e-01 1.2
```

```
09534931775909e+01 1.598199667031198e+01]: -1.875819003590032e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.053386936859832e-01 -1.304040888209297e-01 9.1
67307432344389e+00 6.053271950893222e+00]: -6.655756379290628e-01 (taken from da
tabase)
# Quile log: Fitness value for [1.653184414281981e-01 1.735069055490136e+00 6.80
4112518955824e+00 1.256064803553919e+01]: -3.037794814395682e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [5.889449492784511e-01 7.395991791184260e-01 8.93
3746939996855e+00 7.688211146302455e+00]: -8.938630990332468e-01 (taken from dat
abase)
# Quile log: Fitness value for [3.731415294381901e-01 1.228299461433360e+00 6.93
0605184757385e+00 1.276664983031973e+01]: -1.647954167948954e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.772574284862156e-02 1.274248905031767e+00 4.66
5054866715833e+00 1.664047719017669e+01]: -1.631406077933000e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
```

Note

Evolution result is saved in separate file (not included).

8.1.4.3 binary_identity()

```
template<typename G >
requires chromosome<G> population<G> quile::binary_identity (
            const G & g0,
            const G & g1 )
```

`binary_identity` is an identity recombination.

Template Parameters

G	Some `genotype` specialization.

Parameters

g0	Genotype (first parent).
g1	Genotype (second parent)

Returns

Population consisting of `g0` and `g1`.

Example:

```
#include <iostream>
```

```
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const auto g0 = G::random();
  const auto g1 = G::random();
  std::cout « g0 « '\n';
  std::cout « g1 « '\n';
  const auto h0 = unary_identity(g0)[0];
  std::cout « h0 « '\n';
  const auto p = binary_identity(g0, g1);
  std::cout « p[0] « '\n';
  std::cout « p[1] « '\n';
}
```

Result (might be different due to randomness):

```
0 3 2 5 1 4 6
1 2 6 0 4 3 5
0 3 2 5 1 4 6
0 3 2 5 1 4 6
1 2 6 0 4 3 5
```

8.1.4.4 bit_flipping()

```
template<typename G >
requires binary_chromosome<G> auto quile::bit_flipping (
             probability p )
```

`bit_flipping` returns bit-flipping mutation with parameter `p`.

Template Parameters

G	Some `genotype` specialization.

Parameters

p	Gene mutation probability.

Returns

Bit-flipping mutation operator.

8.1.4.5 cart2polar()

```
template<std::floating_point T>
std::tuple<T, T> quile::cart2polar (
             T x,
             T y )
```

`cart2polar` changes coordinate system from Cartesian to polar.

Template Parameters

T	Argument type and base for return type (floating-point).

Parameters

x	x coordinate in Cartesian coordinate system.
y	y coordinate in Cartesian coordinate system.

Returns

Tuple consisting of coordinates of (r, ϕ) point in polar coordinate system.

Note

$\phi \in [0, 2\pi)$.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  const double x = 0.;
  const double y = 1.;
  std::cout « "(x, y) = (" « x « ", " « y « ")\n";
  const auto [r, p] = quile::cart2polar(x, y);
  std::cout « "(r, phi) = (" « r « ", " « p « ")\n";
  const auto [x2, y2] = quile::polar2cart(r, p);
  std::cout « "(x, y) = (" « x2 « ", " « y2 « ")\n";
}
```

Result:

```
(x, y) = (0, 1)
(r, phi) = (1, 1.5708)
(x, y) = (6.12323e-17, 1)
```

8.1.4.6 cart2spher()

```
template<std::floating_point T>
std::tuple<T, T, T> quile::cart2spher (
              T x,
              T y,
              T z )
```

`cart2spher` changes coordinate system from Cartesian to spherical.

Template Parameters

T	Argument type and base for return type (floating-point).

Parameters

x	x coordinate in Cartesian coordinate system.
y	y coordinate in Cartesian coordinate system.
z	y coordinate in Cartesian coordinate system.

Returns

Tuple consisting of coordinates of (r, θ, ϕ) point in spherical coordinate system.

Note

$\theta \in [0, \pi]$, $\phi \in [0, 2\pi)$.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  const double x = 1.;
  const double y = 0.;
  const double z = 0.;
  std::cout « "(x, y, z) = (" « x « ", " « y « ", " « z « ")\n";
  const auto [r, t, p] = quile::cart2spher(x, y, z);
  std::cout « "(r, theta, phi) = (" « r « ", " « t « ", " « p « ")\n";
  const auto [x2, y2, z2] = quile::spher2cart(r, t, p);
  std::cout « "(x, y, z) = (" « x2 « ", " « y2 « ", " « z2 « ")\n";
}
```

Result:

```
(x, y, z) = (1, 0, 0)
(r, theta, phi) = (1, 1.5708, 0)
(x, y, z) = (1, 0, 6.12323e-17)
```

8.1.4.7 chain_min()

```
template<typename T , std::size_t N>
chain<T, N> quile::chain_min (
              const domain< T, N > & d )
```

`chain_min` returns object of type `chain` filled at each `i` position with `d[i].min()` value.

Template Parameters

T	Chain base type.
N	Chain length.

Parameters

d	Domain.

Returns

Chain based on d.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  const quile::domain<int, 3> d{ quile::range{ 0, 5 },
                                 quile::range{ 1, 6 },
                                 quile::range{ 2, 7 } };
  const quile::chain<int, 3> c0{ 0, 1, 2 };
  const quile::chain<int, 3> c1{ quile::chain_min(d) };
  assert(c0 == c1);
}
```

Result (might be empty):

8.1.4.8 contains()

```
template<typename T , std::size_t N>
bool quile::contains (
             const domain< T, N > & d,
             const std::array< T, N > & p )
```

contains checks if argument p is within domain d and returns true in that case. Otherwise it returns false.

Template Parameters

T	Domain base type.
N	Domain dimensionality.

Parameters

d	Domain.
p	Point to be checked.

Returns

Boolean value describing whether point p is within domain d.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  quile::domain<double, 2> d{ quile::range{ 0., 1.  }, quile::range{ 0., 1.  } };
  std::array<double, 2> p{ .5, .5 };
  std::array<double, 2> q{ 2., 3.  };
  assert(contains(d, p));
  assert(!contains(d, q));
}
```

Result (might be empty):

8.1.4.9 cube()

```
template<typename T >
requires std::floating_point<T> std::integral<T> T quile::cube (
             T x )
```

`cube` returns third power of its argument.

Template Parameters

T	Argument and return type (floating-point or integer type).

Parameters

x	Argument to be raised to the third power.

Returns

Argument raised to the third power, i.e. x^3.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  long x = -80538738812075974;
  long y = 80435758145817515;
  long z = 12602123297335631;
  long k = 42;
  assert(quile::cube(x) + quile::cube(y) + quile::cube(z) == k);
}
```

Result (might be empty):

8.1.4.10 cumulative_probabilities()

```
template<typename G >
requires chromosome<G> selection_probabilities quile::cumulative_probabilities (
             const selection_probabilities_fn< G > & spf,
             const population< G > & p )
```

`cumulative_probabilities` serves for calculation of cumulative selection probabilities, which can be used later in roulette wheel selection or stochastic universal sampling.

Template Parameters

G	Some `genotype` specialization.

Parameters

spf	Selection probabilities function.
p	Population.

Returns

Cumulative selection probabilities.

Note

Cumulative selection probability for index `i` means sum of selection probabilities from index `0` up to `i`. Cumulative selection probability for last genotype in population is by definition equal to `1`.

It is guaranteed that last cumulative probability in returned container is equal to `1` without any numerical error.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout « p[i] « ":  " « sp[i] « ", " « cp[i] « '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
```

```
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

8.1.4.11 cut_n_crossfill()

```
template<typename G >
requires permutation_chromosome<G> population<G> quile::cut_n_crossfill (
             const G & g0,
             const G & g1 )
```

`cut_n_crossfill` is cut-and-crossfill recombination.

Template Parameters

Some	`G` specialization.

Parameters

g0	First parent.
g1	Secong parent.

Returns

Population containing two offspring genotypes.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const variation<G> v0{};
  const auto g0 = G::random();
  const auto g1 = G::random();
  const population<G> p0 = v0(g0, g1);
  assert(p0[0] == g0 && p0[1] == g1);
  const variation<G> v1{ swap_mutation<G> };
  const auto g2 = G::random();
  const auto g3 = G::random();
  const auto p1 = v1(population<G>{ g0, g1, g2, g3 });
  assert(p1.size() == 4);
  std::cout « p1[0] « '\n';
  std::cout « p1[1] « '\n';
  std::cout « p1[2] « '\n';
  std::cout « p1[3] « '\n';
  const variation<G> v2{ cut_n_crossfill<G> };
  try {
    [[maybe_unused]] const auto p2 = v2(population<G>{ g0, g1, g2 });
  } catch (...)  {
    std::cout « "Even number of genotypes is required.\n";
  }
  const variation<G> v3{ swap_mutation<G>, cut_n_crossfill<G> };
  const auto p3 = v3(g0, g1);
  std::cout « p3[0] « '\n';
  std::cout « p3[1] « '\n';
}
```

Result (might be different due to randomness):

```
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 5 0 1 6 3 4 2, 3 1 6 0 5 2 4
4 5 6 3 0 2 1
1 0 4 5 2 6 3
5 0 1 4 3 6 2
4 1 6 0 5 2 3
Even number of genotypes is required.
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
1 5 0 3 4 2 6
0 1 4 5 3 6 2
```

8.1.4.12 evolution() [1/2]

```
template<typename G >
requires chromosome<G> generations<G> quile::evolution (
             const variation< G > & v,
             const populate_0_fn< G > & p0,
             const populate_1_fn< G > & p1,
             const populate_2_fn< G > & p2,
             const termination_condition_fn< G > & tc,
             std::size_t generation_sz,
             std::size_t parents_sz,
             std::size_t max_history = 0 )
```

`evolution` executes evolutionary process.

Template Parameters

G	Some `genotype` specialization.

Parameters

v	Variation.
p0	Mechanism for first generation creation.
p1	Parents selection mechanism.
p2	Selection to the next generation mechanism.
tc	Termination condition.
generation_sz	Generation size.
parents_sz	Size of the parents multiset (should be even).
max_history	Number of generations kept in memory and returned to the caller. Default zero value is special and means keeping and returning all generations.

Returns

Generations produced during evolution (cf. `max_history` argument).

Example:

```
#include <cmath>
#include <cstddef>
#include <fstream>
#include <quile/quile.h>
using namespace quile;
using type = double;
const std::size_t dim = 2;
fitness
f(type x, type y)
{
  return -(x * x + y * y);
}
int
main()
{
  static const domain<type, dim> d0{ range{ -35., +35.  }, range{ -35., +35.  } };
  static const domain<type, 2 * dim> d{ self_adaptive_variation_domain(d0,
                                                                       .001) };
  using G = genotype<g_floating_point<type, 2 * dim, &d>;
  const fitness_function<G> ff = [](const G& g) {
    return f(g.value(0), g.value(1));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const auto p0 = random_population<constraints_satisfied<G>, G>;
  const auto p1 = stochastic_universal_sampling<G>{ fps };
  const auto p2 = adapter<G>(stochastic_universal_sampling<G>{ fps });
  const std::size_t generation_sz{ 1000 };
  const std::size_t parents_sz{ 42 };
  const auto tc = max_fitness_improvement_termination<G>(fd, 10, 0.05);
  const variation<G> v{ self_adaptive_mutation<G>(.002, .002),
                        arithmetic_recombination<G> };
  std::ofstream file{ "evolution.dat" };
  print(file, evolution<G>(v, p0, p1, p2, tc, generation_sz, parents_sz));
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
135605076150335e+01 1.087888184480514e+01]: -1.347580197669043e+01 (taken from d
atabase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
```

```
abase)
# Quile log: Fitness value for [-1.269892419837285e+00 5.600610920462099e-01 1.5
74992500337886e+01 5.375036543150264e+00]: -1.926295184784189e+00 (taken from da
tabase)
# Quile log: Fitness value for [-1.054741505834507e+00 6.756811722622036e-01 6.5
02789087426748e+00 5.428130674046989e+00]: -1.569024690679668e+00 (taken from da
tabase)
# Quile log: Fitness value for [4.518610900865383e-01 2.731480419554952e+00 1.29
4046196412222e+01 6.323685183509894e+00]: -7.665163727146291e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.895858669305523e+00 1.184854220819247e-01 7.4
78855481721993e+00 1.360918242461105e+01]: -3.608318889226842e+00 (taken from da
tabase)
# Quile log: Fitness value for [1.547716625980463e+00 1.824177922655826e+00 6.18
0341656735098e+00 1.119516183733311e+01]: -5.723051847841274e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.337581995874235e+00 2.944374431065633e-01 1.2
09534931775909e+01 1.598199667031198e+01]: -1.875819003590032e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.053386936859832e-01 -1.304040888209297e-01 9.1
67307432344389e+00 6.053271950893222e+00]: -6.655756379290628e-01 (taken from da
tabase)
# Quile log: Fitness value for [1.653184414281981e-01 1.735069055490136e+00 6.80
4112518955824e+00 1.256064803553919e+01]: -3.037794814395682e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [5.889449492784511e-01 7.395991791184260e-01 8.93
3746939996855e+00 7.688211146302455e+00]: -8.938630990332468e-01 (taken from dat
abase)
# Quile log: Fitness value for [3.731415294381901e-01 1.228299461433360e+00 6.93
0605184757385e+00 1.276664983031973e+01]: -1.647954167948954e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.772574284862156e-02 1.274248905031767e+00 4.66
5054866715833e+00 1.664047719017669e+01]: -1.631406077933000e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
```

Note

Evolution result is saved in separate file (not included).

8.1.4.13 evolution() [2/2]

```
template<typename G >
requires chromosome<G> generations<G> quile::evolution (
             const variation< G > v,
             const population< G > & first_generation,
             const populate_1_fn< G > & p1,
             const populate_2_fn< G > & p2,
             const termination_condition_fn< G > & tc,
             std::size_t parents_sz,
             std::size_t max_history = 0 )
```

`evolution` executes evolutionary process.

Template Parameters

G	Some `genotype` specialization.

Parameters

v	Variation.
first_generation	First generation.
p1	Parents selection mechanism.
p2	Selection to the next generation mechanism.
tc	Termination condition.
parents_sz	Size of the parents multiset (should be even).
max_history	Number of generations kept in memory and returned to the caller. Default zero value is special and means keeping and returning all generations.

Returns

Generations produced during evolution (cf. `max_history` argument).

8.1.4.14 exponential_ranking_selection()

```
probability quile::exponential_ranking_selection (
              std::size_t mu,
              std::size_t j )  [inline]
```

`exponential_ranking_selection` is exponential pressure mechanism for ranking selection.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
                         g.data().end(),
                         fitness{ 0.  },
                         [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout « p[i] « ":  " « sp[i] « ", " « cp[i] « '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

8.1.4.15 fitness_threshold_termination()

```
template<typename G >
termination_condition_fn<G> quile::fitness_threshold_termination (
            const fitness_db< G > & fd,
            fitness thr,
            fitness eps )
```

`fitness_threshold_termination` returns condition, which terminates algorithm if at least one genotype reaches `thr` fitness function value with absolute precision `eps`.

Template Parameters

G	Some `genotype` specialization.

Parameters

fd	Database intermediary object.
thr	Fitness function value to achieve.
eps	Fitness function value absolute precision.

Returns

Predicate terminating genetic algorithm after genotype reaching fitness function value `thr` with absolute precision `eps` is found.

8.1.4.16 fn_and()

```
auto quile::fn_and (
            const auto &...  fs )
```

`fn_and` is an higher-order function returning conjunction of its arguments.

Parameters

fs	Callable object returning Boolean value.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  const auto f0 = [](int i) { return i == 42; };
  const auto f1 = [](int i) { return i % 2 == 1; };
  const auto f = quile::fn_and(f0, f1);
  assert(!f(42));
}
```

Result (might be empty):

Note

`fn_and` can be useful to describe complex genetic algorithm termination conditions.

8.1.4.17 fn_or()

```
auto quile::fn_or (
            const auto &...  fs )
```

`fn_or` is an higher-order function returning disjunction of its arguments.

Parameters

fs	Callable object returning Boolean value.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  const auto f0 = [](int i) { return i == 42; };
  const auto f1 = [](int i) { return i % 2 == 1; };
  const auto f = quile::fn_or(f0, f1);
  assert(f(3));
  assert(f(42));
}
```

Result (might be empty):

Note

`fn_or` can be useful to describe complex genetic algorithm termination conditions.

8.1.4.18 Gaussian_mutation()

```
template<typename G >
requires floating_point_chromosome<G> auto quile::Gaussian_mutation (
             typename G::gene_t sigma,
             probability p )
```

`Gaussian_mutation` returns Gaussian mutation operator with standard deviation `sigma` and gene mutation probability `p`.

Template Parameters

G	Some `genotype` specialization.

Parameters

sigma	Standard deviation.
p	Gene mutation probability.

Returns

Gaussian mutation operator.

Example:

```
#include <cmath>
#include <cstddef>
#include <fstream>
```

```
#include <quile/quile.h>
using namespace quile;
using type = double;
const std::size_t dim = 2;
fitness f(type x, type y)
{ return std::cos(0.25 * std::hypot(x, y)) + e<type>; }
int main()
{
  static const domain<type, dim> d{ range{ -10., +10.  }, range{ -10., +10.  } };
  using G = genotype<g_floating_point<type, dim, &d>>;
  const fitness_function<G> ff = [](const G& g) {
    return f(g.value(0), g.value(1));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const auto p0 = random_population<constraints_satisfied<G>, G>;
  const auto p1 = stochastic_universal_sampling<G>{ fps };
  const auto p2 = adapter<G>(stochastic_universal_sampling<G>{ fps });
  const std::size_t generation_sz{ 1000 };
  const std::size_t parents_sz{ 42 };
  const auto tc = max_fitness_improvement_termination<G>(fd, 10, 0.05);
  const type sigma{ .2 };
  const variation<G> v{ Gaussian_mutation<G>(sigma, 1.)  };
  std::ofstream file{ "evolution.dat" };
  print(file, evolution<G>(v, p0, p1, p2, tc, generation_sz, parents_sz));
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
# Quile log: Fitness value for [9.242352316563167e-01 -3.574786511487158e-01]: 3
.687750907372339e+00 (taken from database)
# Quile log: Fitness value for [-1.888821403645154e-01 -6.110098946141196e-01]:
3.705527484695097e+00 (taken from database)
# Quile log: Fitness value for [4.212149307236714e-01 3.557886022656973e-01]: 3.
708796625493519e+00 (taken from database)
# Quile log: Fitness value for [8.901078102456889e-02 1.959430435973608e-01]: 3.
716834783974873e+00 (taken from database)
# Quile log: Fitness value for [9.212283782503832e-01 3.571412425783116e-01]: 3.
687930009993169e+00 (taken from database)
# Quile log: Fitness value for [-7.917489803195674e-01 -5.619060682896583e-01]:
3.688969755626429e+00 (taken from database)
# Quile log: Fitness value for [-1.675759785556307e+00 5.392032212650334e-01]: 3
.622993833216501e+00 (taken from database)
# Quile log: Fitness value for [8.190195997286950e-01 6.561916302794569e-01]: 3.
684060668236501e+00 (taken from database)
# Quile log: Fitness value for [4.212149307236714e-01 3.557886022656973e-01]: 3.
708796625493519e+00 (taken from database)
# Quile log: Fitness value for [9.212283782503832e-01 3.571412425783116e-01]: 3.
687930009993169e+00 (taken from database)
# Quile log: Fitness value for [9.013874579024681e-01 7.228921948493890e-01]: 3.
676850115103187e+00 (taken from database)
# Quile log: Fitness value for [-3.771090104741681e-02 1.095729444623430e+00]: 3
.680952504622896e+00 (taken from database)
# Quile log: Fitness value for [2.264215913063302e-01 7.233765712135378e-01]: 3.
700381103338832e+00 (taken from database)
# Quile log: Fitness value for [-3.161673915653220e-01 3.021505178133335e-01]: 3
.712311006133655e+00 (taken from database)
# Quile log: Fitness value for [-1.338369045118121e-01 5.359239885776699e-01]: 3
.708761758508559e+00 (taken from database)
# Quile log: Fitness value for [3.180415116859693e-01 9.318198757368721e-01]: 3.
688139525071710e+00 (taken from database)
# Quile log: Fitness value for [7.901983326764439e-01 6.883864972673884e-01]: 3.
684156164886526e+00 (taken from database)
# Quile log: Fitness value for [4.212149307236714e-01 3.557886022656973e-01]: 3.
708796625493519e+00 (taken from database)
# Quile log: Fitness value for [6.658705824982647e-01 7.977807910099131e-01]: 3.
684726256783000e+00 (taken from database)
# Quile log: Fitness value for [3.517218843695780e-02 -2.076542170882494e-01]: 3
.716895981275483e+00 (taken from database)
# Quile log: Fitness value for [-1.117894716842838e+00 7.023976324659850e-01]: 3
.664304194556285e+00 (taken from database)
# Quile log: Fitness value for [4.576094058152416e-01 2.796753117115472e-01]: 3.
709307014968485e+00 (taken from database)
# Quile log: Fitness value for [-5.596201452366003e-01 -4.050433739102992e-01]:
3.703405272008298e+00 (taken from database)
```

```
# Quile log: Fitness value for [-3.771090104741681e-02 1.095729444623430e+00]: 3
.680952504622896e+00 (taken from database)
# Quile log: Fitness value for [1.438392788433593e+00 -2.748896333027321e-01]: 3
.652010218018490e+00 (taken from database)
```

Note

Evolution result is saved in separate file (not included).

8.1.4.19 generational_survivor_selection()

```
template<typename G >
requires chromosome<G> population<G> quile::generational_survivor_selection (
            std::size_t sz,
            const population< G > & generation,
            const population< G > & offspring )
```

`generational_survivor_selection` is generational survivor selection mechanism.

Template Parameters

G	Some `genotype` specialization.

Parameters

sz	Generation / offspring size.
generation	Current generation.
offspring	Offspring.

Returns

Offspring.

Exceptions

std::invalid_argument	Exception is raised if `generation` size or `offspring` size is different from `sz`.

8.1.4.20 iota()

```
template<typename T , std::size_t N>
std::array<T, N> quile::iota (
            T t )
```

`iota` returns `std::array` container filled with arithmetic sequence of length `N`, with difference of 1 and starting from value `t`.

Template Parameters

T	Number type.
N	Returned container size.

Parameters

t	Starting value.

Returns

`std::array<T, N>` with consecutive numbers starting from value `t`.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  for (auto x :  quile::iota<int, 7>(0)) {
    std::cout « x « ' ';
  }
  std::cout « '\n';
}
```

Result:

```
0 1 2 3 4 5 6
```

8.1.4.21 linear_ranking_selection()

```
auto quile::linear_ranking_selection (
             double s )  [inline]
```

`linear_ranking_selection` creates linear pressure mechanism for ranking selection.

Parameters

s	So-called *s* parameter.

Returns

Linear pressure mechanism.

Example:

```
#include <iostream>
#include <numeric>
#include <quile/quile.h>
using namespace quile;
template<chromosome G>
fitness
fitness_fn(const G& g)
{
  return std::accumulate(g.data().begin(),
```

```
                        g.data().end(),
                        fitness{ 0.  },
                        [](fitness acc, auto x) { return acc - x * x; });
}
int
main()
{
  const std::size_t n = 8;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<quile::g_integer<int, n, &d>;
  const fitness_db<G> fd{ fitness_fn<G>, constraints_satisfied<G> };
  const auto p = random_population<constraints_satisfied<G>, G>(3);
  const selection_probabilities_fn<G> sp_fns[] = {
    fitness_proportional_selection<G>{ fd },
    ranking_selection<G>{ fd, linear_ranking_selection(2.)  },
    ranking_selection<G>{ fd, exponential_ranking_selection }
  };
  for (auto sp_fn :  sp_fns) {
    const selection_probabilities sp = sp_fn(p);
    const selection_probabilities cp = cumulative_probabilities(sp_fn, p);
    for (std::size_t i = 0; i < p.size(); ++i) {
      std::cout « p[i] « ":  " « sp[i] « ", " « cp[i] « '\n';
    }
  }
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
y on demand)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
6 8 4 1 6 3 6 4: 0.0124611, 0.0124611
5 5 1 5 5 7 1 8: 0.00311526, 0.0155763
1 1 1 1 3 4 0 9: 0.984424, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
6 8 4 1 6 3 6 4: 0.333333, 0.333333
5 5 1 5 5 7 1 8: 0, 0.333333
1 1 1 1 3 4 0 9: 0.666667, 1
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness values for population of size 3
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [5 5 1 5 5 7 1 8]: -215 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
# Quile log: Fitness value for [1 1 1 1 3 4 0 9]: -110 (taken from database)
# Quile log: Fitness value for [6 8 4 1 6 3 6 4]: -214 (taken from database)
```

```
6 8 4 1 6 3 6 4: 0.422319, 0.422319
5 5 1 5 5 7 1 8: 0, 0.422319
1 1 1 1 3 4 0 9: 0.577681, 1
```

8.1.4.22 max() [1/3]

```
fitness quile::max (
            const fitnesses & fs )  [inline]
```

`max` returns maximum fitness function value from `fs`.

Parameters

fs	Fitness function values container.

Returns

Maximum value.

Exceptions

std::runtime_error	Exception is raised if all `fs` elements are equal to `incalculable`.

8.1.4.23 max() [2/3]

```
template<typename G >
requires chromosome<G> fitnesses quile::max (
            const generations< G > & gs,
            const fitness_db< G > & ff )
```

`max` returns maximum fitness function values for each generation from `gs` and database intermediary object `ff`.

Template Parameters

G	Some `genotype` specialization.

Parameters

gs	Sequence of generations.
ff	Database intermediary object.

Returns

Maximum values corresponding to each generation.

Exceptions

std::runtime_error	Exception is raised if fitness function evaluates to `incalculable` for all genotypes from at least one generation.

8.1.4.24 max() [3/3]

```
template<typename G >
requires chromosome<G> fitness quile::max (
            const population< G > & p,
            const fitness_db< G > & ff )
```

`max` returns maximum fitness function value for population `p` and database intermediary object `ff`.

Template Parameters

G	Some `genotype` specialization.

Parameters

p	Population.
ff	Database intermediary object.

Returns

Maximum value.

Exceptions

std::runtime_error	Exception is raised if fitness function evaluates to `incalculable` for all genotypes from `p`.

8.1.4.25 max_fitness_improvement_termination()

```
template<typename G >
termination_condition_fn<G> quile::max_fitness_improvement_termination (
            const fitness_db< G > & ff,
            std::size_t n,
            double frac )
```

`max_fitness_improvement_termination` returns condition, which terminates algorithm after reaching fitness function *plateau*. The algorithm is terminated if after `n` last generations fitness function maximum has not improved *relatively* more than `frac` with respect to the whole evolutionary process.

Template Parameters

G	Some `genotype` specialization.

Parameters

ff	Database intermediary object.
n	Number of *last* generations.
frac	*Plateau flatness*.

Returns

Predicate terminating genetic algorithm after reaching fitness function *plateau*.

Exceptions

std::runtime_error	Exception is raised if fitness function evaluates to `incalculable` for all genotypes from at least one generation.

Note

This condition is not intended for use with `evolution` argument `max_history` different than `0`.

Example:

```
#include <cmath>
#include <cstddef>
#include <fstream>
#include <quile/quile.h>
using namespace quile;
using type = double;
const std::size_t dim = 2;
fitness
f(type x, type y)
{
  return -(x * x + y * y);
}
int
main()
{
  static const domain<type, dim> d0{ range{ -35., +35.  }, range{ -35., +35.  } };
  static const domain<type, 2 * dim> d{ self_adaptive_variation_domain(d0,
                                                                       .001) };
  using G = genotype<g_floating_point<type, 2 * dim, &d>;
  const fitness_function<G> ff = [](const G& g) {
    return f(g.value(0), g.value(1));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const auto p0 = random_population<constraints_satisfied<G>, G>;
  const auto p1 = stochastic_universal_sampling<G>{ fps };
  const auto p2 = adapter<G>(stochastic_universal_sampling<G>{ fps });
  const std::size_t generation_sz{ 1000 };
  const std::size_t parents_sz{ 42 };
  const auto tc = max_fitness_improvement_termination<G>(fd, 10, 0.05);
  const variation<G> v{ self_adaptive_mutation<G>(.002, .002),
                        arithmetic_recombination<G> };
  std::ofstream file{ "evolution.dat" };
  print(file, evolution<G>(v, p0, p1, p2, tc, generation_sz, parents_sz));
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
135605076150335e+01 1.087888184480514e+01]: -1.347580197669043e+01 (taken from d
```

```
atabase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.269892419837285e+00 5.600610920462099e-01 1.5
74992500337886e+01 5.375036543150264e+00]: -1.926295184784189e+00 (taken from da
tabase)
# Quile log: Fitness value for [-1.054741505834507e+00 6.756811722622036e-01 6.5
02789087426748e+00 5.428130674046989e+00]: -1.569024690679668e+00 (taken from da
tabase)
# Quile log: Fitness value for [4.518610900865383e-01 2.731480419554952e+00 1.29
4046196412222e+01 6.323685183509894e+00]: -7.665163727146291e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.895858669305523e+00 1.184854220819247e-01 7.4
78855481721993e+00 1.360918242461105e+01]: -3.608318889226842e+00 (taken from da
tabase)
# Quile log: Fitness value for [1.547716625980463e+00 1.824177922655826e+00 6.18
0341656735098e+00 1.119516183733311e+01]: -5.723051847841274e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.337581995874235e+00 2.944374431065633e-01 1.2
09534931775909e+01 1.598199667031198e+01]: -1.875819003590032e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.053386936859832e-01 -1.304040888209297e-01 9.1
67307432344389e+00 6.053271950893222e+00]: -6.655756379290628e-01 (taken from da
tabase)
# Quile log: Fitness value for [1.653184414281981e-01 1.735069055490136e+00 6.80
4112518955824e+00 1.256064803553919e+01]: -3.037794814395682e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [5.889449492784511e-01 7.395991791184260e-01 8.93
3746939996855e+00 7.688211146302455e+00]: -8.938630990332468e-01 (taken from dat
abase)
# Quile log: Fitness value for [3.731415294381901e-01 1.228299461433360e+00 6.93
0605184757385e+00 1.276664983031973e+01]: -1.647954167948954e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.772574284862156e-02 1.274248905031767e+00 4.66
5054866715833e+00 1.664047719017669e+01]: -1.631406077933000e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
```

Note

Evolution result is saved in separate file (not included).

8.1.4.26 max_fitness_improvement_termination_2()

```
template<typename G >
termination_condition_fn<G> quile::max_fitness_improvement_termination_2 (
            const fitness_db< G > & ff,
            std::size_t n,
            fitness delta )
```

`max_fitness_improvement_termination_2` returns condition, which terminates algorithm after reaching fitness function *plateau*. The algorithm is terminated if after `n` last generations fitness function maximum has not improved *absolutely* more than `delta` with respect to the whole evolutionary process.

Template Parameters

G	Some `genotype` specialization.

Parameters

ff	Database intermediary object.
n	Number of *last* generations.
delta	*Plateau flatness*.

Returns

Predicate terminating genetic algorithm after reaching fitness function *plateau*.

Exceptions

std::runtime_error	Exception is raised if fitness function evaluates to `incalculable` for all genotypes from at least one generation.

Note

This condition is not intended for use with `evolution` argument `max_history` different than `0`.

8.1.4.27 max_iterations_termination()

```
template<typename G >
termination_condition_fn<G> quile::max_iterations_termination (
            std::size_t max )
```

`max_iterations_termination` returns condition, which terminates algorithm after performing `max` loop iterations.

Template Parameters

G	Some `genotype` specialization.

Parameters

max	Number of genetic algorithm loop iteration to perform (number of generations).

Returns

Predicate terminating genetic algorithm after `max` iterations.

8.1.4.28 min() [1/3]

```
fitness quile::min (
            const fitnesses & fs )  [inline]
```

`min` returns minimum fitness function value from `fs`.

Parameters

fs	Fitness function values container.

Returns

Minimum value.

Exceptions

std::runtime_error	Exception is raised if all `fs` elements are equal to `incalculable`.

8.1.4.29 min() [2/3]

```
template<typename G >
requires chromosome<G> fitnesses quile::min (
            const generations< G > & gs,
            const fitness_db< G > & ff )
```

`min` returns minimum fitness function values for each generation from `gs` and database intermediary object `ff`.

Template Parameters

G	Some `genotype` specialization.

Parameters

gs	Sequence of generations.
ff	Database intermediary object.

Returns

Minimum values corresponding to each generation.

Exceptions

std::runtime_error	Exception is raised if fitness function evaluates to `incalculable` for all genotypes from at least one generation.

8.1.4.30 min() [3/3]

```
template<typename G >
requires chromosome<G> fitness quile::min (
            const population< G > & p,
            const fitness_db< G > & ff )
```

`min` returns minimum fitness function value for population `p` and database intermediary object `ff`.

Template Parameters

G	Some `genotype` specialization.

Parameters

p	Population.
ff	Database intermediary object.

Returns

Minimum value.

Exceptions

std::runtime_error	Exception is raised if fitness function evaluates to `incalculable` for all genotypes from `p`.

8.1.4.31 one_point_xover()

```
template<typename G >
requires floating_point_chromosome<G> integer_chromosome<G> binary_chromosome<G> population<G>
quile::one_point_xover (
            const G & g0,
            const G & g1 )
```

`one_point_xover` is one-point crossover recombination.

Template Parameters

Some	`G` specialization.

Parameters

g0	First parent.
g1	Secong parent.

Returns

Population containing two offspring genotypes.

Example:

```
#include <iostream>
#include <quile/quile.h>
#include <vector>
using namespace quile;
namespace {
template<chromosome G, typename...  Rs>
requires(recombination<Rs, G>&&...)  auto random_recombination(Rs&&...  rs)
{
  return
    [r = std::vector<recombination_fn<G>{ rs...  }](const G& g0, const G& g1) {
      return r[random_U<std::size_t>(0, r.size() - 1)](g0, g1);
    };
}
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<double, n>(0., 1.)  };
  using G = genotype<g_floating_point<double, n, &d>;
  const auto g0 = G::random();
  const auto g1 = G::random();
  std::cout « "Parents:\n";
  std::cout « g0 « '\n';
  std::cout « g1 « '\n';
  const recombination_fn<G> r = random_recombination<G>(
    single_arithmetic_recombination<G>, one_point_xover<G>);
  const population<G> p = r(g0, g1);
  std::cout « "Offspring:\n";
  std::cout « p[0] « '\n';
  std::cout « p[1] « '\n';
}
```

Result (might be different due to randomness):

```
Parents:
7.632938301095844e-01 5.380850537336410e-01 8.662196756244862e-01 2.385804670907
881e-01 3.992661030128286e-01 9.637379325762896e-01 8.015222496416907e-01 4.0781
02496612352e-01 6.495688681819360e-02 3.773104410660121e-01 4.470655570561451e-0
1 9.051977045719400e-01 3.754480928190284e-01 5.852131314403002e-01 2.7607692016
30901e-01 6.901601014454185e-01 1.876479488560436e-02 9.691397176956201e-01 2.92
6243459260146e-01 7.015068115063313e-01 5.096234113058886e-01 6.169321355472475e
-02 9.275202166839894e-01 3.908546545463924e-01 8.545031327328174e-01 2.57658215
2773367e-01 6.437327114578221e-01 9.651463910919517e-01 3.518883850377567e-01 9.
158175963353863e-01 5.548562472262517e-01 4.290178207550817e-01
3.567643357649279e-01 1.391264328060377e-01 6.003527872330220e-01 2.862740240145
400e-01 4.883058640665820e-01 9.146947899566816e-01 3.620078563266221e-01 8.7475
99088295999e-01 9.608142268763895e-01 9.611901113709199e-01 8.793766878269520e-0
1 4.577419434232913e-01 8.500066867665092e-02 2.611466104241356e-01 2.0453898352
02442e-02 7.230066340434387e-01 8.718111857170628e-01 5.250218644745700e-01 1.16
1269539668478e-02 1.131626122567013e-01 6.667633275191966e-01 6.620535971330396e
-01 4.047631606157035e-01 3.884035499872193e-01 3.895695861220623e-01 9.19629564
4505946e-01 2.274506717811672e-02 2.862834266068857e-01 9.566141505366095e-02 8.
618512812097814e-01 3.744678094228941e-01 5.132685497467671e-01
Offspring:
7.632938301095844e-01 5.380850537336410e-01 8.662196756244862e-01 2.385804670907
881e-01 3.992661030128286e-01 9.637379325762896e-01 8.015222496416907e-01 4.0781
02496612352e-01 6.495688681819360e-02 3.773104410660121e-01 4.470655570561451e-0
1 9.051977045719400e-01 3.754480928190284e-01 5.852131314403002e-01 2.7607692016
30901e-01 6.901601014454185e-01 1.876479488560436e-02 9.691397176956201e-01 2.92
6243459260146e-01 7.015068115063313e-01 5.096234113058886e-01 6.620535971330396e
-01 4.047631606157035e-01 3.884035499872193e-01 3.895695861220623e-01 9.19629564
4505946e-01 2.274506717811672e-02 2.862834266068857e-01 9.566141505366095e-02 8.
618512812097814e-01 3.744678094228941e-01 5.132685497467671e-01
3.567643357649279e-01 1.391264328060377e-01 6.003527872330220e-01 2.862740240145
400e-01 4.883058640665820e-01 9.146947899566816e-01 3.620078563266221e-01 8.7475
99088295999e-01 9.608142268763895e-01 9.611901113709199e-01 8.793766878269520e-0
1 4.577419434232913e-01 8.500066867665092e-02 2.611466104241356e-01 2.0453898352
02442e-02 7.230066340434387e-01 8.718111857170628e-01 5.250218644745700e-01 1.16
1269539668478e-02 1.131626122567013e-01 6.667633275191966e-01 6.169321355472475e
-02 9.275202166839894e-01 3.908546545463924e-01 8.545031327328174e-01 2.57658215
2773367e-01 6.437327114578221e-01 9.651463910919517e-01 3.518883850377567e-01 9.
158175963353863e-01 5.548562472262517e-01 4.290178207550817e-01
```

8.1.4.32 operator<<() [1/3]

```
template<typename G >
requires chromosome<G> std::ostream& quile::operator<< (
            std::ostream & os,
            const G & g )
```

operator<< prints genotype to the stream.

Parameters

os	Stream to use.
g	Genotype to be printed.

Returns

Reference to the os stream.

Example:

```
#include <cstddef>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  const std::size_t nb = 45;
  quile::genotype<quile::g_binary<nb>> gb{};
  std::cout « "binary:                  " « gb.random_reset() « '\n';
  std::cout « '\n';
  const std::size_t nfp = 4;
  static constexpr const auto dfp{ quile::uniform_domain<double, nfp>(-1.,
                                                                      1.)  };
  quile::genotype<quile::g_floating_point<double, nfp, &dfp>> gfp{};
  std::cout « "floating-point:  " « gfp.random_reset() « '\n';
  std::cout « '\n';
  const std::size_t ni = 32;
  static constexpr const auto di{ quile::uniform_domain<int, ni>(1, 42) };
  quile::genotype<quile::g_integer<int, ni, &di>> gi{};
  std::cout « "integer:                " « gi.random_reset() « '\n';
  std::cout « '\n';
  const std::size_t np = 33;
  quile::genotype<quile::g_permutation<int, np, 0>> gp;
  std::cout « "permutation:        " « gp.random_reset() « '\n';
}
```

Result (might be different due to randomness):

```
binary:         0 1 0 1 0 1 0 0 0 1 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 0
1 1 1 1 0 1 0 1 1 1 0 0 1

floating-point: -3.608911775535568e-01 -5.554340396732966e-01 -5.959424700993771
e-01 -6.539217781767727e-01

integer:        23 25 19 13 9 15 12 17 42 21 33 14 35 9 13 21 3 33 2 36 18 16 10
 12 33 9 17 27 33 11 7 5

permutation:    18 6 16 15 11 32 27 10 13 21 9 7 3 14 19 8 28 24 25 17 23 30 0 2
2 1 31 4 5 2 12 29 26 20
```

8.1.4.33 operator<<() [2/3]

```
template<typename G >
requires floating_point_chromosome<G> std::ostream& quile::operator<< (
             std::ostream & os,
             const G & g )
```

operator<< prints genotype to the stream.

Parameters

os	Stream to use.
g	Genotype to be printed.

Returns

Reference to the os stream.

Note

This overload is dedicated for floating-point genotypes.

Example:

```
#include <cstddef>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  const std::size_t nb = 45;
  quile::genotype<quile::g_binary<nb>> gb{};
  std::cout « "binary:                 " « gb.random_reset() « '\n';
  std::cout « '\n';
  const std::size_t nfp = 4;
  static constexpr const auto dfp{ quile::uniform_domain<double, nfp>(-1.,
                                                                      1.)  };
  quile::genotype<quile::g_floating_point<double, nfp, &dfp>> gfp{};
  std::cout « "floating-point:  " « gfp.random_reset() « '\n';
  std::cout « '\n';
  const std::size_t ni = 32;
  static constexpr const auto di{ quile::uniform_domain<int, ni>(1, 42) };
  quile::genotype<quile::g_integer<int, ni, &di>> gi{};
  std::cout « "integer:               " « gi.random_reset() « '\n';
  std::cout « '\n';
  const std::size_t np = 33;
  quile::genotype<quile::g_permutation<int, np, 0>> gp;
  std::cout « "permutation:        " « gp.random_reset() « '\n';
}
```

Result (might be different due to randomness):

```
binary:          0 1 0 1 0 1 0 0 0 1 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 0
1 1 1 1 0 1 0 1 1 1 0 0 1

floating-point: -3.608911775535568e-01 -5.554340396732966e-01 -5.959424700993771
e-01 -6.539217781767727e-01

integer:         23 25 19 13 9 15 12 17 42 21 33 14 35 9 13 21 3 33 2 36 18 16 10
 12 33 9 17 27 33 11 7 5

permutation:     18 6 16 15 11 32 27 10 13 21 9 7 3 14 19 8 28 24 25 17 23 30 0 2
2 1 31 4 5 2 12 29 26 20
```

8.1.4.34 operator<<() [3/3]

```
template<typename T >
std::ostream& quile::operator<< (
            std::ostream & os,
            const range< T > & r )
```

`operator<<` prints range to the stream.

Parameters

os	Stream to use.
r	Range to be printed.

Returns

Reference to the `os` stream.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  quile::range r{ 0., .42 };
  std::cout « r « '\n';
}
```

Result:

```
[0, 0.42]
```

8.1.4.35 polar2cart()

```
template<std::floating_point T>
std::tuple<T, T> quile::polar2cart (
            T r,
            T phi )
```

`polar2cart` changes coordinate system from polar to Cartesian.

Template Parameters

T	Argument type and base for return type (floating-point).

Parameters

r	r coordinate in polar coordinate system.
phi	ϕ coordinate in polar coordinate system.

Returns

Tuple consisting of coordinates of (x, y) point in Cartesian coordinate system.

Note

$\phi \in [0, 2\pi)$.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  const double x = 0.;
  const double y = 1.;
  std::cout « "(x, y) = (" « x « ", " « y « ")\n";
  const auto [r, p] = quile::cart2polar(x, y);
  std::cout « "(r, phi) = (" « r « ", " « p « ")\n";
  const auto [x2, y2] = quile::polar2cart(r, p);
  std::cout « "(x, y) = (" « x2 « ", " « y2 « ")\n";
}
```

Result:

```
(x, y) = (0, 1)
(r, phi) = (1, 1.5708)
(x, y) = (6.12323e-17, 1)
```

8.1.4.36 print()

```
template<typename G >
requires chromosome<G> void quile::print (
             std::ostream & os,
             const generations< G > & gs,
             const fitness_db< G > * fd = nullptr )
```

`print` prints to the stream `os` information about each genotype from each generation accompanied with optional information about fitness function value.

Template Parameters

G	Some `genotype` specialization.

Parameters

os	Stream to print on.
gs	Generations.
fd	Pointer to the fitness function values database. Default value is equal to `nullptr`.

Note

If `fd` is equal to `nullptr` then information about fitness function values is not printed.

Example:

```
#include <cmath>
#include <cstddef>
#include <fstream>
#include <quile/quile.h>
using namespace quile;
using type = double;
const std::size_t dim = 2;
fitness
f(type x, type y)
{
  return -(x * x + y * y);
}
int
main()
{
  static const domain<type, dim> d0{ range{ -35., +35.  }, range{ -35., +35.  } };
  static const domain<type, 2 * dim> d{ self_adaptive_variation_domain(d0,
                                                                       .001) };
  using G = genotype<g_floating_point<type, 2 * dim, &d>;
  const fitness_function<G> ff = [](const G& g) {
    return f(g.value(0), g.value(1));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const auto p0 = random_population<constraints_satisfied<G>, G>;
  const auto p1 = stochastic_universal_sampling<G>{ fps };
  const auto p2 = adapter<G>(stochastic_universal_sampling<G>{ fps });
  const std::size_t generation_sz{ 1000 };
  const std::size_t parents_sz{ 42 };
  const auto tc = max_fitness_improvement_termination<G>(fd, 10, 0.05);
  const variation<G> v{ self_adaptive_mutation<G>(.002, .002),
                        arithmetic_recombination<G> };
  std::ofstream file{ "evolution.dat" };
  print(file, evolution<G>(v, p0, p1, p2, tc, generation_sz, parents_sz));
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
135605076150335e+01 1.087888184480514e+01]: -1.347580197669043e+01 (taken from d
atabase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.269892419837285e+00 5.600610920462099e-01 1.5
74992500337886e+01 5.375036543150264e+00]: -1.926295184784189e+00 (taken from da
tabase)
# Quile log: Fitness value for [-1.054741505834507e+00 6.756811722622036e-01 6.5
02789087426748e+00 5.428130674046989e+00]: -1.569024690679668e+00 (taken from da
tabase)
# Quile log: Fitness value for [4.518610900865383e-01 2.731480419554952e+00 1.29
4046196412222e+01 6.323685183509894e+00]: -7.665163727146291e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.895858669305523e+00 1.184854220819247e-01 7.4
78855481721993e+00 1.360918242461105e+01]: -3.608318889226842e+00 (taken from da
tabase)
# Quile log: Fitness value for [1.547716625980463e+00 1.824177922655826e+00 6.18
0341656735098e+00 1.119516183733311e+01]: -5.723051847841274e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.337581995874235e+00 2.944374431065633e-01 1.2
09534931775909e+01 1.598199667031198e+01]: -1.875819003590032e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.053386936859832e-01 -1.304040888209297e-01 9.1
67307432344389e+00 6.053271950893222e+00]: -6.655756379290628e-01 (taken from da
tabase)
# Quile log: Fitness value for [1.653184414281981e-01 1.735069055490136e+00 6.80
4112518955824e+00 1.256064803553919e+01]: -3.037794814395682e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [5.889449492784511e-01 7.395991791184260e-01 8.93
3746939996855e+00 7.688211146302455e+00]: -8.938630990332468e-01 (taken from dat
abase)
```

```
# Quile log: Fitness value for [3.731415294381901e-01 1.228299461433360e+00 6.93
0605184757385e+00 1.276664983031973e+01]: -1.647954167948954e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.772574284862156e-02 1.274248905031767e+00 4.66
5054866715833e+00 1.664047719017669e+01]: -1.631406077933000e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
```

Note

Evolution result is saved in separate file (not included).

8.1.4.37 random_engine()

```
std::mt19937& quile::random_engine ( )  [inline]
```

`random_engine` returns pseudo-random number generator engine based on Mersenne Twister.

Returns

Reference to static object with Mersenne Twister engine `std::mt19937` initialized with `std::random_device{}()`.

Example:

```
#include <iostream>
#include <quile/quile.h>
#include <random>
int
main()
{
  const int n = 100;
  double res = 0;
  for (int i = 0; i < n; ++i) {
    res += std::bernoulli_distribution{ .5 }(quile::random_engine()) ?  +1 :  -1;
  }
  std::cout « res / n « '\n';
}
```

Result (might be different due to randomness):

```
0.26
```

8.1.4.38 random_N()

```
template<std::floating_point T>
T quile::random_N (
            T mean,
            T standard_deviation )
```

`random_N` returns random number from normal distribution with mean `mean` and standard deviation `standard_deviation`.

Template Parameters

T	Result type (floating-point).

Parameters

mean	Mean of normal distribution.
standard_deviation	Standard deviation of normal distribution.

Returns

Number drawn from normal distribution.

Example:

```
#include <cmath>
#include <iomanip>
#include <iostream>
#include <map>
#include <quile/quile.h>
using result = std::map<int, int>;
result
draw(double m, double sd, int sz)
{
  result res{};
  for (int i = 0; i < sz; ++i) {
    ++res[static_cast<int>(std::floor(quile::random_N(m, sd)))];
  }
  return res;
}
result
fill_gaps(result res)
{
  for (int i = res.begin()->first; i <= res.rbegin()->first; ++i) {
    res[i];
  }
  return res;
}
void
print(std::ostream& os, const result& r)
{
  const auto f = [&](int n, int sz) {
    os « std::setw(3) « n « ":  ";
    for (int i = 0; i < sz; ++i) {
      os « '*';
    }
    os « '\n';
  };
  for (auto [n, sz] :  r) {
    f(n, sz);
  }
}
int
main()
{
  print(std::cout, fill_gaps(draw(0., 5., 200)));
}
```

Result (might be different due to randomness):

```
-10: ***
 -9: ****
 -8: *********
 -7: *******
 -6: *****
 -5: *********
 -4: ******************
 -3: ******************
 -2: ********
 -1: **********************
  0: ****************
  1: ****************
```

```
 2: ***************
 3: ***********
 4: ********
 5: ********
 6: ************
 7: ****
 8: **
 9:
10: ***
11: *
12: *
13: **
```

8.1.4.39 random_population()

```
template<auto C, typename G >
requires genotype_constraints<decltype(C), G>&& chromosome<G> population<G> quile::random_↩
population (
             std::size_t lambda )
```

`random_population` returns random population of size `lambda`, where each member genotype satisfies predicate `C`.

Template Parameters

C	Proper genotype predicate.
G	Some `genotype` specialization.

Parameters

lambda	Size of returned population.

Returns

Random population.

Example:

```
#include <cassert>
#include <iostream>
#include <quile/quile.h>
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ quile::uniform_domain<int, n>(0, 9) };
  using G = quile::genotype<quile::g_integer<int, n, &d>>;
  const quile::populate_0_fn<G> generator =
    quile::random_population<quile::constraints_satisfied<G>, G>;
  const quile::population<G> p = generator(10);
  assert(p.size() == 10);
  for (auto& g :  p) {
    std::cout « g « '\n';
  }
}
```

Result (might be different due to randomness):

```
7 7 0 5 2 6 2 6 3 0 9 6 2 1 8 2 6 8 2 1 6 2 5 5 5 0 7 1 4 9 8 6
7 0 4 1 0 3 4 2 1 0 9 9 8 7 9 0 3 2 8 2 9 5 9 8 2 2 1 0 4 3 4 9
5 6 5 8 0 5 3 1 9 0 0 9 2 5 5 9 6 6 1 6 7 3 2 7 6 4 2 5 3 7 8 6
8 2 5 0 2 4 2 4 5 0 8 1 6 6 4 7 8 5 6 5 6 3 1 1 5 8 3 6 6 1 0 9
9 3 5 9 1 5 6 2 7 3 6 1 8 6 2 9 0 6 3 1 8 3 0 7 9 6 3 9 6 2 0 4
5 1 4 6 1 6 4 6 4 0 1 6 0 5 3 0 7 3 1 9 2 6 6 8 5 9 0 7 0 3 4 0
9 4 0 2 6 8 3 1 0 4 4 9 3 1 4 9 8 8 6 7 8 0 2 6 8 7 0 3 9 2 2 8
4 0 1 1 5 9 4 9 6 4 3 5 2 6 5 0 0 5 8 9 0 7 3 6 9 9 6 2 3 3 1 8
5 8 0 4 2 1 5 8 3 6 1 1 0 0 0 3 6 2 4 1 0 4 0 5 5 6 5 7 1 5 5 5
6 8 6 6 3 1 1 7 5 0 7 9 9 4 4 0 7 4 0 4 6 7 9 2 7 9 7 7 3 2 2 7
```

8.1.4.40 random_reset()

```
template<typename G >
requires floating_point_chromosome<G> integer_chromosome<G> binary_chromosome<G> auto quile↩
::random_reset (
            probability p )
```

`random_reset` returns random reset mutation with parameter `p`.

Template Parameters

G	Some `genotype` specialization.

Parameters

p	Gene mutation probability.

Returns

Random reset mutation operator.

8.1.4.41 random_U()

```
template<typename T >
T quile::random_U (
            T a,
            T b )
```

`random_U` returns random number from uniform distribution:

- from interval $[a, b]_{\mathbb{R}}$ for floating-point type `T`
- from set $\{a, b\}$ for Boolean type `T`
- from interval $[a, b]_{\mathbb{Z}}$ for integer type `T`

Template Parameters

T	Return type.

Parameters

a	Parameter describing aforementioned interval or set.
b	Parameter describing aforementioned interval or set.

Returns

Value drawn from uniform distribution.

Note

For floating-point types overflow may occur for `std::nextafter(b, std::numeric_limits<T>max())` − `a` (cf. N4861, 26.6.8.2.2).

Example:

```
#include <cmath>
#include <iomanip>
#include <iostream>
#include <map>
#include <quile/quile.h>
using result = std::map<int, int>;
result
draw(int min, int max, int sz)
{
  result res{};
  for (int i = 0; i < sz; ++i) {
    ++res[std::floor(quile::random_U(min, max))];
  }
  return res;
}
result
fill_gaps(result res)
{
  for (int i = res.begin()->first; i <= res.rbegin()->first; ++i) {
    res[i];
  }
  return res;
}
void
print(std::ostream& os, const result& r)
{
  const auto f = [&](int n, int sz) {
    os « std::setw(3) « n « ":  ";
    for (int i = 0; i < sz; ++i) {
      os « '*';
    }
    os « '\n';
  };
  for (auto [n, sz] :  r) {
    f(n, sz);
  }
}
int
main()
{
  print(std::cout, fill_gaps(draw(-10, 10, 200)));
}
```

Result (might be different due to randomness):

```
-10: ***********
 -9: ********
 -8: *********
 -7: *******
 -6: ******
```

```
 -5: *********
 -4: ************
 -3: *********
 -2: **********
 -1: ****************
  0: *******
  1: **********
  2: ************
  3: ***********
  4: ********
  5: **************
  6: *************
  7: ******
  8: **
  9: *********
 10: ***********
```

8.1.4.42 select_calculable()

```
fitnesses quile::select_calculable (
            const fitnesses & fs,
            bool require_nonempty_result = false )  [inline]
```

`select_calculable` returns container with fitness function values equal to `fs`, but with `incalculable` entries skipped.

Parameters

fs	Fitness function values container.
require_nonempty_result	Flag for possible exception.

Returns

Container with values different than `incalculable`.

Exceptions

std::runtime_error	Exception is raised if `require_nonempty_result` is `true` and returned container is empty.

8.1.4.43 select_different_than()

```
template<typename C , typename T >
C quile::select_different_than (
            const C & c,
            T t,
            bool require_nonempty_result )
```

`select_different_than` returns copy of container `c` with deleted elements equal to `t`.

Template Parameters

C	Container type.
T	Type convertible to C container element type.

Parameters

c	Container.
t	Element to be dropped in returned container.
require_nonempty_result	Flag for possible exception.

Returns

Container with values different than `t`.

Exceptions

std::runtime_error	Exception is raised if `require_nonempty_result` is `true` and returned container is empty.

8.1.4.44 self_adaptive_mutation()

```
template<typename G >
requires floating_point_chromosome<G> auto quile::self_adaptive_mutation (
             typename G::gene_t a0,
             typename G::gene_t a1 )
```

`self_adaptive_mutation` returns self adaptive mutation operator with parameters `a0` and `a1`.

Template Parameters

G	Some `genotype` specialization.

Parameters

a0	Self adaptive mutation parameter.
a1	Self adaptive mutation parameter.

Returns

Self adaptive mutation operator.

Note

Due to documentation processing problem the above template declaration is incorrect. Corrected declaration:

```
template<typename G>
requires floating_point_chromosome<G> && (G::size() % 2 == 0)
auto self_adaptive_mutation(typename G::gene_t a0, typename G::gene_t a1)
```

Example:

```
#include <cmath>
#include <cstddef>
#include <fstream>
#include <quile/quile.h>
using namespace quile;
using type = double;
const std::size_t dim = 2;
fitness
f(type x, type y)
{
  return -(x * x + y * y);
}
int
main()
{
  static const domain<type, dim> d0{ range{ -35., +35.  }, range{ -35., +35.  } };
  static const domain<type, 2 * dim> d{ self_adaptive_variation_domain(d0,
                                                                       .001) };
  using G = genotype<g_floating_point<type, 2 * dim, &d>;
  const fitness_function<G> ff = [](const G& g) {
    return f(g.value(0), g.value(1));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const auto p0 = random_population<constraints_satisfied<G>, G>;
  const auto p1 = stochastic_universal_sampling<G>{ fps };
  const auto p2 = adapter<G>(stochastic_universal_sampling<G>{ fps });
  const std::size_t generation_sz{ 1000 };
  const std::size_t parents_sz{ 42 };
  const auto tc = max_fitness_improvement_termination<G>(fd, 10, 0.05);
  const variation<G> v{ self_adaptive_mutation<G>(.002, .002),
                        arithmetic_recombination<G> };
  std::ofstream file{ "evolution.dat" };
  print(file, evolution<G>(v, p0, p1, p2, tc, generation_sz, parents_sz));
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
135605076150335e+01 1.087888184480514e+01]: -1.347580197669043e+01 (taken from d
atabase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.269892419837285e+00 5.600610920462099e-01 1.5
74992500337886e+01 5.375036543150264e+00]: -1.926295184784189e+00 (taken from da
tabase)
# Quile log: Fitness value for [-1.054741505834507e+00 6.756811722622036e-01 6.5
02789087426748e+00 5.428130674046989e+00]: -1.569024690679668e+00 (taken from da
tabase)
# Quile log: Fitness value for [4.518610900865383e-01 2.731480419554952e+00 1.29
4046196412222e+01 6.323685183509894e+00]: -7.665163727146291e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.895858669305523e+00 1.184854220819247e-01 7.4
78855481721993e+00 1.360918242461105e+01]: -3.608318889226842e+00 (taken from da
tabase)
# Quile log: Fitness value for [1.547716625980463e+00 1.824177922655826e+00 6.18
0341656735098e+00 1.119516183733311e+01]: -5.723051847841274e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.337581995874235e+00 2.944374431065633e-01 1.2
09534931775909e+01 1.598199667031198e+01]: -1.875819003590032e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.053386936859832e-01 -1.304040888209297e-01 9.1
67307432344389e+00 6.053271950893222e+00]: -6.655756379290628e-01 (taken from da
tabase)
```

```
# Quile log: Fitness value for [1.653184414281981e-01 1.735069055490136e+00 6.80
4112518955824e+00 1.256064803553919e+01]: -3.037794814395682e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [5.889449492784511e-01 7.395991791184260e-01 8.93
3746939996855e+00 7.688211146302455e+00]: -8.938630990332468e-01 (taken from dat
abase)
# Quile log: Fitness value for [3.731415294381901e-01 1.228299461433360e+00 6.93
0605184757385e+00 1.276664983031973e+01]: -1.647954167948954e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
622507951673398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.772574284862156e-02 1.274248905031767e+00 4.66
5054866715833e+00 1.664047719017669e+01]: -1.631406077933000e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
```

Note

Evolution result is saved in separate file (not included).

8.1.4.45 self_adaptive_variation_domain()

```
template<typename T , std::size_t N>
requires constexpr std::floating_point<T> domain<T, 2 * N> quile::self_adaptive_variation_↩
domain (
            const domain< T, N > & d,
            T lo )  [constexpr]
```

`self_adaptive_variation_domain` creates domain for self-adaptive mutation based on domain for ordinary variation.

Template Parameters

T	Domain base type.
N	Domain dimensionality.

Parameters

d	Domain to use.
lo	Minimum value for σ_i on each direction.

Returns

Domain with dimensionality of `2 * N`, where first part is equal to `d` and the second one consits of ranges of form `range{ lo, 0.5 * std::max(std::fabs(d[i].min()), std↩ ::fabs(d[i].max())) }`.

Example:

```
#include <cmath>
#include <cstddef>
#include <fstream>
#include <quile/quile.h>
using namespace quile;
using type = double;
const std::size_t dim = 2;
fitness
f(type x, type y)
{
  return -(x * x + y * y);
}
int
main()
{
  static const domain<type, dim> d0{ range{ -35., +35.  }, range{ -35., +35.  } };
  static const domain<type, 2 * dim> d{ self_adaptive_variation_domain(d0,
                                                                       .001) };
  using G = genotype<g_floating_point<type, 2 * dim, &d>;
  const fitness_function<G> ff = [](const G& g) {
    return f(g.value(0), g.value(1));
  };
  const fitness_db<G> fd{ ff, constraints_satisfied<G> };
  const fitness_proportional_selection<G> fps{ fd };
  const auto p0 = random_population<constraints_satisfied<G>, G>;
  const auto p1 = stochastic_universal_sampling<G>{ fps };
  const auto p2 = adapter<G>(stochastic_universal_sampling<G>{ fps });
  const std::size_t generation_sz{ 1000 };
  const std::size_t parents_sz{ 42 };
  const auto tc = max_fitness_improvement_termination<G>(fd, 10, 0.05);
  const variation<G> v{ self_adaptive_mutation<G>(.002, .002),
                        arithmetic_recombination<G> };
  std::ofstream file{ "evolution.dat" };
  print(file, evolution<G>(v, p0, p1, p2, tc, generation_sz, parents_sz));
}
```

Result (might be different due to randomness):

```
[Output to this point was skipped.]
135605076150335e+01 1.087888184480514e+01]: -1.347580197669043e+01 (taken from d
atabase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.269892419837285e+00 5.600610920462099e-01 1.5
74992500337886e+01 5.375036543150264e+00]: -1.926295184784189e+00 (taken from da
tabase)
# Quile log: Fitness value for [-1.054741505834507e+00 6.756811722622036e-01 6.5
02789087426748e+00 5.428130674046989e+00]: -1.569024690679668e+00 (taken from da
tabase)
# Quile log: Fitness value for [4.518610900865383e-01 2.731480419554952e+00 1.29
4046196412222e+01 6.323685183509894e+00]: -7.665163727146291e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.895858669305523e+00 1.184854220819247e-01 7.4
78855481721993e+00 1.360918242461105e+01]: -3.608318889226842e+00 (taken from da
tabase)
# Quile log: Fitness value for [1.547716625980463e+00 1.824177922655826e+00 6.18
0341656735098e+00 1.119516183733311e+01]: -5.723051847841274e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.337581995874235e+00 2.944374431065633e-01 1.2
09534931775909e+01 1.598199667031198e+01]: -1.875819003590032e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.053386936859832e-01 -1.304040888209297e-01 9.1
67307432344389e+00 6.053271950893222e+00]: -6.655756379290628e-01 (taken from da
tabase)
# Quile log: Fitness value for [1.653184414281981e-01 1.735069055490136e+00 6.80
4112518955824e+00 1.256064803553919e+01]: -3.037794814395682e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
# Quile log: Fitness value for [5.889449492784511e-01 7.395991791184260e-01 8.93
3746939996855e+00 7.688211146302455e+00]: -8.938630990332468e-01 (taken from dat
abase)
# Quile log: Fitness value for [3.731415294381901e-01 1.228299461433360e+00 6.93
```

```
0605184757385e+00 1.276664983031973e+01]: -1.647954167948954e+00 (taken from dat
abase)
# Quile log: Fitness value for [-1.597083534744854e-01 1.141656041719891e+00 3.3
62250795167398e+00 1.690204989673924e+01]: -1.328885275765060e+00 (taken from da
tabase)
# Quile log: Fitness value for [8.772574284862156e-02 1.274248905031767e+00 4.66
5054866715833e+00 1.664047719017669e+01]: -1.631406077933000e+00 (taken from dat
abase)
# Quile log: Fitness value for [2.011509120541781e+00 1.236125851718313e+00 1.31
1468833172379e+01 3.248770673396150e+00]: -5.574176063309094e+00 (taken from dat
abase)
```

Note

Evolution result is saved in separate file (not included).

8.1.4.46 single_arithmetic_recombination()

```
template<typename G >
requires floating_point_chromosome<G> population<G> quile::single_arithmetic_recombination (
            const G & g0,
            const G & g1 )
```

`single_arithmetic_recombination` is single arithmetic recombination.

Template Parameters

Some	G specialization.

Parameters

g0	First parent.
g1	Secong parent.

Returns

Population containing two offspring genotypes.

Example:

```
#include <iostream>
#include <quile/quile.h>
#include <vector>
using namespace quile;
namespace {
template<chromosome G, typename...  Rs>
requires(recombination<Rs, G>&&...)  auto random_recombination(Rs&&...  rs)
{
  return
    [r = std::vector<recombination_fn<G>{ rs...  }](const G& g0, const G& g1) {
      return r[random_U<std::size_t>(0, r.size() - 1)](g0, g1);
    };
}
}
int
main()
{
  const std::size_t n = 32;
```

```
  static constexpr const auto d{ uniform_domain<double, n>(0., 1.)  };
  using G = genotype<g_floating_point<double, n, &d>;
  const auto g0 = G::random();
  const auto g1 = G::random();
  std::cout « "Parents:\n";
  std::cout « g0 « '\n';
  std::cout « g1 « '\n';
  const recombination_fn<G> r = random_recombination<G>(
    single_arithmetic_recombination<G>, one_point_xover<G>);
  const population<G> p = r(g0, g1);
  std::cout « "Offspring:\n";
  std::cout « p[0] « '\n';
  std::cout « p[1] « '\n';
}
```

Result (might be different due to randomness):

```
Parents:
7.632938301095844e-01 5.380850537336410e-01 8.662196756244862e-01 2.385804670907
881e-01 3.992661030128286e-01 9.637379325762896e-01 8.015222496416907e-01 4.0781
02496612352e-01 6.495688681819360e-02 3.773104410660121e-01 4.470655570561451e-0
1 9.051977045719400e-01 3.754480928190284e-01 5.852131314403002e-01 2.7607692016
30901e-01 6.901601014454185e-01 1.876479488560436e-02 9.691397176956201e-01 2.92
6243459260146e-01 7.015068115063313e-01 5.096234113058886e-01 6.169321355472475e
-02 9.275202166839894e-01 3.908546545463924e-01 8.545031327328174e-01 2.57658215
2773367e-01 6.437327114578221e-01 9.651463910919517e-01 3.518883850377567e-01 9.
158175963353863e-01 5.548562472262517e-01 4.290178207550817e-01
3.567643357649279e-01 1.391264328060377e-01 6.003527872330220e-01 2.862740240145
400e-01 4.883058640665820e-01 9.146947899566816e-01 3.620078563266221e-01 8.7475
99088295999e-01 9.608142268763895e-01 9.611901113709199e-01 8.793766878269520e-0
1 4.577419434232913e-01 8.500066867665092e-02 2.611466104241356e-01 2.0453898352
02442e-02 7.230066340434387e-01 8.718111857170628e-01 5.250218644745700e-01 1.16
1269539668478e-02 1.131626122567013e-01 6.667633275191966e-01 6.620535971330396e
-01 4.047631606157035e-01 3.884035499872193e-01 3.895695861220623e-01 9.19629564
4505946e-01 2.274506717811672e-02 2.862834266068857e-01 9.566141505366095e-02 8.
618512812097814e-01 3.744678094228941e-01 5.132685497467671e-01
Offspring:
7.632938301095844e-01 5.380850537336410e-01 8.662196756244862e-01 2.385804670907
881e-01 3.992661030128286e-01 9.637379325762896e-01 8.015222496416907e-01 4.0781
02496612352e-01 6.495688681819360e-02 3.773104410660121e-01 4.470655570561451e-0
1 9.051977045719400e-01 3.754480928190284e-01 5.852131314403002e-01 2.7607692016
30901e-01 6.901601014454185e-01 1.876479488560436e-02 9.691397176956201e-01 2.92
6243459260146e-01 7.015068115063313e-01 5.096234113058886e-01 6.620535971330396e
-01 4.047631606157035e-01 3.884035499872193e-01 3.895695861220623e-01 9.19629564
4505946e-01 2.274506717811672e-02 2.862834266068857e-01 9.566141505366095e-02 8.
618512812097814e-01 3.744678094228941e-01 5.132685497467671e-01
3.567643357649279e-01 1.391264328060377e-01 6.003527872330220e-01 2.862740240145
400e-01 4.883058640665820e-01 9.146947899566816e-01 3.620078563266221e-01 8.7475
99088295999e-01 9.608142268763895e-01 9.611901113709199e-01 8.793766878269520e-0
1 4.577419434232913e-01 8.500066867665092e-02 2.611466104241356e-01 2.0453898352
02442e-02 7.230066340434387e-01 8.718111857170628e-01 5.250218644745700e-01 1.16
1269539668478e-02 1.131626122567013e-01 6.667633275191966e-01 6.169321355472475e
-02 9.275202166839894e-01 3.908546545463924e-01 8.545031327328174e-01 2.57658215
2773367e-01 6.437327114578221e-01 9.651463910919517e-01 3.518883850377567e-01 9.
158175963353863e-01 5.548562472262517e-01 4.290178207550817e-01
```

8.1.4.47 spher2cart()

```
template<std::floating_point T>
std::tuple<T, T, T> quile::spher2cart (
            T r,
            T theta,
            T phi )
```

`spher2cart` changes coordinate system from spherical to Cartesian.

Template Parameters

T	Argument type and base for return type (floating-point).

Parameters

r	r coordinate in spherical coordinate system.
theta	θ coordinate in spherical coordinate system.
phi	ϕ coordinate in spherical coordinate system.

Returns

Tuple consisting of coordinates of (x, y, z) point in Cartesian coordinate system.

Note

$\theta \in [0, \pi]$, $\phi \in [0, 2\pi)$.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  const double x = 1.;
  const double y = 0.;
  const double z = 0.;
  std::cout « "(x, y, z) = (" « x « ", " « y « ", " « z « ")\n";
  const auto [r, t, p] = quile::cart2spher(x, y, z);
  std::cout « "(r, theta, phi) = (" « r « ", " « t « ", " « p « ")\n";
  const auto [x2, y2, z2] = quile::spher2cart(r, t, p);
  std::cout « "(x, y, z) = (" « x2 « ", " « y2 « ", " « z2 « ")\n";
}
```

Result:

```
(x, y, z) = (1, 0, 0)
(r, theta, phi) = (1, 1.5708, 0)
(x, y, z) = (1, 0, 6.12323e-17)
```

8.1.4.48 square()

```
template<typename T >
requires std::floating_point<T> std::integral<T> T quile::square (
            T x )
```

`square` returns second power of its argument.

Template Parameters

T	Argument and return type (floating-point or integer type).

Parameters

x	Argument to be raised to the second power.

Returns

Argument raised to the second power, i.e. x^2.

Example:

```
#include <cassert>
#include <quile/quile.h>
int
main()
{
  assert(quile::square(3) + quile::square(4) == quile::square(5));
}
```

Result (might be empty):

8.1.4.49 stochastic_mutation()

```
template<typename G >
requires chromosome<G> auto quile::stochastic_mutation (
             const mutation_fn< G > & m,
             probability p )
```

`stochastic_mutation` creates stochastic mutation consisting of `m` applied with probability `p`.

Template Parameters

G	Some `genotype` specialization.

Parameters

m	Mutation.
p	Probability.

Returns

Stochastic mutation.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  const std::size_t n = 32;
  using G = genotype<quile::g_binary<n»;
  mutation_fn<G> m = stochastic_mutation<G>(swap_mutation<G>, 0.5);
```

```
  recombination_fn<G> r = stochastic_recombination<G>(one_point_xover<G>, 0.5);
  const auto g0 = G::random();
  const auto g1 = G::random();
  std::cout « "genotypes:  \n" « g0 « '\n' « g1 « '\n';
  const population<G> p0{ m(g0)[0], m(g1)[0] };
  std::cout « "after mutation:  \n" « p0[0] « '\n' « p0[1] « '\n';
  const population<G> p1{ r(g0, g1) };
  std::cout « "after recombination:  \n" « p1[0] « '\n' « p1[1] « '\n';
}
```

Result (might be different due to randomness):

```
genotypes:
1 0 1 1 0 0 1 0 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 1
0 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0
after mutation:
1 0 1 0 0 0 1 0 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 1 0 1 0 0 1 1 0 1
0 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0
after recombination:
1 0 1 1 0 0 1 0 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 1
0 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0
```

8.1.4.50 stochastic_recombination()

```
template<typename G >
requires chromosome<G> auto quile::stochastic_recombination (
             const recombination_fn< G > & r,
             probability p )
```

`stochastic_recombination` creates stochastic recombination consisting of `r` applied with probability `p`.

Template Parameters

G	Some `genotype` specialization.

Parameters

r	Recombination.
p	Probability.

Returns

Stochastic recombination.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  const std::size_t n = 32;
  using G = genotype<quile::g_binary<n»;
  mutation_fn<G> m = stochastic_mutation<G>(swap_mutation<G>, 0.5);
  recombination_fn<G> r = stochastic_recombination<G>(one_point_xover<G>, 0.5);
  const auto g0 = G::random();
  const auto g1 = G::random();
  std::cout « "genotypes:  \n" « g0 « '\n' « g1 « '\n';
  const population<G> p0{ m(g0)[0], m(g1)[0] };
```

```
  std::cout « "after mutation:  \n" « p0[0] « '\n' « p0[1] « '\n';
  const population<G> p1{ r(g0, g1) };
  std::cout « "after recombination:  \n" « p1[0] « '\n' « p1[1] « '\n';
}
```

Result (might be different due to randomness):

```
genotypes:
1 0 1 1 0 0 1 0 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 1
0 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0
after mutation:
1 0 1 0 0 0 1 0 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 1 0 1 0 0 1 1 0 1
0 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0
after recombination:
1 0 1 1 0 0 1 0 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 1
0 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0
```

8.1.4.51 success()

```
bool quile::success (
            probability success_probability )  [inline]
```

`success` returns true with probability `success_probability` and false with probability `1 - success_probability`, i.e. it implements Bernoulli distribution $\mathrm{B}(1, \mathrm{success\_probability})$.

Parameters

success_probability	Probability of returning `true` value.

Returns

Logic value drawn from $\mathrm{B}(1, \mathrm{success\_probability})$.

Example:

```
#include <iostream>
#include <quile/quile.h>
#include <tuple>
using result = std::tuple<int, int>;
result
draw(quile::probability p, int sz)
{
  int t{ 0 };
  int f{ 0 };
  for (int i = 0; i < sz; ++i) {
    if (quile::success(p)) {
      ++t;
    } else {
      ++f;
    }
  }
  return { t, f };
}
void
print(std::ostream& os, quile::probability p, const result& r)
{
  const auto f = [&](char c, int sz) {
    os « c « ":  ";
    for (int i = 0; i < sz; ++i) {
      os « '*';
    }
    os « '\n';
  };
  os « "probability:  " « p « '\n';
  f('t', std::get<0>(r));
```

```
  f('f', std::get<1>(r));
  os « '\n';
}
int
main()
{
  const int n = 40;
  for (auto p :  { 0., .25, .5, .75, 1.  }) {
    print(std::cout, p, draw(p, n));
  }
}
```

Result (might be different due to randomness):

```
probability: 0
t:
f: ****************************************

probability: 0.25
t: ********
f: ********************************

probability: 0.5
t: ************************
f: ****************

probability: 0.75
t: ******************************
f: **********

probability: 1
t: ****************************************
f:
```

8.1.4.52 swap_mutation()

```
template<typename G >
requires uniform_chromosome<G> population<G> quile::swap_mutation (
             const G & g )
```

`swap_mutation` is swap mutation.

Template Parameters

G	Some `genotype` specialization.

Parameters

g	Genotype.

Returns

Population containing mutated genotype.

Example:

```
#include <cassert>
#include <iostream>
```

```
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const variation<G> v0{};
  const auto g0 = G::random();
  const auto g1 = G::random();
  const population<G> p0 = v0(g0, g1);
  assert(p0[0] == g0 && p0[1] == g1);
  const variation<G> v1{ swap_mutation<G> };
  const auto g2 = G::random();
  const auto g3 = G::random();
  const auto p1 = v1(population<G>{ g0, g1, g2, g3 });
  assert(p1.size() == 4);
  std::cout « p1[0] « '\n';
  std::cout « p1[1] « '\n';
  std::cout « p1[2] « '\n';
  std::cout « p1[3] « '\n';
  const variation<G> v2{ cut_n_crossfill<G> };
  try {
    [[maybe_unused]] const auto p2 = v2(population<G>{ g0, g1, g2 });
  } catch (...)  {
    std::cout « "Even number of genotypes is required.\n";
  }
  const variation<G> v3{ swap_mutation<G>, cut_n_crossfill<G> };
  const auto p3 = v3(g0, g1);
  std::cout « p3[0] « '\n';
  std::cout « p3[1] « '\n';
}
```

Result (might be different due to randomness):

```
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
# Quile log: Variation: 5 0 1 6 3 4 2, 3 1 6 0 5 2 4
4 5 6 3 0 2 1
1 0 4 5 2 6 3
5 0 1 4 3 6 2
4 1 6 0 5 2 3
Even number of genotypes is required.
# Quile log: Variation: 4 5 0 3 6 2 1, 0 1 4 5 2 6 3
1 5 0 3 4 2 6
0 1 4 5 3 6 2
```

8.1.4.53 threshold_termination()

```
template<typename G , typename F >
requires chromosome<G>&& std::predicate<F, G> termination_condition_fn<G> quile::threshold↩
_termination (
             const F & thr )
```

`threshold_termination` returns condition, which terminates algorithm if at least one genotype fulfills predicate `thr`.

Template Parameters

G	Some `genotype` specialization.
F	Predicate type.

Parameters

thr	Predicate identifying searched genotype.

Returns

Predicate terminating genetic algorithm after genotype satisfying `thr` predicate is found.

8.1.4.54 unary_identity()

```
template<typename G >
requires chromosome<G> population<G> quile::unary_identity (
            const G & g )
```

`unary_identity` is an identity mutation.

Template Parameters

G	Some `genotype` specialization.

Parameters

g	Genotype.

Returns

Population consisting of `g`.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  using namespace quile;
  using G = genotype<g_permutation<int, 7, 0»;
  const auto g0 = G::random();
  const auto g1 = G::random();
  std::cout « g0 « '\n';
  std::cout « g1 « '\n';
  const auto h0 = unary_identity(g0)[0];
  std::cout « h0 « '\n';
  const auto p = binary_identity(g0, g1);
  std::cout « p[0] « '\n';
  std::cout « p[1] « '\n';
}
```

Result (might be different due to randomness):

```
0 3 2 5 1 4 6
1 2 6 0 4 3 5
0 3 2 5 1 4 6
0 3 2 5 1 4 6
1 2 6 0 4 3 5
```

8.1.4.55 uniform()

```
template<typename T , std::size_t N>
constexpr bool quile::uniform (
            const domain< T, N > & d )  [constexpr]
```

`uniform` checks whether domain is of form of hypercube.

Template Parameters

T	Domain base type.
N	Domain dimensionality.

Parameters

d	Domain to be checked.

Returns

Boolean value of check result.

Example:

```
#include <quile/quile.h>
constexpr auto d0 = quile::uniform_domain<int, 42>(quile::range{ 0, 100 });
constexpr auto d1 = quile::uniform_domain<int, 42>(0, 100);
static_assert(quile::uniform(d0));
static_assert(quile::uniform(d1));
int
main()
{}
```

Result (might be empty):

8.1.4.56 uniform_domain() [1/2]

```
template<typename T , std::size_t N>
constexpr domain<T, N> quile::uniform_domain (
            const range< T > & r )  [constexpr]
```

`uniform_domain` creates domain, where constraints on each direction are identical, i.e. domain is of form of hypercube.

Template Parameters

T	Domain base type.
N	Domain dimensionality.

Parameters

r	Interval for hypercube construction.

Returns

`N`-dimensional hypercube with edge of `r`.

Example:

```
#include <quile/quile.h>
constexpr auto d0 = quile::uniform_domain<int, 42>(quile::range{ 0, 100 });
constexpr auto d1 = quile::uniform_domain<int, 42>(0, 100);
static_assert(quile::uniform(d0));
static_assert(quile::uniform(d1));
int
main()
{}
```

Result (might be empty):

8.1.4.57 uniform_domain() [2/2]

```
template<typename T , std::size_t N>
constexpr domain<T, N> quile::uniform_domain (
            T lo,
            T hi )  [constexpr]
```

`uniform_domain<T, N>(lo, hi)` is equivalent to `uniform_domain<T, N>(range<T>{ lo, hi})` call. Please see documentation for `uniform_domain` for argument of type `range`.

Example:

```
#include <quile/quile.h>
constexpr auto d0 = quile::uniform_domain<int, 42>(quile::range{ 0, 100 });
constexpr auto d1 = quile::uniform_domain<int, 42>(0, 100);
static_assert(quile::uniform(d0));
static_assert(quile::uniform(d1));
int
main()
{}
```

Result (might be empty):

8.1.5 Variable Documentation

8.1.5.1 binary_chromosome

```
template<typename G >
concept quile::binary_chromosome
```

Initial value:

```
=
  chromosome<G> && binary_representation<typename G::genotype_t>
```

`binary_chromosome` specifies that `T` is some binary type specialization of `genotype`.

Example:

```
#include <iostream>
#include <quile/quile.h>
template<typename G>
requires quile::binary_chromosome<G>
auto
```

```
min_mutation(quile::probability p)
{
  return [=](const G& g) -> quile::population<G> {
    G res{ g };
    const auto& c = G::constraints();
    for (std::size_t i = 0; i < G::size(); ++i) {
      if (quile::success(p)) {
        res.value(i, c[i].min());
      }
    }
    return quile::population<G>{ res };
  };
}
int
main()
{
  const std::size_t n = 32;
  quile::genotype<quile::g_binary<n>> g{};
  std::cout « "before min_mutation:  " « g.random_reset() « '\n';
  const auto h = min_mutation<decltype(g)>(.2)(g)[0];
  std::cout « "after:                              " « h « '\n';
}
```

Result (might be different due to randomness):

```
before min_mutation: 0 0 0 0 1 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 1 0 1 0 1 0 1 1 1 1
 1 1
after:               0 0 0 0 1 1 1 0 0 1 1 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 1 1 1
 1 1
```

8.1.5.2 binary_representation

```
template<typename T >
concept quile::binary_representation = is_g_binary_v<T>
```

`binary_representation` specifies that `T` is some specialization of `g_binary`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
requires quile::binary_representation<T>
struct test
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

8.1.5.3 callable

```
template<typename F , typename R , typename...  Args>
concept quile::callable = std::convertible_to<std::invoke_result_t<F, Args...>, R>
```

`callable` specifies that `F` is convertible to the callable object type, which accepts arguments of types `Args...` and returns value of type `R`.

Template Parameters

F	Type convertible to the callable object type.
R	Return type.
Args	Argument type.

Example:

```
#include <cassert>
#include <quile/quile.h>
template<quile::callable<bool> B>
bool
negate(B b)
{
  return !b();
}
int
main()
{
  assert(negate([]() { return false; }));
}
```

Result (might be empty):

8.1.5.4 chromosome

```
template<typename G >
concept quile::chromosome = is_genotype_v<G>
```

`chromosome` specifies that `T` is some specialization of `genotype`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

8.1.5.5 chromosome_representation

```
template<typename T >
concept quile::chromosome_representation
```

Initial value:

```
=
  floating_point_representation<T> || integer_representation<T> ||
  binary_representation<T> || permutation_representation<T>
```

`chromosome_representation` specifies that `T` is some specialization of one of allowed representations, i.e. `g_floating_point`, `g_integer`, `g_binary` or `g_permutation`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

8.1.5.6 constraints_satisfied

```
template<typename G >
requires chromosome<G> const auto quile::constraints_satisfied = [](const G&) { return true;
}
```

`constraints_satisfied` is predicate stating that all genotypes are proper.

Example:

```
#include <quile/quile.h>
template<typename T, typename G, T t>
requires quile::genotype_constraints<T, G>
struct test
{};
```

```
int
main()
{
  using G = quile::genotype<quile::g_binary<42>>;
  [[maybe_unused]] test<decltype(quile::constraints_satisfied<G>),
                        G,
                        quile::constraints_satisfied<G»
    t{};
}
```

Result (might be empty):

8.1.5.7 e

```
template<std::floating_point T>
const T quile::e = std::numbers::e_v<T>
```

`e` is an approximation of e number.

Template Parameters

T	Floating-point type.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  std::cout « "pi = " « quile::pi<double> « '\n';
  std::cout « "e = " « quile::e<double> « '\n';
  std::cout « "ln 2 = " « quile::ln2<double> « '\n';
}
```

Result:

```
pi = 3.14159
e = 2.71828
ln 2 = 0.693147
```

8.1.5.8 floating_point_chromosome

```
template<typename G >
concept quile::floating_point_chromosome
```

Initial value:

```
=
  chromosome<G> && floating_point_representation<typename G::genotype_t>
```

`floating_point_chromosome` specifies that `T` is some floating-point type specialization of `genotype`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

8.1.5.9 floating_point_representation

```
template<typename T >
concept quile::floating_point_representation = is_g_floating_point_v<T>
```

`floating_point_representation` specifies that `T` is some specialization of `g_floating_point`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

8.1.5.10 genetic_pool

```
template<typename G >
concept quile::genetic_pool = is_population_v<G>
```

`genetic_pool` specifies that `T` is some specialization of `population`.

Example:

```
#include <iostream>
#include <quile/quile.h>
const std::size_t n = 32;
using genotype_t = quile::genotype<quile::g_binary<n>>;
using population_t = quile::population<genotype_t>;
static_assert(quile::is_population<population_t>::value);
static_assert(!quile::is_population_v<genotype_t>);
template<typename T>
requires quile::genetic_pool<T>
struct test
{};
int
main()
{
  [[maybe_unused]] test<population_t> t{};
  population_t p{};
  for (int i = 0; i < 8; ++i) {
    p.push_back(genotype_t{});
  }
  for (auto& g :  p) {
    g.random_reset();
  }
  for (auto& g :  p) {
    std::cout << g << '\n';
  }
}
```

Result (might be different due to randomness):

```
1 1 1 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 1 0 0 1 1 0 1 0
0 0 1 0 1 0 1 1 0 0 0 1 1 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 0 0 0 1
0 0 0 0 1 0 1 0 1 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 1
1 0 0 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 1
0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 1 1 0 0 1 0 1 0 1 0 0 1 1
1 1 1 1 0 0 0 0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 1 0
1 1 1 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1
0 0 1 0 1 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1 0 0 0 0 1 1 1
```

8.1.5.11 genotype_constraints

```
template<typename F , typename G >
concept quile::genotype_constraints = std::predicate<F, G> && chromosome<G>
```

`genotype_constraints` specifies that `F` is some predicate that states which genotypes are proper.

Note

Genotype $g \in X_0 \times \ldots \times X_{c-1} = \prod_{i=0}^{c-1} X_i$, where X_i is equal to $\mathbb{B} = \{\text{false}, \text{true}\}$ or is bounded subset of set of real numbers $\mathbb{R}$ or integer numbers $\mathbb{Z}$.

Proper genotype $g \in G = \{(x_0, \ldots, x_{c-1}) \in \prod_{i=0}^{c-1} X_i \mid Q(x_0, \ldots, x_{c-1})\}$, where Q is predicate stating which genotype is proper.

Set defined with use of predicate is called *extension of predicate*.

Example:
```
#include <quile/quile.h>
template<typename T, typename G, T t>
requires quile::genotype_constraints<T, G>
struct test
{};
int
main()
{
  using G = quile::genotype<quile::g_binary<42>>;
  [[maybe_unused]] test<decltype(quile::constraints_satisfied<G>),
                        G,
                        quile::constraints_satisfied<G»
    t{};
}
```
Result (might be empty):

8.1.5.12 incalculable

```
const fitness quile::incalculable = -std::numeric_limits<fitness>::infinity()
```

`incalculable` is a special value which can be used when given gentotype is not proper (cf. `genotype_constraints`) or to signal some problem in `fitness_function` (e.g. non-convergence).

8.1.5.13 integer_chromosome

```
template<typename G >
concept quile::integer_chromosome
```

Initial value:
```
=
  chromosome<G> && integer_representation<typename G::genotype_t>
```

`integer_chromosome` specifies that `T` is some integer type specialization of `genotype`.

Example:
```
#include <iostream>
#include <quile/quile.h>
template<typename G>
requires quile::integer_chromosome<G> quile::population<G>
min_mutation(const G& g)
{
  G res{ g };
  const quile::probability p = .2;
  const auto& c = G::constraints();
  for (std::size_t i = 0; i < G::size(); ++i) {
    if (quile::success(p)) {
      res.value(i, c[i].min());
    }
  }
  return quile::population<G>{ res };
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ quile::uniform_domain<int, n>(0, 9) };
  quile::genotype<quile::g_integer<int, n, &d>> g{};
  std::cout « "before min_mutation:  " « g.random_reset() « '\n';
  const auto h = min_mutation(g)[0];
  std::cout « "after:                              " « h « '\n';
}
```
Result (might be different due to randomness):

```
before min_mutation: 1 3 8 7 6 6 5 6 0 0 0 3 6 0 4 2 4 3 4 8 2 1 3 6 1 9 8 3 1 2
 6 7
after:               1 3 8 7 6 6 0 6 0 0 0 3 6 0 0 0 4 3 0 8 0 1 3 6 1 9 8 3 1 0
 6 7
```

8.1.5.14 integer_representation

```
template<typename T >
concept quile::integer_representation = is_g_integer_v<T>
```

`integer_representation` specifies that `T` is some specialization of `g_integer`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

8.1.5.15 is_domain_v

```
template<typename T >
constexpr bool quile::is_domain_v = is_domain<T>::value  [inline], [constexpr]
```

`is_domain_v` is helper variable template for `is_domain`.

Example:

```
#include <quile/quile.h>
#include <vector>
using type = quile::domain<double, 42>;
static_assert(quile::is_domain<type>::value);
static_assert(!quile::is_domain_v<std::vector<double»);
template<typename T>
requires quile::set_of_departure<T>
struct test
{};
int
main()
{
  [[maybe_unused]] test<type> t{};
}
```

Result (might be empty):

8.1.5.16 is_g_binary_v

```
template<typename T >
constexpr bool quile::is_g_binary_v = is_g_binary<T>::value  [inline], [constexpr]
```

`is_g_binary_v` is helper variable template for `is_g_binary`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const quile::domain<bool, 42> d{};
using b_type = quile::g_binary<42>;
static_assert(std::is_same_v<b_type::type, bool>);
static_assert(b_type::size() == 42);
static_assert(b_type::constraints() == d);
static_assert(quile::is_g_binary<b_type>::value);
static_assert(!quile::is_g_binary_v<decltype(d)>);
template<typename T>
requires quile::binary_representation<T>
struct test
{};
int
main()
{
  b_type::chain_t c{ quile::chain_min(d) };
  assert(b_type::valid(c));
  assert(b_type::default_chain() == c);
  [[maybe_unused]] test<b_type> t{};
}
```

Result (might be empty):

8.1.5.17 is_g_floating_point_v

```
template<typename T >
constexpr bool quile::is_g_floating_point_v = is_g_floating_point<T>::value  [inline], [constexpr]
```

`is_g_floating_point_v` is helper variable template for `is_g_floating_point`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using fp_type = quile::g_floating_point<double, 42, &d>;
static_assert(std::is_same_v<fp_type::type, double>);
static_assert(fp_type::size() == 42);
static_assert(fp_type::constraints() == d);
static_assert(quile::is_g_floating_point<fp_type>::value);
static_assert(!quile::is_g_floating_point_v<decltype(d)>);
template<typename T>
requires quile::floating_point_representation<T>
struct test
{};
int
main()
{
  fp_type::chain_t c{ quile::chain_min(d) };
  assert(fp_type::valid(c));
  assert(fp_type::default_chain() == c);
  [[maybe_unused]] test<fp_type> t{};
}
```

Result (might be empty):

8.1.5.18 is_g_integer_v

```
template<typename T >
constexpr bool quile::is_g_integer_v = is_g_integer<T>::value  [inline], [constexpr]
```

`is_g_integer_v` is helper variable template for `is_g_integer`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 100>(0, 42) };
using i_type = quile::g_integer<int, 100, &d>;
static_assert(std::is_same_v<i_type::type, int>);
static_assert(i_type::size() == 100);
static_assert(i_type::constraints() == d);
static_assert(quile::is_g_integer<i_type>::value);
static_assert(!quile::is_g_integer_v<decltype(d)>);
template<typename T>
requires quile::integer_representation<T>
struct test
{};
int
main()
{
  i_type::chain_t c{ quile::chain_min(d) };
  assert(i_type::valid(c));
  assert(i_type::default_chain() == c);
  [[maybe_unused]] test<i_type> t{};
}
```

Result (might be empty):

8.1.5.19 is_g_permutation_v

```
template<typename T >
constexpr bool quile::is_g_permutation_v = is_g_permutation<T>::value  [inline], [constexpr]
```

`is_g_permutation_v` is helper variable template for `is_g_permutation`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
requires quile::permutation_representation<T>
struct test
{};
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

8.1.5.20 is_genotype_v

```
template<typename T >
constexpr bool quile::is_genotype_v = is_genotype<T>::value  [inline], [constexpr]
```

`is_genotype_v` is helper variable template for `is_genotype`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<double, 42>(0., 1.)  };
using representation = quile::g_floating_point<double, 42, &d>;
using fp_genotype = quile::genotype<representation>;
static_assert(std::is_same_v<fp_genotype::gene_t, double>);
static_assert(std::is_same_v<fp_genotype::genotype_t, representation>);
static_assert(fp_genotype::size() == 42);
static_assert(fp_genotype::constraints() == d);
static_assert(fp_genotype::uniform_domain);
static_assert(quile::is_genotype<fp_genotype>::value);
static_assert(!quile::is_genotype_v<decltype(d)>);
template<typename T>
requires quile::chromosome_representation<T>
struct test_0
{};
template<typename T>
requires quile::chromosome<T>
struct test_1
{};
template<typename T>
requires quile::floating_point_chromosome<T>
struct test_2
{};
int
main()
{
  fp_genotype::chain_t c{ quile::chain_min(d) };
  assert(fp_genotype::valid(c));
  assert(fp_genotype{}.data() == c);
  [[maybe_unused]] test_0<representation> t0{};
  [[maybe_unused]] test_1<fp_genotype> t1{};
  [[maybe_unused]] test_2<fp_genotype> t2{};
}
```

Result (might be empty):

8.1.5.21 is_population_v

```
template<typename G >
constexpr bool quile::is_population_v = is_population<G>::value  [inline], [constexpr]
```

`is_population_v` is helper variable template for `is_population`.

Example:

```
#include <iostream>
#include <quile/quile.h>
const std::size_t n = 32;
using genotype_t = quile::genotype<quile::g_binary<n>>;
using population_t = quile::population<genotype_t>;
static_assert(quile::is_population<population_t>::value);
static_assert(!quile::is_population_v<genotype_t>);
template<typename T>
requires quile::genetic_pool<T>
struct test
{};
int
main()
{
```

```
  [[maybe_unused]] test<population_t> t{};
  population_t p{};
  for (int i = 0; i < 8; ++i) {
    p.push_back(genotype_t{});
  }
  for (auto& g :  p) {
    g.random_reset();
  }
  for (auto& g :  p) {
    std::cout « g « '\n';
  }
}
```

Result (might be different due to randomness):

```
1 1 1 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 1 0 0 1 1 0 1 0
0 0 1 0 1 0 1 1 0 0 0 1 1 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 0 0 0 1
0 0 0 0 1 0 1 0 1 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 1
1 0 0 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 1
0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 1 1 0 0 1 0 1 0 1 0 0 1 1
1 1 1 1 0 0 0 0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 1 0
1 1 1 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1
0 0 1 0 1 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1 0 0 0 0 1 1 1
```

8.1.5.22 ln2

```
template<std::floating_point T>
const T quile::ln2 = std::numbers::ln2_v<T>
```

$\ln 2$ is an approximation of $\ln 2$ number.

Template Parameters

T	Floating-point type.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  std::cout « "pi = " « quile::pi<double> « '\n';
  std::cout « "e = " « quile::e<double> « '\n';
  std::cout « "ln 2 = " « quile::ln2<double> « '\n';
}
```

Result:

```
pi = 3.14159
e = 2.71828
ln 2 = 0.693147
```

8.1.5.23 mutation

```
template<typename M , typename G >
concept quile::mutation
```

Initial value:

```
= requires(M m, G g)
{
  {
    m(g)
    } -> std::convertible_to<population<G>>;
}
&&chromosome<G>
```

`mutation` specifies that `M` instance applied to `genotype` returns object convertible to `population`.

Example:

```
#include <iostream>
#include <quile/quile.h>
using namespace quile;
namespace {
const auto id = [](auto x) { return x; };
auto
compose()
{
  return id;
}
template<typename F, typename... Fs>
auto
compose(F&& f, Fs&&... fs)
{
  return [=](auto x) { return f(compose(fs...)(x)); };
}
template<chromosome G, typename... Ms>
requires(mutation<Ms, G>&&...) auto mutation_composition(Ms&&... ms)
{
  return [=](const G& g) -> population<G> {
    return { compose([=](const G& g) -> G { return ms(g)[0]; }...)(g) };
  };
}
template<typename G>
requires integer_chromosome<G>
auto
deterministic_mutation(std::size_t pos, typename G::gene_t val)
{
  return [=](G g) -> population<G> {
    g.value(pos, val);
    return population<G>{ g };
  };
}
}
int
main()
{
  const std::size_t n = 5;
  static constexpr const auto d{ uniform_domain<int, n>(0, 9) };
  using G = genotype<g_integer<int, n, &d>;
  const G g{};
  mutation_fn<G> m0 = mutation_composition<G>();
  std::cout « m0(g)[0] « '\n';
  mutation_fn<G> m1 = mutation_composition<G>(deterministic_mutation<G>(1, 1));
  std::cout « m1(g)[0] « '\n';
  mutation_fn<G> m2 = mutation_composition<G>(deterministic_mutation<G>(2, 2));
  std::cout « m2(g)[0] « '\n';
  mutation_fn<G> m12 = mutation_composition<G>(deterministic_mutation<G>(1, 1),
                                               deterministic_mutation<G>(2, 2));
  std::cout « m12(g)[0] « '\n';
  mutation_fn<G> m134 =
    mutation_composition<G>(deterministic_mutation<G>(1, 1),
                            deterministic_mutation<G>(3, 3),
                            deterministic_mutation<G>(4, 4));
  std::cout « m134(g)[0] « '\n';
}
```

Result:

```
0 0 0 0 0
0 1 0 0 0
0 0 2 0 0
0 1 2 0 0
0 1 0 3 4
```

8.1.5.24 N

`quile::N`

Initial value:

```
{
  static void body(auto&& f)
  {
    f(std::integral_constant<T, I>{});
    static_loop<T, I + 1, N>::body(std::forward<decltype(f)>(f));
  }
}
```

8.1.5.25 permutation_chromosome

```
template<typename G >
concept quile::permutation_chromosome
```

Initial value:

```
=
  chromosome<G> && permutation_representation<typename G::genotype_t>
```

`permutation_chromosome` specifies that `T` is some permutation type specialization of `genotype`.

Example:

```
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <quile/quile.h>
template<typename G>
requires quile::permutation_chromosome<G> || quile::uniform_chromosome<G>
auto
n_swap_mutation(std::size_t n)
{
  return [=](const G& g) -> quile::population<G> {
    const auto rnd = []() {
      return quile::random_U<std::size_t>(0, G::size() - 1);
    };
    auto res = g.data();
    for (std::size_t i = 0; i < n; ++i) {
      std::swap(res[rnd()], res[rnd()]);
    }
    return quile::population<G>{ G{ res } };
  };
}
int
main()
{
  const std::size_t n = 32;
  quile::genotype<quile::g_permutation<int, n, 1>> g{};
  std::cout « "before n_swap_mutation:  " « g.random_reset() « '\n';
  const auto h = n_swap_mutation<decltype(g)>(3)(g)[0];
  std::cout « "after:                                  " « h « '\n';
}
```

Result (might be different due to randomness):

```
before n_swap_mutation: 22 12 3 21 10 31 8 17 9 1 32 5 29 27 4 14 19 2 30 16 24
7 25 11 20 18 6 26 15 23 28 13
after:                  22 12 3 6 10 4 8 17 26 1 32 5 29 27 31 14 19 2 30 16 24
7 25 11 20 18 21 9 15 23 28 13
```

8.1.5.26 permutation_representation

```
template<typename T >
concept quile::permutation_representation = is_g_permutation_v<T>
```

`permutation_representation` specifies that `T` is some specialization of `g_permutation`.

Example:

```
#include <cassert>
#include <quile/quile.h>
#include <type_traits>
constexpr const auto d{ quile::uniform_domain<int, 42>(0, 41) };
using p_type = quile::g_permutation<int, 42, 0>;
static_assert(std::is_same_v<p_type::type, int>);
static_assert(p_type::size() == 42);
static_assert(p_type::constraints() == d);
static_assert(quile::is_g_permutation<p_type>::value);
static_assert(!quile::is_g_permutation_v<decltype(d)>);
template<typename T>
requires quile::permutation_representation<T>
struct test
{};
int
main()
{
  p_type::chain_t c{ quile::iota<int, 42>(0) };
  assert(p_type::valid(c));
  assert(p_type::default_chain() == c);
  [[maybe_unused]] test<p_type> t{};
}
```

Result (might be empty):

8.1.5.27 pi

```
template<std::floating_point T>
const T quile::pi = std::numbers::pi_v<T>
```

`pi` is an approximation of π number.

Template Parameters

T	Floating-point type.

Example:

```
#include <iostream>
#include <quile/quile.h>
int
main()
{
  std::cout « "pi = " « quile::pi<double> « '\n';
  std::cout « "e = " « quile::e<double> « '\n';
  std::cout « "ln 2 = " « quile::ln2<double> « '\n';
}
```

Result:

```
pi = 3.14159
e = 2.71828
ln 2 = 0.693147
```

8.1.5.28 recombination

```
template<typename R , typename G >
concept quile::recombination
```

Initial value:

```
= requires(R r, G g)
{
  {
    r(g, g)
    } -> std::convertible_to<population<G>;
}
&&chromosome<G>
```

`recombination` specifies that `M` instance applied to two objects of type `genotype` returns object convertible to `population`.

Example:

```
#include <iostream>
#include <quile/quile.h>
#include <vector>
using namespace quile;
namespace {
template<chromosome G, typename...  Rs>
requires(recombination<Rs, G>&&...)  auto random_recombination(Rs&&...  rs)
{
  return
    [r = std::vector<recombination_fn<G>{ rs...  }](const G& g0, const G& g1) {
      return r[random_U<std::size_t>(0, r.size() - 1)](g0, g1);
    };
}
}
int
main()
{
  const std::size_t n = 32;
  static constexpr const auto d{ uniform_domain<double, n>(0., 1.)  };
  using G = genotype<g_floating_point<double, n, &d>;
  const auto g0 = G::random();
  const auto g1 = G::random();
  std::cout « "Parents:\n";
  std::cout « g0 « '\n';
  std::cout « g1 « '\n';
  const recombination_fn<G> r = random_recombination<G>(
    single_arithmetic_recombination<G>, one_point_xover<G>);
  const population<G> p = r(g0, g1);
  std::cout « "Offspring:\n";
  std::cout « p[0] « '\n';
  std::cout « p[1] « '\n';
}
```

Result (might be different due to randomness):

```
Parents:
7.632938301095844e-01 5.380850537336410e-01 8.662196756244862e-01 2.385804670907
881e-01 3.992661030128286e-01 9.637379325762896e-01 8.015222496416907e-01 4.0781
02496612352e-01 6.495688681819360e-02 3.773104410660121e-01 4.470655570561451e-0
1 9.051977045719400e-01 3.754480928190284e-01 5.852131314403002e-01 2.7607692016
30901e-01 6.901601014454185e-01 1.876479488560436e-02 9.691397176956201e-01 2.92
6243459260146e-01 7.015068115063313e-01 5.096234113058886e-01 6.169321355472475e
-02 9.275202166839894e-01 3.908546545463924e-01 8.545031327328174e-01 2.57658215
2773367e-01 6.437327114578221e-01 9.651463910919517e-01 3.518883850377567e-01 9.
158175963353863e-01 5.548562472262517e-01 4.290178207550817e-01
3.567643357649279e-01 1.391264328060377e-01 6.003527872330220e-01 2.862740240145
400e-01 4.883058640665820e-01 9.146947899566816e-01 3.620078563266221e-01 8.7475
99088295999e-01 9.608142268763895e-01 9.611901113709199e-01 8.793766878269520e-0
1 4.577419434232913e-01 8.500066867665092e-02 2.611466104241356e-01 2.0453898352
02442e-02 7.230066340434387e-01 8.718111857170628e-01 5.250218644745700e-01 1.16
1269539668478e-02 1.131626122567013e-01 6.667633275191966e-01 6.620535971330396e
-01 4.047631606157035e-01 3.884035499872193e-01 3.895695861220623e-01 9.19629564
4505946e-01 2.274506717811672e-02 2.862834266068857e-01 9.566141505366095e-02 8.
618512812097814e-01 3.744678094228941e-01 5.132685497467671e-01
Offspring:
7.632938301095844e-01 5.380850537336410e-01 8.662196756244862e-01 2.385804670907
881e-01 3.992661030128286e-01 9.637379325762896e-01 8.015222496416907e-01 4.0781
```

```
02496612352e-01 6.495688681819360e-02 3.773104410660121e-01 4.470655570561451e-0
1 9.051977045719400e-01 3.754480928190284e-01 5.852131314403002e-01 2.7607692016
30901e-01 6.901601014454185e-01 1.876479488560436e-02 9.691397176956201e-01 2.92
6243459260146e-01 7.015068115063313e-01 5.096234113058886e-01 6.620535971330396e
-01 4.047631606157035e-01 3.884035499872193e-01 3.895695861220623e-01 9.19629564
4505946e-01 2.274506717811672e-02 2.862834266068857e-01 9.566141505366095e-02 8.
618512812097814e-01 3.744678094228941e-01 5.132685497467671e-01
3.567643357649279e-01 1.391264328060377e-01 6.003527872330220e-01 2.862740240145
400e-01 4.883058640665820e-01 9.146947899566816e-01 3.620078563266221e-01 8.7475
99088295999e-01 9.608142268763895e-01 9.611901113709199e-01 8.793766878269520e-0
1 4.577419434232913e-01 8.500066867665092e-02 2.611466104241356e-01 2.0453898352
02442e-02 7.230066340434387e-01 8.718111857170628e-01 5.250218644745700e-01 1.16
1269539668478e-02 1.131626122567013e-01 6.667633275191966e-01 6.169321355472475e
-02 9.275202166839894e-01 3.908546545463924e-01 8.545031327328174e-01 2.57658215
2773367e-01 6.437327114578221e-01 9.651463910919517e-01 3.518883850377567e-01 9.
158175963353863e-01 5.548562472262517e-01 4.290178207550817e-01
```

8.1.5.29 set_of_departure

```
template<typename T >
concept quile::set_of_departure = is_domain_v<T>
```

`set_of_departure` specifies that `T` is some specialization of `domain`.

Example:

```
#include <quile/quile.h>
#include <vector>
using type = quile::domain<double, 42>;
static_assert(quile::is_domain<type>::value);
static_assert(!quile::is_domain_v<std::vector<double»);
template<typename T>
requires quile::set_of_departure<T>
struct test
{};
int
main()
{
  [[maybe_unused]] test<type> t{};
}
```

Result (might be empty):

8.1.5.30 termination_condition

```
template<typename F , typename G >
concept quile::termination_condition = std::predicate<F, std::size_t, generations<G>>
```

`termination_condition` specifies that `F` is some predicate that states when evolution should be finished.

8.1.5.31 uniform_chromosome

```
template<typename G >
concept quile::uniform_chromosome = chromosome<G> && G::uniform_domain
```

`uniform_chromosome` specifies that `T` is some specialization of `genotype` satisfying uniformity condition.

Example:

```
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <quile/quile.h>
template<typename G>
requires quile::permutation_chromosome<G> || quile::uniform_chromosome<G>
auto
n_swap_mutation(std::size_t n)
{
  return [=](const G& g) -> quile::population<G> {
    const auto rnd = []() {
      return quile::random_U<std::size_t>(0, G::size() - 1);
    };
    auto res = g.data();
    for (std::size_t i = 0; i < n; ++i) {
      std::swap(res[rnd()], res[rnd()]);
    }
    return quile::population<G>{ G{ res } };
  };
}
int
main()
{
  const std::size_t n = 32;
  quile::genotype<quile::g_permutation<int, n, 1>> g{};
  std::cout « "before n_swap_mutation:  " « g.random_reset() « '\n';
  const auto h = n_swap_mutation<decltype(g)>(3)(g)[0];
  std::cout « "after:                                  " « h « '\n';
}
```

Result (might be different due to randomness):

```
before n_swap_mutation: 22 12 3 21 10 31 8 17 9 1 32 5 29 27 4 14 19 2 30 16 24
7 25 11 20 18 6 26 15 23 28 13
after:                  22 12 3 6 10 4 8 17 26 1 32 5 29 27 31 14 19 2 30 16 24
7 25 11 20 18 21 9 15 23 28 13
```

Index