EKUMEN

# Implementing Particle Filters With Ranges

NAHUEL ESPINOSA

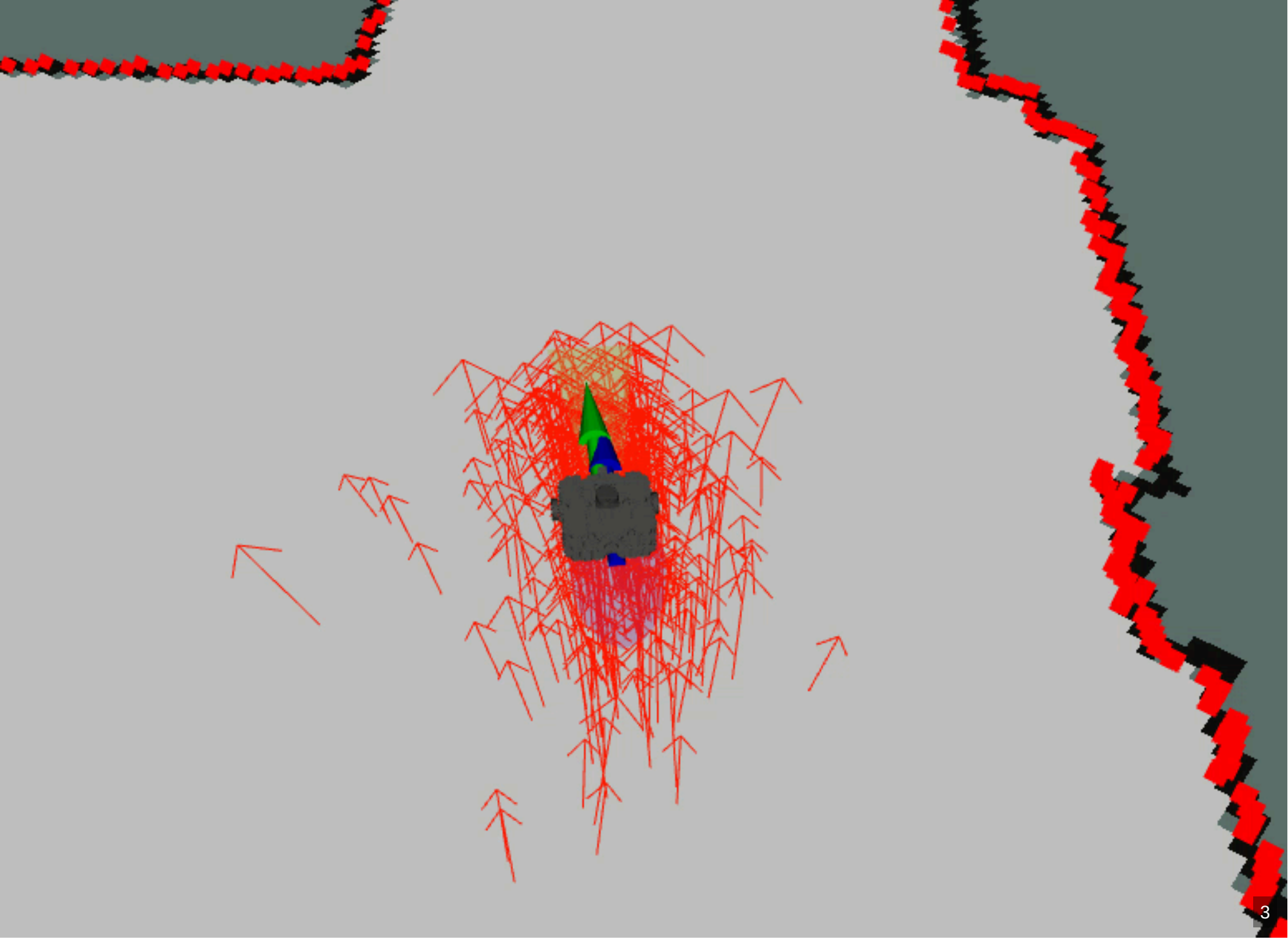

# ABOUT ME 🇦🇷

Robotics Software Engineer
(2020-present)

EKUMEN

Repository Maintainer

BELUGA

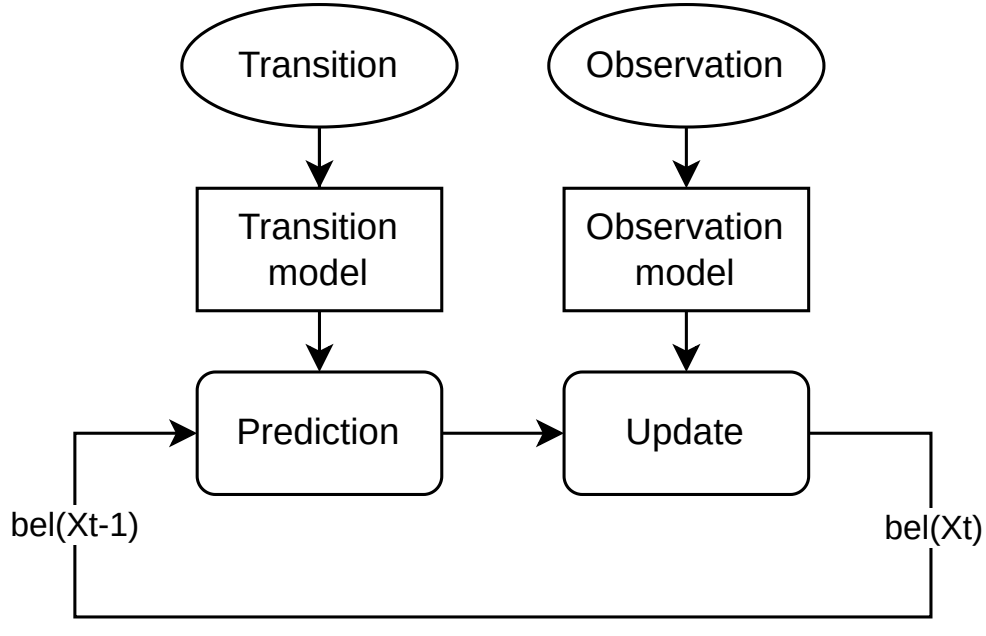


# TALK OVERVIEW

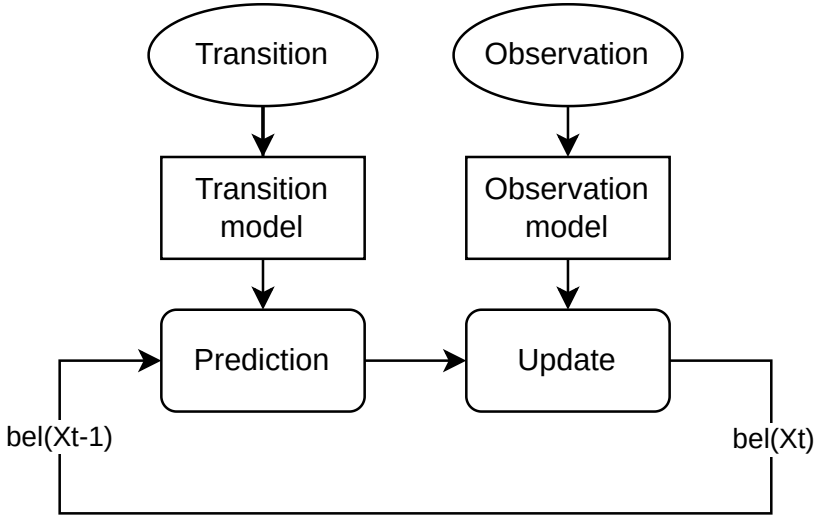
- Brief introduction to Particle Filters
- Brief mention to the C++ Ranges library
- Implementation walkthrough of a Particle Filter using C++23
- Practical recommendations and remarks

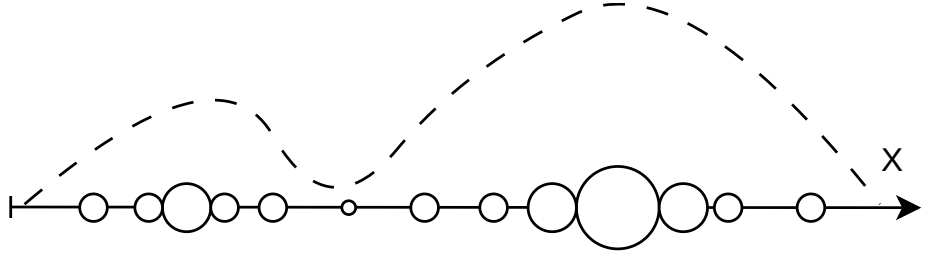
# BAYESIAN FILTERS

Algorithms used to estimate the internal state of a dynamical system given **noisy observations** and **random perturbations** in the system itself.



# PARTICLE FILTERS

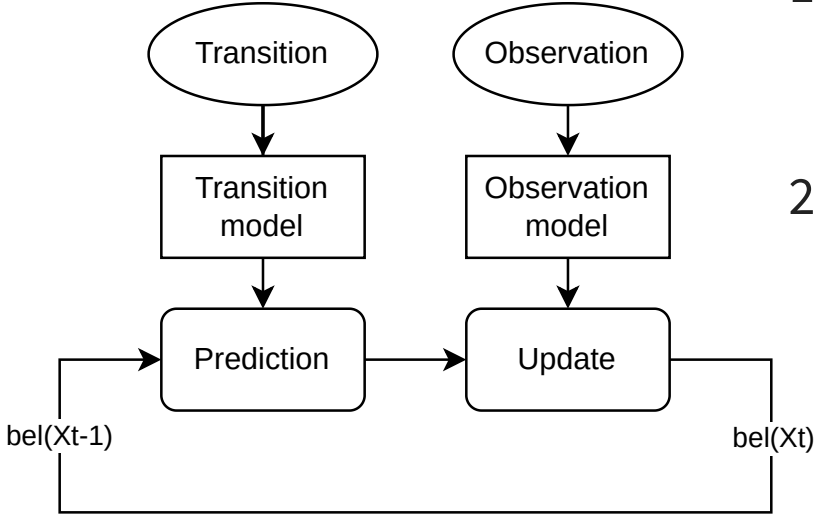



BELIEF - Represented by a set of particles

PARTICLE - A single hyphotesis of the **state** of the system with an associated **weight**

# PARTICLE FILTERS




1. Prediction
   - Compute new states for each particle (transition model)
2. Update
   - Compute weights for each particle (observation model)
   - **Resample**
     - Duplicate particles with high weights
     - Eliminate particles with low weights

# MONTE CARLO LOCALIZATION (MCL)


`STATE` - Position and orientation of the robot

`TRANSITION` - Control commands, odometry, etc.

`OBSERVATION` - Sensor data (LIDAR, Camera, etc.)

# C++ RANGES LIBRARY

Extension and generalization of the algorithm and iterator libraries.

```
(C++98) algorithm(begin(range), end(range)) → result
```

# C++ RANGES LIBRARY

Extension and generalization of the algorithm and iterator libraries.

```
(C++98) algorithm(begin(range), end(range)) → result
```

```
(C++20) algorithm(range) → result
```

```
(C++20) adaptor(range) → view
```

```
(C++20) range | adaptor → view
```

```
(C++20) range | adaptor1 | adaptor2 → view
```

# C++ RANGES LIBRARY

Extension and generalization of the algorithm and iterator libraries.

```
(C++98) algorithm(begin(range), end(range)) → result

(C++20) algorithm(range) → result

(C++20) adaptor(range) → view

(C++20) range | adaptor → view

(C++20) range | adaptor1 | adaptor2 → view

(C++23) range | adaptor1 | user_defined_adaptor | adaptor2 → view

(C++23) range | to<non-view> → non-view
```

# C++ RANGES LIBRARY

- **Particle sets** are *ranges*
- **Particle filters** are complex *algorithms* applied to *ranges*

# C++ RANGES LIBRARY

- **Particle sets** are *ranges*
- **Particle filters** are complex *algorithms* applied to *ranges*

The *Ranges* library provides the tools to write these algorithms in a way that is:

- Easy to read
- Less error-prone
- Easy to reuse

# CONCEPTS

```
1  template <typename T>
2  concept ParticleLike =
3      std::is_object_v<T> &&
4      requires(T a)
5      {
6          { a.state };
7          { a.weight } -> std::convertible_to<double>;
8      };
9
10 template <typename F, typename T>
11 concept StateUpdateFn =
12     ParticleLike<T> &&
13     requires(F f, T t) { { t.state = f(t.state) }; };
14
15 template <typename F, typename T>
16 concept ReweightFn =
17     ParticleLike<T> &&
18     requires(F f, T t) { { t.weight = f(t.state) }; };
```

# CONCEPTS

```cpp
template <typename T>
concept ParticleLike =
    std::is_object_v<T> &&
    requires(T a)
    {
        { a.state };
        { a.weight } -> std::convertible_to<double>;
    };

template <typename F, typename T>
concept StateUpdateFn =
    ParticleLike<T> &&
    requires(F f, T t) { { t.state = f(t.state) }; };

template <typename F, typename T>
concept ReweightFn =
    ParticleLike<T> &&
    requires(F f, T t) { { t.weight = f(t.state) }; };
```

# CONCEPTS

```cpp
template <typename T>
concept ParticleLike =
    std::is_object_v<T> &&
    requires(T a)
    {
        { a.state };
        { a.weight } -> std::convertible_to<double>;
    };

template <typename F, typename T>
concept StateUpdateFn =
    ParticleLike<T> &&
    requires(F f, T t) { { t.state = f(t.state) }; };

template <typename F, typename T>
concept ReweightFn =
    ParticleLike<T> &&
    requires(F f, T t) { { t.weight = f(t.state) }; };
```

# CONCEPTS

```cpp
template <typename T>
concept ParticleLike =
    std::is_object_v<T> &&
    requires(T a)
    {
        { a.state };
        { a.weight } -> std::convertible_to<double>;
    };

template <typename F, typename T>
concept StateUpdateFn =
    ParticleLike<T> &&
    requires(F f, T t) { { t.state = f(t.state) }; };

template <typename F, typename T>
concept ReweightFn =
    ParticleLike<T> &&
    requires(F f, T t) { { t.weight = f(t.state) }; };
```

# PARTICLE FILTER ALGORITHM

The filter function should perform two steps:

- Update states and weights
- Resample

```
template <ParticleLike T>
void filter(
    std::vector<T>& particles,
    StateUpdateFn<T> auto state_update_fn,
    ReweightFn<T> auto reweight_fn) {

    ...
}
```

# PARTICLE FILTER ALGORITHM

## UPDATE STATES AND WEIGHTS

### std::views::transform

```
1  template <ParticleLike T>
2  void filter(
3      std::vector<T>& particles,
4      StateUpdateFn<T> auto state_update_fn,
5      ReweightFn<T> auto reweight_fn) {
6
7      auto states = particles | std::views::transform(&T::state);
8      auto weights = particles | std::views::transform(&T::weight);
9
10     ...
11 }
```

# PARTICLE FILTER ALGORITHM

## UPDATE STATES AND WEIGHTS

### std::ranges::transform

```cpp
template <ParticleLike T>
void filter(
    std::vector<T>& particles,
    StateUpdateFn<T> auto state_update_fn,
    ReweightFn<T> auto reweight_fn) {

    auto states = particles | std::views::transform(&T::state);
    auto weights = particles | std::views::transform(&T::weight);

    std::ranges::transform(states, states.begin(), std::move(state_update_fn));
    std::ranges::transform(states, weights.begin(), std::move(reweight_fn));

    ...
}
```

# PARTICLE FILTER ALGORITHM

## RESAMPLE

### std::discrete_distribution

```
template <ParticleLike T>
void filter(
    std::vector<T>& particles,
    StateUpdateFn<T> auto state_update_fn,
    ReweightFn<T> auto reweight_fn) {

    auto states = particles | std::views::transform(&T::state);
    auto weights = particles | std::views::transform(&T::weight);

    std::ranges::transform(states, states.begin(), std::move(state_update_fn));
    std::ranges::transform(states, weights.begin(), std::move(reweight_fn));

    auto gen = std::mt19937{std::random_device()()};
    auto dist = weights | std::ranges::to<std::discrete_distribution<std::size_t>>();

    auto new_particles = std::vector<T>{};
    new_particles.reserve(particles.size());

    for (std::size_t i = 0; i < particles.size(); ++i) {
        new_particles.push_back(particles[dist(gen)]);
    }

    std::swap(particles, new_particles);
```

# PARTICLE FILTER ALGORITHM

## RESAMPLE

### std::discrete_distribution

```
void filter(
    std::vector<T>& particles,
    StateUpdateFn<T> auto state_update_fn,
    ReweightFn<T> auto reweight_fn) {

    auto states = particles | std::views::transform(&T::state);
    auto weights = particles | std::views::transform(&T::weight);

    std::ranges::transform(states, states.begin(), std::move(state_update_fn));
    std::ranges::transform(states, weights.begin(), std::move(reweight_fn));

    auto gen = std::mt19937{std::random_device()()};
    auto dist = weights | std::ranges::to<std::discrete_distribution<std::size_t>>();

    auto new_particles = std::vector<T>{};
    new_particles.reserve(particles.size());

    for (std::size_t i = 0; i < particles.size(); ++i) {
        new_particles.push_back(particles[dist(gen)]);
    }

    std::swap(particles, new_particles);
}
```

# PARTICLE FILTER ALGORITHM

## RESAMPLE

```cpp
template <ParticleLike T>
void filter(
    std::vector<T>& particles,
    StateUpdateFn<T> auto state_update_fn,
    ReweightFn<T> auto reweight_fn) {

    auto states = particles | std::views::transform(&T::state);
    auto weights = particles | std::views::transform(&T::weight);

    std::ranges::transform(states, states.begin(), std::move(state_update_fn));
    std::ranges::transform(states, weights.begin(), std::move(reweight_fn));

    auto gen = std::mt19937{std::random_device()()};

    particles = sample(particles, gen) |
                std::views::take(particles.size()) |
                std::ranges::to<std::vector>();
}
```

1. Generate random samples (with replacement)
2. Take N of those samples
3. Create a new particle set

# SAMPLE VIEW

| `std::ranges::sample?` `(C++20)` | `custom view` |
|---|---|
| Algorithm (eager) | View (lazy) |
| Fixed size | Infinite |
| Without replacement | With replacement |
| All elements have the same probability | Probability determined by the weights |

# SAMPLE VIEW

## DEFINITION

Given a particle range and a random number generator (RNG), generate a **lazy** computed **infinite** sequence of random samples (**with replacement**).

```
sample(particles, gen) → view
```

# SAMPLE VIEW

## DEFINITION

Given a particle range and a random number generator (RNG), generate a **lazy** computed **infinite** sequence of random samples (**with replacement**).

```
sample(particles, gen) → view
```

## IMPLEMENTATION

- `std::generator`
- `std::views::generate` (C++26)
- `std::views::view_interface`

# SAMPLE VIEW

## `std::generator` (C++23)

```cpp
template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
requires std::ranges::random_access_range<R> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample(R&& particles, RNG& gen) {
    auto dist = particles |
                std::views::transform(&P::weight) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}
```

# SAMPLE VIEW

## `std::generator` (C++23)

```cpp
template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
requires std::ranges::random_access_range<R> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample(R&& particles, RNG& gen) {
    auto dist = particles |
                std::views::transform(&P::weight) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}
```

# SAMPLE VIEW

## `std::generator` (C++23)

```cpp
template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
requires std::ranges::random_access_range<R> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample(R&& particles, RNG& gen) {
    auto dist = particles |
                std::views::transform(&P::weight) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}
```

# SAMPLE VIEW

## `std::generator` (C++23)

```cpp
template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
requires std::ranges::random_access_range<R> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample(R&& particles, RNG& gen) {
    auto dist = particles |
                std::views::transform(&P::weight) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}
```

# SAMPLE VIEW

## `std::generator` (C++23)

```cpp
template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
requires std::ranges::random_access_range<R> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample(R&& particles, RNG& gen) {
    auto dist = particles |
                std::views::transform(&P::weight) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}
```

# SAMPLE VIEW

## `std::generator` (C++23)

```
1  template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
2  requires std::ranges::random_access_range<R> &&
3           std::uniform_random_bit_generator<RNG> &&
4           ParticleLike<P>
5  std::generator<const P&> sample(R&& particles, RNG& gen) {
6      auto dist = particles |
7                  std::views::transform(&P::weight) |
8                  std::ranges::to<std::discrete_distribution<std::size_t>>();
9
10     while (true) {
11         co_yield particles[dist(gen)];
12     }
13 }
```

# SAMPLE VIEW

## `std::generator` (C++23)

```cpp
template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
requires std::ranges::random_access_range<R> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample(R&& particles, RNG& gen) {
    auto dist = particles |
                std::views::transform(&P::weight) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}
```

```cpp
auto input = std::vector{Particle{}};
auto output = sample(input, gen);  // ✅
```

```cpp
auto output = sample(std::vector{Particle{}}, gen);  // ❌ temporary will be destroyed
```

# SAMPLE VIEW

Take a range by value?

```cpp
template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
requires std::ranges::random_access_range<R> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample(R particles, RNG& gen) {
    auto dist = particles |
                std::views::transform([](const P& p) { return p.weight; }) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}
```

```cpp
auto input = std::vector{Particle{}};
auto output = sample(input, gen);  // ✅ (copying the container)
```

```cpp
auto output = sample(std::vector{Particle{}}, gen);  // ✅
```

# SAMPLE VIEW

Take a view by value!

```
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
requires std::ranges::view<V> &&
         std::ranges::random_access_range<V> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample(V particles, RNG& gen) {
    auto dist = particles |
                std::views::transform([](const P& p) { return p.weight; }) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}
```

```
auto input = std::vector{Particle{}};
auto output = sample(input, gen);  // ❌ std::vector is not a view
```

```
auto output = sample(std::vector{Particle{}}, gen);  // ❌ std::vector is not a view
```

# COMPILER ERROR

```
ranges_base.h:576:13:   required for the satisfaction of 'view<V>'
    [with V = std::vector<Particle, std::allocator<Particle> >]
ranges_base.h:577:39: note: the expression 'enable_view<_Tp>
    [with _Tp = std::vector<Particle, std::allocator<Particle> >]' evaluated to 'false'
  577 |       = range<_Tp> && movable<_Tp> && enable_view<_Tp>;
      |                                       ^~~~~~~~~~~~~~~
```

EKUMEN

# SAMPLE VIEW

Use `std::views::all` to convert any range into a view

```
1  template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
2  requires std::ranges::view<V> &&
3           std::ranges::random_access_range<V> &&
4           std::uniform_random_bit_generator<RNG> &&
5           ParticleLike<P>
6  std::generator<const P&> sample_impl(V particles, RNG& gen) {
7      auto dist = particles |
8                  std::views::transform([](const P& p) { return p.weight; }) |
9                  std::ranges::to<std::discrete_distribution<std::size_t>>();
10
11     while (true) {
12         co_yield particles[dist(gen)];
13     }
14 }
15
16 template <typename R, typename RNG>
17 requires std::ranges::viewable_range<R>
18 auto sample(R&& particles, RNG& gen) {
19     return sample_impl(std::views::all(std::forward<R>(particles)), gen);
20 }
```

# SAMPLE VIEW

Use `std::views::all` to convert any range into a view

```cpp
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
requires std::ranges::view<V> &&
         std::ranges::random_access_range<V> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample_impl(V particles, RNG& gen) {
    auto dist = particles |
                std::views::transform([](const P& p) { return p.weight; }) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}

template <typename R, typename RNG>
requires std::ranges::viewable_range<R>
auto sample(R&& particles, RNG& gen) {
    return sample_impl(std::views::all(std::forward<R>(particles)), gen);
}
```

```cpp
auto input = std::vector{Particle{}};
auto output = sample(input, gen);  // ✅
```

```cpp
auto output = sample(std::vector{Particle{}}, gen);  // ❌ does not compile (yet)
```

# COMPILER ERROR

# COMPILER ERROR

```
sample.h:18:27: error: no match for 'operator|'
    (operand types are
        'std::ranges::owning_view<std::vector<Particle> >' and
        'std::ranges::views::__adaptor::_Partial<...>')

   18 |     auto dist = particles |
      |                 ~~~~~~~~~~^
   19 |                 std::views::transform([](const P& p) { return p.weight; }) |
      |                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ...

ranges_base.h:808:13:   required for the satisfaction of 'viewable_range<_Range>'
    [with _Range = std::ranges::owning_view<std::vector<Particle, std::allocator<Particle> > >&]
ranges_base.h:810:11: note: no operand of the disjunction is satisfied
  809 |       && ((view<remove_cvref_t<_Tp>> && constructible_from<remove_cvref_t<_Tp>, _Tp>)
      |          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  810 |           || (!view<remove_cvref_t<_Tp>>
      |           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  811 |               && (is_lvalue_reference_v<_Tp>
      |               ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  812 |                   || (movable<remove_reference_t<_Tp>>
      |                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

EKUMEN

# SAMPLE VIEW

Use `std::ranges::ref_view` to iterate over an owning view without moving it

```
1  template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
2  requires std::ranges::view<V> &&
3           std::ranges::random_access_range<V> &&
4           std::uniform_random_bit_generator<RNG> &&
5           ParticleLike<P>
6  std::generator<const P&> sample_impl(V particles, RNG& gen) {
7      auto dist = std::ranges::ref_view(particles) |
8                  std::views::transform([](const P& p) { return p.weight; }) |
9                  std::ranges::to<std::discrete_distribution<std::size_t>>();
10
11     while (true) {
12         co_yield particles[dist(gen)];
13     }
14 }
15
16 template <typename R, typename RNG>
17 requires std::ranges::viewable_range<R>
18 auto sample(R&& particles, RNG& gen) {
19     return sample_impl(std::views::all(std::forward<R>(particles)), gen);
20 }
```

EKUMEN

# SAMPLE VIEW

Use `std::ranges::ref_view` to iterate over an owning view without moving it

```
1  template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
2  requires std::ranges::view<V> &&
3           std::ranges::random_access_range<V> &&
4           std::uniform_random_bit_generator<RNG> &&
5           ParticleLike<P>
6  std::generator<const P&> sample_impl(V particles, RNG& gen) {
7      auto dist = std::ranges::ref_view(particles) |
8                  std::views::transform([](const P& p) { return p.weight; }) |
9                  std::ranges::to<std::discrete_distribution<std::size_t>>();
10
11     while (true) {
12         co_yield particles[dist(gen)];
13     }
14 }
15
16 template <typename R, typename RNG>
17 requires std::ranges::viewable_range<R>
18 auto sample(R&& particles, RNG& gen) {
19     return sample_impl(std::views::all(std::forward<R>(particles)), gen);
20 }
```

# SAMPLE VIEW

Use `std::ranges::ref_view` to iterate over an owning view without moving it

```cpp
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
requires std::ranges::view<V> &&
         std::ranges::random_access_range<V> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
std::generator<const P&> sample_impl(V particles, RNG& gen) {
    auto dist = std::ranges::ref_view(particles) |
                std::views::transform([](const P& p) { return p.weight; }) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    while (true) {
        co_yield particles[dist(gen)];
    }
}

template <typename R, typename RNG>
requires std::ranges::viewable_range<R>
auto sample(R&& particles, RNG& gen) {
    return sample_impl(std::views::all(std::forward<R>(particles)), gen);
}
```

```cpp
auto input = std::vector{Particle{}};
auto output = sample(input, gen);  // ✅
```

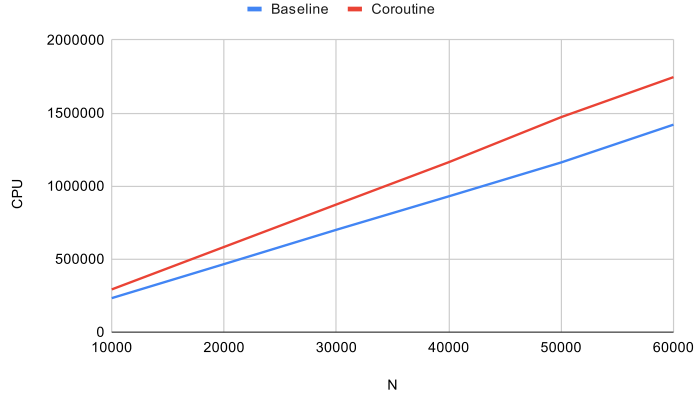
```cpp
auto output = sample(std::vector{Particle{}}, gen);  // ✅
```

# SAMPLE VIEW

EKUMEN

## PERFORMANCE COMPARISON

Initialize a vector by taking N samples from a small particle set.




GCC 14.1.0 -std=gnu++23 -O3 -DNDEBUG

# SAMPLE VIEW

## std::views::generate? (Tier 1 candidate for C++26 Ranges)

```cpp
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
requires std::ranges::view<V> &&
         std::ranges::random_access_range<V> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
auto sample_impl(V particles, RNG& gen) {
    auto dist = std::ranges::ref_view(particles) |
                std::views::transform([](const P& p) { return p.weight; }) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    return std::views::generate(
        [particles, dist = std::move(dist), &gen] {
            return particles[dist(gen)];
        });
}

template <typename R, typename RNG>
requires std::ranges::viewable_range<R>
auto sample(R&& particles, RNG& gen) {
    return sample_impl(std::views::all(std::forward<R>(particles)), gen);
}
```

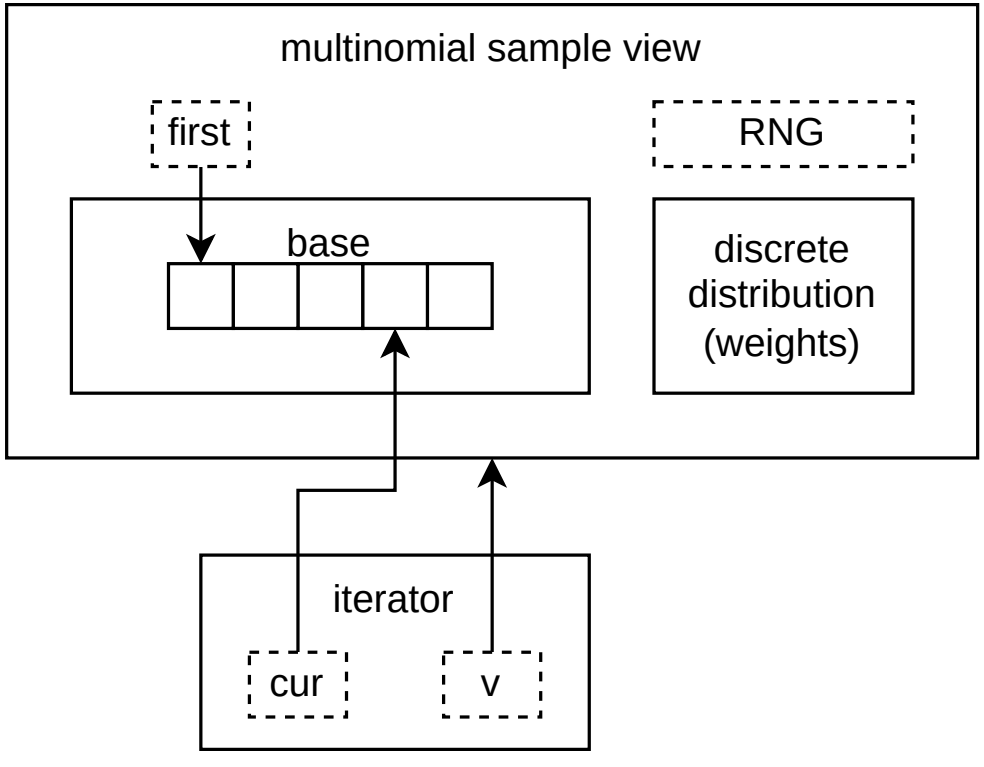
# SAMPLE VIEW

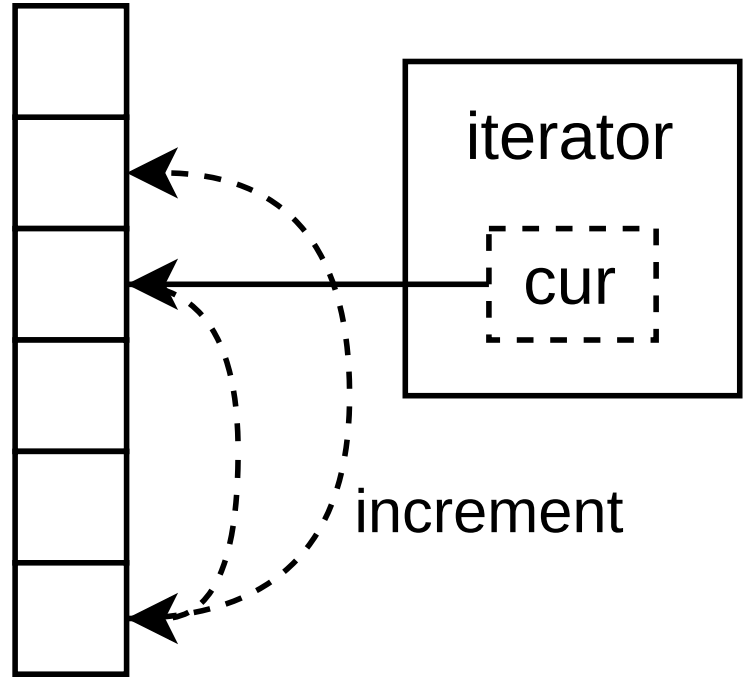
## std::views::generate? (Tier 1 candidate for C++26 Ranges)

```cpp
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
requires std::ranges::view<V> &&
         std::ranges::random_access_range<V> &&
         std::uniform_random_bit_generator<RNG> &&
         ParticleLike<P>
auto sample_impl(V particles, RNG& gen) {
    auto dist = std::ranges::ref_view(particles) |
                std::views::transform([](const P& p) { return p.weight; }) |
                std::ranges::to<std::discrete_distribution<std::size_t>>();

    return std::views::generate(
        [particles, dist = std::move(dist), &gen] {
            return particles[dist(gen)];
        });
}

template <typename R, typename RNG>
requires std::ranges::viewable_range<R>
auto sample(R&& particles, RNG& gen) {
    return sample_impl(std::views::all(std::forward<R>(particles)), gen);
}
```

# SAMPLE VIEW

## DESIGN CHOICES



# SAMPLE VIEW

## std::ranges::view_interface

```
1 template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
2 requires std::ranges::view<V> &&
3          std::ranges::random_access_range<V> &&
4          std::uniform_random_bit_generator<RNG> &&
5          ParticleLike<P>
6 class sample_view : public std::ranges::view_interface<sample_view<V, RNG, P>> {
7     ...
8 };
```

# SAMPLE VIEW

```cpp
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
class sample_view : public std::ranges::view_interface<sample_view<V, RNG, P>> {
public:
    sample_view(V base, RNG& gen)
      : base_{std::move(base)},
        first_{std::ranges::begin(base_)},
        gen_{std::addressof(gen)},
        dist_{
            std::ranges::ref_view(base_) |
            std::views::transform([](const P& p) { return p.weight; }) |
            std::ranges::to<std::discrete_distribution<std::size_t>>()
        } {}

    sample_view(const sample_view&) = delete;
    sample_view(sample_view&&) = default;
    sample_view& operator=(const sample_view&) = delete;
    sample_view& operator=(sample_view&&) = default;

    auto begin() { return iterator{this}; }
    auto end() const noexcept { return std::unreachable_sentinel; }

private:
    V base_;
```

# SAMPLE VIEW

```
    sample_view(V base, RNG& gen)
      : base_{std::move(base)},
        first_{std::ranges::begin(base_)},
        gen_{std::addressof(gen)},
        dist_{
            std::ranges::ref_view(base_) |
            std::views::transform([](const P& p) { return p.weight; }) |
            std::ranges::to<std::discrete_distribution<std::size_t>>()
        } {}

    sample_view(const sample_view&) = delete;
    sample_view(sample_view&&) = default;
    sample_view& operator=(const sample_view&) = delete;
    sample_view& operator=(sample_view&&) = default;

    auto begin() { return iterator{this}; }
    auto end() const noexcept { return std::unreachable_sentinel; }

private:
    V base_;
    std::ranges::const_iterator_t<V> first_;
    RNG* gen_;
    std::discrete_distribution<std::size_t> dist_;
```

# SAMPLE VIEW

```
 8          dist_{
 9              std::ranges::ref_view(base_) |
10              std::views::transform([](const P& p) { return p.weight; }) |
11              std::ranges::to<std::discrete_distribution<std::size_t>>()
12          } {}
13
14      sample_view(const sample_view&) = delete;
15      sample_view(sample_view&&) = default;
16      sample_view& operator=(const sample_view&) = delete;
17      sample_view& operator=(sample_view&&) = default;
18
19      auto begin() { return iterator{this}; }
20      auto end() const noexcept { return std::unreachable_sentinel; }
21
22  private:
23      V base_;
24      std::ranges::const_iterator_t<V> first_;
25      RNG* gen_;
26      std::discrete_distribution<std::size_t> dist_;
27
28      auto next() { return first_ + dist_(*gen_); }
29
30      class iterator { ... };
31
```

# SAMPLE VIEW

```
11              std::ranges::to<std::discrete_distribution<std::size_t>>()
12          } {}
13
14      sample_view(const sample_view&) = delete;
15      sample_view(sample_view&&) = default;
16      sample_view& operator=(const sample_view&) = delete;
17      sample_view& operator=(sample_view&&) = default;
18
19      auto begin() { return iterator{this}; }
20      auto end() const noexcept { return std::unreachable_sentinel; }
21
22  private:
23      V base_;
24      std::ranges::const_iterator_t<V> first_;
25      RNG* gen_;
26      std::discrete_distribution<std::size_t> dist_;
27
28      auto next() { return first_ + dist_(*gen_); }
29
30      class iterator { ... };
31
32      static_assert(std::input_iterator<iterator>);
33  };
```

# SAMPLE VIEW

```cpp
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
class sample_view : public std::ranges::view_interface<sample_view<V, RNG, P>> {
public:
    sample_view(V base, RNG& gen)
      : base_{std::move(base)},
        first_{std::ranges::begin(base_)},
        gen_{std::addressof(gen)},
        dist_{
            std::ranges::ref_view(base_) |
            std::views::transform(&P::weight) |
            std::ranges::to<std::discrete_distribution<std::size_t>>()
        } {}

    sample_view(const sample_view&) = delete;
    sample_view(sample_view&&) = default;
    sample_view& operator=(const sample_view&) = delete;
    sample_view& operator=(sample_view&&) = default;

    auto begin() { return iterator{this}; }
    auto end() const noexcept { return std::unreachable_sentinel; }

private:
    V base_;
```

# SAMPLE VIEW

Deduction guide to construct a `sample_view` from any range

```
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
class sample_view : public std::ranges::view_interface<sample_view<V, RNG, P>> {
    ...
};

template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
sample_view(R&&, RNG&) -> sample_view<std::views::all_t<R>, RNG, P>;

template <typename R, typename RNG>
auto sample(R&& range, RNG& gen) {
    return sample_view{std::forward<R>(range), gen};
}
```

# SAMPLE VIEW

Deduction guide to construct a `sample_view` from any range

```
1  template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
2  class sample_view : public std::ranges::view_interface<sample_view<V, RNG, P>> {
3      ...
4  };
5
6  template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
7  sample_view(R&&, RNG&) -> sample_view<std::views::all_t<R>, RNG, P>;
8
9  template <typename R, typename RNG>
10 auto sample(R&& range, RNG& gen) {
11     return sample_view{std::forward<R>(range), gen};
12 }
```

# SAMPLE VIEW

Deduction guide to construct a `sample_view` from any range

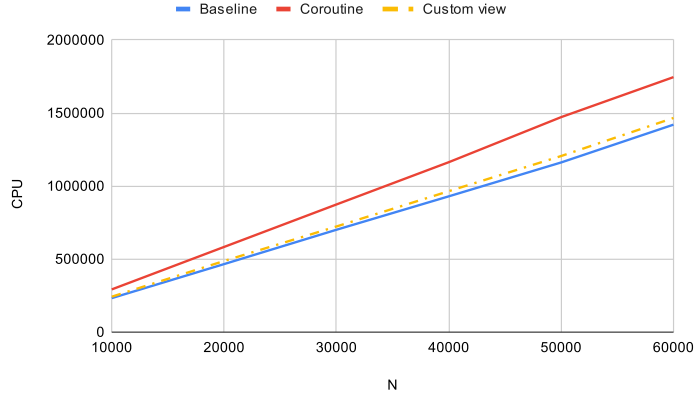
```cpp
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
class sample_view : public std::ranges::view_interface<sample_view<V, RNG, P>> {
    ...
};

template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
sample_view(R&&, RNG&) -> sample_view<std::views::all_t<R>, RNG, P>;

template <typename R, typename RNG>
auto sample(R&& range, RNG& gen) {
    return sample_view{std::forward<R>(range), gen};
}
```

# SAMPLE VIEW

EKUMEN

## PERFORMANCE COMPARISON

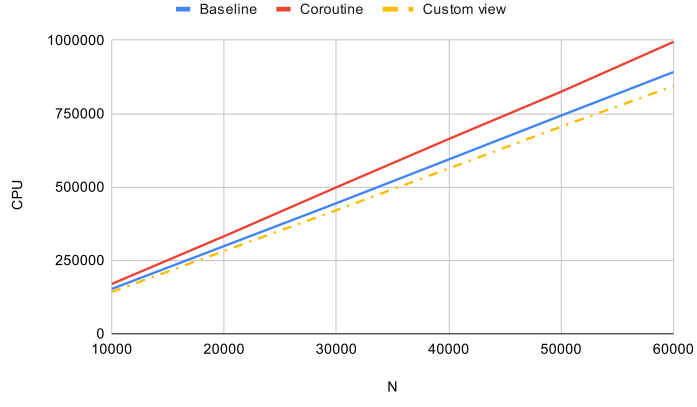
Initialize a vector by taking N samples from a small particle set.




GCC 14.1.0 -std=gnu++23 -O3 -DNDEBUG (Intel(R) Xeon(R))

# SAMPLE VIEW

Initialize a vector by taking N samples from a small particle set.




GCC 14.1.0 -std=gnu++23 -O3 -DNDEBUG (AMD Ryzen 9 5900HX)

# SAMPLE VIEW

```
sample(particles, gen) → view ✅
```

# SAMPLE VIEW

```
sample(particles, gen) → view ✅
```

```
sample(gen)(particles) → view ❌
```

```
particles | sample(gen) → view ❌
```

# SAMPLE VIEW

```
sample(particles, gen) → view ✅
```

```
sample(gen)(particles) → view ❌
```

```
particles | sample(gen) → view ❌
```

What do we need to fix?

```
sample(gen) → C (range adaptor closure object)
```

*If C is a range adaptor closure object and R is a range, these two expressions are equivalent:*

*C(R)*

*R | C*

# SAMPLE VIEW

```
1  template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
2  class sample_view : public std::ranges::view_interface<sample_view<V, RNG, P>> {
3      ...
4  };
5
6  template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
7  sample_view(R&&, RNG&) -> sample_view<std::views::all_t<R>, RNG, P>;
8
9  template <typename R, typename RNG>
10 auto sample(R&& range, RNG& gen) {
11     return sample_view{std::forward<R>(range), gen};
12 }
```

# SAMPLE VIEW

```
template <typename V, typename RNG, typename P = std::ranges::range_value_t<V>>
class sample_view : public std::ranges::view_interface<sample_view<V, RNG, P>> {
    ...
};

template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
sample_view(R&&, RNG&) -> sample_view<std::views::all_t<R>, RNG, P>;

template <typename R, typename RNG>
auto sample(R&& range, RNG& gen) {
    return sample_view{std::forward<R>(range), gen};
}

template <typename RNG>
auto sample(RNG& gen) {
    ...
}
```

# SAMPLE VIEW

## `std::ranges::range_adaptor_closure` (C++23)

```cpp
template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
class sample_view : public std::ranges::view_interface<sample_view<R, RNG, P>> {
    ...
};

template <typename R, typename RNG, typename P = std::ranges::range_value_t<R>>
sample_view(R&&, RNG&) -> sample_view<std::views::all_t<R>, RNG, P>;

template <typename RNG>
class sample_closure : public std::ranges::range_adaptor_closure<sample_closure<RNG>> {
public:
    explicit sample_closure(RNG& gen) : gen_{gen} {}

    template <typename R>
    auto operator()(R&& range) const {
        return sample_view{std::forward<R>(range), gen_};
    }

private:
    RNG& gen_;
};

template <typename R, typename RNG>
```

# SAMPLE VIEW

## `std::ranges::range_adaptor_closure` (C++23)

```
 9 template <typename RNG>
10 class sample_closure : public std::ranges::range_adaptor_closure<sample_closure<RNG>> {
11 public:
12     explicit sample_closure(RNG& gen) : gen_{gen} {}
13
14     template <typename R>
15     auto operator()(R&& range) const {
16         return sample_view{std::forward<R>(range), gen_};
17     }
18
19 private:
20     RNG& gen_;
21 };
22
23 template <typename R, typename RNG>
24 auto sample(R&& range, RNG& gen) {
25     return sample_view{std::forward<R>(range), gen};
26 }
27
28 template <typename RNG>
29 auto sample(RNG& gen) {
30     return sample_closure{gen};
31 }
```

# SAMPLE VIEW

```
sample(particles, gen) → view ✅
```

```
sample(gen)(particles) → view ✅
```

```
particles | sample(gen) → view ✅
```

## RECAP

| | |
|---|---|
| `sample_view` | A view class that implements the main iteration logic |
| `sample_closure` | A pipable object that captures configuration arguments and can create a view given a range |
| `sample` | A function that can create a view or a closure depending on the arguments |

# SEPARATE CONTAINERS

```
1 const auto states = std::vector<std::array<double, 50>>{...};
2 const auto weights = std::vector<double>{...};
```

# SEPARATE CONTAINERS

```
const auto states = std::vector<std::array<double, 50>>{...};
const auto weights = std::vector<double>{...};
```

```
template <typename S, typename W>
struct ParticleRef {
    const S& state;
    const W& weight;

    static_assert(ParticleLike<ParticleRef>);
};

auto to_particle = [](const auto& state, const auto& weight) {
    return ParticleRef{state, weight};
};

auto to_state = [](const auto& p) -> decltype(auto) {
    return p.state;
};

const auto new_states = zip_transform(to_particle, states, weights) |
                        sample(gen) |
                        take(1'000) |
                        transform(to_state) |
                        to<std::vector>();
```

# SEPARATE CONTAINERS

```
1 const auto states = std::vector<std::array<double, 50>>{...};
2 const auto weights = std::vector<double>{...};
```

```
 1 template <typename S, typename W>
 2 struct ParticleRef {
 3     const S& state;
 4     const W& weight;
 5
 6     static_assert(ParticleLike<ParticleRef>);
 7 };
 8
 9 auto to_particle = [](const auto& state, const auto& weight) {
10     return ParticleRef{state, weight};
11 };
12
13 auto to_state = [](const auto& p) -> decltype(auto) {
14     return p.state;
15 };
16
17 const auto new_states = zip_transform(to_particle, states, weights) |
18                         sample(gen) |
19                         take(1'000) |
20                         transform(to_state) |
21                         to<std::vector>();
```

# SEPARATE CONTAINERS

```
const auto states = std::vector<std::array<double, 50>>{...};
const auto weights = std::vector<double>{...};
```

```
template <typename S, typename W>
struct ParticleRef {
    const S& state;
    const W& weight;

    static_assert(ParticleLike<ParticleRef>);
};

auto to_particle = [](const auto& state, const auto& weight) {
    return ParticleRef{state, weight};
};

auto to_state = [](const auto& p) -> decltype(auto) {
    return p.state;
};

const auto new_states = zip_transform(to_particle, states, weights) |
                        sample(gen) |
                        take(1'000) |
                        transform(to_state) |
                        to<std::vector>();
```

# OTHER RANGE ADAPTORS

```
/* Resample algorithm in MCL */
particles = particles |
            sample(gen) |
            take(particles.size()) |
            to<std::vector>();
```

# OTHER RANGE ADAPTORS

```
/* Resample algorithm in MCL */
particles = particles |
            sample(gen) |
            take(particles.size()) |
            to<std::vector>();
```

```
/* Resample algorithm in AMCL (Adaptive Monte-Carlo Localization) */
particles = particles |
            sample(gen) |
            take_until_kld() |
            to<std::vector>();
```

# OTHER RANGE ADAPTORS

```
/* Resample algorithm in MCL */
particles = particles |
            sample(gen) |
            take(particles.size()) |
            to<std::vector>();
```

```
/* Resample algorithm in AMCL (Adaptive Monte-Carlo Localization) */
particles = particles |
            sample(gen) |
            take_until_kld() |
            to<std::vector>();
```

```
/* Recovery strategy to deal with the kidnapped robot problem */
particles = particles |
            sample(gen) |
            random_intersperse(map_distribution, probability, gen) |
            take_until_kld() |
            to<std::vector>();
```

# PRACTICAL RECOMMENDATIONS

- Leverage concepts and static assertions as much as possible

# PRACTICAL RECOMMENDATIONS

- Leverage concepts and static assertions as much as possible
- Beware of semantic requirements: *cheaply-copyable*, *regular-invocable*

# PRACTICAL RECOMMENDATIONS

- Leverage concepts and static assertions as much as possible
- Beware of semantic requirements: *cheaply-copyable*, *regular-invocable*
- Check out papers and source:
  - P2387R3: Pipe support for user-defined range adaptors
  - P2214R0: A Plan for C++23 Ranges
  - P2760R1: A Plan for C++26 Ranges

# PRACTICAL RECOMMENDATIONS

EKUMEN

## RANGE-V3 (C++17)

https://godbolt.org/z/b98e5eb9E

Source Editor: C++ source #1

```
#include <random>

#include <range/v3/view/facade.hpp>
#include <range/v3/view/transform.hpp>
#include <range/v3/range/conversion.hpp>

#include <range/v3/view/take_exactly.hpp>

#include <fmt/core.h>
#include <fmt/ranges.h>
#include <fmt/format.h>

struct Particle {
    int state;
    double weight;
};

template <class V, class RNG, class P = ranges::range_value_t<V>>
class sample_view : public ranges::view_facade<sample_view<V, RNG>, ranges::infinite> {
```

Edit on Compiler Explorer

# PRACTICAL RECOMMENDATIONS

## RANGE-V3 (C++17)

https://github.com/Ekumen-OS/beluga


# FINAL REMARKS

- We can use the STL to implement composable range adaptors for domain specific applications like particle filters.

# FINAL REMARKS

- We can use the STL to implement composable range adaptors for domain specific applications like particle filters.
- Ranges are useful to implement complex algorithms in a way that's straightforward to write, easy to read, and easy to reuse.



# THANKS!

https://github.com/nahueespinosa/presentation-cppcon2024