Particle Filter for Robot Localization

Cole Marco, Swasti Jain

October 17, 2023

1 Introduction

Robot localization is a fundamental problem in robotics, where the goal is to determine the precise position and orientation (pose) of a robot within its environment. One widely used approach for solving this problem is the Particle Filter (PF) algorithm. This conceptual writeup will explain what a particle filter is, how to use it for robot localization, and our implementation of it.

2 What is a Particle Filter?

A particle filter is a Bayesian filtering technique commonly used in probabilistic robotics and estimation. It serves as a recursive, sequential Monte Carlo method that aids in the estimation of a dynamic system's state while accounting for uncertainty. Its versatility makes it suitable for various applications, such as simultaneous localization and mapping (SLAM), object tracking, and state estimation in systems with nonlinear and non-Gaussian characteristics.

Particle filters find their utility in scenarios where the objective is to estimate a system's evolving state in the presence of uncertainty. This could encompass anything from tracking the position and velocity of a moving object to determining the configuration of a robotic system or any other dynamic parameter.

The foundation of particle filters lies in Bayesian filtering, where they maintain a probability distribution over potential states of the system. This distribution, known as the posterior distribution, continually updates as new observations become available.

The core of a particle filter is its particles, each representing a plausible state of the system. These particles are drawn from the posterior distribution and evolve over time in accordance with the system's dynamics and the incoming observations.

Particle filters operate in two fundamental steps:

Prediction Step: In this step, particles are advanced in time according to the system's dynamic model. This represents how the system's state evolves over time when no new observations are available.

Update Step: During the update step, particles receive weights based on their consistency with the available observations. Particles that align well with

the observations receive higher weights, while those that deviate receive lower weights. This weighting process utilizes the likelihood of the observations given each particle's state.

To ensure a diverse representation of the state space, particle filters employ a resampling step after updating the particle weights. This step emphasizes particles with higher weights, making them more likely to be selected and replicated, while particles with lower weights may be removed. This resampling process helps prevent particle degeneracy, a situation where only a few particles have significant weights.

Particle filters are **great in certain applications**:

- They can effectively handle systems that are nonlinear and non-Gaussian, offering flexibility in real-world applications.
- They provide not only a point estimate of the state but also a measure of uncertainty by estimating the entire posterior distribution.
- Their capability to track multiple hypotheses in a multi-modal distribution is particularly useful in complex, dynamic scenarios.

However, particle filters do have **limitations**:

- In high-dimensional state spaces, an extensive number of particles may be required for accurate estimation, which can impose significant computational demands.
- They may encounter challenges in scenarios with sudden and abrupt changes or when the posterior distribution becomes highly non-Gaussian.

Despite these limitations, particle filters are a powerful tool for state estimation and have been instrumental in numerous fields, particularly in robotics and tracking applications. Some **real world examples include**:

- Simultaneous Localization and Mapping (SLAM): Robots use particle filters to estimate their own position and create a map of their environment, combining sensor data and motion models for navigation (We are just tackling the localization part of SLAM in this case).
- Object Tracking: Particle filters are used in computer vision and tracking systems to follow and predict the movement of objects in video sequences, such as tracking vehicles on the road or monitoring people in surveillance footage.
- Aircraft State Estimation: In aviation, particle filters are employed to estimate the state of an aircraft by combining data from various sensors, particularly in cases of unreliable or conflicting sensor data.

3 Pseudocode Walkthrough

We thought a good way to gain a full conceptual understanding of the particle filter, was to think it out in pseudocode. An algorithm is designed to be a strict series of steps, so bringing it into code (where we already break everything down into small sub-tasks) gave us a more intuitive approach to scaffold it. In figure 1, we have the example we came up with. We'll explain in depth each step of the algorithm below:

```
while not converged:
    # Prediction: Update particle poses based on motion model
    for i in range(num_particles):
        particles[i] = motion_model(particles[i], odometry_data)

    # Update weights based on observation (Laser Scan)
    for i in range(num_particles):
        weights[i] = calculate_weight(particles[i], laser_scan_data, map)

    # Normalize weights
    total_weight = sum(weights)
    for i in range(num_particles):
        weights[i] /= total_weight

    # Resampling: Create a new set of particles based on weights
    new_particles = []
    for _ in range(num_particles):
        selected_particle = select_particle_with_replacement(particles, weights)
        perturbed_particle = perturb_particle(selected_particle)
        new_particles.append(perturbed_particle)
    particles = new_particles

    # Check for convergence (e.g., if the variance of particle poses is below a threshold)
    if is_converged(particles):
        converged = True

# Estimate robot pose as the weighted average of particles
estimated_pose = calculate_weighted_average(particles, weights)
```

Figure 1: Pseudocode Example

3.1 Particle Initialization

To begin the localization process, we initialize a set of particles. These particles represent potential positions and orientations of the robot within the known map. The number of particles (`num_particles`) is determined in advance based on computational resources and accuracy requirements.

```
for i in range(num_particles):
    particles[i] = generate_random_particle()
```

3.2 Iterative Localization

The localization process is iterative and continues until convergence, which typically involves assessing the consistency and accuracy of particle poses.

```
while not converged:
```

3.3 Prediction Step

In the prediction step, we update each particle's pose based on the robot's estimated motion. This is achieved by applying a motion model to each particle.

```
for i in range(num_particles):
    particles[i] = motion_model(particles[i], odometry_data)
```

3.4 Update Step

In the update step, we calculate the weight of each particle based on how well its pose aligns with the observed laser scan data and the known map.

```
for i in range(num_particles):
    weights[i] = calculate_weight(particles[i], laser_scan_data, map)
```

3.5 Weight Normalization

To ensure that the weights represent valid probabilities, we normalize them by dividing each weight by the sum of all weights.

```
total_weight = sum(weights)
for i in range(num_particles):
    weights[i] /= total_weight
```

3.6 Resampling Step

Resampling is a crucial step in the Particle Filter algorithm. It involves selecting particles with replacement based on their weights. Particles with higher weights are more likely to be selected, leading to a more accurate representation of the robot's pose. After each point is chosen, an amount of perturbation dependant on the spread of the particles is applied.

```
new_particles = []
for _ in range(num_particles):
    selected_particle = select_particle_with_replacement(particles, weights)
    perturbed_particle = perturb_particle(selected_particle)
    new_particles.append(perturbed_particle)
particles = new_particles
```

3.7 Convergence Check

The convergence check determines whether the algorithm has reached a satisfactory estimate of the robot's pose. We decided on checking whether the variance of particle poses falls below a predefined threshold.

```
if is_converged(particles):
    converged = True
```

3.8 Pose Estimation

Once the Particle Filter has converged, the estimated robot pose is calculated as the weighted average of the particles. This weighted average provides a robust estimate of the robot's pose within the known map.

```
estimated_pose = calculate_weighted_average(particles, weights)
```

3.9 Termination

At the end of the algorithm, the robot's pose is represented by `estimated_pose`, which is the result of the Particle Filter localization process.

This Particle Filter algorithm allows a robot to estimate its pose within a known map by integrating motion information from odometry and sensor data from laser scans.

4 Our Implementation

Because we already turned this problem into code, it was trivial to move our abstract solution into a specific Python implementation using ROS. Before actually coding however, we looked through the starter code to make sure we didn't miss any useful helper functions that might already be provided. This gave us a good idea of the structure code overall, and how our algorithm would work in tandem with the rest of the ROS system. Once we started implementing, we wanted to focus on making a robust design, so we added a couple additional features that allows our particle filter to converge more easily and error correct. We'll detail the key parts of our implementation below, and how they differ from the algorithm explanation above:

4.1 Custom Classes and Objects

4.1.1 Particle(Object) Class

The `Particle` class serves as a fundamental element for representing hypotheses or particles related to the robot's pose. Each particle encapsulates key attributes:

- **x**: The x-coordinate of the hypothesis relative to the map frame.

- **y**: The y-coordinate of the hypothesis relative to the map frame.
- **theta**: The yaw angle (theta) of the hypothesis relative to the map frame.
- **w**: The particle weight, which quantifies the particle's likelihood given observed data. Note that this class does not enforce the normalization of particle weights, a step typically addressed in the particle filter's update process.

The constructor allows for the creation of new `Particle` instances with user-specified values for the x, y, theta, and weight attributes, with default values of (0.0, 0.0, 0.0, 1.0) if not provided. The `as_pose` method is a useful utility for converting a `Particle` into a `geometry_msgs/Pose` message, often employed in ROS (Robot Operating System) environments for representing the robot's pose.

In a particle filter, instances of the `Particle` class collectively represent a spectrum of hypotheses about the robot's pose. These particles are pivotal for estimating and tracking the state of the robot as it navigates its environment and receives sensor observations. The associated weights for each particle play a central role in assessing the probability of each hypothesis given the observed data.

4.1.2 ParticleFilter(node) Class

The `ParticleFilter` class represents the central component of a particle filter-based robot localization system. It inherits from the ROS class `node`, so it can be spun up in a ROS environment. It is responsible for managing a cloud of particles (of the `Particle` class) that collectively estimate the robot's pose within an environment. Key attributes and functionalities of this class include:

- **Attributes**: The class has various attributes that define parameters and components used in the particle filter, including frame names, scan topics, particle count, and movement thresholds.
- **Odometry and Laser Data**: The class subscribes to odometry and laser scan data, which are fundamental for the update process.
- **Update with Odometry**: It updates particle positions using odometry data, accounting for the robot's movement.
- **Publishing Particles**: The class publishes the particle cloud, enabling visualization in tools like RViz.
- **Transforms**: The class manages the transformation of coordinates and ensures that they are correctly aligned.
- **Multi-Threading**: To address single-threaded execution limitations, the particle filter runs its primary loop in a separate thread.
- **Update Robot Pose**: It updates the robot's estimated pose based on the particles, allowing for both mean and most-likely pose estimation.

- **Integration with ROS**: The class is integrated with the Robot Operating System (ROS) framework, making use of ROS messages and topics for communication.

The `ParticleFilter` class coordinates the movement, weighting, and resampling of particles to estimate the robot's pose accurately in a dynamic environment. It exemplifies the core functionality of a particle filter.

4.2 Particle Initialization Step

Our particle initialization function takes a timestamp and an optional xy_theta argument representing the mean position and orientation. If xy_theta is not provided, it defaults to the robot's odometry pose. The function then generates a particle cloud with a specified number of particles (typically representing potential robot positions) around the provided or default pose. Each particle is perturbed with random noise to account for uncertainty in position and orientation. After initializing the particles, the function further normalizes them and updates the robot's estimated pose based on this initial distribution.

4.3 Prediction Step

Next, the `update_particles_with_odom` function is responsible for updating the particles using the odometry pose information. The primary goal of this step is to predict how the robot's pose (position and orientation) has changed since the last update based on odometry data, and apply this change to each of the particles. Here's how the code accomplishes this prediction:

- First, it converts the received odometry pose (`odom_pose`) into a set of coordinates (`new_odom_xy_theta`) that represent the robot's new position (x, y) and orientation (theta).
- It then calculates the change in position and orientation between the new odometry pose and the pose when the particles were last updated. This change is computed as a tuple (`delta`) with three components: the change in x-coordinate, the change in y-coordinate, and the change in the yaw angle (theta).
- If there was a previous odometry pose (`current_odom_xy_theta`), it subtracts the corresponding coordinates from the new odometry pose to determine the change. This step is essential for tracking the robot's movement over time.
- The code calculates the magnitude of the change (`d`) in position, which represents the linear distance traveled by the robot since the last update.
- Finally, for each particle in the particle cloud, it updates the particle's orientation by adding the change in yaw (theta) and updates the particle's position by moving it in the direction of its orientation by a distance of `d`. This simulates how the particles propagate based on the robot's motion.

These updated particles will later be refined using sensor measurements during the correction step of the particle filter, ultimately providing an improved estimate of the robot's pose.

4.4 Update Step

The `update_particles_with_laser` function's primary purpose is to update the weights of particles based on laser scan data, allowing the particle filter to refine its estimate of the robot's pose by aligning particles with the observed environment. Here's a breakdown of how the code accomplishes this correction:

- The function takes two inputs: `r`, which represents the distance readings to obstacles obtained from the laser scanner, and `theta`, which denotes the angle relative to the robot's frame for each corresponding reading.
- For each particle, we run through the readings, and the code calculates a new position (x, y) by adding the laser scan's range (`r[i]`) to the particle's current position and taking into account the particle's orientation (`theta[i]`). This step simulates where the laser beams would intersect obstacles from the particle's perspective.
- We're given a helper function that uses an occupancy field to calculate the distance from an (x,y) point to the nearest obstacle in the map. If the obstacle distance is neither a NaN (Not-a-Number) nor infinity (indicating valid data), the code keeps track of the error. After all laser scans have been evaluated, the code assigns the particle's weight as the reciprocal of the average error, effectively assigning higher weights to particles that are closer to obstacles and align well with the laser scan data.

In essence, the `update_particles_with_laser` function leverages laser scan data to update particle weights, assigning higher weights to particles that are consistent with the observed environment.

4.5 Weight Normalization Step

The `normalize_particles` function ensures that the sum of all particle weights equals 1.0, thereby representing a valid probability distribution over the particles. Here's a breakdown of how the code accomplishes weight normalization:

- For each particle, the code accumulates the weight of each particle into the `sum` variable. This effectively calculates the sum of all particle weights, reflecting the total likelihood of all particles.
- The code then steps through each particle again and performs the actual weight normalization for each particle. It checks whether the `sum` is not equal to zero, ensuring that the division is valid. If `sum` is not zero, it divides the particle's weight (`w`) by the `sum`. This step scales each particle's weight to ensure that the sum of all weights equals 1.0, creating a valid probability distribution.

- If, by any chance, the `sum` is equal to zero, indicating that all particle weights are zero (which can happen in certain situations), the code assigns a uniform weight to each particle, ensuring that they sum up to 1.0. This step prevents division by zero and maintains a valid probability distribution.

4.6 Pose Estimation Step

The `update_robot_pose` function is responsible for updating the estimate of the robot's pose based on the particle cloud. Here's how the code accomplishes this task:

- The code loops through all particles and determines the one with the highest weight. The code sets the robot's pose (`robot_pose`) to be the pose of that particle. This effectively represents the most likely pose of the robot, which is the mode of the particle distribution. The robot's pose is updated with respect to the map frame.
- Additionally, the code checks if it has received odometry data (`odom_pose`). If odometry data is available, it uses this data to update the transformation between the map frame and the odometry frame. This step ensures that the map-to-odometry transform is correctly set based on the robot's estimated pose.
- In case there is no odometry data received, the code logs a warning, indicating that it cannot set the map-to-odometry transform due to the absence of odometry information. This situation may occur in scenarios where odometry data is not available.

4.7 Resampling Step

The `resample_particles` function is responsible for resampling the particles based on their weights. This step ensures that particles with higher weights are more likely to be selected, improving the representation of the robot's pose in response to new observations.

Here's how the code accomplishes this task:

- The code employs a helper function (`draw_random_sample`) to resample the particles based on their weights. The function creates a new list (`new_particles`) of particles by randomly selecting particles from the current cloud according to their weights. Particles with higher weights have a greater chance of being selected, improving the representation of the particle cloud.
- Additionally, the code calculates the bounds for perturbing the resampled particles. It computes the variances of the particle cloud's x, y, and theta values, which represent the uncertainty in each dimension.

- For each particle, the code perturbs the position and orientation of each particle to introduce variability. The perturbations are drawn from a normal distribution centered around zero with variances derived from the particle cloud's uncertainties (`bound_x`, `bound_y`, and `bound_t`). These perturbations help prevent particle degeneracy and maintain diversity in the particle cloud.

4.8 Termination/Convergence Check

The previous steps repeat over and over again until the user exits the program. As it runs, the Neato's laser scan slowly starts to align with the map visualized in Rviz. An optional step would be to cut off the localization once it converges on a pose estimate that's accurate enough. We didn't include this feature in the final implementation, since it's more interesting seeing the particle filter continue to work, but we could add back the functionality very easily just by looking at the spread in the distribution of the particles. If the variance in the x, y, and theta of all the particles is very low, then we know the particles are all right next to each other and aligned in direction. We could add a check at the end of the loop to see whether the variances are less than a specific tolerance, and then terminate the program if it's converged. Without this check, the program continues in a loop, always updating its estimated pose. After running for a while, the pose estimate maintains an almost exact location, shifting around a little bit as the user continues driving. There seems to be more uncertainty on the theta of the final pose, as the rotation of the alignment visualization is the most significant change (visually) between each estimation once the filter has converged.

5 Challenges

5.1 What We Struggled With Most

One significant challenge was the accidental processing of NaN (Not-a-Number) data when implementing the particle filter. The sensor data would occasionally output r values as NaN or even as infinity so we had to process our sensor data to prevent unintended outcomes. Another challenge was implementing the use of quaternion-based calculations to make our code robust. We eventually opted for more simplistic trigonometry to achieve the same results.

5.2 What We Would Do Different Next Time

A definitive next step would be to try implementing quaternion calculations. Even though our neato is primarily concerned with the 2 Dimensional obstacles, it is imperative to our learning and to making our code more robust to account for the real world which is in 3 Dimensions. ROS Actions were an interest that we were not able try out. It would be interesting to try and expand our current method of subscribing to topics and receiving messages with ROS actions.