# Implementing the Particle Filter

Adela Wee and Michelle Sit
Computational Robotics Fall 2014
Oct. 16, 2014

## Introduction

The particle filter is sort of a mix of various approaches and concepts relevant to probabilistic robotics, which is a subset of reactive control paradigms. Probablistic robotics combines an approximate model with noisy sensors  through mathematical constructs such as Bayes rule.  The problem we're tackling here is localization of the robot in a known map that has a wide variety of possibilities (large areas) or a map with similar features (therefore high in global uncertainty).  is an example of successful applications of the filter to a world filled with uncertainty.

From a technical standpoint, the particle filter uses the Bayes' theorem and approximates its posteriors by taking a random distribution of a finite number of samples and updates based on the last known position. Basically the Bayes rule is used to estimate the state of a dynamic system by looking at sensor measurements.  Bayes filters assume that past and future data are conditionally independent if you know the current state-- assuming you have a Markov environment.  The particle filter is a type of algorithm based off of the Markov chain Monte Carlo (MCMC) methods, which was developed by the team of physicists working on mathematical physics and the atomic bomb during World War II. The MCMC methods attempt to solve a problem by taking in a bunch of random numbers and mapping those to some set of outputs in order to generate a solution.  Although the method has been around since the late 1940s, it was not widely used until the 1990s-- in fact, the term "particle filter" dates back to 1997.  The application of the algorithm to robotics is just about as old as the students in the class!

In our specific representation, there are a few key elements.  The particle filter has three frames of reference: the base frame, the map frame, and the odometry frame. The base frame is the frame relative to the robot, the map frame is the global view of the world that includes the robot, and the odometry frame is the frame relative to the wheels or odometry sensors (encoders) that is how the robot thinks where it is. Our particles consist of tuples (x,y, theta, weight), where x,y, and theta are coordinates of our hypotheses relative to the map frame.  We first initialize the cloud of particles, see if we've passed our distance threshold, then update our probabilities based on odometry data, then laser scan data, then re-sample the particles and update the robot's position from there.  The resampling step picks particles with greater likelihoods more frequently than the other 300 particles, which results in a fewer number of visible particles as the robot gets driven around the map.

# Our Steps to Making it Work

## Creating the Particle Cloud

We created an array that would store our particle cloud and populated it with particles. To introduce some randomness in the particles, each particle's tuple were scaled with a random value as they were created.

## Getting the Particles to Update Based on Odometry

There are three steps to updating the particles based on odometry which requires a rotation to the new center of the robot, a translation to get to that new coordinate, and another rotation to line up with the robot's heading. As seen in Figure 1, the robot rotates first by the angle (theta_k- theta), where theta_k is the arctan(delta_y/delta_x) to face the new position. It then translates by distance formed by the hypotenuse (sqrtt((delta_x)^2 +(delta_y)^2), and then rotates again to the final heading of the robot. Since the particles each face different directions, they are individually updated with changes in the robot's position.

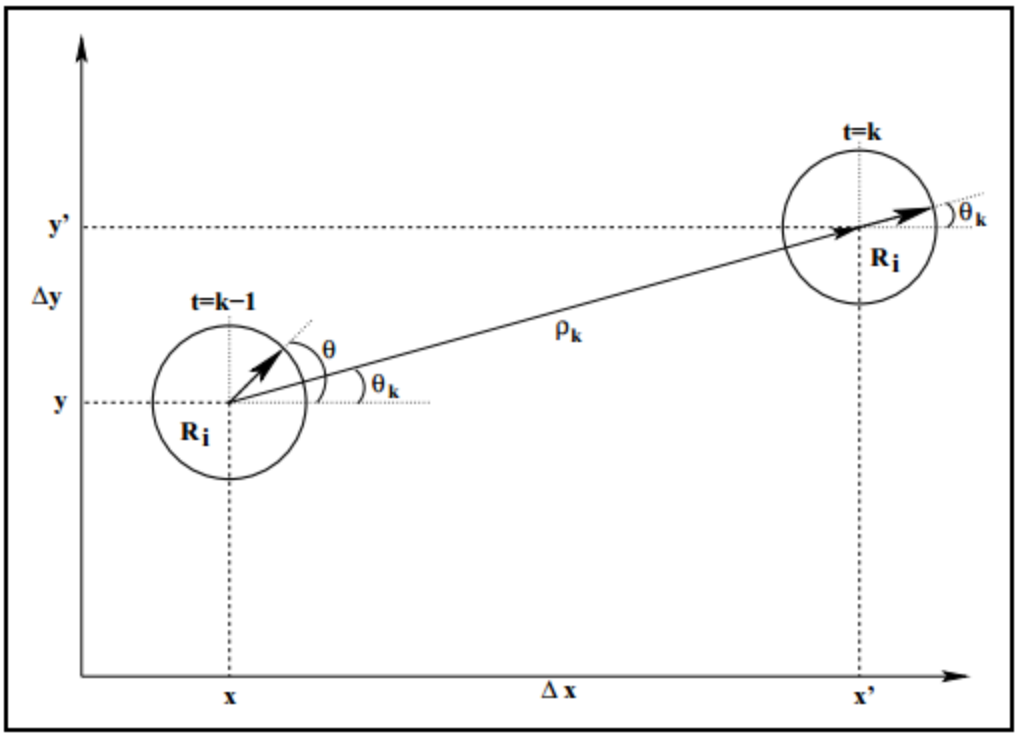


*Figure 1: Diagram of two robot positions: before the robot has moved and after. In the first position, the robot rotates, translates, and rotates again when it updates its position.*

## Getting the Particles to Update Based on Laser Scans

The robot takes several laser scans at specific angles to measure the distances of objects surrounding it. Then we calculate the distance between each particle and the closest objects to it at those angles.

Particles with distances similar in value to the robot's laser scan distances are assigned higher weights via a Gaussian distribution formula $\mathbf{e}^{-d*d/(2*sigma*sigma)}$ where sigma is an arbitrary interval in the gaussian distribution measured from the peak of the normal distribution and d is the distance between the particle and the closest object for a given spot in the occupancy field. In our first pass of the algorithm, we just calculated the likelihood of a specific particle based on the lidar measurement and the closest object at zero degrees. We forgot that we had to calculate the average probability for a particle at different angles (which became very apparent by the time we got down to running the particle filter on the neato).

### Update Robot Pose

Since each particle represents a hypothesis of where the robot is on the map, the robot takes the particle with the highest weight and assigns its position to that particle's coordinates.

### Refining the Particle Filter

We added to the resampling algorithm such that the particles with higher probabilities were picked more often. When the filter runs for longer periods of time, the number of visible particles decreases as the likelihood of a few particles becomes greater than the majority. Figure 2 shows a particle cloud after the algorithm had been running for a minute or so.

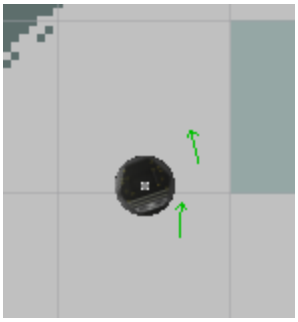

*Figure 2: Particle cloud after it has resampled several times-- it's hard to see, but in this image we actually have three particles around the robot, and since we updated the position of the robot in simulation such that the robot would jump to whatever particle had the highest probability of being where the robot actually was, the robot was actually on top of the third particle in this image. It was very challenging keeping track of where the robot was since it kept hopping from spot to spot!*

### Normalization of Particles

To normalize the particles, we summed all of the particle weights and then divided each particle by the total weight. By normalizing the particles, we create a standardized method for comparing the particles which later plays a significant role during the resampling process because particles are chosen based on their relative weights.

## Field Testing

### Recording Data and Building a Map with SLAM

We decided to use the Hector SLAM package to create a map that we could utilize to determine our location with our particle filter. However after collecting the map data, it became apparent that something

was wrong with our particle filter.  After checking in with the ninjas, it seemed that our particles weren't actually updating based off of the lidar data, which was pretty evident when our laser data didn't update or align with the map, as shown in figure 3.

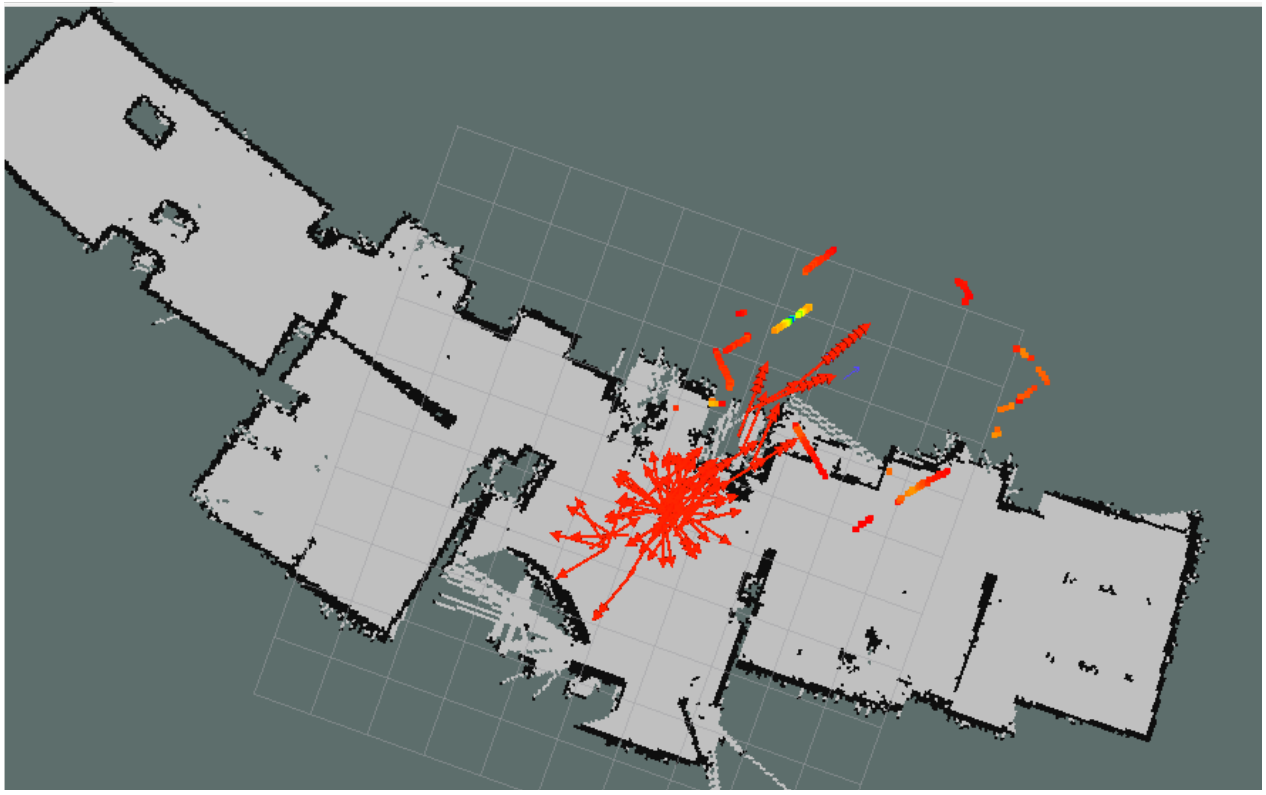

*Figure 3 shows the map we created with hector SLAM of the 3rd floor of the Campus center with our overlayed view of our particles-- the big cluster in the center (or origin) is where the particles were first initialized, and as we drove the robot around for about a minute, we would get particles that would track the robot fairly well (this is what you're seeing with the stacked arrows to the right of the initial cluster.  The multicolored data is actually our overlayed laser scan, although it was unclear as to why the laser data was not aligned with our map.*

## Reflection

Challenges Faced?

This project had us spinning our wheels and bashing our heads over and over. Both team members understood the problem-- we were pretty clear on which parts of the code we needed to implement and in what order.  The trouble was with our python knowledge-- neither one of us was familiar with object-oriented code in Python, and a lot of our problems stemmed from syntax issues.

Did you learn any interesting lessons for future robotic programming projects?  These could relate to working on robotics projects in teams, working on more open-ended (and longer term) problems, or any other relevant topic.

Ask questions early, and often.

Pair programming is really useful for catching bugs as they're introduced.

## Future Improvements

What would you do to improve your project if you had more time?
For future projects, it would be interesting to explore creating a Hector SLAM program that allows the robot to create its own map and understand where it is in relation to the map.  We feel limited by the fact that we must use a premade map for our particle filter.

## Sources

Particle Filters for Mobile Robot Localization, D. Fox, S. Thrun, W. Burgard, F. Dellaert, published in 2001.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.9914&rep=rep1&type=pdf

A Short History of Markov Chain Monte Carlo: Subjective Recollections from Incomplete Data, C.Robert and G. Casella. *Statistical Science*, 2011, Vol 0, No. 00, p 1-14. DOI: 10.1214/10-STS351
http://www.stat.ufl.edu/archived/casella/Papers/MCMCHistory.pdf

Particle Filters in Robotics, S. Thrun, Proc UAI 2002, pp511-518
http://arxiv.org/pdf/1301.0607.pdf

## Appendix A: Code

All code is located in the folder comprobo/src/my_pf and we only extensively modified pf_level1.py.

Commands to run in the simulator:

```
$ roslaunch neato_simulator neato_tb_playground.launch
$ roslaunch my_pf test_my_pf.launch map_file:=`rospack find neato_2dnav`/maps/playground.yaml
use_sim_time:=true
```

or if you want the smaller map that loads much faster:

```
$ roslaunch my_pf test_my_pf.launch map_file:=`rospack find neato_2dnav`/maps/playground_smaller.yaml
use_sim_time:=true
```

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Running on the Neato

```
$roslaunch neato_node bringup.launch host:=IP_ADDRESS_OF_THE_PI
$rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

reduced speeds to 0.054709495 and turn speed to 0.43046721 (init speeds were at 0.5 and 1)

Run laser data: $rosrun rviz rviz

Hector mapping: $roslaunch neato_2dnav hector_mapping_neato.launch
Saving the map: mapserver savemap (or something similar)

Run it: roslaunch my_pf test_my_pf.launch map_file:=`rospack find my_pf`/cc_map.yaml

On shutdown: $ ~/comprobo2014/shutdown_pi.sh