Udacity Self Driving Cars Nanodegree- Kidnapped Vehicle project

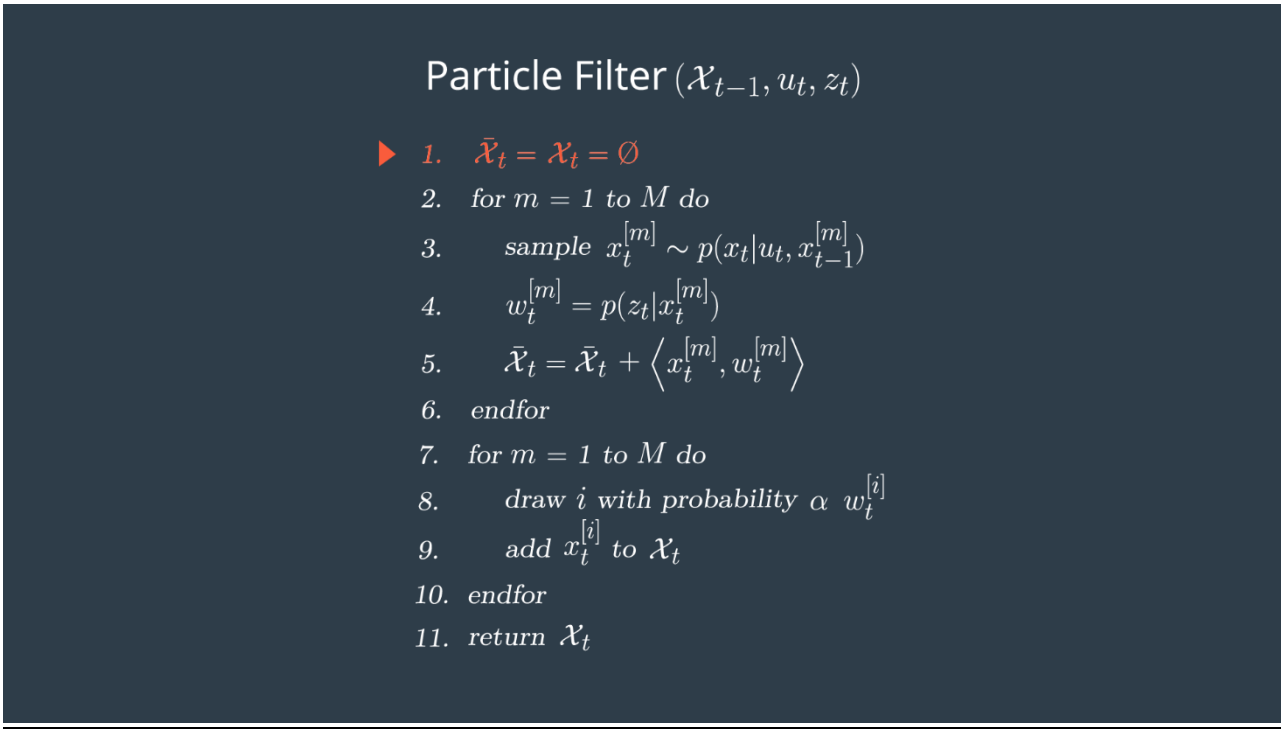
Objective:

The objective of this project was to localize a car thru an implementation of a 2-D particle filter in C++. The particle filter was given a map and some initial localization information (analogous to what a GPS would provide). At each time step, the particle filter also got observation and control data.

Approach:

Pseudo code:

The following image shows the Pseudo code for the project:

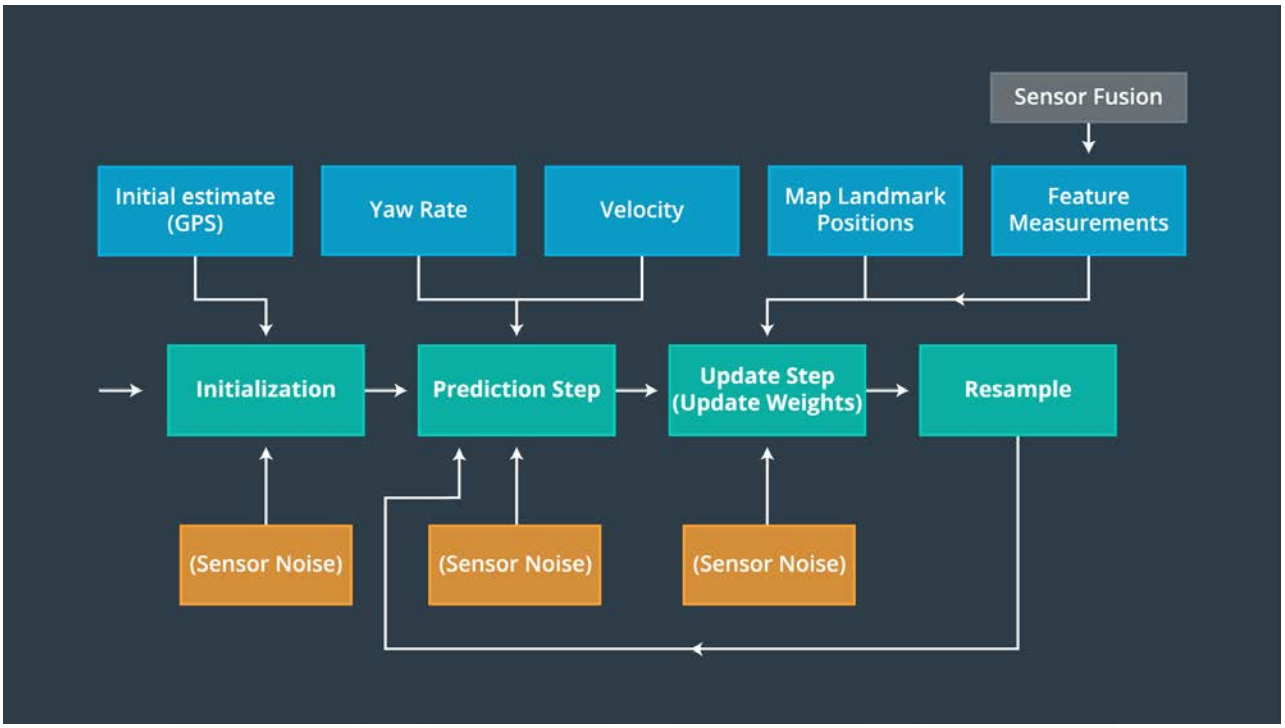


Steps:

1. Initialization for the filter
2. Prediction for the filter
3. Data Association
4. Update each particle's weight
5. Resampling

Once the resampling is done, the new set of particles represent the Bayes' filter posterior.

Flowchart for the particle filter algorithm:



For the project; we had to write the C++ code in the particle_filter.cpp file for these steps. This required implementing the functions for the particle filter class. Main.cpp, GPS data (and the supporting header files were provided. Following are the details of the implementation steps:

Initialization:

We had to decide how many particles we want to use (I chose 100). If we have too few particles, we may not have enough particles to cover all of the likely positions of the car; so we may miss the correct position. If we have too many particles, it will slow down the filter and prevent it from localizing the self-driving car in real time.

For a self-driving car, the sample of particles is done around the initial estimate provided by GPS. For the project, the particles were initialized by sampling from a Gaussian distribution; taking into account Gaussian sensor noise around the initial GPS position and heading estimates. In C++, we use the standard library normal_distribution and default_random_engine to sample positions around the GPS measurement. Each particles' weight is initialized to 1.

Prediction:

We have to predict where the car will be at the next time step. For each particle, we have to update the particle's location based on velocity and yaw rate measurements, while accounting for Gaussian sensor noise.

The following are the equations for updating x, y and the yaw angle when the yaw rate is not equal to zero:

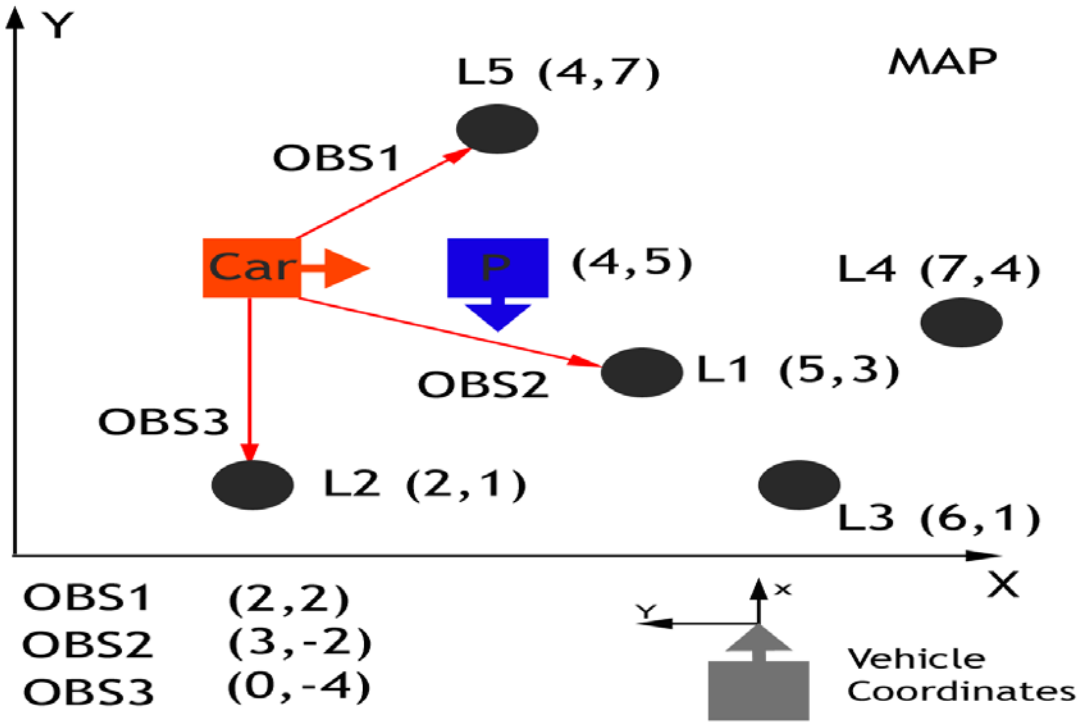
$$x_f = x_0 + \frac{v}{\dot{\theta}}[sin(\theta_0 + \dot{\theta}(dt)) - sin(\theta_0)]$$

$$y_f = y_0 + \frac{v}{\dot{\theta}}[cos(\theta_0) - cos(\theta_0 + \dot{\theta}(dt))]$$

$$\theta_f = \theta_0 + \dot{\theta}(dt)$$

Data Association:

This is the process of matching Lidar measurements to the map landmarks using the Nearest Neighbor technique. To show how it is done, we will take an example in the following image:



In the picture above, we have a car (**ground truth position**) that observes (within the sensor range) three nearby landmarks, labeled OBS1, OBS2, and OBS3; respectively. Each observation measurement has x, and y values in the car's coordinate system. We have a particle "P" (**estimated position of the car**) above with position (4, 5) on the map with heading -90 degrees.

The first task is to transform each observation marker from the vehicle's coordinates to the map's coordinates, with respect to our particle. Observations in the car coordinate system can be transformed into map coordinates (xm and ym) by passing car observation coordinates (xc and yc), map particle coordinates xp and yp), and rotation angle (-90 degrees) through a homogenous transformation matrix. This homogenous transformation matrix, shown below, performs rotation and translation.

$$\begin{bmatrix} x_m \\ y_m \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & x_p \\ \sin\theta & \cos\theta & y_p \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix}$$

Which results in the following equations:

$$x_m = x_p + (\cos\theta \times x_c) - (\sin\theta \times y_c)$$

$$y_m = y_p + (\sin\theta \times x_c) + (\cos\theta \times y_c)$$

Let us say, the transformed observations are TOBS1, TOBS2 and TOBS3 for observations OBS1, OBS2 and OBS3 respectively. Using the above equations; values for TOBS1, TOBS2 and TOBS3 will be (6, 3), (2, 2) and (0, 5) respectively.

The next step is to associate each transformed observation with a land mark identifier. To do this we must associate the closest landmark to each transformed observation. In the above example image, TOBS1, TOBS2 and TOBS3 will be associated with landmarks L1,L2 and L2/L5 respectively.

Update weights:

The particles final weight will be calculated as the product of each measurement's Multivariate-Gaussian probability density; given by the following equation.

$$P(x,y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\left(\frac{(x-\mu_x)^2}{2\sigma_x^2} + \frac{(y-\mu_y)^2}{2\sigma_y^2}\right)}$$

In the above equation; x and y are the observations in map coordinates and mu(x) and mu(y) are the

coordinates of the nearest landmarks.

Weights need to be normalized so that their values are between 0 and 1. These weights are used as probabilities in the resampling function.

Resampling:

The weights of the particle and the C++ standard library discrete_distribution function is used to update the particles to the Bayesian posterior distribution. The discrete_distribution works roughly in the following manner: Particle with the higher weights (and copies of these particles) are more likely to be sampled in proportion of their weights and particles with the lower weights are more likely to be discarded.

Results:

Evaluation of the particle filter is done by using the weighted mean error function using the ground truth position of the car and the weights of the particles as inputs; at each time step.

In the image and video below, the blue circle is the best particle (i.e. the particle with the highest weight) that we have. The blue lines are lasers from the best particle to the landmarks; the green lines are lasers from the car to the landmarks; and since they pretty much overlap, that means the particle filter is working well in localizing the car (estimating the position of the car). Note: The blue lines are visible in the image from the video snapshot from the course; but they are not visible in the video below from my program run.