Localisation in a Known Environment

Batuhan Ozgur BASAL

MSc Intelligent Systems and Robotics Faculty of Technology, De Montfort University, Leicester

Abstract

Localisation is among the most substantial capabilities demanded of a mobile robot since knowing where the robots is a requirement for generating selections regarding its behavior. For that purpose, this paper describes how the location of the robot named P3DX was determined in the 2D environment. In this study, Robot Operating System (ROS) [1] and Python programming language [2] were used via The Construct platform [3]. To predict a vehicle's position and orientation (pose), the Monte Carlo Localization (MCL) [4] technique is utilized. A predefined map of the surroundings, sonar sensor information, and odometry data are all used in the process.

Key words : The Construct, ROS, Python, Localisation, Monte Carlo Localisation, P3DX, 2D environment, MCL

Introduction

A localization approach is one of the most crucial criteria for an intelligent mobile robot. It is used to predict the mobile robot's position and orientation using the measurements and current information including a map of the area, its starting location, or identified landmarks. Since any interior mobile robot cannot execute its given activities autonomously without exact information of its location in the surroundings, localization is essential. This article concentrates on the robot localization situation with the help ROS and Pioneer P3DX mobile robot. ROS reduces the degree of expertise necessary to engage on robotics tasks. Numerous organizations may find it simpler to begin in programming or create sophisticated solutions faster as a result of this. As well as in this study, the advantages provided by ROS were used and the localization of the robot was provided to the users. The Pioneer P3DX model [5] mobile robots are the most widely used robots in teaching and development across the globe. They are the primary tool for sophisticated intelligent robots due to their adaptability, dependability, and longevity. They are also easily adaptable, and durable enough to withstand decades of usage in the workshop and field.

Localisation technique

The MCL method employs a particle filter to determine the robot's location in order to localize it. The particles illustrate the variety of possible robot configurations. Every particle indicates one of the robot's various states. As the robot travels through the 2D world and uses a sonar sensor to perceive various portions of it, the particles cluster all over a specific point. An odometry sensor detects the robot's movement.

```
Algorithm MCL(X_{t-1}, u_t, z_t):
    X̄_t = X_t = ∅
    for m = 1 to M:
        x_t^[m] = motion_update(u_t, x_{t-1}^[m])
        w_t^[m] = sensor_update(z_t, x_t^[m])
        X̄_t = X̄_t + ⟨x_t^[m], w_t^[m]⟩
    endfor
    for m = 1 to M:
        draw x_t^[m] from X̄_t with probability ∝ w_t^[m]
        X_t = X_t + x_t^[m]
    endfor
    return X_t
```

Figure 1- Pseudocode of MCL algorithm

The approach usually begins with a uniformly distributed random particles over the coordinate world, indicating that the robot seems to have no idea at which location it is and is similarly probable to be anywhere. When the robot travels, the particles displace in order to estimate the updated state following the motion. Once the robot detects information, the particles are resampled using Bayes filter [6] which determines how successful the true observed information matches the expected condition. The particles will eventually settle on the robot's true location.

The robot estimates its updated position depending on the odometry sensor by assigning generated momentum to every one of the particles throughout the motion update [7]. When a robot travels ahead, for instance, all of these particles travel ahead in their individual trajectories, regardless of which path the particles face. When a robot turns 45 degrees rotation, all particles rotate 45 degrees rotation regardless of where they are. But no odometry sensor is precise in the actual world: it might overshoot or undershoot the applicable level of displacement. As a result, the model should consider for noise. As a result, the particles will eventually divide throughout the update. This is to be predicted, as moving aimlessly without detecting the surroundings makes a robot's location less certain.

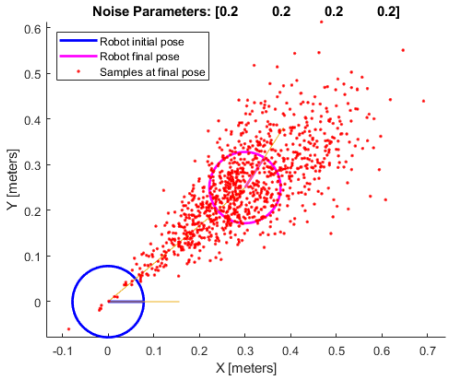

Figure 2- Odometry motion noise

Whenever the robot detects its surroundings, it updates their particles to better represent its location. The robot calculates the chance that it might detect what its detectors have really detected if it were in the condition of the particle. It gives every particle a weight equivalent to the likelihood. Afterwards, with likelihood corresponding to weight, it selects new particles arbitrary given the previous assumption. Particles that match sensor data are highly probable to be selected whereas particles that don't match

sensor data are barely selected. As a result, particles settle on a more accurate representation of the robot's state [8].

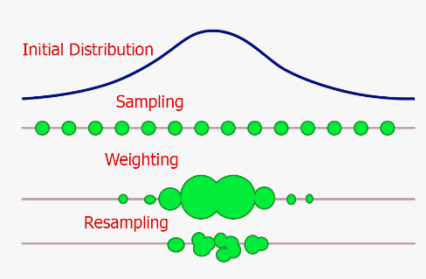

Figure 3 – Illustration on the resampling

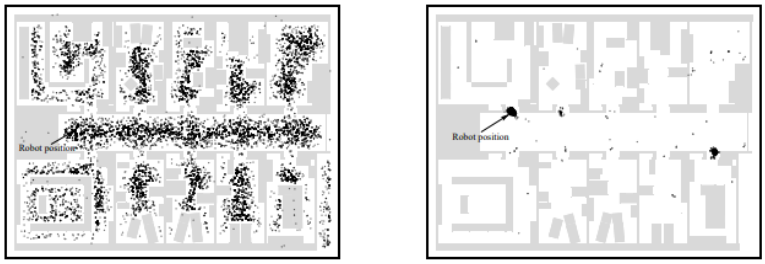

Figure 4 – Particle Filter implementation

Software Implementation

The software application consists of 3 functions. The first of these is the initialization of the particle cloud, the second is the updating of this particle cloud, and finally the estimation of the robot's position and orientation. To put it another way, the Monte Carlo Localization method is as follows:

- Initialize uniformly distributed group of particles.
- Engage with the surroundings to get information using the sonar sensor of the robot.
- Examine each particle in the sample and give a weight to each one depending on how closely it matches the information obtained. The "weight" is determined by our probability that a particle accurately represents the robot's position and orientation.
- Generate a new collection of particles by resampling from particles that have higher in terms of probability weight. Particles with greater weights colonize the group for the following process.
- Start again by replacing the previous group of particles using a new one.

Initialise_particle_cloud()

```
def initialise_particle_cloud(self, initialpose):

    pose_arrays = PoseArray()
    i = 0
    while i < 500:
        random_gauss_number = gauss(0, 1)
        rotational_dist = vonmisesvariate(0, 5)
        pose_objects = Pose()
        pose_objects.position.x = initialpose.pose.pose.position.x + (random_gauss_number * self.ODOM_TRANSLATION_NOISE)
        pose_objects.position.y = initialpose.pose.pose.position.y + (random_gauss_number * self.ODOM_DRIFT_NOISE)
        pose_objects.orientation = rotateQuaternion(initialpose.pose.pose.orientation,((rotational_dist - math.pi) * self.ODOM_ROTATION_NOISE))
        pose_arrays.poses.append(pose_objects)
        i += 1
    return pose_arrays
```

Figure 5 - Displaying the initialise_particle_cloud method

Particles come first among the basic factors in the MCL algorithm. Without them, robot localization cannot be successful. The function called *initialise_particle_cloud()* is required to create these particles.

The total number of particles to be used in this project has been determined as 500. The probability that the particles settling on the true destination rises as the number of particles rises. A smaller particle count, on the other hand, is quicker. Depending on the weights of particle groups, the number of particles changes constantly under limitations. This modification contributes to the reduction of particle numbers over duration, allowing for highly effective localisation. After determining the number of particles, first, a new object is created and synchronized to the *PoseArray()* message. This new object contains a list of the pose objects. Therefore, the function is required to return this list.

There are a few parameters that need to be added to the list before returning. these are odometry model rotation noise, Odometry model x axis (forward) noise and Odometry model y axis (forward) noise. Since the particles need to be uniformly distributed around the map, The corresponding noises are added to the instantaneous poses of the robot in the x and y axes. In addition to these, von Mises distribution is used instead of Gaussian distribution in rotation noise. Since this distribution fits the rotation noise more.

All operations are done under the while loop. The reason for this is that different noise values will be selected for each particle, these values are obtained through the *random.gauss* and *random.vonmisesvarite* functions for 500 times (for every particle in the cloud).

Update_particle_cloud

```
def update_particle_cloud(self, scan):

    prob_of_weight = []

    for pose_object in self.particlecloud.poses:
        prob_of_weight.append(self.sensor_model.get_weight(scan, pose_object))

    obtained_pose_arrays = self.obtained_value(prob_of_weight)

    for pose_object in obtained_pose_arrays.poses:
        pose_object = self.updated_noise(pose_object)

    self.particlecloud = obtained_pose_arrays
```

Figure 6- Displaying the update_particle_cloud method

After sending or dispersing the particles to the 2d map, the next step is to find the location of the mobile robot in the map using these particles. Some of these particles are so far away from the robot that they become redundant. In such a case, it is necessary to replace these particles with particles closer to the robot. This process is done with the help of the sonar sensor that p3dx has. This sonar sensor includes spectrum sensor-specific properties, 2-D map data for the robot's surroundings, and reading noise features. The sensor computes the probability of the readings provided the robot's present pose using the variables with observation data.

First, initializing an empty list named *prob_of_weight* to store the probability weighting of each Pose in *self.particlecloud* by using the *self.sensor_model.get_weight()* method. The function named *get_weight()* is in the sensor_model.py and returns probability using sonar sensor data. After adding the probabilities of each particle to the *probe_of_weight* list, next is the method to eliminate these particles that have low probability.

A new function is used for this based on Roulette-wheel selection method [9], and this function is called *obtained_value*. As in Roulette wheel selection, high-probability weights are selected more, while low-probability weights are selected much less or not at all.

```
def obtained_value(self, prob_array):

    pose_arrays = PoseArray()
    for each_pose in range(len(self.particlecloud.poses)):
        total_value = random.random() * sum(prob_array)
        total_weight_probability = 0
        indicator = 0
        while total_weight_probability < total_value:
            total_weight_probability += prob_array[indicator]
            indicator = indicator + 1
        pose_arrays.poses.append(copy.deepcopy(self.particlecloud.poses[indicator - 1]))
    return pose_arrays
```

Figure 7- Displaying the obtained_value method

Particles with high probability weight are stored in an object named obtained_pose_array. Next, to maintain the particle cloud distributed wide enough to ensure with any shifts in the robot's position, resampling noise must be applied to each new particle. With the for loop, noise is added to the particles using *updated_noise* method.

Estimate_pose()

```
def estimate_pose(self):
    predicted_pose = Pose()
    x_value = 0
    y_value = 0
    z_value = 0
    w_value = 0

    for each_object in self.particlecloud.poses:
        x_value += each_object.position.x
        y_value += each_object.position.y
        z_value += each_object.orientation.z
        w_value += each_object.orientation.w

    predicted_pose.position.x = x_value / 500
    predicted_pose.position.y = y_value / 500
    predicted_pose.orientation.z = z_value / 500
    predicted_pose.orientation.w = w_value / 500

    #rpy = tf.transformations.euler_from_quaternion(predicted_pose.position.x , predicted_pose.position.y, predicted_pose.orientati
    orientation_list = [predicted_pose.position.x, predicted_pose.position.y, predicted_pose.orientation.z, predicted_pose.orientat
    (roll, pitch, yaw) = euler_from_quaternion(orientation_list)

    print('Robots X position: ', predicted_pose.position.x)
    print('Robots Y position: ', predicted_pose.position.y)
    print('Robots Heading: ', math.degrees(yaw))
    return predicted_pose
```

Figure 8- Displaying the estimate_pose method

This function predicts the robot based on particle position and orientation. This process that determines the mean pose of all particles in the particle cloud.

First, x,y,z,w values are initialized by giving them 0's. These objects are aligned with the axis corresponding to each particle in the particle cloud. Then, the Pose() message is stored inside a new object named *predicted_pose*. Each of the objects obtained before with x, y, z, w values is divided by the total number of particles, 500, and equaled to this newly created object with its corresponding axis *(predicted_pose.position.x, predicted_pose.position.y, predicted_pose.position.z, predicted_pose.position.w*).

Next, to obtain the Heading (theta) of the Robot, It was necessary to get the yaw value. For this, the *euler_from_quaternion* method, one of the functions provided by ROS, was used and yaw roll pitch values were reached. Finally, X position, Y position, and heading of the robot is reflected to the output.

Testing and Results

Finally, after adding required topics from rviz and running the node.py code, the localisation of the mobile robot is obtained through rviz visualization tool in the online platform named The Construct.

There are very important factors in applying the MCL algorithm. These changes have many effects from the distribution of the particles on the 2D environment to the localization success of the P3DX mobile robot. Some of them are values of the parameters of the noises, total number of particles used to locate the robot, and the number of predicted readings from the sonar sensor.

First, it is very important to determine the total number of particles. This is because the particles themselves determine the localization of the robot. To successfully sample particles throughout all possible configurations of a system, localization necessitates a greater number of particles. The chance of efficient combination of the particles on the actual state increases as the number of particles increases. This wide distribution affects initial efficiency significantly before particles rejoining together and the number of particles may be lowered. This number was calculated by trial and error, and the one with the best result was selected, and this number was determined as 500.

Another variable was the Odometry noise values. Again, optimization was achieved by giving different numbers by trial and error. The range of the parameter named *self.UPDATED_NOISE* was found to be uniformly distributed between the values (100,120). If this value range is set to (1.5), it has been observed that the particles were concentrated at the selected point and did not show a distribution on the map. As a result of increasing the values of the odometry motion parameters, it has been observed that the particles in the map move on their own and no localization was found.

Finally, the value of the *self.NUMBER_PREDICTED_READINGS* parameter was changed, and the results were observed. Although not a great change was observed, it was observed that the increase in this value decreased the localization speed. Therefore, 50 was chosen as the optimal value.

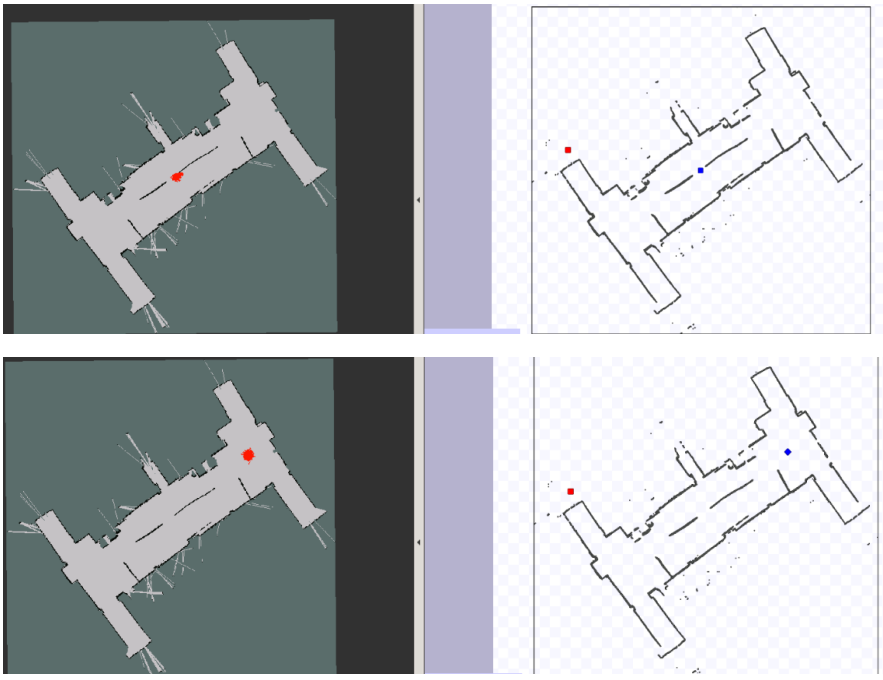

Figure 9- Testing results

Conclusion

Many various strategies have been developed and utilized in the software development process. Trial and error testing was used to fine-tune the settings. The experiment has been successfully completed and the localization of the robot has been achieved. The robot reports its position at least once a second to the output console. As observed as a result of the tests performed in the experiment, localization errors were observed depending on the position of the robot. These errors are often like many locations on the map. Spaces , structure of walls and path to these empty rooms. To prevent this, it is necessary to move the robot and show it to the sensor from different angles. Thus, the particle cloud can be updated and successfully pinpoint the robot's position.

References:

[1] Ros.org. 2022. *ROS: Home*. [online] Available at: <https://www.ros.org/> [Accessed 13 May 2022].

[2] Python.org. 2022. *Welcome to Python.org*. [online] Available at: <https://www.python.org/> [Accessed 13 May 2022].

[3] The Construct. 2022. *The Construct: A Platform to Learn ROS-based Advanced Robotics Online*. [online] Available at: <https://www.theconstructsim.com/> [Accessed 13 May 2022].

[4] Fox, D. and Burgard, W. and Dellaert, F (1999) Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In: Proceedings of the National Conference on Artificial Intelligence, July 1999, Orlando, Florida, USA. Berlin: ResearchGate, pp. 343-349

[5] Generationrobots.com. 2022. [online] Available at: <https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA.pdf> [Accessed 13 May 2022].

[6] A. J∅sang, "Generalising Bayes' theorem in subjective logic," *2016 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, 2016, pp. 462-469, doi: 10.1109/MFI.2016.7849531.

[7] M. Bilgin and T. Ensari, "Robot localization with Monte Carlo method," *2017 Electric Electronics, Computer Science, Biomedical Engineerings' Meeting (EBBT)*, 2017, pp. 1-7, doi: 10.1109/EBBT.2017.7956755.

[8] T. Li, S. Sun and J. Duan, "Monte Carlo localization for mobile robot using adaptive particle merging and splitting technique," *The 2010 IEEE International Conference on Information and Automation*, 2010, pp. 1913-1918, doi: 10.1109/ICINFA.2010.5512017.

[9] L. Zhang, H. Chang and R. Xu, "Equal-Width Partitioning Roulette Wheel Selection in Genetic Algorithm," *2012 Conference on Technologies and Applications of Artificial Intelligence*, 2012, pp. 62-67, doi: 10.1109/TAAI.2012.21.

APPENDIX

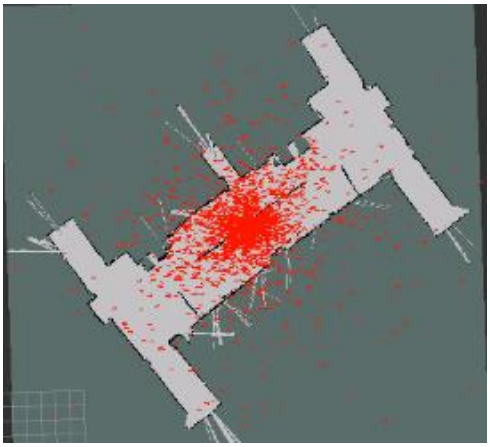

Figure 10 - Total particles set as 10000

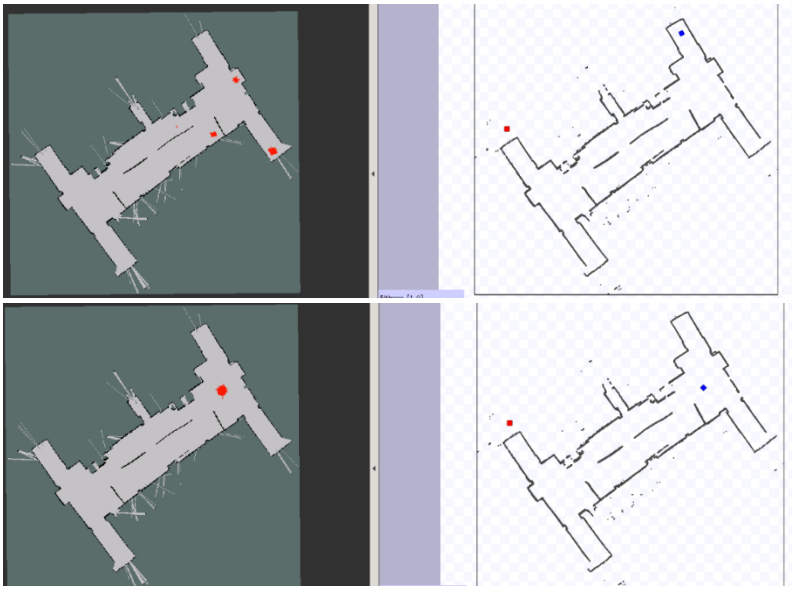

Figure 11- After moving the robot, particles correct themselves

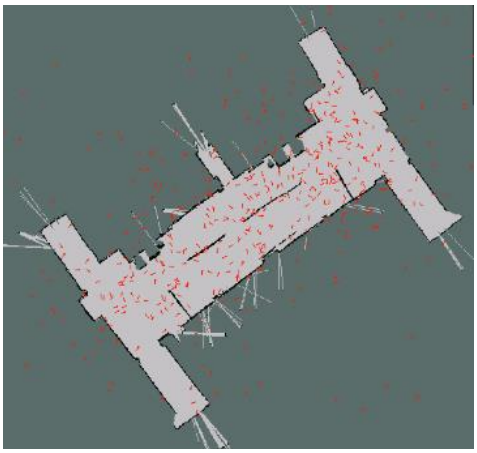

Figure 12 - When the noise parameters set higher than usual

Figure 13- When the noise parameters set 0, stops updating