CSE 490R P1 - Localization using Particle Filters
Due date: Sun, Jan 28 - 11:59 PM

1 Introduction

In this assignment you will implement a particle filter to localize your car within a known map. This will provide a global pose ($\mathbf{x} =< x, y, \theta >$) for your car where ($x$, y) is the 2D position of the car within the map and θ is the heading direction w.r.t the map's frame of reference. You will be implementing different parts of the particle filter, namely the motion model, sensor model, and the overall filtering framework. You will also be tuning various parameters that govern the particle filter to gain intuition on the filter's behaviour.

Provided is skeleton code (in Python) to begin this assignment, multiple bag files of recorded data with results from the TAs' implementation of the particle filter, and maps of Sieg 322, the 3rd floor of Sieg and the 4th floor of the Paul Allen Center. All provided reference bags are from runs in the 4th floor of the Paul Allen Center - you can replay them to test the implementations of different parts of your particle filter and compare the results to the provided baseline results (your results need not match exactly).

Submission items for this homework includes all written code, all generated plots, and a PDF or txt file answering the questions listed in the following sections. Additionally, there will be a final demo where we will meet with each team to test their particle filter on actual runs through the 3rd floor of the Sieg building – we will also be talking to each team separately to evaulate their understanding of different parts of the assignment (and to ascertain the contributions of each team member).

Next, we discuss each part of the assignment in further detail.

2 Motion Model

In the first part of this assignment, you will implement and evaluate two different motion models: a simple motion model based on wheel odometry information and the kinematic car model. As discussed in class, a motion model specifies the probability distribution $p(\mathbf{x}_t|\mathbf{x}_{t-1}, u_t)$, i.e. the probability of reaching a pose $\mathbf{x}_t$ given that we apply a control u_t from pose $\mathbf{x}_{t-1}$. Unlike a traditional Bayes filter which requires us to explicitly represent the posterior over possible future poses (i.e. explicitly compute $p(\mathbf{x}_t|\mathbf{x}_{t-1}, u_t)$), we only need to be able to draw samples from this distribution for the particle filter:

$$\mathbf{x}'_t \sim p(\mathbf{x}_t|\mathbf{x}_{t-1}, u_t) \tag{1}$$

where $\mathbf{x}'_t$ is a possible next pose (x_t, y_t, θ_t) sampled from the motion model. We do this by having our motion models output a change in state $\Delta x, \Delta y, \Delta\theta$.

A motion model is critical to successful state estimation: the more accurate we are at predicting the effect of actions, the more likely we are to localize accurately. We will explore two motion models in this assignment: the odometry motion model that uses encoder information to make future pose predictions and the kinematic car model which models the kinematics of the racecar. You will implement a sampling algorithm for each motion model and investigate and compare the characteristics of both models, their predictions and their corresponding noise distributions.

2.1 Odometry Motion Model

Our first motion model is based on sensor odometry information – odometry in this case refers to the explicit pose information provided by the car in the "/vesc/odom" topic. Strictly speaking,

odometry provides us observations $\hat{\mathbf{x}}$ of the robot's pose $\mathbf{x}$ (these observations are located in an arbitrary reference frame based on the robot's initial position, not the map), but we choose to treat it as a motion model rather than a sensor model. Thus our controls for the odometry motion model are a pair of poses measured through odometry: $u_t =< \hat{\mathbf{x}}_t, \hat{\mathbf{x}}_{t-1} >$ where $\hat{\mathbf{x}}_{t-1}$ refers to the previous odometry pose and $\hat{\mathbf{x}}_t$ is the current. Given this pair of poses, we can measure the observed change in pose $\Delta x, \Delta y, \Delta\theta$ through co-ordinate transformations. The odometry motion model applies this observed change in pose to the previous robot's pose $\mathbf{x}_{t-1}$ to get predictions of the next pose $\mathbf{x}_t$:

$$\Delta\mathbf{x} :=< \Delta x, \Delta y, \Delta\theta >= f(u_t) \tag{2}$$

$$\mathbf{x}_t = g(\mathbf{x}_{t-1}, \Delta\mathbf{x}) \tag{3}$$

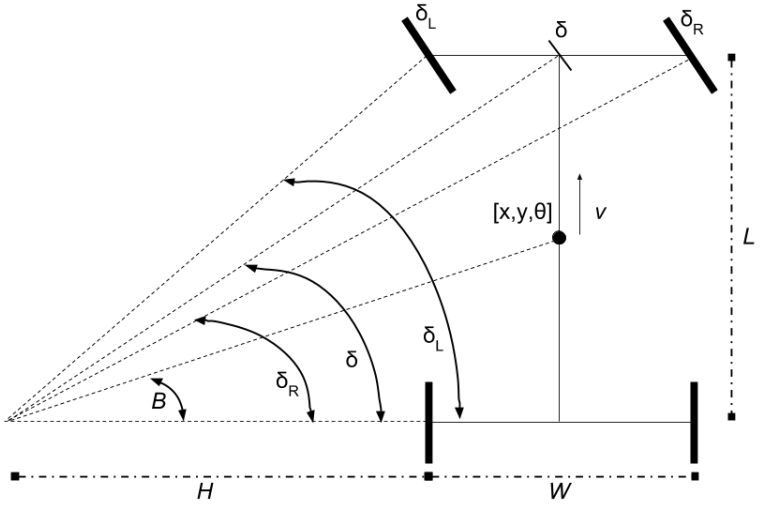
where the function f measures the change in pose from odometry and g applies it to the previously estimated robot pose.

You will be implementing this motion model in the file called **MotionModel.py** which has some skeleton code to get you started.

2.2 Kinematic Model

Instead of relying on the odometry information, we can explicitly derive a motion model based on the kinematic structure of the car. As we discussed in class, our controls for the racecar are the speed of the car v and its steering angle δ, i.e. $u_t =< v_t, \delta_t >$. The kinematic car model explicitly models the effect of this action u_t on the previously estimated robot pose $\mathbf{x}_{t-1}$, predicting a velocity for the 2D position $\Delta x, \Delta y$ and orientation $\Delta\theta$ of the robot which we integrate forward in time to get the future pose $\mathbf{x}_t$. Further details on the derivation of this model are provided below (and in the class notes):

$$\begin{aligned}
\Delta x &= v\cos\theta \\
\Delta y &= v\sin\theta \\
\Delta\theta &= \frac{v}{L}\sin 2\beta \\
\tan\beta &= \tan\delta/2
\end{aligned}$$



$$\begin{aligned}
\tan\delta_r &= \frac{L}{W+H} \\
\tan\delta_l &= \frac{L}{H} \\
\tan\delta &= \frac{L}{H+W/2} \\
\tan\beta &= \frac{L/2}{H+W/2}
\end{aligned}$$

Figure 1: Simplified Kinematic Model of the robot car. State consists of position and heading angle (x, y, θ). Controls consist of velocity and wheel angle (v, δ).

You will also be implementing this motion model in the file called **MotionModel.py** which has some skeleton code to get you started.

2.3 Noise

Both the models described above make deterministic predictions, i.e. given an initial pose $\mathbf{x}_{t-1}$ and control u_t there is a single predicted future pose $\mathbf{x}_t$. This would be acceptable if our model is perfect, but as you have seen we have made many simplistic assumptions to derive this model. In general, it is very hard to model any physical process perfectly. In practice, we would like our model to be robust to potential errors in modeling. We do this by allowing our model to be probabilistic - given any pose $\mathbf{x}_{t-1}$ and control u_t, our model can predict a distribution over future states $\mathbf{x}_t$, some being more likely than others (we can measure this through $p(\mathbf{x}_t|\mathbf{x}_{t-1}, u_t)$. We achieve this by adding noise directly to the model's predictions (odometry motion model) or by adding noise to the controls (kinematic car model) input to the model:

- **Odometry motion model noise**: For the odometry motion model you will add noise to the observed change in pose $\Delta\mathbf{x}$. A simple noise model is to perturb each dimension of the change in pose $< \Delta x, \Delta y, \Delta\theta >$ by independent zero-mean Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ where σ is the standard deviation of the noise (tuned by you). The resulting perturbed change in pose: $(\Delta x + \epsilon_x, \Delta y + \epsilon_y, \Delta\theta + \epsilon_\theta)$ can now be applied to the previous pose $\mathbf{x}_{t-1}$ to get a "noisy" estimate of the predicted next pose $\mathbf{x}_t$.
- **Kinematic car model noise**: For the kinematic motion model, the noise will be added directly to your controls (rather than the predicted change in pose). Once again, our noise can be sampled from an independent zero-mean Gaussian (different per-dimension) $\epsilon \sim \mathcal{N}(0, \sigma^2)$ with a tunable standard deviation σ. The resulting perturbed controls: $(v_t + \epsilon_v, \delta_t + \epsilon_\delta)$ can be used to compute the next pose $\mathbf{x}_t$ based on the equations described in the previous section.

You will implement these noise models in **MotionModel.py**.

2.4 Questions

We will now compare the two different motion models and their corresponding noise models to get a better idea of their behavior:

1. **Visualizing the posterior**: Given a particular pose $\mathbf{x}_{t-1}$ and control u_t, our "noisy" motion model can predict multiple future states $\mathbf{x}_t$ due to the added Gaussian random noise. Visualize this distribution over future states by sampling multiple times (say 1000 times) from our motion model (for a fixed $\mathbf{x}_{t-1}, u_t$) and plotting the x and y values of the samples. Answer the following questions: Do the distributions from the two motion models differ? If so, how? How does the distribution vary when you increase the noise in position (or speed) significantly compared to the orientation (or steering angle) and vice-versa? Show plots for each and include the noise parameters, previous pose $\mathbf{x}_{t-1}$ and control u_t in the caption.
2. **Open loop rollout**: The particle filter (or the general Bayes filter) uses the prediction-correction cycle to recursively estimate state. Assume that you do not have access to any observations – the correction step cannot be applied. What do you expect to see if we run only the prediction step recursively as we drive the robot? Is the resulting state estimate good? Let us try to visualize this – using the two bag files provided (loop.bag and circle.bag), estimate the state using only the motion model and compare to the "ground truth" state estimates provided in the bags. For this test, you can turn off the noise addition - your motion model predictions are deterministic (so you only need a single particle). You can also assume that the initial pose of the robot is provided to you. Plot the resulting state trajectory from this run against the "ground truth" data on the map. What do you see? Is the motion model able to correctly predict the pose of the robot? Justify.
3. **Noise propagation**: In the previous question we looked at the predictions of the motion model without any noise added. Here, we visualize how the motion model propagates noise over time as the robot moves – Given a starting pose for the robot and a sequence of controls (of fixed length, say 20 steps), visualize the distribution over future states (similar to visualizing the posterior) for each of these steps. Note: You will have an initial sample set of 500 particles which you will predict forward in time based on the current control to get future predictions and repeat this in a loop for 20 future steps without resampling. Again, you have one initialization followed by multiple predictions without resampling. Choose

a sequence of 20 timesteps from the two bag files provided (loop.bag and circle.bag) and generate plots of the future particle distributions using both the motion models. How does the noise propagate? Do you see any explicit difference between the two motion models? Explain.

HINT: The parameters (σ) for the Gaussian noise terms need to be chosen to best capture the variability of the model acting in the real world. You can also make these noise parameters dependent on the control parameters (for example, it might be physically realistic to say that there is more uncertainty in your predictions if you move faster).

HINT: Figures 5.3, 5.10, and 8.10, and their respective sections from the textbook *Probabilistic Robotics*, may be useful to understanding the effects of motion models and noise.

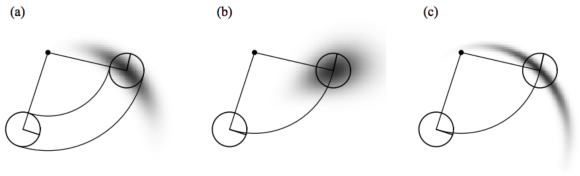


Figure 5.3 The velocity motion model, for different noise parameter settings.

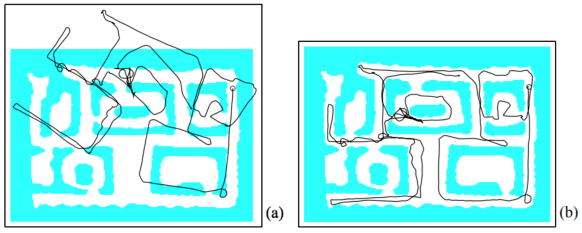


Figure 8.10 (a) Odometry information and (b) corrected path of the robot.

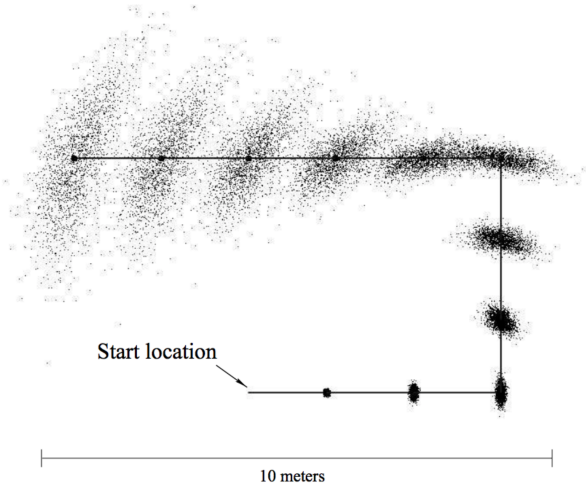


Figure 5.10 Sampling approximation of the position belief for a non-sensing robot. The solid line displays the actions, and the samples represent the robot's belief at different points in time.

2.5 Submission

Submit **MotionModel.py**, your plots for the noise/motion model analyses along with explanations for the different behaviors observed in the plots. In addition, answer the following questions:

1. In your opinion, which motion model works better with the Particle filter? What is the benefit of the odometry model versus kinematic model?
2. What do you think are the weaknesses of these models, and what improvements would you suggest? If you were to build your own model, how would you do it?
3. Include your answers to the questions asked in section 2.4.

3 Sensor Model

The sensor model captures the probability $p(\mathbf{z}_t|\mathbf{x}_t)$ that a certain observation $\mathbf{z}_t$ can be obtained from a given robot pose $\mathbf{x}_t$ (assuming that the map of the environment is known). In this assignment, $\mathbf{z}_t$ is the LIDAR data from the laser sensor mounted near the front of the robot. Your task in this section is to implement and analyze the LIDAR sensor model discussed in class.

3.1 Model

The LIDAR sensor shoots out rays into the environment at fixed angular intervals and returns the measured distance along these rays (or nan for an unsuccesful measurement). Therefore, a single

LIDAR scan z has a vector of distances along these different rays $\mathbf{z} = \left[z^1, z^2, ..., z^N\right]$. Given the map of the environment (m), these rays are conditionally independent of each other, so we can rewrite the sensor model likelihood as follows:

$$p(\mathbf{z}_t|\mathbf{x}_t, m) = p(z^1, z^2, ..., z^N|\mathbf{x}_t, m) = p(z^1|\mathbf{x}_t, m) * p(z^2|\mathbf{x}_t, m) * ... * p(z^N|\mathbf{x}_m,) \quad (4)$$

where the map m is fixed (we will not use m explicitly in any further equations, it is assumed to be provided). As we can see, to evaluate the likelihood it is sufficient if we have a model for measuring the probability of a single range/bearing measurement (each ray has a bearing and range) given a pose $p(z^i|\mathbf{x}_t)$.

We will compute this in the following manner: First, we will generate a simulated observation $\hat{\mathbf{z}}_t$ given that the robot is at a certain pose $\mathbf{x}_t$ in the map. We do this by casting rays into the map from the robot's pose and measuring the distance to any obstacle/wall the ray encounters. This is very much akin to how laser light is emitted from the LIDAR itself. We then quantify how close each ray in this simulated observation $\hat{\mathbf{z}}_t^i$ is close to the real observed data $\mathbf{z}_t^i$ providing us an estimate of $p(z^i|\mathbf{x}_t)$. Overall, this requires two things: 1) A way to compute the simulated observation given a pose and 2) A model that measures closeness between the simulated and real data. For the purpose of this assignment, you are provided with a fast, optimized method (implemented in C++/CUDA) to compute the simulated observation - you have access to a ray casting library called rangelibc. The second part is what you will be implementing - a model that measures how likely you are to see a real observation $\mathbf{z}_t^i$ given that you are expected to see a simulated observation $\hat{\mathbf{z}}_t^i$. This LIDAR sensor model we use is a combination of the following four curves, with certain mixing parameters and hyper-parmeters (tuned by you) that weigh each of the four components against the other. More details on this model can be found in Section 6.3.1 of *Probabilistic Robotics* as well as in the lecture notes.

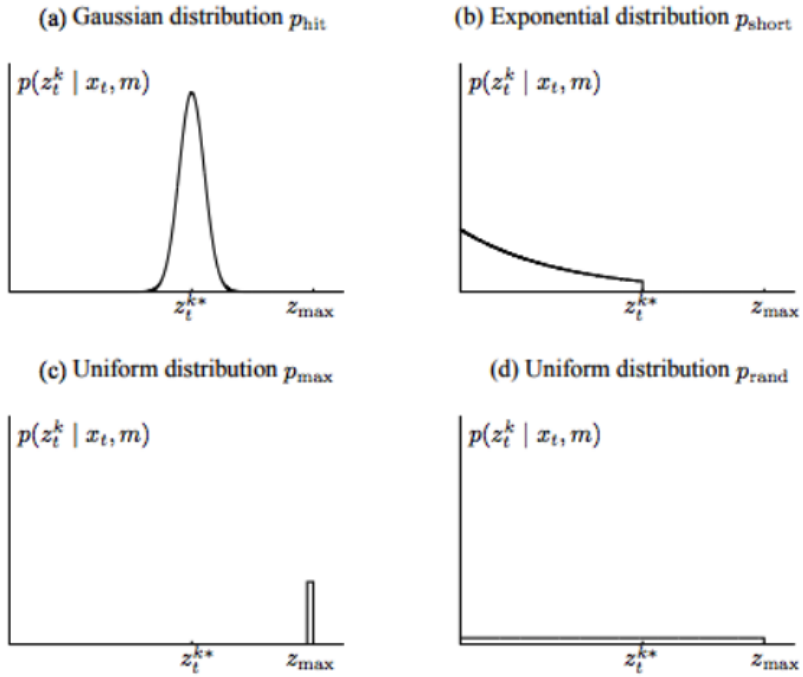


Figure 6.2 Components of the range finder sensor model. In each diagram the horizontal axis corresponds to the measurement z_t^k, the vertical to the likelihood.

Given that the LIDAR range values are discrete (continuous values are converted to discrete pixel distances on the 2D map), we do not have to generate the model on-line and can instead pre-calculate them. This will be represented as a table of values, where each row is the actual measured value and the column is the expected value for a given LIDAR range. Pre-computing this table allows for faster processing during runtime. During run-time, we use the aforementioned ray casting library to generate simulated range measurements. This is then used in your calculated look-up table, and compared to the LIDAR output.

For this part of the assignment, you will implement the sensor model in the file **SensorModel.py**. In particular, the function `precompute_sensor_model()` should return a `numpy` array containing a tabular representation of the sensor model. The code provided will use your array as a look-up table to quickly compare new sensor values to expected sensor values, returning the final sensor model likelihood: $p(\mathbf{z}_t|\mathbf{x}_t)$.

3.2 Particle Likelihood

For a given LIDAR scan, there are potentially many locations in a map where that data could come from. In this part of the assignment, you are going to measure the probability that a given LIDAR scan came from any position on the map. You will **plot** the likelihood map of the provided LIDAR scans (in the `bags/laser_scans` directory of the skeleton code) within the map of the 3rd floor of Sieg Hall. You can do this by discretizing the map (this discretization can be as simple as iterating through all possible x, y values for each point in the map, and then iterating through all θ values the car could be in at that point) and evaluating the implemented sensor model at each robot pose $\mathbf{x}_t$ to compute $p(\mathbf{z}_t|\mathbf{x}_t)$. Finally you can assign per 2D positon (x, y) a value that corresponds to the orientation with the maximum probability at that point (for that x, y find the maximum over all possible orientations). The resulting plot (normalized) should be a heatmap of likelihood values; regions of higher likelihood should be a lighter color than regions of low likelihood. For each of the three given scans, provide this heatmap.

Of course, too fine-grain of a discretization can take a lot of time to process. Therefore, you will also **plot** a line comparing the resolution of your discretization against computation time. The amount of discretization is akin to number of particles, but with known, fixed spaces between the x, y, θ values.

Figure 6.7 from *Probabilistic Robotics* is a good reference.

3.3 Submission

Turn in **SensorModel.py**, the likelihood and processing time plots along with explanations for the behaviors observed in the plots. Also, answer the following questions:

1. How would you incorporate additional sensors into a sensor model? Describe how you would use the IMU in this way.
2. For localizing a robotic car, what additional sensors would be useful? For a robotic hand, what additional sensors would be useful for estimation?

4 Particle Filter

The bayes filter consists of a motion model that predicts the movement of the robot, and compares the expected sensor values after this movement to the sensor values collected from the real robot. In the particle filter, your belief over the robot pose $Bel(\mathbf{x}_t)$ is represented as a list of M particles (these are nothing but samples from the belief distribution). Each particle i has a pose $\mathbf{x}_t^i$ which represents a hypothetical robot at that particular pose; using many particles allows for us to more accurately represent the belief distribution.

4.1 Implementation

In this part of the assignment, you will be implementing a particle filter to track our belief over the robot's pose over time with a fixed number of particles. The code from the prior sections will be referenced to in the file **ParticleFilter.py**.

From a high level, this is how the particle filter works. Initially, we have a prior belief over where the robot is: $Bel(\mathbf{x}_0)$. You are provided with a clicking interface that enables you to specify the initial pose of the robot - this will specify $Bel(\mathbf{x}_0)$. We will represent this belief using a fixed number of M particles - initially all these particles will be at the clicked pose. In order to specify a pose, look for a button labeled '2D Pose Estimate' along the top bar of the RVIZ interface. After clicking this button, you can specify a position and orientation by clicking and dragging on the map. Next, each particle will be propagated forward in time using the motion model (provided the controls). Using ray-casting on the car's GPU, we will generate a simulated LIDAR observation for each particle

which we will compare to the real LIDAR data from the laser sensor. This now assigns a "weight" for each particle - particles with higher weights are more likely given the motion and sensor model updates. Finally, we will re-sample from this distribution to update our belief over the robot's pose $Bel(x_t)$ - this resampling step gets rid of particles with low belief and concentrates our distribution over particles with higher belief. We repeat these steps recursively.

In prior sections, you have already implemented algorithms for sampling from two motion models (with noise) and a sensor model. The key step remaining is to implement the re-sampling step (discussed in the next section) and to connect all the parts together in a tight loop that should run in real-time. The particle filter does involve multiple hyper-parameter settings that you will have to tune for good performance, such as the number of particles to use, the amount to sub-sample the LIDAR data, when to resample, etc. More details can be found in the provided skeleton code.

4.2 Re-sampling

You will implement two re-sampling procedures for your particle filter in **ReSample.py**. First is the naive re-sampler: we draw a random number per particle and choose a specific particle based on the value of this random number (and the particle's weight). (Hint: `np.random.choice` is a good place to start.)

The second re-sampler is a low-variance re-sampler, which only draws a single random number. This is detailed in table 4.4 from *Probabilistic Robotics* and implemented as follows:

1: **Algorithm Low_variance_sampler($\mathcal{X}_t, \mathcal{W}_t$):**
2: $\bar{\mathcal{X}}_t = \emptyset$
3: $r = \text{rand}(0; M^{-1})$
4: $c = w_t^{[1]}$
5: $i = 1$
6: for $m = 1$ to M do
7: $u = r + (m-1) \cdot M^{-1}$
8: while $u > c$
9: $i = i + 1$
10: $c = c + w_t^{[i]}$
11: endwhile
12: add $x_t^{[i]}$ to $\bar{\mathcal{X}}_t$
13: endfor
14: return $\bar{\mathcal{X}}_t$

Table 4.4 Low variance resampling for the particle filter. This routine uses a single random number to sample from the particle set $\mathcal{X}$ with associated weights $\mathcal{W}$, yet the probability of a particle to be resampled is still proportional to its weight. Furthermore, the sampler is efficient: Sampling M particles requires $O(M)$ time.

4.3 Coding and Debugging

Provided in the `bags` directory of the skeleton code are bag files of collected data – LIDAR values, and reference car locations based on the TA implemented particle filter. We strongly advise you to test on this data to get your implementation started - while it is not necessary (and probably not possible due to randomization) to match the TA results you should see similar tracking performance (without losing track) when running the particle filter. A key point to note is that your code will ultimately be run on the robot for the demo so you should make the code as efficient as possible so that it can run in real-time.

4.4 Extra Credit 1

Implement global localization at the start of the particle filter. So far, you've assumed access to a clicking interface which lets you manually initialize the robot's pose. By implementing an algorithm for global localization, you can automate this process. There are multiple ways to do this: a naive way would be to increase the number of particles significantly until you are able to localize well globally right at the start. A better way would be to first initialize a large number of particles, weight these particles by the sensor model and sample a fixed number of M particles from this weighted distribution. Can you come up with better ways to solve this problem? For this extra credit question, you will have to implement an algorithm that reliably localizes the robot at the start of the particle filter run without having to explicitly providing an initialization via clicking. Note that it does not have to lock on to the exact location right at the start as long as it is able to refine it's belief estimate and converge to the correct distribution as the robot moves. The TA's will test this by having you re-initialize your particle filter in a new location without human input.

4.5 Extra credit 2

Solve the kidnapped robot problem. A harder version of the global localization problem is called the kidnapped robot problem - say that your robot has been tracking the pose quite well, but suddenly a malicious agent picks the robot up while covering the LIDAR and moves it to a different location within the same map. Can your particle filter recover and correctly re-estimate this new position that the robot is in? Unlike the initial global localization problem, you now have an additional problem to solve - you have to figure out if you have been kidnapped and need to re-initialize your filter. Propose and implement an algorithm for solving this problem - the TAs will test this by moving your robot to a new location in the middle of tracking while covering the LIDAR.

4.6 Submission

Submit your **ParticleFilter.py** code, generated plots and the answers to the following questions:

1. What is the variance of the two different re-sampling methods? You can compare this by measuring the variance of the particle draws - for a fixed distribution of particle weights by how much do the identities of the final sampled particles vary? You will generate a plot of this variance measure as the number of particles increases in fixed steps, say for five distinct values of M (100, 200, 500, 1000, 4000).
2. When calculating a single x, y, θ estimate of the car from the belief distribution of the particle filter, you can take the mean or the maximum particle; describe the situations in which the mean or the max particle would calculate a bad estimate. Is there another estimate that is more robust?
3. What are the effects of the number of particles on the performance of the system? Show a plot of the difference between your estimate and the provided data; the x axis is time, the y axis is the root mean squared error of the difference. Also plot the computation time per iteration of the filter as the number of particles increases. Do this for 200, 800, 1600, 4000, 8000 particles. What number achieves the best trade-off between speed and accuracy?

5 Demo

In your team's demo to the TA's, you will need to:

1. Show your particle filter working on the bag file provided for debugging.
2. Show the particle filter working on the robot car. The TAs will then drive your car around (possibly at 2x the default speed), and your particle filter should track the real world location of the car.

Additionally, the TAs will quantify each member's contribution to the final result with a quick QA right after your demo.