Identify Problem: Particle Filter

1: Odometry is Flawed

Odometry has an inherent flaw. It is a dead reckoning algorithm, meaning it calculates the robot position using the previous position and incorporating displacement[1] and heading measurements. Any inaccuracies in the displacement measurement will result in an inaccurate position estimation.

For example, if a y displacement measurement is 1 inch more than what it should be, then the resulting position estimate will be off by 1 inch. This is what is called error, the difference between what the position should be and the actual measured position.

There are many ways for error to be incorporated:

- Wheel slip: If the bot accelerates too fast, the wheels can lose traction with the ground and track more or less displacement than they should. This is exactly why we use dedicated tracking wheels, as they will slip far less than the drive wheels that are being driven by motors.
- Tipping: If the robot accelerates/decelerates way too fast the tracking wheels can lose contact with the ground all together, resulting in great amounts of error.
- Sensor noise: Our sensors are noisy, meaning that they are not perfectly accurate. The rotation sensors can track more or less than they should. The inertial measurement unit (IMU), which are used for heading, measures angular acceleration. This means that it must double integrate its measurements to find heading, which results in even greater compounding error.

Furthermore, odometry has no mechanism for correcting this error. Thus, any incorporated error will remain in the position estimate forever, resulting in

[1]Displacement: The difference between the starting and ending positions. Has both direction and quantity.

Identify Problem: Particle Filter

accumulating error. By the end of an autonomous routine, the error can grow to several inches, necessitating a solution to this error.

Here are the most common ways that this error is addressed:

- Reduce acceleration: Prevents the tracking wheels from slipping and tipping, and thus reduces error.
- Assume error is consistent: One can measure the error at a certain point in the auton, and, assuming the error will be consistent between runs, the error can be accounted for by setting the position at that point. For example, if odom is off by `(3, -2)` after the first part of an auton, one could set the pose to `currentPose - (3, -2)` at that point.
- Use the field: Another common method is to push the robot against one of the walls. This lets you figure out one of the coordinates (x or y), as long as there's no game elements between you and the wall.

But these all come with caveats that reduce the amount of points that our autons can score:

- Reduce acceleration: This means that the auton will be slower and will not be able to score as many points.
- Assume error is consistent: This is never true. For instance, the field tiles at a competition will always behave differently from the ones used in practice. Other robots can interfere with your auton and push the bot in a way that results in different error. This results in less consistent autons and lower scoring autons
- Use the field: Ramming into walls requires that time be taken away from scoring activities, leading to less time to score points in auton.

In a head to head match, scoring less points means that we have less chance to win the autonomous bonus, which often makes or breaks high level matches. In skills, scoring less points means that we are less likely to score high on the leaderboard.

Identify Problem: Particle Filter

Additionally, tracking wheels take a lot of space and are a pain to design for. In specific, our last bot's intake geometry was really hard to get working consistently, because our tracking wheels forced the instascore to start higher.

In summary, there's a few problems with this naive approach to position tracking using odometry:

1. Odometry assumes that sensors are perfectly accurate
2. Odometry has no method of recovering from error.
3. Accurate odometry requires tracking wheels which take up valuable space

2: A Better Way?

Lets focus on #2 for a moment. How can we tell if the robot's position estimate is inaccurate? It's pretty simple for us, non-robotic human beings. We can just look at the robot and figure out the robot is over there. How do we do that? Perhaps we can teach the robot to do that?

Perhaps you find the robot's approximate position relative to a landmark with a known position, such as the ladder or a wall stake. With this knowledge, you can simply add or subtract from the known object's position to find the robot's position. Alright, so we just need to identify the landmarks around the robot, and find its position relative to them. Simple, right?

No, no it is not. To do this we'd need some form of computer vision to see what object we're looking at and where it is relative to the camera. In V5RC, the closest thing to this is the vision sensor, which is woefully unequipped for this task. It is likely possible, but the vision sensor is not very consistent. It doesn't work well under different lighting conditions, and it can get confused by other robots or objects outside the field. This would require lots of work to get consistent.

This landmark method essentially boils down to measuring the relative position to known objects. Maybe rather than landmarks we use the field walls. There are

a couple key benefits to using the field walls: they are always there, no matter the current year's game, and they're practically impossible to miss. The latter point is the most important as it makes it super simple to find the distance to the walls.

Alright, so we just point some distance sensors at the walls and as long as we also know the heading of the robot, we can compute the robot's position. Simple, right?

No, no it is not. I glossed over many important details here. Perhaps you tried to figure out how to compute the position from this information, and found it was impossible due to the symmetry of the field. Additionally, the distance sensors are much noisier than our traditional sensors. They can also measure things that aren't the walls such as other robots or the ladder. Therefore, this approach will require a robust mechanism for handling noisy sensors (problem #1).

Despite this method's many problems, we believe them to be easier to overcome than those of computer vision. In specific, we only need to figure out whether the current position estimate is accurate, for which purpose, this method will more than suffice (we'll explain this later). The distance sensors will be much easier and simpler to work with than the vision sensors. Most importantly, this method has been proven to work by 2654E Echo[1] in their world record skills run.

3: Filters

Noise is often defined as an error which is added to a true signal. Noise can come from many different sources. In the case of odometry, everything that impacts the accuracy of a tracking wheel (wheel slip, sensor noise, etc.) can be classified as noise.

[1]Sourced from discussions with Alex of 2654E

Identify Problem: Particle Filter

We need some method of removing this noise and accessing the true signal. This is known as filtering and it is not unique to position tracking. As such, there is plenty of research into the topic. Our **research** and understanding of the subject is largely based upon Roger Labbe's online book, *Kalman and Bayesian Filters in Python*[1]. For a complete and intuitive understanding of the subject, we highly suggest reading it.

The book describes several filters that could be applied to our problem, but before we can decide which is best, we must first first define our problem.

We'd like to determine the x, y and heading of the robot. Therefore, x, y, and theta are known as our system state, where the system is the robot. These variables are also known as observed variables since we can observe them with the distance sensors and the IMU. To improve the filter's performance it may also be beneficial to include hidden variables, such as velocities, but for now these shall suffice.

All of these filters rely on a mechanism for converting a state into the expected measurement. This is known as a measurement function. In our case, we must find the distance we'd expect the sensor to measure given the pose[2] of the robot. For demonstration purposes, we'll work out a simplified example measurement function:

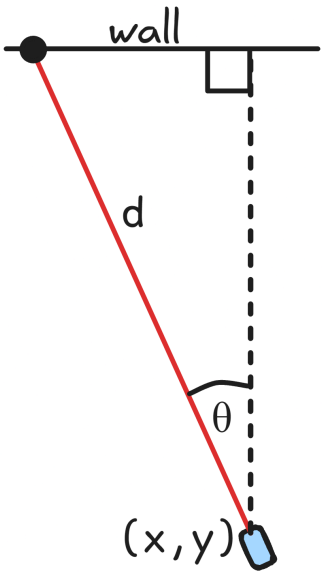

Figure 27.1: Laser hitting wall diagram

[1] *Kalman and Bayesian Filters in Python*: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

[2] Pose: Represents both the position and the orientation of an object

Identify Problem: Particle Filter

We will assume that the laser's pose is the same as the robot's, and that the laser is hitting the topmost wall for simplicity. See Figure 1 for a visual representation of the situation.

Let $h(x, y, \theta) = d$ be the measurement function, where x, y, θ represent a potential robot state, and d is the expected distance measurement.
Let w_y be the y coordinate of the field wall.

Now we can use some trig to find $h(x, y, \theta)$:

$$\cos\theta = \frac{w_y - y}{d}$$

$$d = \frac{w_y - y}{\cos\theta}$$

$$h(x, y, \theta) = \frac{w_y - y}{\cos\theta}$$

One of the most important factors in choosing a filter is the linearity of both the process model and the measurement function. In this context, a function $f(x)$ is linear if it has the following properties:

- Additivity: $f(x + b) = f(x) + f(b)$
- Homogeneity: $f(ax) = af(x)$

This simply means that $f(x)$ must be linearly proportional to the sum of all its inputs. Consequently, $f(0)$ must equal 0.

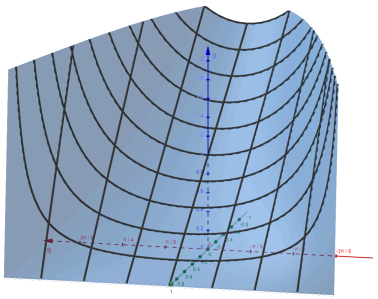

Figure 27.2: Measurement Function in 3D. The red axis is θ, blue is d, green is y, and $x = 0$.

To figure out which filter to use, we must first evaluate the linearity of our measurement function. We can do this by visualizing it 3d as shown in Figure 2. If it were linear, we should expect a flat plane, which is not the case.

Brainstorming: Particle Filter

Now, we should have sufficient knowledge to evaluate our filter choices:

Filter Name	Pros	Cons
Kalman Filter (KF)	• Simple • Requires little computation	• Struggles with nonlinearities
Extended Kalman Filter (EKF)	• Handles nonlinearities • Requires little computation	• Requires math that we don't yet understand
Unscented Kalman Filter (UKF)	• Medium simplicity • Handles nonlinearities	• Requires medium computation
Particle Filter (PF)	• Medium simplicity • Robustly handles nonlinearities	• Requires high computation • Difficult to troubleshoot

From this table, Andrew decided that UKF and PF would be our best options, as they could handle the nonlinearities of our measurement function and system model. In July, Andrew chose to use the UKF, because the particle filter would require more computation than thought available on the brain.

Recently, Andrew heard of 2654E Echo's world record auton skills, which used a PF. Combined with the difficulties Andrew had experienced with the UKF and the immense simplicity of the PF, this encouraged us to switch to the PF. So, what is a particle filter?

4: The Particle Filter Algorithm

Essentially, the particle filter uses a bunch of randomly generated potential states to model the probability distribution of the state. We can then do manipulate these potential states to represent the probability given a measurement or time passed.

Identify Problem: Particle Filter

The algorithm is fairly simple:

1. **Randomly generate a bunch of particles**: Each of these particles is a potential state, with an associated weight. This weight represents the likelihood/probability that the particle is the true state. They should all start with equal weight.
2. **Move particles (Predict)**: Move the particles according to how you predict the system should behave (system model). Any uncertainty in this prediction should be represented by moving the particles noisily.
3. **Update**: Update the weights of each particle based on the measurement. Particles that closely match the measurement are weighted higher than those that don't.
4. **Resample**: Remove highly unlikely particles and replace them with copies of more likely particles.
5. **Compute Estimate**: Optionally, compute the weighted average of the particles to be used elsewhere (e.g. motion algorithms).
6. **Repeat steps 2-5**

Particle filters are actually a family of algorithms, but the one described here is one that applies to our problem. This description is also rather broad, but we'll describe the algorithm in more detail as we code it.

This use of a finite number of random points to compute a result is known as a Monte Carlo method. Additionally, the process of tracking the robot position is known as localization. Hence, the application of a particle filter, a Monte Carlo method, to the problem of robot localization is known as *Monte Carlo Localization* (MCL).

5: Programming MCL

Andrew decided it would be best to split our implementation into 2 parts.

Program: Particle Filter

1. The `ParticleFilter` class: This would be an abstract implementation of the particle filter algorithm that could be applied to any problem.
2. The `Localization` class: This would use the `ParticleFilter` class for the purpose of robot localization.

This enables Andrew to write the robot specific `Localization` class without worrying about the implementation details of the particle filter algorithm.

Additionally, the more particles that are used, the more accurate the estimates from the particle filter will be. But, each iteration of the particle filter must also take shorter than 10ms, because new sensor data is made available every 10ms. Due to this, it is imperative that the particle filter not squander any compute resources on the brain so we can get the most accurate estimates possible. This will play a big role in how the code for MCL is designed.

5.1: The `ParticleFilter` Class

This class must expose the following functionality whilst being as performant as possible:

- Move the particles according to a system model
- Update the particles according to a measurement
- Compute the mean of the particles

What type should the particles be? Since the PF should be as abstract as possible, it actually shouldn't specify a specific type for the particles. Instead, we'll take a template parameter `P` to represent the particle type.

There are two ways to represent the particles: a `std::vector<P>` and a `std::array<P, N>`. The main difference being that `std::vector<P>` can be resized, whereas `std::array<P, N>` has a fixed size of `N`. Since the number of particles should never change in length, Andrew decided to use a `std::array<P, N>`, which might be a little more performant.

Program: Particle Filter

In testing, Andrew ran into a problem with a large `std::array<P, N>` that seemed to be related to it not fitting in the stack. Using a `std::unique_ptr<std::array<P, N>>`, resolved this problem as it allocates the `std::array` on the heap. Since, each particle also needs a weight, we'll also have a member of type `std::unique_ptr<float, N>`.

include/pf/pf.h — C++

```
11  /**
12   * @brief Abstract object for Particle Filter algorithm
13   *
14   * @tparam P Particle type. Must be addable by P, and multipliable by float.
15   * @tparam N Number of particles.
16   */
17  template <typename P, size_t N> class ParticleFilter {
18    private:
…                                                   …
24      std::unique_ptr<std::array<P, N>> m_particles;
25      /** @brief Weights corresponding to the particles. */
26      std::unique_ptr<std::array<float, N>> m_weights;
152 };
```

5.1.1: Predict step

Andrew decided that the abstractions for the particle filter should be done through composition rather than inheritance, as this often makes for cleaner and more readable code. Therefore, to enable any arbitrary prediction algorithm / system model to be used, we'll create an abstract class named `Predictor`. By abstract, we mean that it cannot be constructed, because not all of its methods are implemented. But, how can a method not be implemented and still work?

Typically, methods compile to global functions that also take a `this` param, but virtual methods are different. Virtual methods are actually function pointers[1] that are stored alongside the other members of the class. For virtual methods to

[1]Function pointer: Basically, a way to put a function in a variable. Often used for callbacks.

work, the object to which the virtual method belongs, must be a pointer or a reference, otherwise object slicing will occur.

We could use `std::function<void(std::array<P, N>&)>` here instead, but declaring a struct for this clarifies intent.

include/pf/pf.h C++

```
19      /**
20       * @brief Ensures only one operation occurs on particle filter at a time.
21       */
22      pros::Mutex m_mutex;
…                                                   …
43      /** @brief Moves particles and incorporates noise. */
44      struct Predictor {
45          /**
46           * @brief Modifies particles param according to system model and
47           * incorporates noise.
48           */
49          virtual void predict(std::array<P, N>& particles) = 0;
50      };
…                                                   …
67      /** Moves particles and incorporates noise. */
68      void predict(Predictor& predictor) {
69        m_mutex.take();
70        predictor.predict(*m_particles);
71        m_mutex.give();
72      }
```

The `virtual` keyword, combined with the `= 0` part, specifies that the method is *pure* virtual. Pure virtual tells the compiler that this function is not defined for this class, and thus makes the class abstract. Since virtual methods only work on references and pointers to the class, you can only hold a reference or pointer to the abstract class. Virtual methods can have a small performance impact, but

[1]Epoch: Iteration of the filter algorithm

because this function is only called once per epoch,[1] it will only have a nominal impact.

Also, while the predict function is running, the RTOS scheduler could context switch, stealing control of the processor. This could result in the particles only being partially mutated, which would cause problems if another task tried to estimate the mean of the particles. For this reason, Andrew uses a mutex for all the particle filter methods to ensure that no two tasks are mutating or reading the particle filter data at the same time.

5.1.2: The Update step

In our implementation, we have combined the resampling step and the normalizing, as these will almost always be done in sequence. We'll start with the resampling part.

The resampling algorithm must be abstract as there are many different possible resampling algorithms to choose from. This enables Andrew to easily test different resampling techniques and measure their impacts.

The resampler will have two responsibilities:

1. Pick the particles to copy.
2. Decide whether resampling is necessary.

Looking back, it would have been better to separate these functions as they should not be tied together.

5.1.2.1: Update: Resampling

Despite this, Andrew used the effective N method suggested in the book for responsibility #2:

$$\hat{N}_{\text{eff}} = \frac{1}{\sum w^2}$$

Program: Particle Filter

include/pf/resamplers/resampler.h C++

```cpp
    /** @returns Whether resampling should be performed. */
    virtual bool shouldResample(const std::array<float, N>& weights) {
      float sumOfSquares = 0;
      for (const float& weight : weights) { sumOfSquares += weight * weight; }
      /** Effective N - Approximates number of particles meaningfully
       * contributing to probability distribution  */
      const float effN = 1 / sumOfSquares;
      return effN < (N / 2.f);
    }
```

This method is virtual so that child classes of the resampler can override it and use their own algorithms. It takes a `const` reference to weights to ensure that the `Resampler` does not mutate the weights.

For the actual resampling, `Resampler` exposes a pure virtual method called `resample()`. To figure out which particles to select, it must take the weights of the particles. To convey the which particles the `Resampler` chose there are two options:

1. Mutate the particle array: Could be potentially more performant, but would complicate implementations of `resample()`
2. Return an array of particle indices to copy: Reduce complexity of `resample()`, but may be slower than #1.

Andrew chose #2 because it would reduce code duplication. If it did cause performance problems, it could be measured and refactored later.

Andrew chose to have `resample()` mutate an existing indices array. This meant that `resample()` would not need to construct a new indices array upon every call which could be quite costly. Rather than passing a reference / pointer to the indices, a `std::unique_ptr<>` was used to clarify intent.

include/pf/resamplers/resampler.h C++

```cpp
8   /**
```

Program: Particle Filter

include/pf/resamplers/resampler.h (C++)

```
 * @brief Resamples particles
 * @tparam N Number of particles/weights
 */
template <size_t N> struct Resampler {
…                                                    …
    /** @brief Modifies indices array to contain selected indices. */
    virtual std::unique_ptr<std::array<size_t, N>>
    resample(const std::array<float, N>& weights,
             std::unique_ptr<std::array<size_t, N>> indices) = 0;
};
```

5.1.2.2: Update: Measurements

The update step must be able to incorporate any number and any type of measurement. There is no universal method to convert a measurement into the impact it should have on the particle weights. For this reason, Andrew decided to create an abstract `Sensor` class that would define this for itself:

include/pf/sensors/sensor.h (C++)

```
/**
 * @brief Updates weights with measurement.
 * @tparam P Particle type. Must be addable by P, and multipliable by float.
 * @tparam N Number of particles.
 */
template <typename P, size_t N> struct Sensor {
    /**
     * @brief Modifies weights param based on measurement.
     * Does not need to normalize weights.
     */
    virtual void update(const std::array<P, N>& particles,
                        std::array<float, N>& weights) = 0;
};
```

Importantly, `update()` takes a reference to the particles array rather than a copy, which ensures that time is not wasted copying the particles into a new array.

Program: Particle Filter

Since any number of Sensors may be used, the `update()` method of the `ParticleFilter` class takes a `std::vector<std::unique_ptr<Sensor>>&`. `std::vector` was chosen as it enabled the use of a variable number rather than a fixed number, which could be helpful in some cases. A `std::unique_ptr` was used to prevent any problems with dangling pointers.

Furthermore, the weights are assumed to be normalized for the resampler, so we must normalize them after incorporating the measurements.

include/pf/pf.h C++

```
28      /**
29       * @brief Used to allocate memory once for resampling. Is not shared
30       * between calls to the particle filter.
31       */
32      std::unique_ptr<std::array<size_t, N>> m_indices;
33
34      /**
35       * @brief Used to allocate memory once for resampling. Is not shared
36       * between calls to the particle filter.
37       */
38      std::unique_ptr<std::array<P, N>> m_newParticles;
39    public:
40      using Resampler = resamplers::Resampler<N>;
41      using Sensor = sensors::Sensor<P, N>;
…                                                 …
52      /**
53       * @brief Utility for normalizing weights
54       * @time O(N)
55       */
56      static void normalizeWeights(std::array<float, N>& weights) {
57        float sum = std::reduce(weights.begin(), weights.end());
58        // TODO: Investigate if this is necessary/correct
59        // Avoid round off to 0
60        sum += FLT_MIN * N;
61        for (float& weight : weights) {
62          weight += FLT_MIN;
63          weight /= sum;
```

include/pf/pf.h C++

```
64        }
65      }
…                                                   …
74      /** Updates weights, and resamples if needed. */
75      void update(std::vector<std::unique_ptr<Sensor>>& sensors,
76                  Resampler& resampler) {
77        m_mutex.take();
78        // Update weights
79        for (std::unique_ptr<Sensor>& sensor : sensors)
80          sensor->update(*m_particles, *m_weights);
81        // Normalize weights
82        normalizeWeights(*m_weights);
83
84        // TODO: Remove the need for m_indices and m_newParticles
85        // (Some resamplers may not need them)
86
87        // If should resample, then resample
88        if (resampler.shouldResample(*m_weights)) {
89          // printf("[%ims] Resampling!\n", pros::millis());
90          // Get indices of particles to keep
91          m_indices = resampler.resample(*m_weights, std::move(m_indices));
92          // Perform the actual resampling
93          for (size_t i = 0; i < N; i++)
94            (*m_newParticles)[i] = (*m_particles)[(*m_indices)[i]];
95          m_particles.swap(m_newParticles);
96          // Reset weights
97          std::fill(m_weights->begin(), m_weights->end(), 1.f / N);
98        }
99        m_mutex.give();
100     };
```

`m_indices` and `m_newParticles` are used so that `update()` doesn't need to reconstruct the arrays on every call, and thus improving performance.

5.1.3: Compute the Mean

Compute the weighted average like so:

Program: Particle Filter

$$\mu = \frac{1}{N}\sum_{i=1}^{N} w^i x^i$$

Here, the notation x^i indicates the i^{th} particle. The superscript is used as the subscript will often be used for time steps (e.g. x^i_{k+1}).

include/pf/pf.h C++

```
102     /**
103      * @brief Computes the weighted average of the particles, aka the state
104      * estimate.
105      * @time O(N)
106      */
107     P getMean() {
108       m_mutex.take();
109       P mean;
110       for (size_t i = 0; i < N; i++)
111         mean += (*m_particles)[i] * (*m_weights)[i];
112       m_mutex.give();
113       return mean;
114     }
```

5.1.4: Wrapping Up `ParticleFilter`

How should initial knowledge about the state be conveyed to the particle filter? One method would be to pass this knowledge when initially constructing the particle filter, but this wouldn't work well for an auton that could be run multiple times. Another would be to create a `Predictor` and use `predict()`, but this would be clunky and difficult to understand.

Instead, Andrew opted to provide a method that would call a callback on the data. This callback could then be used for other applications, such as printing the particles for debugging. `std::unique_ptr<>` is used again to clarify the intent of the callback and its arguments.

include/pf/pf.h C++

```
116     struct Data {
117         std::unique_ptr<std::array<P, N>> particles;
118         std::unique_ptr<std::array<float, N>> weights;
```

Program: Particle Filter

include/pf/pf.h C++

```
119     };
120
121     /**
122      * @brief Lets the particles and weights be modified by a function.
123      * @param func Function to given ability to modify the particles.
124      */
125     void modify(std::function<Data(Data)> func) {
126       m_mutex.take();
127       Data data {
128           .particles = std::move(m_particles),
129           .weights = std::move(m_weights),
130       };
131       Data newData = func(std::move(data));
132       m_particles = std::move(newData.particles);
133       m_weights = std::move(newData.weights);
134       m_mutex.give();
135     }
```

Finally, we must actually initialize the particle filter:

include/pf/pf.h C++

```
144     /** Create a particle filter. */
145     ParticleFilter(std::unique_ptr<std::array<P, N>> particles)
146       : m_particles(std::move(particles)),
147         m_weights(std::make_unique<std::array<float, N>>()),
148         m_indices(std::make_unique<std::array<size_t, N>>()),
149         m_newParticles(std::make_unique<std::array<P, N>>()) {
150       std::fill((*m_weights).begin(), (*m_weights).end(), 1. / N);
151     }
```

5.2: The `Localization` Class

Now, we must use the `ParticleFilter<P, N>` for MCL. 2654E said they used 500 particles with x and y as the state, and were able to achieve sub 5ms times. Thus, Andrew chose to use these parameters as a starting baseline. Heading

was not included, because trig operations are especially costly and heading doesn't accumulate error nearly as fast as x and y in Andrew's testing.

A 2d vector of $< x, y >$ is used to represent these particles. In the code, Andrew decided to use the Eigen library[1] for these vectors. Eigen is a linear algebra library for C++ which is used outside of V5RC for numerous applications. It is header only, so Liam Teale, the lead developer of LemLib, was able to easily port it to the v5 brain.

Rather than writing it himself, Andrew chose to use this library for a few reasons:

- Eigen would make use of the brain CPU's NEON processor,[2] which lets floating point operations run in parallel, thus reducing the processing time for each epoch.
- Andrew wouldn't have to write his own linear algebra functions, thus saving programming time.
- Eigen is specifically designed to be as performant as possible.

This meant that it would reduce the time to get the MCL working as efficiently as possible.

Eigen lets us configure the scalar storage type, which is simply the type that is used to represent the value of entry in the matrix/vector. Andrew chose to use floating point numbers rather than integers despite the slowness of floating point operations, because it would be easiest and if done wrong, integers could break things. Next, Andrew chose to use single-precision floats (`float`) rather double-precision (`double`), because operations on them would be faster and he theorized the extra precision would not be necessary. Both of these choices would be something that could be revisited later to improve performance.

---

[1] Eigen Library: https://eigen.tuxfamily.org
Eigen Template for PROS: https://github.com/lemlib/Eigen

[2] Arm NEON: https://www.arm.com/technologies/neon

Program: Particle Filter

Similarly to `std::vector` vs `std::array`, Eigen also supports dynamic and fixed sized vectors in the form of `Eigen::VectorXf` (dynamic) vs `Eigen::Vector2f` (fixed). Here, choosing the fixed length vector is very important as any performance gains will be multiplied by the number of particles.

Since the abstraction of the `ParticleFilter` is performed via composition rather than inheritance, the `Localization` does not extend the `ParticleFilter` class. Instead, it holds a `ParticleFilter` object, like so:

include/pf/localization.h C++

```
17  class Localization : Subsystem {
18    private:
19      pros::Mutex m_mutex;
20    public:
21      // Number of particles
22      static constexpr size_t N = 500;
23      /** Particle type */
24      typedef Eigen::Vector2f P;
25      /** Particle filter type */
26      typedef ParticleFilter<P, N> PF;
…                                          …
52    private:
53      PF m_pf;
…                                          …
104 };
```

`Localization` extends `Subsystem`, so that it runs every 10ms, as this is how quickly new sensor data is made available to the program. Similar to the `ParticleFilter`, `Localization` also has a `m_mutex` member in order to prevent any multitasking edge cases.

Here, Andrew uses the `typedef` keyword to define a type alias for `P` and `PF`, and the and `constexpr` keyword to define a compile time value. This is done to make the code a bit more succinct, and will be very helpful later.

Program: Particle Filter

To debug `Localization` easily, it was critical that it didn't require a robot to test. Andrew decided to use the vexide's vex-v5-qemu,[1] which enables robot code to be tested without a robot, without compilation for a different target. Unfortunately, this sim doesn't yet implement devices, and Andrew decided it would be too difficult to contribute this.

Instead, Andrew decided to simulate the sensors within the user program, by simply providing faked sensors and odometry to the `Localization` object. This made it much easier to simulate the sensors, especially for odometry, as only the pose measurements would have to be calculated, rather than having to figure out the sensor measurements that should be read.

Now we must implement the `Predictor`, `Resampler`, and `Sensor` classes, whilst being careful to enable them to be simulated.

5.2.1: Implementing `Predictor`

For our prediction algorithm, we will use the displacement measurements from odometry. Why not implement odometry as a sensor? Well, since odometry is simple to use to move the particles, we can implement it here by just adding the x and y displacement to the particles. This makes it a little easier for the PF to figure out what's going on and may be more performant than the alternative. Additionally, this method was proven to work by 2654E Echo.

Despite currently using LemLib's odometry, Andrew wanted the ability to swap in different odometry implementations and use simulated odom. Therefore, `Localization` contains an abstract `std::shared_ptr<Odom>` object, enabling abstraction by composition. This `Odom` abstract odom class is defined like so:

include/pf/odom/odom.h C++

```
5  /**
6   * @brief Interface for odometry sensors.
```

[1] vex-v5-qemu: https://github.com/vexide/vex-v5-qemu

Program: Particle Filter

include/pf/odom/odom.h

```cpp
 */
class Odom {
  public:
    /** @brief Calculates new pose with new data */
    virtual void update() = 0;

    /** @returns Const ref to current pose. Theta is in standard radians. */
    virtual const lemlib::Pose& getPose() const = 0;

    /** @brief Sets pose to newPose. Theta must be in standard radians. */
    virtual void setPose(lemlib::Pose pose) = 0;

    /** @brief Calibrates sensors */
    virtual void calibrate() = 0;

    /** @returns Whether the sensors have been calibrated yet */
    virtual bool hasCalibrated() const = 0;
};
```

Then, LemLib's odom can be wrapped in a `Odom` class like so:

include/pf/odom/lemlib.h

```cpp
/**
 * @brief Object wrapper for LemLib odometry using tracking wheels.
 */
class LemLibOdom : public Odom {
  private:
    /** Used to setup odom */
    lemlib::OdomSensors m_sensors;
    /** Used to setup odom */
    lemlib::Drivetrain m_drivetrain;

    /** @brief Updated by update() and setPose(). Used to return references.
     * Theta is in standard radians. */
    lemlib::Pose m_pose;

    bool m_hasCalibrated = false;
  public:
    /** @returns Const ref to current pose. Theta is in standard radians. */
    const lemlib::Pose& getPose() const override;
    /** @returns Mutable ref to current pose. Theta is in standard radians. */
    lemlib::Pose& getPose();
```

include/pf/odom/lemlib.h

```cpp
    /** @brief Sets pose to newPose. Theta must be in standard radians. */
    void setPose(lemlib::Pose newPose) override;

    /** @brief Calibrates odometry sensors, if it hasn't been done before */
    void calibrate() override;

    bool hasCalibrated() const override;

    /** Calculates new pose using sensor data. */
    void update() override;

    LemLibOdom(lemlib::OdomSensors sensors, lemlib::Drivetrain drivetrain);
};
```

src/pf/odom/lemlib.cpp

```cpp
const lemlib::Pose& LemLibOdom::getPose() const { return m_pose; }

lemlib::Pose& LemLibOdom::getPose() { return m_pose; }

void LemLibOdom::setPose(lemlib::Pose pose) {
  m_pose = pose;
```

Program: Particle Filter

src/pf/odom/lemlib.cpp (C++)

```
12    // Convert to compass radians
13    pose.theta = M_PI_2 - pose.theta;
14    lemlib::setPose(pose, true);
15  }
16
17  void LemLibOdom::update() {
18    if (m_hasCalibrated) {
19      lemlib::update();
20      m_pose = lemlib::getPose(true);
21      // Convert to standard radians
22      m_pose.theta = M_PI_2 - m_pose.theta;
23    }
24  }
25
26  bool LemLibOdom::hasCalibrated() const { return
    m_hasCalibrated; }
27
28  LemLibOdom::LemLibOdom(lemlib::OdomSensors sensors,
29                         lemlib::Drivetrain drivetrain)
30    : m_sensors(sensors), m_drivetrain(drivetrain), m_pose {0, 0}
    {}
…                                  …
75  void LemLibOdom::calibrate() {
76    if (hasCalibrated()) return;
77
78    const bool calibrateImu = true;
```

src/pf/odom/lemlib.cpp (C++)

```
79    // calibrate the IMU if it exists and the user doesn't specify
      otherwise
80    if (m_sensors.imu != nullptr && calibrateImu)
      calibrateIMU(m_sensors);
81    // initialize odom
82    if (m_sensors.vertical1 == nullptr)
83      m_sensors.vertical1 = new lemlib::TrackingWheel(
84          m_drivetrain.leftMotors, m_drivetrain.wheelDiameter,
85          -(m_drivetrain.trackWidth / 2), m_drivetrain.rpm);
86    if (m_sensors.vertical2 == nullptr)
87      m_sensors.vertical2 = new lemlib::TrackingWheel(
88          m_drivetrain.rightMotors, m_drivetrain.wheelDiameter,
89          m_drivetrain.trackWidth / 2, m_drivetrain.rpm);
90    m_sensors.vertical1->reset();
91    m_sensors.vertical2->reset();
92    if (m_sensors.horizontal1 != nullptr) m_sensors.horizontal1-
      >reset();
93    if (m_sensors.horizontal2 != nullptr) m_sensors.horizontal2-
      >reset();
94    setSensors(m_sensors, m_drivetrain);
95
96    // rumble to controller to indicate success
97    pros::c::controller_rumble(pros::E_CONTROLLER_MASTER, ".");
98    m_hasCalibrated = true;
99  }
```

To actually use this odom data for the particle filter, `Localization` has a nested `struct` named `OdomPredictor` which extends `PF::Predictor`. This `OdomPredictor` has two roles:

- Move the particles according to displacement measurements
- Incorporate noise

To add noise, 2 dimensional gaussian, or normal, distribution will be sampled from. Sampling from a multidimensional gaussian is surprisingly annoying to do efficiently, and thus Andrew wrote a helper for it:

include/pf/util/stats.h (C++)

```
35  template <size_t N> struct MultivariateGaussian {
36      Eigen::Vector<float, N> mu;
37      Eigen::Matrix<float, N, N> sigma;
38
39      class Sampler {
40        private:
41          const MultivariateGaussian m_dist;
42          std::normal_distribution<float> m_rngDist;
43          const Eigen::Matrix<float, N, N> L;
44          Eigen::Vector<float, N> z;
```

include/pf/util/stats.h (C++)

```
45
46          friend MultivariateGaussian;
47
48          Sampler(const MultivariateGaussian& dist)
49            : m_dist(dist), m_rngDist {0, 1},
              L(m_dist.sigma.llt().matrixL()) {}
50        public:
51          /**
52           * @brief Samples once from the distribution.
53           *
```

Program: Particle Filter

include/pf/util/stats.h (C++)

```
54          * @param result Where to store the sample
55          */
56         void sampleOnce(Eigen::Vector<float, N>& result) {
57           for (size_t i = 0; i < N; i++) { z(i) =
             m_rngDist(rng); }
58           result = m_dist.mu + L * z;
59         }
60
61         /**
62          * @brief Samples many times from the distribution.
63          *
64          * @tparam M Number of samples
65          * @param result Where to store the samples
66          */
67         template <size_t M>
68         void sampleMany(std::array<Eigen::Vector<float, N>, M>&
           result) {
69           for (auto& sample : result) { sampleOnce(sample); }
70         }
71       };
72
73       Sampler makeSampler() const { return Sampler(*this); }
74
75       /**
76        * @brief Calculates the mean of the samples.
77        *
78        * @tparam M Number of samples
79        * @param samples Array of samples
80        * @return Eigen::Vector<float, N> Mean of the samples
81        *
82        * @time ignoring N: O(M)
83        */
84       template <size_t M> static Eigen::Vector<float, N>
85       calcMuFromSamples(const std::array<Eigen::Vector<float, N>,
          M>& samples) {
86         Eigen::Vector<float, N> mu = Eigen::Vector<float,
           N>::Zero();
```

include/pf/util/stats.h (C++)

```
87         for (const auto& sample : samples) mu += sample / M;
88         // mu /= M;
89         return mu;
90       }
91
92       /**
93        * @brief Calculates the properties of the samples,
          assuming they are
94        * gaussian.
95        *
96        * @tparam M Number of samples
97        * @param samples Array of samples
98        * @return MultivariateGaussian representation of the
          samples
99        *
100       * @time ignoring N: O(M)
101       */
102      template <size_t M> static MultivariateGaussian<N>
103      fromSamples(const std::array<Eigen::Vector<float, N>, M>&
         samples) {
104        // TODO: investigate performance and numerical stability,
           improving it if
105        // necessary
106        Eigen::Vector<float, N> mu = calcMuFromSamples(samples);
107
108        Eigen::Matrix<float, N, N> sigma = Eigen::Matrix<float,
           N, N>::Zero();
109        for (const auto& sample : samples) {
110          Eigen::Vector<float, N> diff = sample - mu;
111          sigma += diff * diff.transpose() / M;
112        }
113        // sigma /= M;
114
115        return {.mu = mu, .sigma = sigma};
116      }
117    };
```

A separate sampler object is used to share functionality between sampling once and multiple times whilst still caching the data that is required for each sample.

To create a `MultivariateGaussian`, we need a mean and a covariance[1] matrix. The noise should be unbiased, and thus centered around 0. The covariance is something that should be tuned, so it will be exposed as a parameter.

[1] Covariance: A measure of the corelation between random variables

Program: Particle Filter

To tell the `OdomPredictor` how much displacement has been measured, Andrew opted to mutate a member of `OdomPredictor`, as this was the simplest possible solution. This results in:

include/pf/localization.h C++

```
59      /** @brief Moves particle based on lemlib odom readings */
60      struct OdomPredictor : PF::Predictor {
61          /**
62           * @brief Represents noise in system model.
63           * @symbol Q
64           */
65          Eigen::Matrix2f m_covar;
66          /** @brief Represents change in position, according to LemLib odom. */
67          P m_displacement;
68          /** @brief Updates particles according to LemLib odom. */
69          void predict(std::array<P, N>& particles) override;
70
71          OdomPredictor(Eigen::Matrix2f covar);
72      } m_predictor;
```

src/pf/localization.cpp C++

```
129 void Localization::OdomPredictor::predict(std::array<P, N>& particles) {
130   auto noiseSampler =
131       stats::MultivariateGaussian<2> {.mu = {0, 0}, .sigma = m_covar}
132           .makeSampler();
133
134   Eigen::Vector<float, 2> noise;
135   for (auto& particle : particles) {
136     noiseSampler.sampleOnce(noise);
137     particle += m_displacement + noise;
138   }
139 }
140
141 Localization::OdomPredictor::OdomPredictor(Eigen::Matrix2f covar)
142   : m_covar(covar), m_displacement({0, 0}) {}
```

5.2.2: Resampler

For resampling, Andrew chose to start with the systematic resampler as it was simple to implement and should be quite effective. The easiest way to understand systematic resampling is with a diagram:

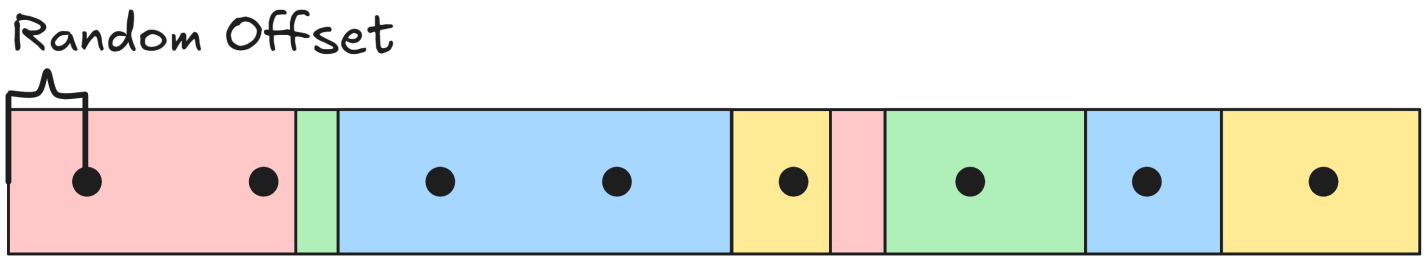

Figure 27.3: Diagram of Systematic Resampling. Boxes represent a particles where the width represents the weight of the particle.

Essentially, we create a dot with a random offset between **0** and $\frac{1}{N}$. Then we find the box, or particle in which the dot is in. Next, we record this particle's index and make a new dot $\frac{1}{N}$ away from the previous dot. The actual implementation of this algorithm looks like this:

include/pf/resamplers/systematic.h (C++)

```
9  /** @brief Implements Systematic Resampling */
10 template <size_t N> class Systematic : public Resampler<N>
   {
11   private:
12     /** @brief Returns random number between 0 and 1 */
13     std::function<float()> m_random;
14   public:
15     std::unique_ptr<std::array<size_t, N>> resample(const
       std::array<float, N>& weights,
16                   std::unique_ptr<std::array<size_t, N>>
                     indices) override{
17       float offset = m_random() / N;
18
19       /** Current sum. */
20       float currSum = 0;
21       /** Value we want to find. */
22       float position = offset;
23       /** Index of weights */
24       size_t i_w = 0;
25       /** Index of new indices */
26       size_t i_i = 0;
27
```

include/pf/resamplers/systematic.h (C++)

```
28       for (; i_w < N; i_w++) {
29         // Update sum
30         currSum += weights[i_w];
31
32         // If current weight is at desired position
33         while (currSum > position) {
34           // Then add it to the indices
35           (*indices)[i_i] = i_w;
36           ++i_i;
37           // And move to next position
38           position += 1.f / N;
39         }
40       }
41
42       return std::move(indices);
43     };
44
45     /** @param random Returns random number between 0 and
       1 */
46     Systematic(std::function<float()> randomFunc) :
       m_random(randomFunc) {}
47 };
```

Program: Particle Filter

5.2.3: Sensors

Since any kind of sensor could be used, such as a distance sensor, a vision sensor or a line tracker, Andrew wanted to make sure that the `Localization` class worked with any type of sensor. To enable this, `Localization` is not be responsible for constructing the sensors, and instead takes them as parameters. Andrew chose to use `std::vector<std::shared_ptr<Sensor>>` to store the sensors inside `Localization`, because this would enable the sensors to be accessed outside `Localization` if needed.

For now we will only be using distance sensors, so Andrew only had to implement the Sensor interface for the distance sensor. In the code, Andrew decided to name the distance sensor class `Laser`, as this was more succinct and sounded cooler. `Laser` would be an abstract class, enabling code to be shared between the simulated laser and the real v5 laser. The primary goal of the laser class is to figure out how likely a particle is given a certain distance measurement.

This is implemented using Bayes' theorem which is as follows:

$$P(x|z) = \frac{P(z|x)P(x)}{P(z)}$$

- $P(x|z)$ is the probability of x, our particle being the true state, given z, our measurement, which is what we want.
- $P(z|x)$ is the probability of the measurement given the particle is the true state which is easy for us to compute.
- $P(x)$ is the probability of the particle being the true state without knowing the measurement. This is equivalent to the weight of the particle before incorporating the measurement, and is often called the prior.
- $P(z)$ is the probability of the measurement without knowing anything about the state. Rather than calculating this, we instead normalize all the particles which has the same effect.

Program: Particle Filter

This leaves pretty much just $P(z|x)$ for Andrew to figure out. To simplify this, Andrew assumes that the distance measurement is gaussian. The gaussian's mean should be the measurement and its variance will be provided by the subclass. Therefore, $P(z|x)$ can be calculated by evaluating the gaussian's probability density function at the expected measurement of the particle.

To find the expected measurement of the particle, Andrew use a modified version of algorithm presented in this Stack Overflow post:[1]

include/pf/util/lines.h C++

```
10 struct Segment {
11     /** Start point */
12     Eigen::Vector2f start;
13     /**
14      * @brief Represents the vector from start that
        creates the end points
15      * end = start + vec
16      */
17     Eigen::Vector2f vec;
…                              …
28 };
29
30 struct Ray {
31     Eigen::Vector2f start;
32     /** Must have a length of 1 */
33     Eigen::Vector2f direction;
34
35     /**
36      * @brief Finds the distance to line segment
37      * @returns Distance to line segment if the lines
        intersect.
38      * @returns std::nullopt if the lines do not
        intersect.
39      */
40     std::optional<float> distanceTo(const Segment&
        other) const;
…                              …
54 };
```

src/pf/util/lines.cpp C++

```
22 std::optional<float> Ray::distanceTo(const Segment&
   other) const {
23   // Source: https://stackoverflow.com/a/565282
24
25   auto& q = other.start;
26   auto& s = other.vec;
27   auto& p = start;
28   auto& r = direction;
29
30   const float r_x_s = r.cross(s);
31   // Ray and segment are parallel / colinear
32   if (r_x_s == 0) return std::nullopt;
33
34   const auto q_minus_p = q - p;
35
36   // u = ((q - p) x r) / (r x s)
37   const float u = (q_minus_p.cross(r)) / r_x_s;
38
39   // Ray doesn't hit segment
40   if (u < 0 || u > 1) return std::nullopt;
41
42   // t = ((q - p) x s) / (r x s)
43   // Represents distance
44   const float t = (q_minus_p.cross(s)) / r_x_s;
45
46   // Ray facing away from segment
47   if (t < 0) return std::nullopt;
48
49   return t;
50 }
```

[1] https://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect/565282#565282

Then, to use this function, Andrew must be able to convert the laser into a ray object, so `Laser` exposes a `toRay()` function. This requires a knowledge about the particle pose and the relative pose of the laser to the robot. The position of the particle comes from the particle itself and a reference to the current odometry heading. The relative position is stored inside the laser and is configured when the laser is constructed.

include/pf/sensors/laser.h C++

```
25    protected:
26      using Vec2 = Eigen::Vector2f;
…                                                    …
30    protected:
31      /** Position of sensor relative to center of bot. */
32      const Vec2 m_pos;
33      /** Angle of sensor relative to bot. Should be in standard radians. */
34      const float m_theta;
35
36      /** Holds theta of bot in standard radians. */
37      const float& m_botTheta;
…                                                    …
40    public:
…                                                    …
53      /** Creates a ray from this laser object. */
54      lines::Ray toRay(const Vec2& robotPos) const;
```

src/pf/sensors/laser.cpp C++

```
8   lines::Ray Laser::toRay(const Vec2& robotPos) const {
9     const float theta = m_botTheta + m_theta;
10    const float hypot = m_pos.norm();
11
12    return {robotPos + Vec2(hypot * Vec2 {cos(theta), sin(theta)}),
13            {cos(theta), sin(theta)}};
```

To get the measurement from the subclass, the `Laser` class exposes a pure `virtual sense()` method which returns a `std::optional<Measurement>`. `std::optional` was chosen due to the capacity for a sensor to not return any-

thing. For example, this could occur if the object is too far for the laser to sense or the laser is disconnected.

`Measurement` is a struct that holds both a `float dist` and `float stdDev` which represent the distance and the standard deviation of the measurements from the sensor.

include/pf/sensors/laser.h (C++)

```
40    public:
41      struct Measurement {
42          float dist;
43          float stdDev;
44      };
45
46      /**
47       * @brief Returns the distance sensed by the sensor and the variance in the
48       * reading.
49       * @return Measurement of the sensor.
50       */
51      virtual std::optional<Measurement> sense() = 0;
```

To extend the sensor class, a separate nested class is used, in order to prevent all of the subclasses and the `Laser` class from needing to be in the header file due templates.

include/pf/sensors/laser.h (C++)

```
56    /**
57     * @brief Implements the PF sensor for the laser sensor.
58     * Also prevents the rest of the Laser code from being in
       the header file.
59     *
60     * @tparam N Number of particles/weights.
61     */
62    template <size_t N> struct Sensor
63      : public sensors::Sensor<Eigen::Vector2f, N> {
64        Laser& laser;
65
66        void update(const std::array<Vec2, N>& particles,
67                    std::array<float, N>& weights) override {
68          auto measurement_opt = laser.sense();
69          if (!measurement_opt) return;
70          auto& measurement = *measurement_opt;
71          lines::Ray ray = laser.toRay({0, 0});
```

include/pf/sensors/laser.h (C++)

```
72          const Vec2 startOffset = ray.start;
73          const stats::NormPDF pdf {measurement.dist,
            measurement.stdDev};
74
75          ray.start = particles[0];
76
77          // TODO: Improve safety (ray might not intersect
            anything)
78          // Assume all particles will intersect the same line
79          auto seg = laser.obstacles[ray.findFirstIntersect(laser
80                                          .value_or(std::pair
                                            {0.f, 0u})
81                                          .second];
82
83          for (size_t i = 0; i < particles.size(); i++) {
84            // TODO: make cleaner
85            ray.start = particles[i] + startOffset;
86            auto intersectOpt = ray.distanceTo(seg);
```

Program: Particle Filter

include/pf/sensors/laser.h C++

```
87            if (const auto intersect = intersectOpt) {
88              weights[i] *= pdf.eval(*intersect);
89            } else {
90              if (const auto intersect =
91                      ray.findFirstIntersect(laser.obstacles)) {
92                weights[i] *= pdf.eval(intersect->first);
93              } else weights[i] = 0;
94            }
```

include/pf/sensors/laser.h C++

```
95          }
96        };
97
98        Sensor(Laser& laser) : laser(laser) {}
99    };
```

Finally, this can be easily implemented by the `V5Laser` class like so:

include/pf/sensors/v5laser.h C++

```
8   /**
9    * @brief Implements the laser sensor for the V5 laser sensor.
10   */
11  class V5Laser : public Laser {
12    private:
13      pros::Distance& m_sensor;
14
15      template <size_t N> friend struct V5LaserSensor;
16
17      V5Laser(const Laser::Config config, pros::Distance& sensor);
18    public:
19      std::optional<Measurement> sense() override;
20  };
21
22  /**
23   * @brief Implements the PF sensor for the V5 laser sensor.
24   */
25  template <size_t N> struct V5LaserSensor : public Laser::Sensor<N> {
26      V5Laser m_laser;
27
28      V5LaserSensor(const Laser::Config config, pros::Distance& sensor)
29        : m_laser(config, sensor), Laser::Sensor<N>(m_laser) {}
30    private:
31      using Laser::Sensor<N>::laser;
32  };
```