Particle Filter

Rihab Laroussi, and Davina Sanghera

November 20, 2024

Abstract

Localizing a robot accurately is a challenging problem due to uncertainties in motion and sensor data. One effective method for addressing this problem is the particle filter, a type of Monte Carlo (MC) method. Monte Carlo methods use a finite number of randomly sampled points to compute results, generating enough points to get a representative sample of the problem. These points are then processed through the system being modeled, and results are computed based on the transformed points. This is the essence of the particle filter. This Bayesian filter algorithm is applied to thousands of particles, with each particle representing a possible state for the system. Each particle is then weighted based on its likelihood or representing the true state of the system. In this lab, we explore the implementation of a particle filter for a differential drive mobile robot, focusing on both the prediction and update phases.

1 Introduction

The purpose of this lab was to implement and test a particle filter for a differential drive mobile robot. The particle filter is a probabilistic approach that helps in estimating the robot's pose and in this lab, we implement it using sensor data and process models. The lab was divided into two main parts: the prediction and the update phases of the particle filter. In the following sections, we describe the steps taken during the lab, the results obtained, and any challenges faced along the way.

1.1 Report Organization:

This report is organized into several sections:

- Section 2: Details the prediction phase of the particle filter, the update phase, and the resampling methods.
- Section 3: Describes the simulation results and observations.
- Section 4: Discusses the issues encountered during the lab and how they were resolved.
- Section 5: Provides a summary of the key findings.

2 Particle Filter

There are several steps to implement a particle filter localization algorithm. During each iteration, odometry data is used to update the movement of each particle. Subsequently, particles are assigned weights based on the similarity between actual range sensor readings and predicted measurements. These predicted measurements are derived from the updated particle states and a map of the environment. Finally, the particle set is resampled according to these weights. The following diagram illustrates the generic particle filter algorithm used.

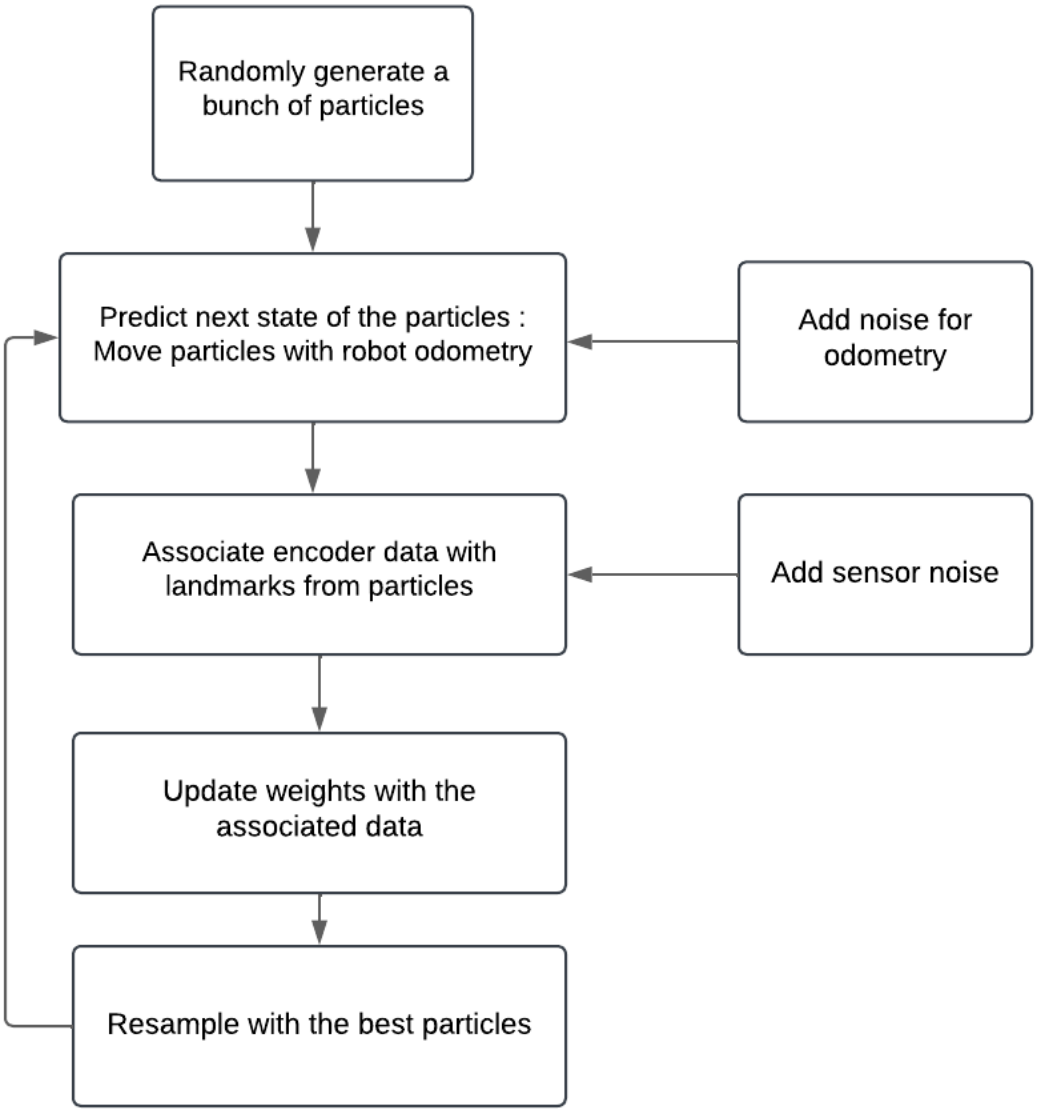


fig.1: Generic Particle Filter Algorithm

2.1 Prediction only

Initially, the particles are distributed around the initial pose x0 with uncertainty P0. In the prediction step of the Bayes algorithm, the sample process model updates our belief about the system's state. With the particles, each one represents a potential position for the robot. If we instruct the robot to move forward and turn by some degree, then we'd apply these movements to each particle. However, since the robot's controls are imperfect and won't move precisely as instructed, it's crucial to add noise to each particle's movement. This step is necessary to accurately reflect the uncertainty in the

system. Without accounting for this uncertainty, the particle filter can't properly represent the probability distribution of the robot's position.

Implementation:

First, when the robot is moving around, we need to track its movement. For that we read the sensors from ReadEncoders() to see how much each wheel has moved and then form a control input vector u with these displacements. We also need to know how certain these measurements are, which is given by the process noise covariance Qk.

```
def GetInput(self):
    """
    Get the input for the motion model.

    :return: * **uk, Qk**. uk: input vector (:math:`u_k={}^B[\Delta L~\Delta R]^T`), Qk: covariance of the input noise

    **To be completed by the student**.
    """

    zsk, rsk = self.robot.ReadEncoders() # get wheel encoder readings
    uk = np.array([zsk[0, 0], zsk[1, 0]]) # store as motion model input
    Qk = np.diag(np.array([0.2 ** 2, 0.02 ** 2, np.deg2rad(2) ** 2])) # noise of the input

    return uk, Qk
```

Next, we need to take these inputs and apply them to each particle to predict where the robot might be. This is where we use the SampleProcessModel() method, to sample the motion model to propagate the particles.

```
def SampleProcessModel(self, particle, u, Q):
    """
    Apply the process model to a single particle.

    **To be completed by the student**.
    """
    xk_1 = particle
    noise = Q

    R = self.robot.wheelRadius # radius of robot's wheels
    pulses = self.robot.pulse_x_wheelTurns # number of pulses for a full wheel rotation
    wheel_base = self.robot.wheelBase # distance between the two wheels

    nl, nr = (u[0]), (u[1]) # uk = [nl, nr] + noise sampled from Q
    dl = nl * (2*pi*R) / pulses # left wheel displacement
    dr = nr * (2*pi*R) / pulses # right wheel displacement
    dk = (dl + dr)/2 + noise[0][0] # distance travelled by robot aka forward distance
    yaw = ((dr - dl) / wheel_base) + noise[2][0] # angle that the robot has turned

    uk = Pose3D(np.array([[dk], [0 + noise[1][0]], [yaw]])) # change in position (dk along x axis, 0 along y axis and yaw angle change)
    xk = xk_1.oplus(uk) # updated position

    return xk
```

And then we update the state of each particle based on the robot's movement using the Prediction() method.

```python
def Prediction(self, u, Q):
    """
    Predict the next state of the system based on a given motion model.

    This function updates the state of each particle by predicting its next state using a motion model.

    :param u: input vector
    :param Q: the covariance matrix associated with the input vector
    :return: None

    """
    updated_particles = self.particles.copy()
    for i, particle in enumerate(updated_particles):
        noise = np.random.multivariate_normal([0,0,0], Q).reshape(3,1) # noise used in the motion model
        x_bar = self.SampleProcessModel(particle, u, noise) # predicted next state of the particle
        updated_particles[i] = x_bar

    self.particles = updated_particles # update the particle set
```

2.2 Update

Going back to the Discrete Bayes chapter, we updated the probability distribution by the likelihood of the new measurement. Similarly, in this particle filter, each particle's weight is adjusted based on how likely it is that the particle represents the true state given the new measurements. This process ensures that particles that better match the measurements receive higher weights.

def update (likelihood, prior):

  posterior = prior * likelihood
  return normalize(posterior)

In our case, each of our particles has a position and an associated weight that reflects how well it aligns with the measurements. By normalizing these weights to sum up to one, we transform them into a probability distribution. Particles that are closer to the robot generally receive higher weights than those that are farther away.

Implementation:

Initially, we collect the current sensor measurements that provide information about the features.

```
def GetMeasurements(self):
    """
    Get the measurements for the observation model.

    :return: * **zf, Rf**. zf: list of measurements, Rf: list of covariance matrices of the measurements

    **To be completed by the student**.
    """

    zf, Rf = self.robot.ReadRanges() # get readings (zf) and measurement covariance (Rf)

    return zf, Rf
```

This is done by the ReadRanges() method, which simulates the process of reading distances to various features in the map. At the predefined frequency, it first iterates over all known features, and for each feature, it calculates the distance from the robot's current position to the feature with an added noise. The method returns readings, each containing the feature id and the noisy distance to that feature, plus measurement-cov which represents the covoriance matrix of the noise.

```
def ReadRanges(self):

    readings = []
    measurement_cov = []
    std_reading_noise = 0.02
    measurement_cov = np.array([[std_reading_noise**2]])

    # Take the readings at the predefined frequency
    if self.k % self.xy_feature_reading_frequency == 0:
        # TODO: to be completed by the student
        for ftr_id, feature_pos in enumerate(self.M):
            ftr_range = np.linalg.norm(self.xsk[:2] - feature_pos)  # range from current position to feature position
            noisy_range = ftr_range + np.random.normal(0, std_reading_noise) # add noise of the reading
            readings.append([ftr_id, noisy_range])

    else:
        return [], []

    return readings, measurement_cov
```

The next step is to compute the likelihood that each particle's state matches the sensor measurements using the ObservationalModel method. The distance is calculated using the Euclidean Distance between the particle's predicted position and the feature. It is then used to determine the likelihood of this distance given the measurement noise using the probability density function.

```python
def ObservationModel(self, particle, z, R):
    """
    Compute the measurement probability of a single particle with respect to a single measurement.

    **To be completed by the student**.
    """

    distance = np.linalg.norm(particle[0:2] - self.M[z[0]]) # euclidean distance between the particle and the landmark
    likelihood = pdf(mean = z[1], sigma = np.sqrt(R), x = distance) # likelihood of the measurement

    return likelihood
```

Lastly, we adjust the particles weights based on how well they match the sensor measurements, we normalize the weights, and then perform resampling to ensure that particles with higher weights are more likely to be selected. Here we check if the effective number of the particles (Neff) is less than half the number of particles, in this case resampling is triggered to assure diversity and focus on the most probable particles. This step is executed by the update() method.

```python
def Update(self, z, R):

    particle_weights = np.ones_like(self.particle_weights) # initialise particle weights to 1

    for l, measurement in enumerate(z):
        for i, particle in enumerate(self.particles):
            likelihood = self.ObservationModel(particle, measurement, R) # calculate the likelihood of the measurement
            particle_weights[i] *= likelihood # update particle's weight with the likelihood

    # Normalize weights
    particle_weights /= len(z) # normalise by the number of measurements
    particle_weights /= np.sum(particle_weights) # normalise the weights so they sum to 1
    self.particle_weights = particle_weights # update the particle weights

    # Resample if necessary
    n_eff = 1 / sum(self.particle_weights)**2
    if(n_eff < len(self.particle_weights)/2):
        self.Resample()
```

2.3 Resampling

In particle resampling, particles assigned very low weights are discarded because they don't significantly contribute to the probability distribution of the robot's state. The algorithm focuses on replicating particles with higher weights, increasing their prevalence in the new particle set. This ensures that most particles effectively represent the true probability distribution.

Implementation:

We implemented two resampling methods, Roulette Wheel Resampling and Stochastic Universal Sampling.

a. Roulette Wheel Resampling

This technique prioritizes particles with higher weights, ensuring they have a greater chance of being selected. First, we normalize the particle weights so that their sum equals the total weight W. Then, we generate a random threshold r within the range of 0 to W. We iterate through the particles, accumulating their weights until the cumulative weight exceeds r. The corresponding particle is then selected. This process is repeated M times (where M is the number of particles), each time resetting r and repeating the selection. As a result, particles with higher weights are likely duplicated, while those with lower weights are less likely to be chosen. After resampling, the selected particles represent a more accurate probability distribution of the robot's state. This step is done by the RouletteWheelResampling () method, which ensures that our particle filter maintains a diverse and accurate set of particles, improving the overall localization performance.

```python
def RouletteWheelResampling(self):
    ''' This method is the Roulette Wheel version of resampling'''

    """Resample particles using the Roulette Wheel Resampling method."""
    M = len(self.particles)  # number of particles
    W = sum(self.particle_weights)  # total weight
    r = np.random.uniform(0, W)  # random threshold
    c = 0
    resampled_particles = []

    # Resample M particles
    for x in range(M):
        i = 0
        while not (c < r):
            c = c + self.particle_weights[i] # cumulative weight
            i = i + 1

        resampled_particles.append(self.particles[i])
        r = np.random.uniform(0, W)

    self.particles = resampled_particles
    self.particle_weights = np.ones(len(self.particles)) / len(self.particles) # reset weights to 1
```

b. Stochastic Universal Resampling

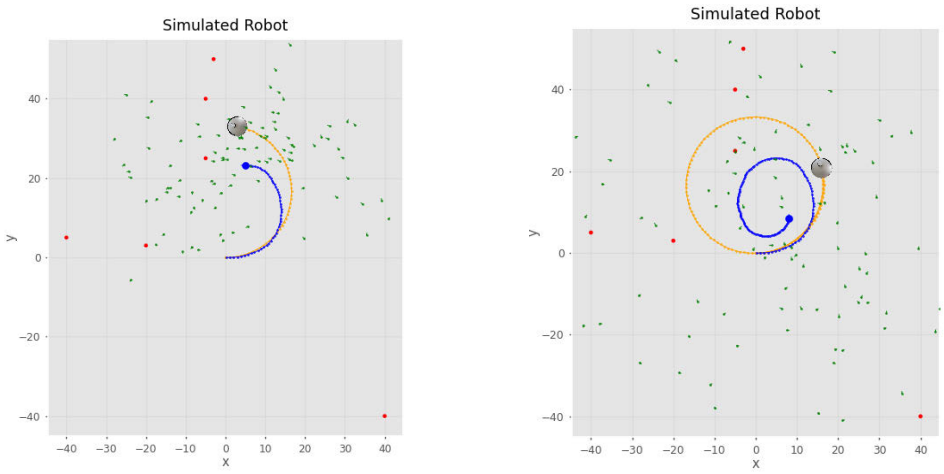
This method is slightly different from the Roulette Wheel Resampling. Instead of focusing on one fixed point, SUS uses multiple evenly spaced pointers to select particles. This ensures that each particle is evaluated more systematically, reducing the variance in selection. We start by determining the number of particles M and calculating the total weight W. A random starting point r is chosen between 0 and W/M. For each particle, a pointer u is calculated by adding r to m * W/M for each m from 0 to M-1. We go through the cumulative weights until u is exceeded, selecting the corresponding particle. This step is done by StochasticUniversalResampling() method , which ensures that particles with higher weights are more likely to be selected, thus providing a more accurate and diverse set of particles for the particle filter.

```
def StochasticUniversalResampling(self):
    ''' This method is the Stochastic Universal version of resampling'''

    M = len(self.particles)  # number of particles
    W = sum(self.particle_weights)  # total weight
    r = np.random.uniform(0, W/M) # random starting point
    c = self.particle_weights[0]
    i = 0
    resampled_particles = []

    for m in range(M):
        u = r + (m * W/M)
        while u > c:
            i = i + 1
            c = c + self.particle_weights[i] # cumulative weight
        resampled_particles.append(self.particles[i])

    self.particles = resampled_particles
    self.particle_weights = np.ones(len(self.particles)) / len(self.particles) # reset weights to 1
```

3. Simulation

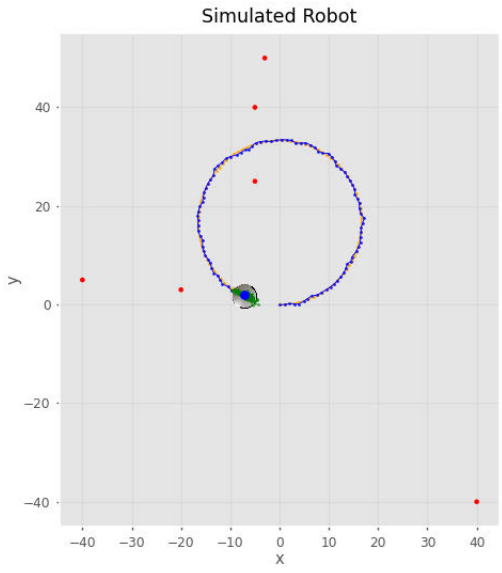
The simulation involves running the particle filter on a differential drive mobile robot model. The robot's movement was simulated using control inputs, and range measurements were used to update the particle weights. The evolution of the particles over time was visualized, showing how they converged towards the true position of the robot as more measurements were incorporated.

3.1 Result One:



During the prediction phase, the particles were observed to spread out from their initial positions as the robot moved. This demonstrated the effect of control inputs on the predicted states and allowed us to visualize how uncertainty in the robot's motion affected the distribution of particles.

3.2 Result Two:



The result shows the particle filter successfully localizing the robot. The particles converge over time toward the robot's true position, reflecting the integration of motion and measurement updates. This demonstrates the filter's effectiveness in handling uncertainty and accurately estimating the robot's state.

4. Issues

Due to the object-oriented structure of our code, one issue we encountered was debugging our code to identify which function was causing errors during the localization simulation. For example, we encountered an issue of the weights sometimes summing to zero which meant the get_mean_particle() function was halting the execution of the simulation in these instances. Another issue we encountered was with variables being of incompatible types when outputted from one function and then passed as input to other functions. To solve both of these issues, we learnt the importance of debugging each function one by one, in the sequential order of the program and printing out important information such as the type, size and content of the variables being used within each function. This allowed us to resolve the type issue as well as the weight summation issue as we managed to identify that we weren't normalizing the weights so that they summed to 1 in the Update() function.

5. Conclusions

In conclusion, from result two it is clear that the particle filter performs well when localizing the robot. This success is due to the particle filter's ability to handle non-linear motion, which is often present in robotic systems. Furthermore, the process of robot localization involves much uncertainty; this uncertainty is produced from multiple parts of the process such as the noise from motion or the noise in sensor measurements. By representing the robot's belief of its state as a set of weighted particles, the particle filter inherently captures this uncertainty. Therefore, this particular approach to robot localization has allowed us to produce satisfactory localization results that lie close to the ground truth of the robot's movement, even in the midst of noise being added from multiple sources. In addition, the repeated process of the filter where the particle set is updated through prediction, updates and resampling, ensures that errors are accounted for and therefore don't accumulate. This stands in contrast to the previous lab where we implemented dead reckoning localization and the accumulation of error led to a poorer localization performance, where the simulation was farther from the robot's ground truth.