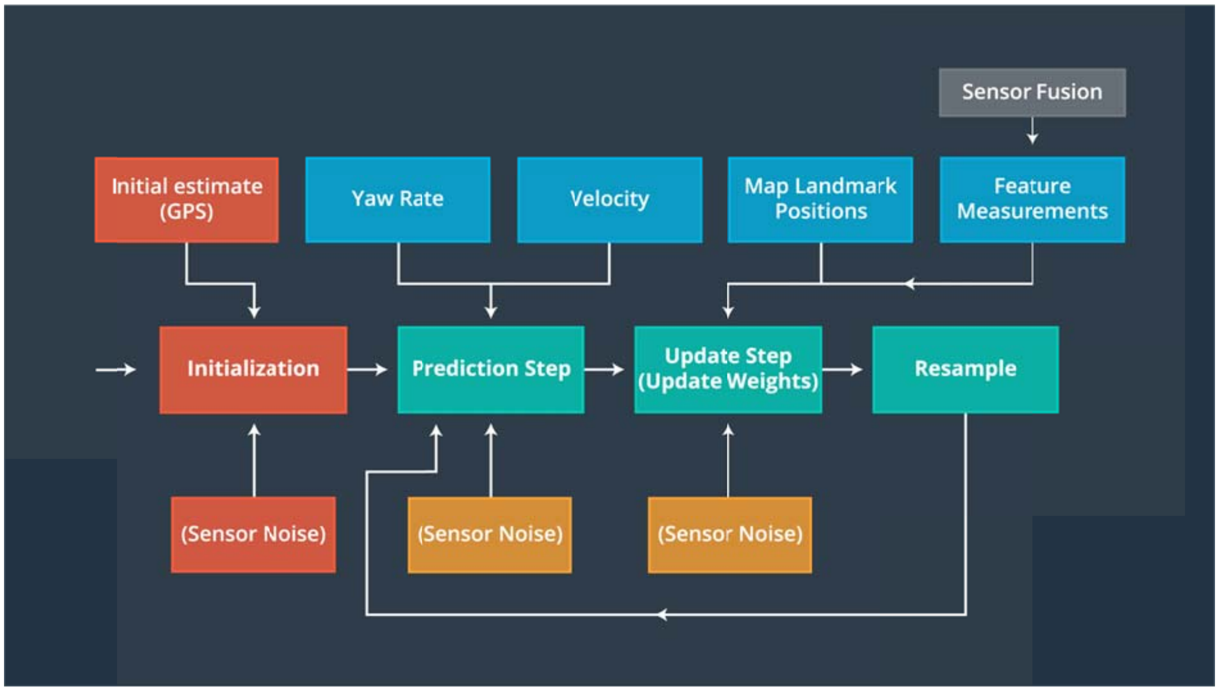
Specific explanations and Results:

In this project, the structure of the coding follows the below block diagram from courses to implement a 2D particle filter by using C++:

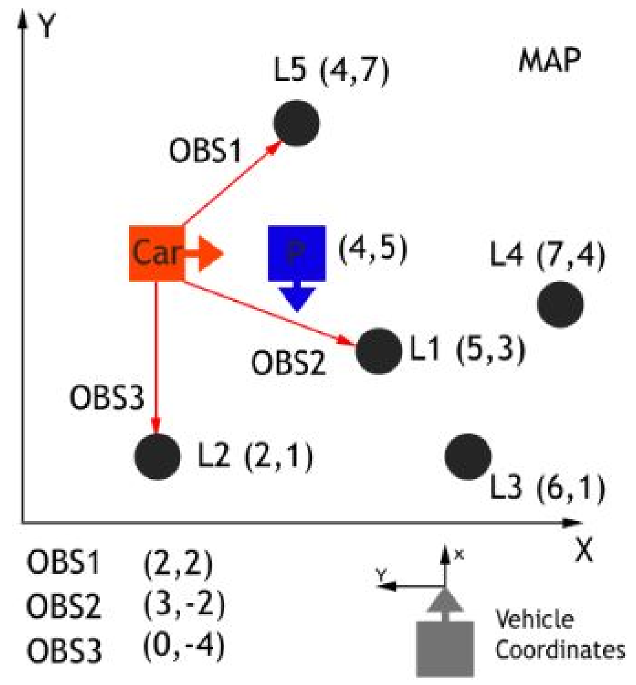


Initial GPS information and map is given to particle filter. However GPs data is not accurate enough and can't meet required accuracy for a self-driving car (3-10cm, that's why particle filter is necessary). Then filter will receive control data and observation data from sensors in different time steps.

The sensor information are received in vehicle coordination system (vehicle direction), hence a transformation is required to map the observations into global system (Homogenous Coordinate Transformation, lesson 5 section 17).

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

Refer to below image (Lesson 5 section 14, and section 17) from lessons for more insight (L*: landmarks, OBS*: observations, Car: ground truth location of the vehicle, P: predicted location)



To calculate particle final weights, learnings from lesson 5 section 19 are used. The particles final weight will be calculated as the product of each measurement's Multivariate-Gaussian probability density.

The Multivariate-Gaussian probability density has two dimensions, x and y. The mean of the Multivariate-Gaussian is the measurement's associated landmark position and the Multivariate-Gaussian's standard deviation is described by our initial uncertainty in the x and y ranges. The Multivariate-Gaussian is evaluated at the point of the transformed measurement's position. The formula for the Multivariate-Gaussian can be seen below. x and y is the observation and μx and μy are the coordinates of associated landmark. The final weight of particle is product of all probabilities:

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

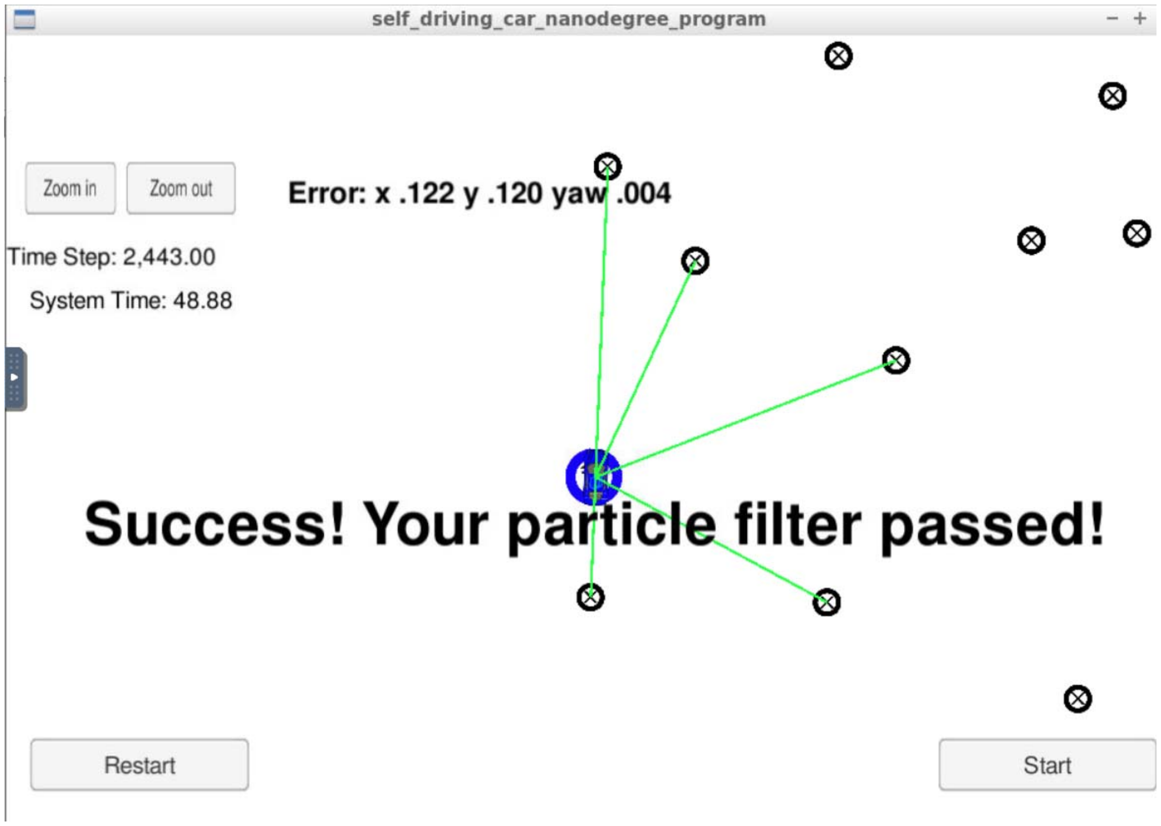
Resampling is done to filter best accurate particles with higher weights t ogive best predictions of vehicle position. Then this location is measured against ground truth to calculate errors.

The results of simulation are shown in below table and image (test run for 40 particles). Errors basically show the difference between predicted location and ground truth location.

RMSE for x: 0.122

RMSE for y: 0.120

RMSE for yaw: 0.004



Error: x .122 y .120 yaw .004

Time Step: 2,443.00

System Time: 48.88

Success! Your particle filter passed!

Other trials also were done for 20 and 100 number of particles, all were satisfying the results.

Overview

This repository contains all the code needed to complete the final project for the Localization course in Udacity's Self-Driving Car Nanodegree.

Submission

All you will need to submit is your `src` directory. You should probably do a `git pull` before submitting to verify that your project passes the most up-to-date version of the grading code (there are some parameters in `src/main.cpp` which govern the requirements on accuracy and run time).

Project Introduction

Your robot has been kidnapped and transported to a new location! Luckily it has a map of this location, a (noisy) GPS estimate of its initial location, and lots of (noisy) sensor and control data.

In this project you will implement a 2 dimensional particle filter in C++. Your particle filter will be given a map and some initial localization information (analogous to what a GPS would provide). At each time step your filter will also get observation and control data.

Running the Code

This project involves the Term 2 Simulator which can be downloaded here

This repository includes two files that can be used to set up and install uWebSocketIO for either Linux or Mac systems. For windows you can use either Docker, VMware, or even Windows 10 Bash on Ubuntu to install uWebSocketIO.

Once the install for uWebSocketIO is complete, the main program can be built and ran by doing the following from the project top directory.

1. mkdir build
2. cd build
3. cmake ..
4. make
5. ./particle_filter

Alternatively some scripts have been included to streamline this process, these can be leveraged by executing the following in the top directory of the project:

1. ./clean.sh
2. ./build.sh
3. ./run.sh

Tips for setting up your environment can be found here

Note that the programs that need to be written to accomplish the project are src/particle_filter.cpp, and particle_filter.h

The program main.cpp has already been filled out, but feel free to modify it.

Here is the main protocol that main.cpp uses for uWebSocketIO in communicating with the simulator.

INPUT: values provided by the simulator to the c++ program

// sense noisy position data from the simulator

["sense_x"]

["sense_y"]

["sense_theta"]

// get the previous velocity and yaw rate to predict the particle's transitioned state

["previous_velocity"]

["previous_yawrate"]

// receive noisy observation data from the simulator, in a respective list of x/y values

["sense_observations_x"]

["sense_observations_y"]

OUTPUT: values provided by the c++ program to the simulator

// best particle values used for calculating the error evaluation

["best_particle_x"]

["best_particle_y"]

["best_particle_theta"]

//Optional message data used for debugging particle's sensing and associations

// for respective (x,y) sensed positions ID label

["best_particle_associations"]

// for respective (x,y) sensed positions

["best_particle_sense_x"] <= list of sensed x positions

["best_particle_sense_y"] <= list of sensed y positions

Your job is to build out the methods in `particle_filter.cpp` until the simulator output says:

```
Success! Your particle filter passed!
```

Implementing the Particle Filter

The directory structure of this repository is as follows:

```
root
|   build.sh
|   clean.sh
|   CMakeLists.txt
|   README.md
|   run.sh
|
|___data
|   |
|   |   map_data.txt
|
|
|___src
    |   helper_functions.h
    |   main.cpp
    |   map.h
    |   particle_filter.cpp
    |   particle_filter.h
```

The only file you should modify is `particle_filter.cpp` in the `src` directory. The file contains the scaffolding of a `ParticleFilter` class and some associated methods. Read through the code, the comments, and the header file `particle_filter.h` to get a sense for what this code is expected to do.

If you are interested, take a look at `src/main.cpp` as well. This file contains the code that will actually be running your particle filter and calling the associated methods.

Inputs to the Particle Filter

You can find the inputs to the particle filter in the `data` directory.

The Map*

`map_data.txt` includes the position of landmarks (in meters) on an arbitrary Cartesian coordinate system. Each row has three columns

1. x position
2. y position
3. landmark id

All other data the simulator provides, such as observations and controls.

- Map data provided by 3D Mapping Solutions GmbH.

Success Criteria

If your particle filter passes the current grading code in the simulator (you can make sure you have the current version at any time by doing a `git pull`), then you should pass!
The things the grading code is looking for are:

1. **Accuracy**: your particle filter should localize vehicle position and yaw to within the values specified in the parameters `max_translation_error` and `max_yaw_error` in `src/main.cpp`.

2. **Performance**: your particle filter should complete execution within the time of 100 seconds.

How to write a README

A well written README file can enhance your project and portfolio. Develop your abilities to create professional README files by completing this free course.