Monte Carlo Localization in Python

Hrittik Roy, Nadezhda Kharitonova, Nicolas Grichn

We implement the montecarlo localization algorithm with custom world simulator on python. It uses particle filters to estimate posteriors over robot poses. We demonstrate this algorithm and discuss its shortcomings and observed failure cases.

I. Introduction

MONTE Carlo Localization algorithm is one of the most popular localization algorithms to date. Our objective in this paper is to describe how this algorithm works demonstrate its workings when its successful, document its failure cases and discuss its shortcomings.

Localization refers to the process of determining where a mobile robot is located with respect to its environment. In this we case we will be tackling the localization problem in a specific context. We assume that our environmnet is static and unchanging. We also assume that we are given a map of the environment. We will use the information from onboard sensors and the odometry information to localize our robot. Monte Carlo Localization is an improvement of the Markov Localization method. Instead of maintaining a belief for every point in the state space, in Monte Carlo Localization we only maintain belief for specific particles. This simple augmentation allows Monte Carlo localization to be successful in much larger configuration spaces and complex environments.

We simulate both the environment and the robot in python. The environment is a gridworld with random obstacles between edges of different cells that prevent movement between them. The robot is meant to simulate a diffrential drive robot like thymio or Turtlebot Burger etc. The sensor in this robot measures the distance to closest obstacles in four different directions (North, East, South, West). **This sensor was implemented after the discussion and comments in presentation.** We previously had a simpler sensor which worked in the following way. It didn't take into account the robots orientation only its location in the map. From that location it looked at the closest obstacle to the north, east, south and west. And this was the sensor reading for the robot. Whereas now the robot's sensor reading are exactly similar to ones we find in Mighty Thymio. It looks for obstacles in the axis of one of its sensors and reports the distance to it. We will describe implementation details below.

II. Monte Carlo Localization Algorithm

In Monte Carlo localization we represent our current belief about our state by particles. The table below shows the basic MCL algorithm which is obtained by substitution appropriate probabilistic motion models and perceptual models. The basic MCL algorithm represents belief $bel(x_t)$ by as set of M particles $X_t = \{x_t^{[1]}, x_t^{[2]}, ..., x_t^{[M]}\}$. We illustrate this using the example of a one-dimensional hallway.

A. Properties

1) Theoretically MCL can converge to almost any distribution of practical importance.

Algorithm 1 Monte Carlo Localization

1: Algorithm MCL (X_{t-1}, u_t, z_t, m) :
2: $\bar{X}_t = X_t = \Phi$
3: **while** $1 \leq i \leq M$ **do**
4: $\quad x_t^m = sample\ motion\ model(u_t, x_{t-1}^m)$
5: $\quad w_t^m = measurement\ model(u_t, x_t^m, m)$
6: $\quad \bar{X}_t = X_t + < x_t^m, w_t^m >$
7: $\quad w_{avg} = w_{avg} + \frac{1}{M} w_t^m$
8: **end while**
9: **while** $1 \leq i \leq M$ **do**
10: $\quad$ draw i with probability proportional to w_t^i
11: $\quad$ add x_t^i to X_t
12: **end while**
13: return X_t

Fig. 1. Monte Carlo localization in one dimension

2) The accuracy of the approximation can be determined and controlled by the number of particles used for localization. Increasing the number fo particles increases the accuracy of the approximation.
3) MCL is non-parametric and hence it can represent complex multi-modal probability distributions as well as simple gaussian distributions.

B. Limitations

MCL, cannot recover from robot kidnapping, or global localization failures. We demonstrate this in one of the experiments below. This is its main limitation that we discovered. As the position is acquired, particles at places other than the most likely pose gradually disappear. At some point, particles only "survive" near a single pose, and the algorithm is unable

to recover if this pose happens to be incorrect. This can occur either by accident in complex environments which have high degree of local similarity as we observe in many of our experiments or it can be induced artificially by kidnapping and respawning the robot in a random location as the particles start to cluster at a few specific locations.

In practice we observe that a little hyper-parameter tuning can be very effective in alleviating this problem.

III. Methodology and Implementation Details

A. Environment

The environment is a 2D numpy array. Each cell is coded as a 4-bit number, with a bit value taking 0 if there is a wall and 1 if there is no wall. The 1s register corresponds with a square's top edge, 2s register the right edge, 4s register the bottom edge and the 8s register the left edge. This part of the code was inspired by the Beacon Based Particle Filter project[4]. However the robot simulation, the sensor mechanism and the core localization algorithm is completely different. Each element of the numpy array is a cell and each has a dimension. The robot is free to move inside a cell with no obstacles.

B. Robot Simulation

We simulate a differential drive robot as a tuple of three numbers that gives us its exact pose. It belongs to the same class as particles. It has the ability to move within cells and across cells as long there isn't a well between them. It is also able to make sensor readings from the environment. Sensor reading mechanism is described below.

C. Sensor Mechanism

We implemented two versions of the sensor mechanism in our project. One was before the presentation and a second version based on the feedback from the presentation.

The current version of the robot has proximity sensors attached to it at angles 0, 90, 180 and 270. It looks along each of the lines at this angle and finds the near obstacle in this line. The distance to this obstacle is the reading of the sensor. Basically, it looks at the north, east, south and west directions from the reference frame of the robot and finds the nearest obstacles in these directions. It works exactly like the proximity sensors in mighty thymio. The only difference is that there is no limit up-to which the sensor can detect obstacles whereas in a Mighty Thymio proximity sensors can detect obstacles only up-to 2 meters and the configuration fo sensors is different. Although these features would not be difficult to implement.

The first version of the sensor didn't take into account the orientation fo the robot only its location. From its location the sensor would look at the global north, east, south and west directions and find the nearest obstacles in these directions. The difference between this version and the current version is that the current version works from the reference frame of the robot while this version works from the reference frame of the world. Consequently it was much easier to implement but it had a much harder time learning the exact pose, specially, the orientation of the robot.

D. Monte Carlo Localization

Now we will specify the final ingredients for the monte carlo localization algorithm: Motion algorithm for the robot, Computing the weight of the particles and resampling methods for the particles. After that we will summarise the whole algorithm briefly.

The robot moves in the following way: It heads in straight line without turning. It follows this path until it hits an obstacle. At that point it randomly chooses a new direction and moves in a straight line until it hits another obstacle and it repeats the same process again.

The weight of a particle should measure how likely it is to be at a location close to the location of the robot. We compute the weight of a particle by computing the value of the gaussian kernel function with inputs as the sensor readings of the particle and robot. So if their sensor readings are very close the kernel function assumes the maximum value and hence the particle has a higher weight. We also tested Laplacian and linear kernel but they either didn't make any difference or gave worse results.

For resampling particles we sample from the existing particles proportional to their weights and then add gaussian noise to its location. Adding a sufficiently large noise is vital because otherwise the resampled particles coincide with each other and then the next step if the Monte Carlo Localization is like running the localization algorithm with a lot fewer particles. We demonstrate this in an experiment.

Finally, we briefly summarise the whole process. We simulate an environment and a robot as described above. We begin by randomly generating particles and randomly spawning the robot at some unknown location. The robot moves in the environment according to the algorithm described above. It collects sensor information. The particles also follow the same instructions for moving. At each step the weight of each particle is computes based on how similar their sensor readings are to that of the robots. based on these weights the particles are resampled and this process continues until all the particles cluster around the exact location of the robot.

E. Code

Please find the code for the program in the following link: https://drive.google.com/drive/folders/1OT3a1qyg-C0PQ8cv-kCoIzhHyqG3bsW-?usp=sharing

IV. Results

A. Successful Localization

We display some instances of successful localization below for environments fo different sizes:

We test our algorithm in environments of size 5, 15, 25, 35 and 45 for several seed values. Curiously we don't observe a straightforward relationship between convergence time and size of the environment. The complexity of the environment is the biggest factor in how long or if at all we converge to the robot. Environments that look quite similar in many local regions are harder to localize in. However, increasing the number of particles makes even these environments tractable.

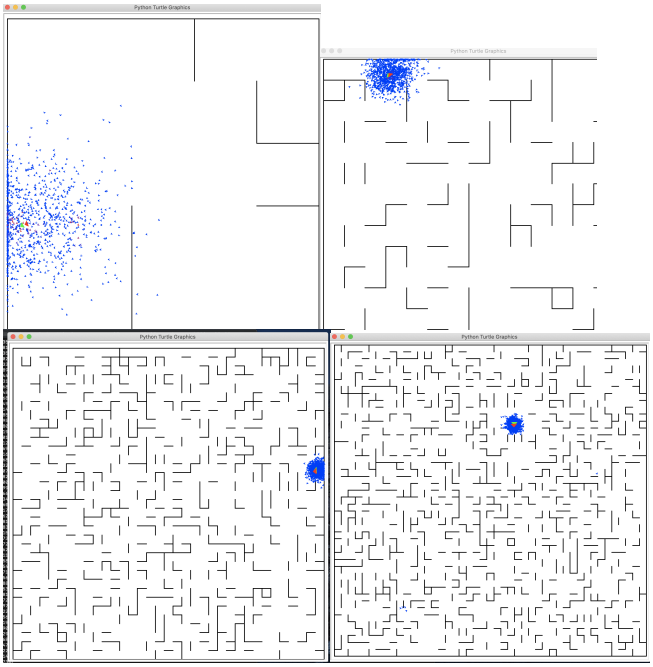

Fig. 2. Examples of Successful localization

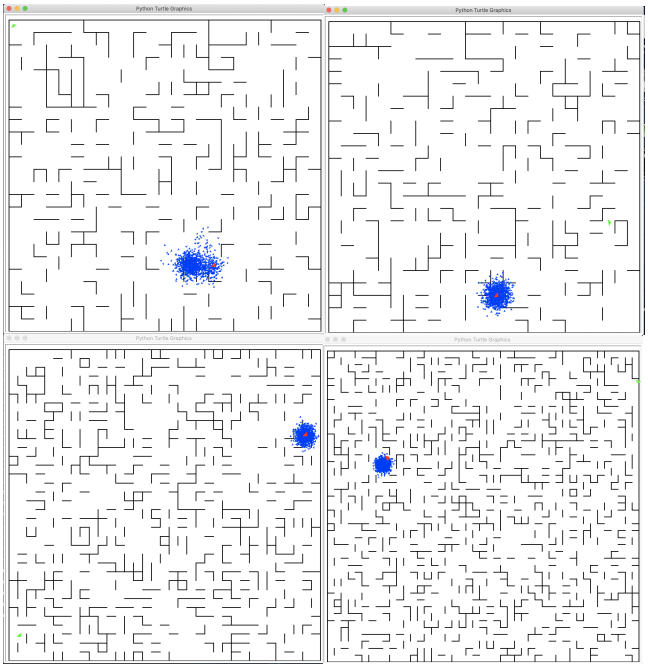

Fig. 4. Examples of Failed attempts at localization

But rendering these experiments were not computationally feasible. Below we plot the curve of steps till convergence against size of the environment. We take the average for all the seeds that we ran on. Perhaps we would see a more clear increasing trend if we had done the experiment on even more seeds.

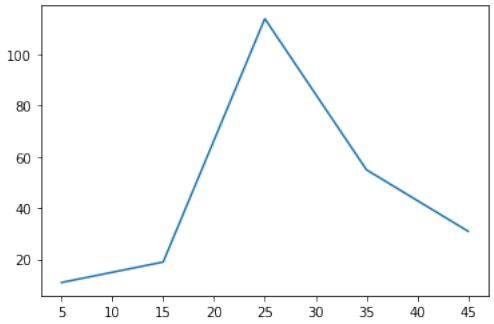

Fig. 3. Steps for convergence vs Size of the Environment

towards the particle. We know that without intervention these particles will converge to the robot from previous experiments. But in this experiment after 15 steps we pick up the robot from its current location and place it a random spot in the map. At this point the particles have already started clustering around a point. And the algorithm does not recover from this intervention.

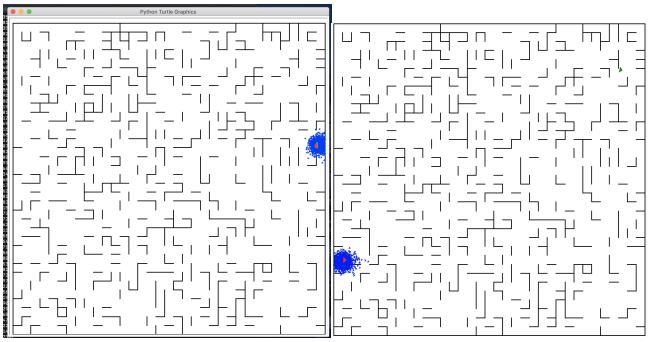

Fig. 5. Kidnapping a robot

B. Failure Cases

We display some instances of failure of the algorithm:

Behavior of the algorithm when it fails is the same. As we discussed above particles only “survive” near a single pose, and the algorithm is unable to recover if this pose happens to be incorrect. And the algorithm is not able to recover from this situation.

1) Kidnapping the Robot

The above situation occurred naturally because of the complexity of the environment. However, this situation can also be induced artificially. In the following experiment we allow the the particles to evolve and we observe as they slowly converge

2) Necessity of noise

In resampling the particles based on their weight it is necessary to add some noise otherwise we don’t have convergence. We have discussed this above. We demonstrate it empirically what happens when we add noise vs when we don’t. We see that particles coincide at a single point and it happens to be wrong and the algorithm is unable to recover from it.

C. Remarks

We would also like to make two additional remarks about the algorithm based on our observations. Firstly, we would like to talk about the termination condition. We say that the

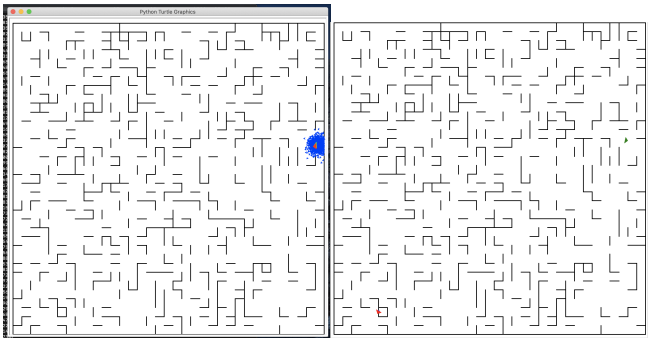

Fig. 6. Noise

algorithm converges when the distance between the actual pose and the estimated pose is smaller than some tolerance. however we should not refer to the actual pose since it is what we are trying to infer. An alternative method might be to terminate when most particles have a similar weight. however we didn't find this to be effective. So far it is not clear to us what would be a good termination condition without referring to the actual pose or visual intuition. Secondly, we would also like to note that this algorithm seems very sensitive to hyper-parameter tuning. For, example for one of our experiments we didn't get convergence after over 100 steps but as soon as we increase the standard deviation of the noise we found that the algorithm converged in just 19 steps. There are also no optimal hyper-parameter since they depend on the value of other hyper-parameters. We didn't get time to fully explore this aspect of the algorithm. We also have the option of adjusting the magnitude of sensor and odometry noise. We would have liked to see the effects on convergence of tuning this noise. Unfortunately we didn't have time for that analysis either.

V. Conclusions

In this project we present the Monte Carlo Localization Algorithm. We demonstrate a way of simulating the algorithm in Python. We demonstrate a few situations where it works. We also discuss some of its limitations and demonstrate them empirically. We also present a few interesting observations for which we don't have a clear explanations.

Bibliography

[1] Lecture Slides from the Course

[2] Probabilistic Robotics. Sebastian Thrun, Wolfram burgard, Dieter Fox.

[3] Elements of Robotics. Moti Ben-Ari, Francesco Mondada.

[4] Beacon Based Particle Filter project: https://github.com/mjl/particle_filter_demo