Particle Filter for Robot Localization

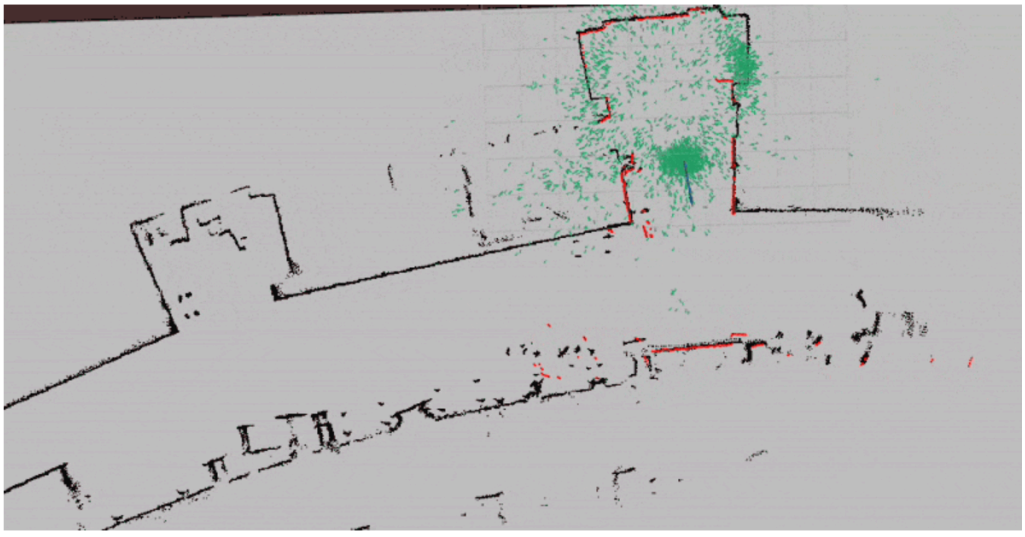
By: Andrew Kurtz and Ananth Kutuva

10/23/2025

Introduction

In this project, we designed a particle filter to perform robot localization given a map. A particle filter is an algorithm that uses brute force to compare real sensor readings to many synthetic sensor readings generated from simulated robots (called particles!) spread across a map. The locations of the particles whose synthetic sensor readings more closely match the real robot are prioritized in a resampling step. The particles also match the robot's movements in the simulated world, which enables the filter to differentiate between similar locations. The real robot's location can then be deduced from the particles' distribution.

This picture illustrates the particle filter in action. The green arrows are particles, and the blue arrow is the estimated robot's pose. The red dots show the robot's current laser scan transformed into the map frame using the estimated robot pose. The better the red dots align with the map, the better the pose. An animated version of this picture, showing the particle filter in action, can be found in the README of this repository.



Overview

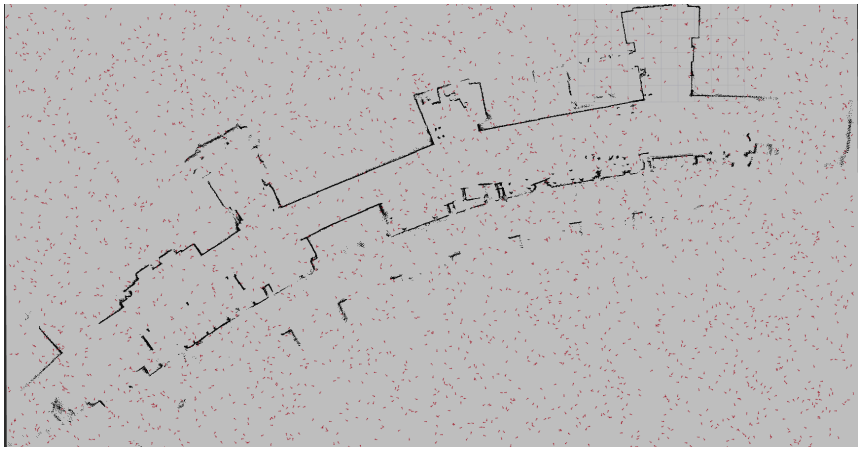
To make the algorithm more concrete, we outline the high level overview of the algorithm in pseudocode form here. Each of the named functions represent key design decisions in the algorithm, so detailed overviews of each is given in the implementation section.

```
# Initialize particles. If given pose estimate sample around it,
otherwise randomly across map
initialize_particle_cloud()

# Main particle filter loop
while running particle filter:
      read odom data
      read corresponding lidar data
      if moved or turned far enough since last update:
            # Update each particle's location to match robot movement
            update_particles_with_odom()

            # Compare laser scan to particle's surroundings to update
            each particle's weight
            update_particles_with_laser()

            # Given the weighted distribution, estimate robot's location
            update_robot_pose()

            # Resample particles based weighted distribution
            resample_particles()
```

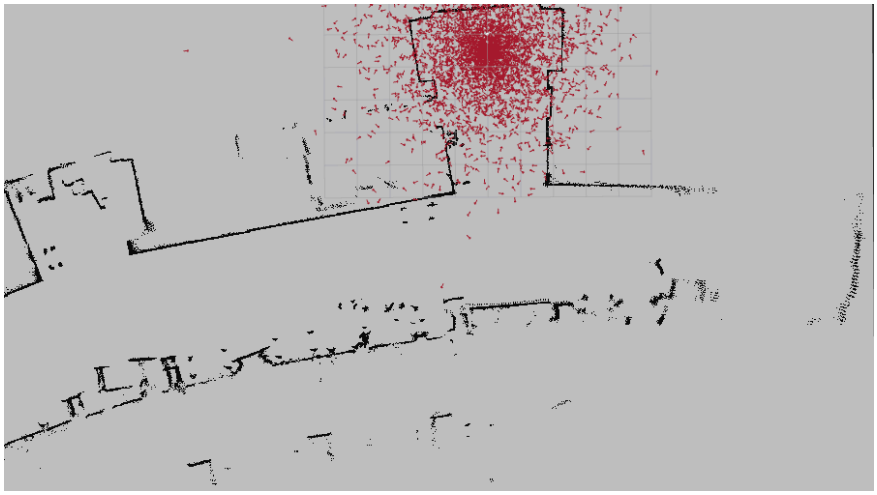
Implementation

Particle Initialization - initialize_particle_cloud()

To initialize the particles, there are two options depending on what information is given. If no initial pose estimate is given, it distributes the particles randomly across the map with random headings as shown in the picture below. This approach works because, given enough particles, it will explore every part of the map enough to know what to explore further through resampling. If the map has impossible places (outside the building), these particles should be filtered out due to high scan error.



If an initial pose estimate is provided, it samples particles from a circle with a radius of 2 meters centered on the initial estimate, as shown in this picture.



Mathematically, if the initial pose estimate is $(\hat{x}, \hat{y})$, for particle i, it samples a radius and angle in polar coordinates, then converts to Cartesian according to these equations.

$$radius_i \sim Normal(0, 2)$$
$$\theta_i \sim Uniform(0, 2\pi)$$
$$x_i = \hat{x} + radius_i * cos(\theta_i)$$
$$y_i = \hat{y} + radius_i * sin(\theta_i)$$

Adding noise this way enables convergence even if the estimate is up to 2 meters off in each direction.

Motion Update - update_particles_with_odom()

The motion update can be split into two parts. The first part is updating the particle's locations, and the second is their heading direction. To conceptualize updating the particle's location, we thought of it in two steps. The first is calculating how far the robot moved relative to the last pose. The second is applying this relative motion to each particle.

To find how far the robot moved relative to the last pose, you can subtract the coordinates between the poses in the odom frame to get a difference vector.

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} \Delta x_t \\ \Delta y_t \end{bmatrix} - \begin{bmatrix} \Delta x_{t-1} \\ \Delta y_{t-1} \end{bmatrix}$$

This vector won't be relative to the heading in the last pose, so we need to rotate it to break it up into the parts that are straight forward and side to side. We can leverage the fact that the robot and odom frame was initialized pointing along the positive x-axis. This allows us to rotate the difference vector back to the initialization rotation, so that the x coordinate will correspond to forward and the y to side to side. I will use $\widehat{\Delta x}$ and $\widehat{\Delta y}$ to denote this rotated vector, and θ_{t-1} corresponds to the robot's heading in the odom frame in the last pose.

$$\begin{bmatrix} \widehat{\Delta x} \\ \widehat{\Delta y} \end{bmatrix} = \begin{bmatrix} cos(-\theta_{i-1}) & -sin(-\theta_{i-1}) \\ sin(-\theta_{i-1}) & cos(-\theta_{i-1}) \end{bmatrix} * \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

Given $\widehat{\Delta x}$ and $\widehat{\Delta y}$ you can find the new location of each particle x_{p_t}, and y_{p_t} by rotating this relative difference vector by $\theta_{p_{t-1}}$ to align the relative movement to the particle's heading. Then, add this vector to the particle's position to get the new position

$$\begin{bmatrix} x_{p_t} \\ y_{p_t} \end{bmatrix} = \begin{bmatrix} \Delta x_{t-1} \\ \Delta y_{t-1} \end{bmatrix} + \begin{bmatrix} cos(\theta_{p_{t-1}}) & -sin(\theta_{p_{t-1}}) \\ sin(\theta_{p_{t-1}}) & cos(\theta_{p_{t-1}}) \end{bmatrix} * \begin{bmatrix} \widehat{\Delta x} \\ \widehat{\Delta y} \end{bmatrix}$$

The new particle's angle is much simpler. It is simply the difference between the robot's positions plus the particle's old angle.

$$\theta_{p_t} = \theta_{p_{t-1}} + (\theta_t - \theta_{t-1})$$

Particle Weighting - update_particles_with_laser()

This method takes the laser scan data from the actual robot and projects it onto each of the particles, using the nearest object algorithm to assign a weight to each particle based on the error between the actual scan data and the objects around the particle.

Given an array of radius and theta values from the scan data of the Neato, we would iterate through them and remove radius values greater than a certain value and their corresponding theta values.

For each scan endpoint around the original robot, we converted its location from polar (r, theta) to cartesian (x, y) using these equations.

$$x = radius * cos(\theta)$$
$$y = radius * sin(\theta)$$

We iterated through the particles and used their current heading to create a rotation matrix (shown below), which rotates the original scan endpoints using matrix multiplication to align with each particle in the world frame.

$$\begin{bmatrix} x_{rot} \\ y_{rot} \end{bmatrix} = \begin{bmatrix} cos(\theta_{particle}) & -sin(\theta_{particle}) \\ sin(\theta_{particle}) & cos(\theta_{particle}) \end{bmatrix} * \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

We then translated the scan endpoints, projecting them onto the particle frame using the given particle x and y coordinates and these equations.

$$x_f = x_{rot} + x_{particle}$$
$$y_f = y_{rot} + y_{particle}$$

Next, for each particle, we iterated through each scan endpoint and used the Occupancy Field to find the closest obstacle to the endpoint. We made a list of these raw error values for each particle location. If the endpoint was outside of the map, we gave it a large error instead of finding the distance to an object.

To turn these errors into an initial weight, $\widehat{w_p}$, for the particle, we plugged each endpoint error into an exponential decay equation, then averaged them across particles. We found the exponential decay to be useful in distinguishing good scan matches from ok matches. This is shown in the equation below, where N is the number of scan endpoints per particle and $error_i$ is the error of the ith endpoint.

$$\widehat{w_p} = \frac{1}{N}\sum_{i=0}^{N} e^{-100 * (error_i)^2}$$

To turn $\widehat{w_p}$ into the actual particle weight, w_p, you can normalize by the sum of the weights across all particles. The equation below shows this, where N is the number of particles.

$$w_p = \frac{\widehat{w_p}}{\sum_{i=o}^{N} w_i}$$

Update Estimated Robot Position - update_robot_pose()

To estimate the robot's pose from the cloud of particles, we average the position and directions of the five particles with the greatest weights. While this is a simple approach that certainly isn't perfect, it works well and has a couple of key advantages. The first advantage is that if you have multiple clouds, it will pick the one with the best scan matches, rather than one with the most particles. This makes it correct from a poor initial convergence faster because if a few particles discover a better location with better scans, it can immediately switch to that location rather than

waiting a few iterations for the majority of particles to resample to that location. Another benefit is that it is interpretable and doesn't require tuning. We considered using a clustering algorithm like density and spatial-based clustering, but decided against it because it would require tuning depending on the number of particles and map size, making it harder to test and iterate quickly.

The main disadvantage of this technique is that if the 5 best particles are split across multiple clusters, then the average will be somewhere between them, rather than trying to pick the more likely cluster.

Resampling Particles - resample_particles()

To resample a new particle cloud from the current weighted cloud, we select a particle from the distribution, then add some noise to its position and angle. Adding this noise enables us to search the space around the particle in an effort to get as close to the robot's location as possible. Without it, the algorithm would settle for a 'good enough' location.

Mathematically, let's say we sample from the particle distribution and get particle p with coordinates $(\hat{x}\ ,\ \hat{y})$ and angle $\hat{\theta}$. The new particle's location will be noted by $(x,\ y)$ with θ.

Similarly to particle initialization from a pose estimate, the noise will be generated from a circular Gaussian distribution around the sampled point. To distinguish between the positional noise generated from polar coordinates and heading direction noise, the polar angle is $\theta_{position\ noise}$ and the heading noise is $\theta_{heading\ noise}$.

$$radius_{position\ \ noise} \sim Normal(0,\ 0.1)$$
$$\theta_{position\ noise} \sim Uniform(0,\ 2\pi)$$
$$x\ =\ \hat{x}\ +\ radius_{noise}\ *\ cos(\theta_{position\ noise})$$
$$y\ =\ \hat{y}\ +\ radius_{noise}\ *\ sin(\theta_{position\ noise})$$

To get the new angle, we can sample angular noise with a standard deviation of 0.26 radians, which is about 15 degrees.

$$\theta_{heading\ noise} \sim Normal(0,\ 0.26)$$
$$\theta\ =\hat{\theta}\ +\ \theta_{heading\ noise}$$

Challenges

Some challenges we faced were conceptually understanding the implementation of the particle filter before writing any lines of code. We originally thought we were comparing every synthetic laser scan beam of each particle to the actual Neato and formulating an error value based on this. We realized it was too computationally expensive to calculate an exact synthetic laser scan

for each particle in each time step, so we needed to use a likelihood field to precompute an approximation. It took us a few rounds of ideation and pseudocode, but we were eventually able to concretely understand how we could compare the scans from the Neato, convert them to a usable format for our nearest distance algorithm, project them onto the particles, and calculate a weight for that particle.

Once we had the code written and had tested the functionality of each function separately, it was challenging to make the particle filter work as a whole and properly converge. Initially, there was too much noise being added to the angle in the resampling step. This caused the motion update to disperse the particles and prevent convergence. The next problem was slow and incorrect convergence. After adding debugging statements to print the distribution of the particle's weights, we realized our error to weight conversion was resulting in good particles having too similar of weights to bad particles, meaning the resampling wasn't focusing on the good areas. We switched our error to weight conversion to be a negative exponential, meaning a high error would get an exponentially low weight, and a low error would get a much higher weight. Finally, it would converge well when given an initial pose estimate, but not without one. To compensate for this, we increased the number of particles by a factor of 10 to 3000, which enabled the whole map to be properly sampled.

Improvements

There are several ways we could improve the particle filter. First, we could restrict particle generation to occur only within the valid area of the MAC building rather than across the entire map. This would eliminate particles that begin outside the building and never contribute useful information, allowing us to reduce the total number of particles while maintaining accuracy.

Second, we could develop a more robust method for estimating the robot's position. Currently, the robot's estimated pose is determined by averaging the positions and headings of the five highest-weighted particles. While simple, this approach can produce inaccurate results if the best particles are split between multiple clusters. A better approach would involve clustering particles based on both spatial proximity and weight, then selecting the cluster with the greatest total weight as the estimated robot position.

Finally, converting more of the for loop computations into NumPy operations would improve computational efficiency and convergence speed. This would allow the algorithm to process more particles in less time, which could significantly improve both accuracy and speed.

Lessons

One of the main takeaways was how to tackle a complex problem when you don't know where to start. The process that worked well in this project was to gain a strong understanding of the task, especially the relevant math if the task requires it. Then, you can begin to think about how you can turn this into logic and a repeatable process by developing pseudocode to plan the implementation step by step. This is also a good time to think through what existing code you

can use. Finally, once it is all planned out, you can actually write the code. Sometimes it seems like these are extra steps, but we found it was a lot faster to write the code and debug it when it comes from a strong foundation, and you know what should be happening at each step.

Another takeaway was how to work with starter code or an existing codebase that seems complicated and unapproachable at first. It was useful for us to develop a strong, high-level understanding of the problem and approach first, then read through the code to fill in the details and answer questions. This enabled us to create a strong mental model that was both conceptual and rooted in the code. It was also helpful for the last step in the planning process when we were figuring out what code existed and what we needed to write.

The final takeaway was to test everything individually before combining behaviors and running the system as a whole. Depending on what you're testing, it may take some creativity to test an individual step. One technique that was useful to do this was to change parameters or to temporarily modify the code. For example, to debug the odom update step, we reduced the number of particles to 1 and turned off the resampling step. Testing as we went allowed us to catch lots of bugs that would've been much harder to track down if they were only a step in the larger algorithm that had downstream consequences.