Lab 5 Report: Localization on the Racecar

Team 22

Benjamin Rich
Dinuri Rupasinghe
Sophia Wang
Steven Liu
Jose Soto
Editor for Lab 5: Benjamin Rich

16.405/6.4200

April 14, 2023

Contents

1 Introduction (Author: Sophia Wang)

The purpose of Lab 5: Monte Carlo Localization (MCL) is to determine the robot's orientation and position in a known environment, a critical problem in the field of robotics. Localization is a fundamental functionality required by an autonomous robot since the knowledge of its own location in time is an essential prerequisite to making decisions about future actions, such as path planning. Additionally, the ability to avoid dangerous situations such as collisions and unsafe conditions can be related to specific places in the robot environment. Localization is important both for path planning and obstacle avoidance.

We are specified to utilize ROS software given by the course staff of Robotics, Science, and Systems. We will be utilizing the LiDAR sensor data we receive from the racecar, odometry data from the wheels and steering, the racecar router for transferring this data, and built-in ROS packages to implement the Python scripts we will be creating.

The goal of this lab is to successfully implement MCL on the racecar to obtain odometry estimates and potential particle locations. MCL is an algorithm for robots to establish position and orientation in a known environment using a particle filter. The algorithm uses a particle filter to represent the distribution of likely states, with each particle representing a possible state. Over time, the estimates of particles' positions converge to the robots true location.

We tackle the technical problem by utilizing these main steps: implementing a motion model, implementing a sensor model, and implementing a particle filter. The particle filter uses both the motion model and sensor model to listen to laser and odometry data and publish new odometry data based on adapted particle probabilities and resampling. Throughout each step, we use unit tests where a racecar-simulator with a small testing map is generated, and the robot is placed in a predetermined location. Additionally, when integrating and adjusting the algorithm to hardware, we publish our estimated pose to the simulator for easy comparison with a ground truth map.

A challenge in the implementation of MCL is that ROS callbacks are not necessarily threadproof. Additionally, for real-time performance in the racecar, we must publish transformations from the map to the car's position at a rate greater than 20 Hz since at race speeds (10 meters per second), the poses we publish could be as much as half a meter apart, which could interfere with controller performance.

2 Technical Approach (Author: Dinuri Rupasinghe)

2.1 Overview

The process of implementing Monte Carlo Localization on our racecar includes three main parts: the motion model, the sensor model, and the particle filter. The particle filter will use the motion model and sensor model to manipulate data from the racecar. Specifically, we will be subscribing to lidar data, odometry data, and inital pose data from the racecar and passing it into the motion model and sensor model. The particle filter will resample probabilities of the car's position and publish a new odometry message estimate and a new transform message.

2.2 Motion Model (Author: Steven Liu)

The motion model calculates how pose estimates change based on odometry data measured from the car. In particular, for a given pose (or for each pose in an array of poses), we compose with it a transformation derived from the odometry $\Delta x, \Delta y, \Delta\theta$ in the car frame.
Because we are working in two dimensions, poses consist of three components x, y, θ, representing two Cartesian coordinates and a rotation relative to a fixed world frame. To compute a new pose, we only need to convert the odometry, which is represented relative to the car, to the same frame as the poses. This

requires us to rotate the transformation using the rotation matrix

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

which can be used to rotate the odometry $\Delta x, \Delta y$ into the world frame. $\Delta\theta$ is simpler to handle: we only need to add it to θ directly. Thus, we get a new pose

$$\begin{aligned} x' &= x + \Delta x \cos\theta - \Delta y \sin\theta \\ y' &= y + \Delta x \sin\theta + \Delta y \cos\theta \\ \theta' &= \theta + \Delta\theta \end{aligned}$$

The computation thus far has been fully deterministic. However, while the odometry is quite good, it is not infinitely accurate. Thus, adding noise to the final pose is desirable. This is accomplished by adding three uncorrelated normally distributed random values to the components x, y, θ of the final pose.

2.3 Sensor Model (Author: Benjamin Rich)

A crucial aspect of Monte Carlo Localization is the ability to approximate the probability of various poses being the actual robot location, given some sort of observation. In the case of our car, a pose is defined by (x, y, θ) relative to the map frame, and our observation is an array of distances from a scan of our Hokuyo LIDAR. In order to establish the probablity of a pose, we first need to establish the probability of a certain distance being recieved as part of a LIDAR scan.

LIDAR Distance Probability Distribution: Multiple distributions are combined into one to create a continuous probability disribution that approximaes the probability of a given LIDAR scan outputting a distance z_k given a hypothesis distance x_k. In short, four possible 'events' that could affect the reading are accounted for:

1. The probability of detecting a known obstacle as expected in the map is modeled using a Gaussian distribution (p_{hit}) centered a the hypothesis distance (i.e. the Gaussian distribution is maximum at the actual distance).

$$p_{hit}(z_k^{(i)}|x_k, m) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k^{(i)} - d)^2}{2\sigma^2}\right) & \text{if} \quad 0 \leq z_k \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$$

2. The probability of a short measurement caused by LIDAR reflections or other errors is modeled as a downward-sloping distribution (p_{short}).

$$p_{short}\left(z_k^{(i)}|x_k, m\right) = \frac{2}{d} \begin{cases} 1 - \frac{z_k^{(i)}}{d} & \text{if} \quad 0 \leq z_k^{(i)} \leq d \text{ and } d \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

3. The probability of a missed measurement due to a transparent surface or other errors is modeled as a spike in probability at the maximum range (p_{max}).

$$p_{max}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{\epsilon} & \text{if} \quad z_{max} - \epsilon \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$$

4. The probability of a random measurement is modeled as a small uniform distribution (p_{rand}).

$$p_{rand}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$$

These four distributions are weighted by parameters that sum to one, as in the below equations:

$$p(z_k^{(i)}|x_k,m) = \alpha_{hit}\cdot p_{hit}(z_k^{(i)}|x_k,m)+\alpha_{short}\cdot p_{short}(z_k^{(i)}|x_k,m)+\alpha_{max}\cdot p_{max}(z_k^{(i)}|x_k,m)+\alpha_{rand}\cdot p_{rand}(z_k^{(i)}|x_k,m)$$

$$\alpha_{hit}+\alpha_{short}+\alpha_{max}+\alpha_{rand}=1$$

In our case, we used the default $\alpha_{hit}=0.74, \alpha_{short}=0.07, \alpha_{max}=0.07, \alpha_{rand}=0.12, \sigma_{hit}=8.0$ as suggested for this lab.

A visual of the resulting distribution is shown below in Figure 1:

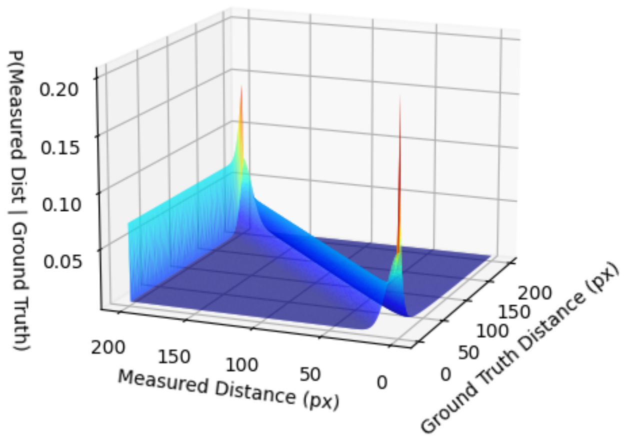
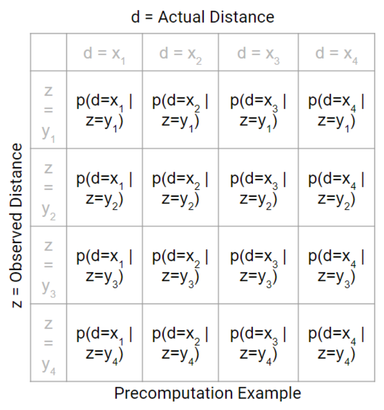


Figure 1: Plot of $p(z_k^{(i)}|x_k,m)$; the Probability of Receiving a Distance z_k Given a Hypothesis Distance x_k.

Having the ability to compute this distribution is useful, but the computation time can be high. Instead, we seek to improve computation time via creating a 'look-up' table.

Discretization and Look-Up Table Creation: We precompute a table of probabilities, where columns are the actual distance d, rows are the measured distance z_k, and the value at a given row and column is $p(z_k|d)$. The probabilities are properly normalized across columns such that the probability of all locations given d sum to 1.0. This table allows us to quickly look up the probability of a distance scan given our actual distance. Figure 2 shows a visual image of an example look-up table as described.


d = Actual Distance

z = Observed Distance

	d = x$_1$	d = x$_2$	d = x$_3$	d = x$_4$
z = y$_1$	p(d=x$_1$ | z=y$_1$)	p(d=x$_2$ | z=y$_1$)	p(d=x$_3$ | z=y$_1$)	p(d=x$_4$ | z=y$_1$)
z = y$_2$	p(d=x$_1$ | z=y$_2$)	p(d=x$_2$ | z=y$_2$)	p(d=x$_3$ | z=y$_2$)	p(d=x$_4$ | z=y$_2$)
z = y$_3$	p(d=x$_1$ | z=y$_3$)	p(d=x$_2$ | z=y$_3$)	p(d=x$_3$ | z=y$_3$)	p(d=x$_4$ | z=y$_3$)
z = y$_4$	p(d=x$_1$ | z=y$_4$)	p(d=x$_2$ | z=y$_4$)	p(d=x$_3$ | z=y$_4$)	p(d=x$_4$ | z=y$_4$)

Precomputation Example

Figure 2: Example Image Showing a Discretized Probability Look-Up Table.

Having created a look-up table, we can finally use it to evaluate the probability of a pose.

Evaluating the Probability of a Pose: Given a LIDAR scan (an observation) and several hypothesis poses, we can estimate the probability of each pose. We first generate a simulated LIDAR scan for each pose using ray tracing (i.e. given a position and orientation on the map, and given knowledge of how the LIDAR scan performs a scan, trace points from the LIDAR out to the map to get estimated LIDAR scan distances). Then, we compute the product probability of each distance in a single scan given the actual distance observed, and thus are able to output the probability of each particle being at the robots actual position in the map! A flowchart for this process is shown in Figure 3.

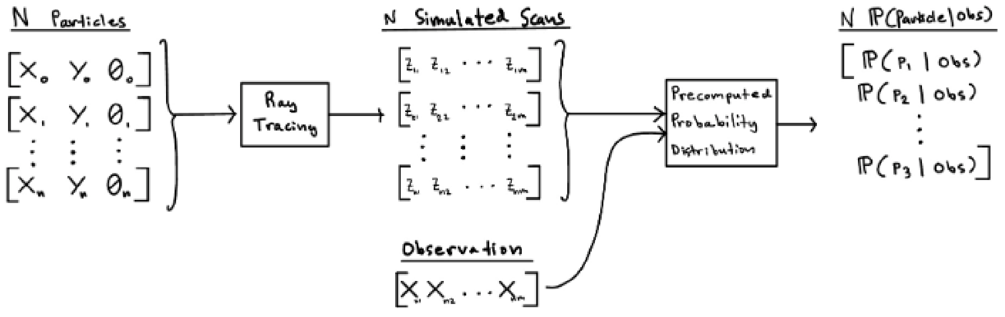


Figure 3: Flowchart Showing the Process of Computing the Probability of Estimated Poses (Particles) Given observed LIDAR Data.

We then implement the above methods in code.

Details on Code Implementation: Our sensor model is implemented using a class structure. Upon initialization, the class creates and stores the discretized table. Then, upon receiving a LIDAR scan and poses through a function call, the probability of each pose is computed and returned for use in the particle filter! All processes in the evaluation of probabilities are implemented with numpy matrix operations to avoid significant computation time.

2.4 Particle Filter (Author: Dinuri Rupasinghe

We have implemented a motion model and sensor model. Now, we can implement Monte Carlo Localization by incorporating both of these models into one formal script - this is the particle filter. The particle filter is composed of three main steps. First, we must obtain the initial pose and create a set of initial particles based on this initial pose. Then, we must publish an odometry message when we obtain odometry data and publish an odometry message when we obtain lidar scan data. Below is a diagram of this workflow.

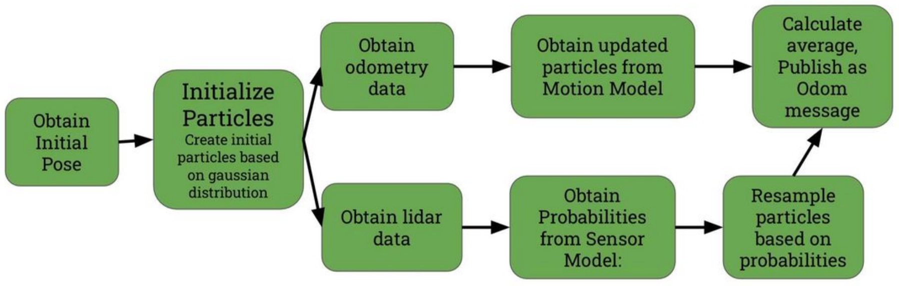

Obtain Initial Pose
Initialize Particles Create initial particles based on gaussian distribution
Obtain odometry data
Obtain updated particles from Motion Model
Calculate average, Publish as Odom message
Obtain lidar data
Obtain Probabilities from Sensor Model:
Resample particles based on probabilities

Figure 4: Diagram of Particle Filter Implementation

Initial Pose and Particles: we are able to access an initial pose of our racecar by accessing the initial pose, of type *PoseWithCovarianceStamped*. In order to access this attribute by placing the car in different positions on our map using a 2D pose estimate, we must add the Pose with Covariance topic in RViz. Thus, we have the ability to access the position and covariance of the initial position. It is important to access this data because it enables us to obtain an inference on an appropriate initial set of particles.

In order to generate a random set of initial particles, we used a python numpy function to obtain a random sample of a multivariate normal distribution. Our group chose this specifically because we wanted to create a set of particles based on as much data as possible, and this particular function, *random.multivariatenormal* creates a distribution based on both the initial pose and the covariance of the initial pose.

Obtain Pose from Motion Model: One of our goals is to publish new odometry data when the robot obtains odometry data. In order to do this, we utilize our motion model, since we can obtain a new set of particles based on our current set of particles and current odometry data. By recording time between the aggregation of current odometry data and the moment we attempt to calculate new odometry data, we can obtain current odometry data.
Once we obtain the most up to date odometry data, we are able to generate a set of updated particles by passing our current particles and current odometry into an evaluate function of our motion model. We then calculate the average of these particles and publish this average as an odometry message and as a pose transform message.

Obtain Pose from Sensor Model: Another goal of the particle filter is to generate new odometry data while factoring in the probabilities calculated by the sensor model. We calculate probabilities of each particle's position by using the sensor model's evaluate function and passing in a set of particles and lidar scan data. Then we conduct a resample of the particles. The purpose of resampling is to generate a new set of particles based on particles that are likely to be present. We performed this resample by using the numpy function *random.choice*, and generated particles given the current particles, the amount of particles we desire, and the probabilities calculated by the sensor model. We also inserted a clause stating that there is a thirty percent chance that we return the current version of particles rather than resample based on their probabilities. We implemented this because there is a chance that our racecar does randomly choose the same set of particles again. After resampling, once again, we calculate the average of these updated particles from the resample and publish this average as an odometry message and as a pose transform message.

Calculating Average of Particles: As a note, we calculate the x and y averages of our particles as one would calculate them classically. However, we accounted differently for the average of circular coordinates because simply taking the average of the angles will lead to an inaccurate calculation. Instead, we first take the average of the sine and cosine components and then obtain an average angle based on the average sine and cosine, utilizing arctangent.

Quaternions: It is important to note that some of the attributes of the data that we subscribe to from the racecar is in the form of quaternions rather than angles in units of degrees. We therefore applied a transformation using an import of *tf.transformations as trans*, which one can see is applied in our code.

Locking and Threading: Lastly, a large reason why our simulation ran into errors was because multiple processes were running at once. Since our racecar was retrieving both odometry data and laser scan data at the same time, our particle filter attempted to subscribe to an odometry and lidar callback at the same time. We fixed this by applying a series of "locks". Thus, the both the motion model and the sensor model portions of our particle filter can run sequentially rather than at the same time.

3 Experimental Evaluation (Author: Benjamin Rich)

3.1 Overview

Having implemented our motion model, sensor model, and particle filter, we sought to qualify the performance of our MCL implementation in simulation and in real life. To do so we use a three-pronged approach. We qualified our system in simulation using the gradescope autograder provided and we qualitatively tested our system in the Stata basement by making a comprehensible video comparing the performance of our tracking to the actual position. We measured the convergence rate of our particles both on the car and in simulation and measured cross-tracking error in simulation.

3.2 Gradescope Autograder

The Gradescope Autograder provided by staff provides a useful performance metric. In these simulations, our particle filter receives simulated odometry data and laser scans and estimates the position of the car with respect to ground truth. This is done in the presence of no odometry noise, some odometry noise, and more odometry noise. The resultant plots are shown in Figure 4.

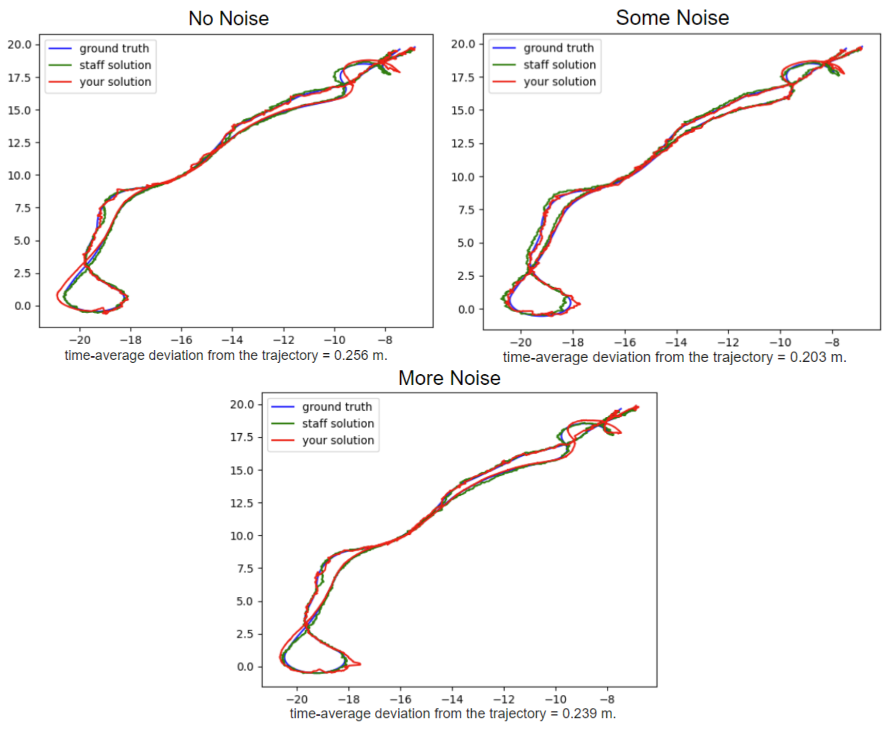


Figure 5: Simulation Plots. From left to right, top down: No Odometry Noise, Some Noise, More Noise.

Visually, our solution performs well. However, we can notice that especially with no or more noise, our solution struggles to track tight turns as well as perhaps it should. This suggests there is room for improvement in our motion model for how it handles tight turns. For each simulation as shown in Figure 4, Gradescope provides a time-averaged deviation from the trajectory. These results are summarized in Table 1.

Experiment	No Noise	Some Noise	More Noise
Time-Averaged Deviation from Trajectory	0.256m	0.203m	0.239m

These values are good, but not great. A deviation of $0.2 - 0.26$m or of $0.66 - 0.84$ feet may be acceptable for some applications but certainly could be too high in situations that require higher precision (such as navigating a complex area). However, in practice, we found this deviation to be not very noticeable.

3.3 Video Demonstration

A video demonstration of MCL in the Stata Basement was creative for the purposes of qualitative observation-based evaluation of our performance. The video can be accessed at:

https://drive.google.com/file/d/1GLzSGjiPZLCoxgMg-a0pmlcAc0vJkYSs/view?usp=sharing

An example frame from our video is shown in Figure 5.

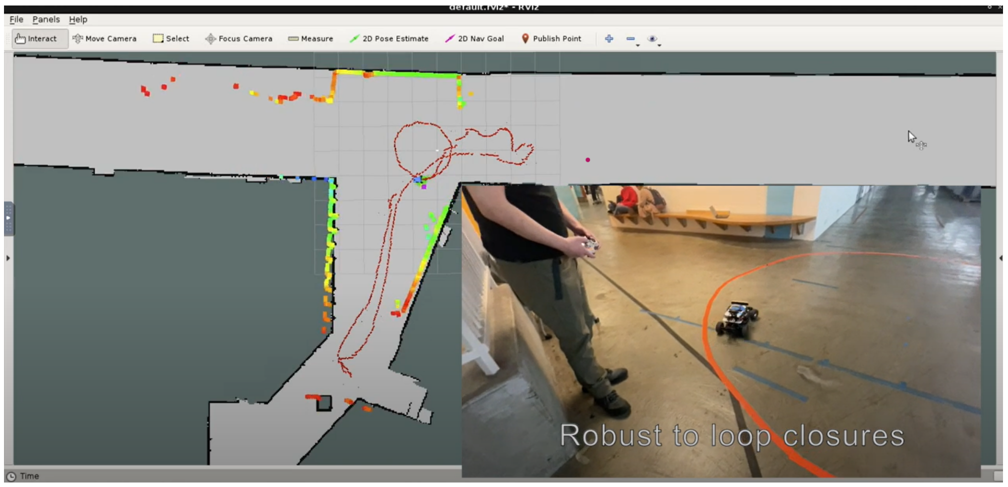


Figure 6: Example Frame from Demonstration Video. The red dotted line behind the car represents the car's previous estimated positions. The video shows the car's true position.

The video shows us that in general our MCL implmentation performs very well! We were able to drive the car around via only viewing its estimated position. In general; the car's slime-trail shows that the car tracks well with respect to it's true position. We can see that when the car performs a full circle of a turn the estimated position forms a 'loop-closure' as expected. It is worth noting that the laser scans have a harder time matching to a pose when the robot car is in the uppermost hallway. This is to be expected, as the map given for the Stata basement was created before a series of large crates were placed in the uppermost hallway, thus the map no longr reflects accurate real-world conditions in the uppermost hallway.

3.4 Convergence Time

In order to ensure that our implementation works as expected, we measured the convergence time of our particles (pose estimates). If our implementation were working correctly, we would expect that as we randomly generate poses and resample them, they would converge toward the true position with time. I.e., they would start spread out but converge upon the real location. Thus, we devised a metric to measure the time it takes for particles to converge.

Definition of Convergence Time: We sought to evaluate convergence time both on the physical car and in simulation. Recall that a pose p is given by (x, y, θ). We defined 'unconverged' as when the standard deviation of *one* of x, y and θ is above x_{up}, y_{up}, or θ_{up} respectively. In other words, we consider the particles unconverged when one of x, y, θ is above a large standard deviation. $x_{up}, y_{up}, \theta_{up}$ were determined experimentally. We initialized the car in a 'False' location (i.e. we told the car it was in a fake position) and measured the initial standard deviations of the x, y, θ components of particles across all particles, and defined these as x_{up}, y_{up}, θ. In practice, the standard deviation is only this high when the car is first initialized in new position or it is fed false data. We defined 'converged' similarly; when *all* of x, y, θ are below $x_{low}, y_{low}, \theta_{low}$, where $x_{low}, y_{low}, \theta_{low}$ were chosen experimentally when the car is confident in its position (i.e. is estimating it's position correctly). Then, convergence time is simply given by the time it takes to go from 'unconverged' to 'converged'. This was measured using $rospy.get\_time()$ in our code. Mathematically, we can write the following expressions to define our measurement metrics:

$$\text{unconverged} \implies (std(x) > x_{up}) \lor (std(y) > y_{up}) \lor (std(\theta) > \theta_{up}) \tag{1}$$

$$\text{converged} \implies (std(x) < x_{low}) \land (std(y) < y_{low}) \land (std(\theta) < \theta_{low}) \tag{2}$$

$$t_{convergence} = t_{converged} - t_{unconverged} \tag{3}$$

In simulation, we can get an unconverged state by initializing a new location. In the real world, we could get an unconverged state by feeding the car 'fake' LIDAR scans (i.e. obstructing it's view) and confusing it of it's position. Then we allow the particles to return to a converged state. The results are summarized below in Table 2.

Location	Real-World (N=6)	Simulation (N=20)
Average Convergence Time (s)	0.09	0.07

These results tell us that on average, our particles converge in under a 10th of a second both on the real car and in simulation, which is a decently fast rate. However, it is worth noting that the standard deviation of convergence time can be large, as the convergence time depends on a number of random factors including how the particles are initialized, how close he robot's estimated position is to its true position, etc. For example, both on he physical car and in simulation, we saw occasional convergence times on the order of 0.2 or 0.3 seconds. So even though the average convergence time is low, in some cases it could be considerably higher.

3.5 Cross-Track Error

Finally, we measured cross-tracking errors in simulation. We listened to the transformation published between *map* and $base_link$ and compared the translational components to our estimated position in simulation. An example plot of cross track errors obtained by listening to the aforementioned transformation, and publishing the difference with respect to our pose, is show in Figure 6.

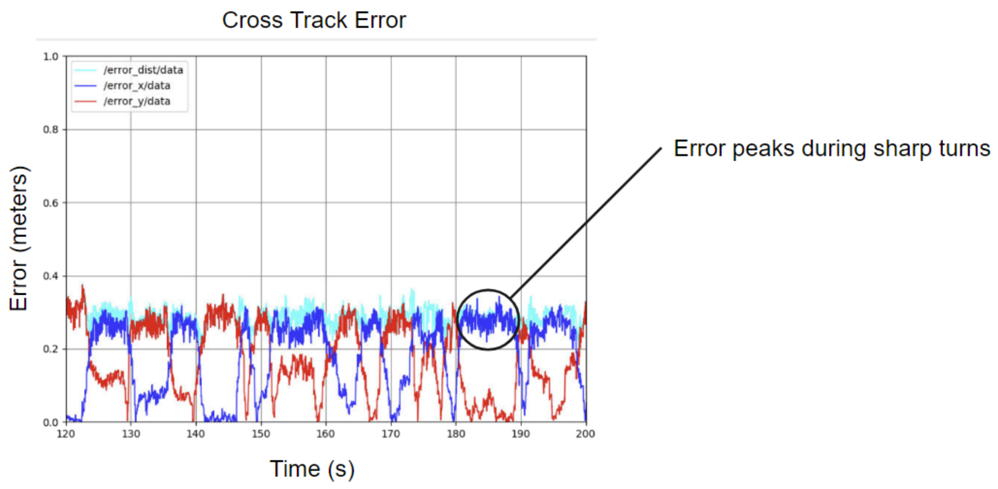


Figure 7: Cross Track Error Collected over 80 Seconds in Simulation

In the above plot, one can view somewhat of a saw-tooth pattern in error. We believe this is primarily due to sharp turns. We manually controlled the car in simulation while collecting the data used in Figure 6, and cross-tracking error would increase for one direction and sometimes decrease for another when the car was rapidly turned. However, we see that the error remains at or below ≈ 0.3 meters for the entirety of the test. This is consistent with our gradescope score of around $0.2 - 0.25$ meters avg deviation.

3.6 Summary

Overall, our car performs well following a line in simulation, has < 0.1s average convergence rate, and on average has $0.2 - 0.25$ deviation from a ground truth position. It performs well when tested in the Stata basement, but is sensitive to sharp turns.

4 Potential Improvements (Author: Steven Liu)

Our localization algorithm performs quite well. However, due to time constraints we were not able to do extensive tuning of the parameters used in the motion model and sensor model. As such, it is likely that we could improve performance by a good amount without changing our overall approach.
Speed improvements could also almost certainly be made. The car did not move extremely fast during testing, but as speeds get higher cadences will have to get faster to keep up. There remain some places in our code where we believe we can optimize, in particular by using numpy more extensively.
One notable consistent flaw in our performance is that the estimated pose has a tendency to ”understeer” by a small amount; that is, the car thinks it is turning slightly less than it actually is. We were not able to diagnose the underlying cause of this, so it is difficult to speculate what a plausible fix would be.

5 Conclusion (Author: Jose Soto)

Overall, our team can confidently say that we developed and implemented Monte Carlo Localization for our vehicle. We were successfully able to implement all three components: the motion model to create odometry estimates, sensor model to explore the probabilities of our particles and the particle filter to refine our odometry estimates using our sensor model. Even after implementing noise, we found our algorithm was able to accurately follow the ground truth solution in simulation. The path to hardware was difficult, requiring several substantial changes, but we were able to reach great localization in our final product.

Looking ahead, our team has a few improvements to be made in preparation for our final deliverable at the end of the course which includes: making our localization algorithm more robust to turns, and further improving the latency and refresh rate of the algorithm - especially useful as we plan to traverse the Stata basement at race speeds - which could interfere with the performance of our MCL program.

6 Lessons Learned

6.1 Benjamin Rich

This lab was probably the most time-consuming one thus far. I learned that it really pays to sit and think and wait until you have a solid grasp of a concept before beginning the code. Rushing to the code prior to truly understand the concept sometimes works in that as you code it you begin to understand it better, but this approach often wastes time. The end result of this lab was really interesting and made the large effort required to get it working totally worth it.

6.2 Dinuri Rupasinghe

I enjoyed this lab because I learned about localization in a more comprehensive, quantitative way. I am glad that I was able to become even more familiar with the general syntax of subscribers and publishers, specifically in our implementation of the particle filter. I am grateful that our team has been proactive in documenting their technical work and will utilize comments and doc strings more often in the future, as I feel that they were extremely helpful for when our team had to work remotely, especially over spring break.

6.3 Sophia Wang

This lab was an interesting look into transitioning from software to hardware since threading and seeing how our rates in simulation differed from rates on the robot were both challenging. In terms of debugging, I learned that debugging is most productive in small groups of two to three and also in modules with unit tests. Sometimes, we would gather to debug but found it was hard since only one laptop was connected to the robot at a time, and we were all huddled around a single screen. Overall, I enjoyed this lab and found it very exciting to visualize its results as the racecar traced its own location across the Stata basement.

6.4 Steven Liu

This lab was, in terms of technical content, definitely the most interesting of the labs so far; working out the kinks of the motion model and particle filter were gratifying, if quite time-consuming. It's also the first lab for which I was building off of other people's code in a significant way, rather than being the one to provide a first draft of a module, since Dinuri and Sophia got started on the particle filter before I took a serious look at it. This has made me reconsider to some extent how I write code in collaborative environments, specifically with leaving more informative comments and documentation.

6.5 Jose Soto

In my opinion this was one of the most technically challenging and interesting labs that we've worked on so far. Spring break falling in the middle of the lab could have posed some communication issues for our team, but I think we handled it well and were able to deliver well despite the circumstances of some members being able to work on the lab while others weren't able to.

7 Appendix

7.1 Particle Filter Code

```
#!/usr/bin/env python2
import numpy as np
import rospy
from sensor_model import SensorModel
from motion_model import MotionModel

from std_msgs.msg import Float32
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseWithCovarianceStamped,TransformStamped,Pose,PoseArray, Point
import tf.transformations as trans
import tf


from threading import Lock

import tf2_ros

class ParticleFilter:
    def __init__(self):

        self.measure_convergence_rate = False
        self.measuring = False
        self.measure_time = None
        self.measure_error = False # only works in SIMULATION
        if self.measure_error:
            self.gt = Point()
            self.pos = Point()
            self.has_pos = False
            #self.tf_sub = rospy.Subscriber("/tf",TransformStamped, self.tf_callback, queue_size=10)
            self.error_pub_x = rospy.Publisher("/error_x", Float32, queue_size=10)
            self.error_pub_y = rospy.Publisher("/error_y", Float32, queue_size=10)
            self.error_pub_dist = rospy.Publisher("/error_dist", Float32, queue_size=10)
            self.gt = (None, None, None) # x,y,z
            self.tf_listener = tf.TransformListener()


        self.num_particles = 200
        self.motion_model = MotionModel()
        self.sensor_model = SensorModel()
        self.last_odom = None
        self.last_odom_time = rospy.get_time()

        # Establish thread locking for the two callbacks updating the particle list
        self.particle_lock = Lock()

        # Get parameters
        self.particle_filter_frame = \
                rospy.get_param("~particle_filter_frame")


        scan_topic = rospy.get_param("~scan_topic", "/scan")
        odom_topic = rospy.get_param("~odom_topic", "/odom")

        self.probabilities = None

        self.initial_pose = np.array([0,0,0])
        self.particles = np.zeros((200,3)) #np.array([0,0,0])


        self.transform_pub = tf2_ros.TransformBroadcaster()
        self.pose_sub  = rospy.Subscriber("/initialpose", PoseWithCovarianceStamped, self.pose_callback, queue_size=1)
        self.laser_sub = rospy.Subscriber(scan_topic, LaserScan, self.lidar_callback, queue_size=1)
        self.odom_sub  = rospy.Subscriber(odom_topic, Odometry, self.odom_callback, queue_size=1)
        self.odom_pub  = rospy.Publisher("/pf/pose/odom", Odometry, queue_size = 1)
        self.particle_pub = rospy.Publisher("particles", PoseArray, queue_size = 1)

    def calc_avg(self, particles):
        '''
        Take average of the calculated particles returned by the evaluate function of motion or sensor model
        '''

        avg_x = np.average(particles[:,0], weights=self.probabilities, axis = 0)
        avg_y = np.average(particles[:,1], weights=self.probabilities, axis = 0)
        avg_cos = np.average(np.cos(particles[:,2]), weights=self.probabilities, axis=0)
        avg_sin = np.average(np.sin(particles[:,2]), weights=self.probabilities, axis=0)
        theta_pos = np.arctan2(avg_sin,avg_cos)

        return [avg_x, avg_y, theta_pos]

    def tf_callback(self, data):
        self.gt = data
        if self.has_pos:
            x = self.pos.pose.pose.position.x
            y = self.pos.pose.pose.position.y
            error_x = abs(x-self.gt[0])
            error_y = abs(y-self.gt[1])
            error_dis = np.sqrt(error_x**2+error_y**2)
            self.error_pub_x.publish(error_x)
            self.error_pub_y.publish(error_y)
            self.error_pub_dist.publish(error_dis)


    # Callback functions
    def pose_callback(self,data):
        '''
        Gets initial pose from rviz data
        Remember to add posewithcovariance topic on rviz
        '''
        with self.particle_lock:

            self.initial_pose = np.array([data.pose.pose.position.x,
                                          data.pose.pose.position.y,
                                          2*np.arctan2(data.pose.pose.orientation.z,data.pose.pose.orientation.w)])
            self.initial_cov = np.array([[data.pose.covariance[0],data.pose.covariance[1],data.pose.covariance[5]],
                                         [data.pose.covariance[6],data.pose.covariance[7],data.pose.covariance[11]],
                                         [data.pose.covariance[30],data.pose.covariance[31],data.pose.covariance[35]]])
            self.particles = np.random.multivariate_normal(self.initial_pose,self.initial_cov, size = self.num_particles)


    def publish_pose(self,pose):
        avg = pose
        z,w = np.sin(avg[2]/2),np.cos(avg[2]/2)
        quat = trans.quaternion_about_axis(avg[2],(0,0,1))

        odom_msg = Odometry()
        odom_msg.header.stamp = rospy.Time.now()
        odom_msg.header.frame_id = 'map'
        odom_msg.pose.pose.position.x = avg[0]
        odom_msg.pose.pose.position.y = avg[1]
        odom_msg.pose.pose.position.z = 0
        odom_msg.pose.pose.orientation.x = quat[0]
        odom_msg.pose.pose.orientation.y = quat[1]
        odom_msg.pose.pose.orientation.z = z
        odom_msg.pose.pose.orientation.w = w

        self.odom_pub.publish(odom_msg)
        if self.measure_error:
            self.pos = odom_msg
            self.tf_listener.waitForTransform("map","base_link", rospy.Time(), rospy.Duration(1.0))
            trans_vect, _ = self.tf_listener.lookupTransform("map","base_link",rospy.Time(0))
            self.tf_callback(trans_vect)
            if not self.has_pos:
                self.has_pos = True


        pose_transform = TransformStamped()
        pose_transform.header.stamp = rospy.Time.now()
        pose_transform.header.frame_id = '/map'
        pose_transform.child_frame_id = self.particle_filter_frame # 'base_link_pf' for the simulator, 'base_link' for the car
        pose_transform.transform.translation.x = avg[0]
        pose_transform.transform.translation.y = avg[1]
        pose_transform.transform.translation.z = 0
        pose_transform.transform.rotation.x = quat[0] #0
        pose_transform.transform.rotation.y = quat[1] #0
        pose_transform.transform.rotation.z = z #z
        pose_transform.transform.rotation.w = w #w
        self.transform_pub.sendTransform(pose_transform)

    def odom_callback(self,data):
        '''
        Uses motion model
        '''
        if self.particles is None: return

        with self.particle_lock:

            now = rospy.get_time()
            now_odom = np.array([data.twist.twist.linear.x,
                                 data.twist.twist.linear.y,
                                 data.twist.twist.angular.z])

            if self.last_odom is None: odom = now_odom*(now-self.last_odom_time)
            else: odom = (now_odom+self.last_odom)*(now-self.last_odom_time)/2

            self.last_odom = now_odom
            self.last_odom_time = now

            self.updated_particles = self.motion_model.evaluate(self.particles, odom)
            avg = self.calc_avg(self.updated_particles)

            self.publish_pose(avg)
            self.particles = self.updated_particles.copy()

            # ADDING STUFF TO MEASURE CONVERGENC RATE
            if self.measure_convergence_rate:
                dev1,dev2,dev3 = self.particles.std(axis=0)
                #print([dev1,dev2,dev3])
                # NOTE: up thresholds determined by giving false initialization and examinind stds
                up_threshold_1 = 0.13
                up_threshold_2 = 0.13
                up_threshold_3 = 0.13
                # NOTE: low threshold determined by giving true initilization and examining STDS
                low_threshold = 0.07
                if dev1 > up_threshold_1 or dev2 > up_threshold_2 or dev3 > up_threshold_3:
                    print('began measuring')
                    self.measuring = True
                    self.measure_time = rospy.get_time()
                if dev1 <= low_threshold and dev2 <= low_threshold and dev3 <= low_threshold and self.measuring:
                    print('end measuring')
                    curr = rospy.get_time()
                    self.measuring = False
                    diff = curr - self.measure_time
                    print('Convergence Time = ' + str(diff))
            # END OF CHANGES!




            if False: #Publish particles for debugging purposes
```

```
                particle_msg = PoseArray()
                particle_msg.header.stamp = rospy.Time.now()
                particle_msg.header.frame_id = '/map'

                for p in self.particles:
                    pose = Pose()
                    z,w = np.sin(p[2]/2),np.cos(p[2]/2)
                    pose.position.x = p[0]
                    pose.position.y = p[1]
                    pose.orientation.z = z
                    pose.orientation.w = w

                    particle_msg.poses.append(pose)

                self.particle_pub.publish(particle_msg)


    def lidar_callback(self, data):
        '''
        Uses sensor model
        '''
        if self.particles is None: return
        if np.random.rand() > 0.3: return
        with self.particle_lock:

            probs = self.sensor_model.evaluate(self.particles, np.array(data.ranges),1) # IF ON RACECAR, CHANGE 1 to 11
            probs /= sum(probs)
            self.probabilities = probs

            particle_resample = self.particles[np.random.choice(self.particles.shape[0], size=self.particles.shape[0], p=probs),

            avg = self.calc_avg(particle_resample)

            self.publish_pose(avg)

            self.particles = particle_resample.copy()

if __name__ == "__main__":
    rospy.init_node("particle_filter")
    pf = ParticleFilter()
    rospy.spin()
```

7.2 Sensor Model Code

```
import numpy as np
from localization.scan_simulator_2d import PyScanSimulator2D
import matplotlib.pyplot as plt
# Try to change to just `from scan_simulator_2d import PyScanSimulator2D`
# if any error re: scan_simulator_2d occurs

import rospy
import tf
from nav_msgs.msg import OccupancyGrid
from tf.transformations import quaternion_from_euler

class SensorModel:

    def __init__(self):
        # Fetch parameters
        self.map_topic = rospy.get_param("~map_topic")
        self.num_beams_per_particle = rospy.get_param("~num_beams_per_particle")
        self.scan_theta_discretization = rospy.get_param("~scan_theta_discretization")
        self.scan_field_of_view = rospy.get_param("~scan_field_of_view")
        self.lidar_scale_to_map_scale = rospy.get_param("~lidar_scale_to_map_scale", 1.0)
        ####################################
        # TODO
        # Adjust these parameters
        self.alpha_hit = 0.74
        self.alpha_short = 0.07
        self.alpha_max = 0.07
        self.alpha_rand = 0.12
        self.sigma_hit = 8.0

        # Your sensor table will be a ‘table_width‘ x ‘table_width‘ np array:
        self.table_width = 201
        self.z_max = self.table_width-1
        ####################################

        # Precompute the sensor model table
        self.sensor_model_table = None
        self.precompute_sensor_model()

        # Create a simulated laser scan
        self.scan_sim = PyScanSimulator2D(
                self.num_beams_per_particle,
                self.scan_field_of_view,
                0, # This is not the simulator, don’t add noise
                0.01, # This is used as an epsilon
                self.scan_theta_discretization)

        # Subscribe to the map
        self.map = None
        self.map_set = False
        rospy.Subscriber(
                self.map_topic,
                OccupancyGrid,
                self.map_callback,
                queue_size=1)

    def precompute_sensor_model(self):

        """
        Generate and store a table which represents the sensor model.

        For each discrete computed range value, this provides the probability of
        measuring any (discrete) range. This table is indexed by the sensor model
        at runtime by discretizing the measurements and computed ranges from
        RangeLibc.
        This table must be implemented as a numpy 2D array.

        Compute the table based on class parameters alpha_hit, alpha_short,
        alpha_max, alpha_rand, sigma_hit, and table_width.

        args:
            N/A

        returns:
            No return type. Directly modify ‘self.sensor_model_table‘.
            columns are d values!
            200x200
        """

        zmax = self.table_width-1
        sigma = self.sigma_hit
        alphaHit = self.alpha_hit
        alphaShort = self.alpha_short
        alphaMax = self.alpha_max
        alphaRand = self.alpha_rand
        table_width = self.table_width

        plot = False # if want to plot the probability distribution. Requires matplotlib.pyplot imported as plt
        checksum = False # if want to print sum’s by column (to make sure ~1.0 for proper normalization)
        def phit(zk,d):
            if 0<=zk<=zmax:
                return 1.0/((2.0*np.pi*sigma**2.0)**(0.5))*np.exp(-1.0*(((zk-d)**2.0)/(2.0*sigma**2.0)))
            return 0.0

        def pshort(zk,d):
            if zk >= 0 and zk<=d and d != 0:
                return 2.0/d*(1-(float(zk)/d))
            return 0.0

        def pmax(zk,d):
            if zk == zmax:
                return 1.0
            return 0.0

        def prand(zk,d):
            if zk>=0 and zk <= zmax:
                return 1.0/zmax
            return 0.0

        def getP(zk,d): # everything except hit
            return alphaShort*pshort(zk,d)+alphaMax*pmax(zk,d)+alphaRand*prand(zk,d)

        #compute PHIT prior to others, columns are d
        out = []
        out = np.zeros((table_width,table_width))
        for i in range(table_width): # z
            for j in range(table_width): # d
                out[i][j] = phit(i,j)

        out = out/out.sum(axis=0)
        out *= alphaHit

        #compute other part of distribution
        for i in range(table_width):
            for j in range(table_width):
                out[i][j] += getP(i,j)

        # normalize out
        out = out/out.sum(axis=0)
        self.sensor_model_table = out

    def downsample(self, arr, spacing, mode = 'direct'):
        if mode == 'avg':
            end = spacing * (len(arr) // spacing)
            return np.mean(arr[:end].reshape(-1, spacing), axis=1)
        else:
            return  arr[0::spacing]



    def evaluate(self, particles, observation, spacing=1):

        """
        Evaluate how likely each particle is given
        the observed scan.

        args:
            particles: An Nx3 matrix of the form:

                [x0 y0 theta0]
                [x1 y0 theta1]
                [    ...     ]

            observation: A vector of lidar data measured
                from the actual lidar.
        returns:
           probabilities: A vector of length N representing
               the probability of each particle existing
               given the observation and the map.
        """

        observation = self.downsample(observation,spacing)

        # UNCOMMENT THE NEXT 3 LINES IF CODE ON ACTUAL CAR!!!
        scans = self.scan_sim.scan(particles)
        # for i in range(len(scans)):
        #     scans[i] = self.downsample(scans[i],spacing)

        z_k = np.clip(np.array(observation)/ (self.map_resolution*self.lidar_scale_to_map_scale), a_min=0, a_max = self.z_max) #
        d = np.clip(scans / (self.map_resolution*self.lidar_scale_to_map_scale), a_min = 0, a_max = self.z_max) # clip scans
        probs = np.prod(self.sensor_model_table[np.rint(z_k).astype(np.int32), np.rint(d).astype(np.int32)], axis=1)**(1/2.2) #

        return probs



    def map_callback(self, map_msg):
        # Convert the map to a numpy array
        self.map_resolution = map_msg.info.resolution

        self.map = np.array(map_msg.data, np.double)/100.
        self.map = np.clip(self.map, 0, 1)

        # Convert the origin to a tuple
        origin_p = map_msg.info.origin.position
        origin_o = map_msg.info.origin.orientation
        origin_o = tf.transformations.euler_from_quaternion((
                origin_o.x,
                origin_o.y,
                origin_o.z,
                origin_o.w))
        origin = (origin_p.x, origin_p.y, origin_o[2])

        # Initialize a map with the laser scan
        self.scan_sim.set_map(
                self.map,
                map_msg.info.height,
                map_msg.info.width,
                map_msg.info.resolution,
                origin,
                0.5) # Consider anything < 0.5 to be free

        # Make the map set
        self.map_set = True

        print("Map initialized")
```

7.3 Motion Model Code

```
import numpy as np
import rospy

class MotionModel:

    def __init__(self):

        self.DETERMINISTIC = rospy.get_param(rospy.search_param('deterministic'))

        #Constants for uncertainty - 1,2 are rotational, 3,4 are translational
        self.alpha = {1: 0.03,
                      2: 0.03,
                      3: 0.02,
                      4: 0.02} #Arbitrary values, no idea if they make sense
        self.xy_noise = 0.02
        self.theta_noise = 0.02

    def eps_b(self,b,n=2):
        '''
        Sample a single value from a distribution with mean zero and variance b

        args:
            b: A float, the overall variance of the distribution
            n: An integer for how many uniform samples to average
                Defaults to 2 (triangular)

        returns:
            A randomly-generated float
        '''
        c = np.sqrt(abs(3*b/n))
        return sum(np.random.rand(n))*2*c-c*n

    def evaluate(self, particles, odometry):
        """
        Update the particles to reflect probable
        future states given the odometry data.

        args:
            particles: An Nx3 matrix of the form:

                [x0 y0 theta0]
                [x1 y0 theta1]
                [    ...     ]

            odometry: A 3-vector [dx dy dtheta]

        returns:
            particles: An updated matrix of the
                same size
        """

        N = particles.shape[0]
        for i in range(N):
            # Rotate the position components of the odometry to the world frame
            cos,sin = np.cos(particles[i,2]),np.sin(particles[i,2])
            rotated_displacement = (cos*odometry[0]-sin*odometry[1],
                                    sin*odometry[0]+cos*odometry[1])

            if self.DETERMINISTIC:
                # Add the displacements to the pose
                particles[i,0] += rotated_displacement[0]
                particles[i,1] += rotated_displacement[1]
                particles[i,2] += odometry[2]
            else:
                particles[i,0] += rotated_displacement[0]+np.random.normal(0,self.xy_noise)
                particles[i,1] += rotated_displacement[1]+np.random.normal(0,self.xy_noise)
                particles[i,2] += odometry[2]+np.random.normal(0,self.theta_noise)
                '''
                # Adapted from Probabilistic Robotics, sample_motion_model_odometry

                # Decompose motion into three components
                delta_rot_1 = np.arctan2(rotated_displacement[1],
                                         rotated_displacement[0])-particles[i,2]
                delta_trans = (odometry[0]**2+odometry[1]**2)**0.5
                delta_rot_2 = odometry[2]-delta_rot_1

                # Add noise
                delta_rot_1 -= self.eps_b(abs(self.alpha[1]*delta_rot_1)+\
                                          self.alpha[2]*delta_trans)
                delta_trans -= self.eps_b(self.alpha[3]*delta_trans+\
                                          abs(self.alpha[4]*(delta_rot_1+delta_rot_2)))
                delta_rot_2 -= self.eps_b(abs(self.alpha[1]*delta_rot_2)+\
                                          self.alpha[2]*delta_trans)

                # Update the pose
                particles[i,0] += delta_trans*np.cos(particles[i,2]+delta_rot_1)
                particles[i,1] += delta_trans*np.sin(particles[i,2]+delta_rot_1)
                particles[i,2] += delta_rot_1+delta_rot_2
                '''

        return particles
```