3.PARTICLE FILTER.

In the third exercise, of the udacity self driving engineering 2nd term, we're asked to implement a 2 dimension particle filter, which allows a kidnapped car to localize itself, under a time constraint of 100 seconds. The fundamental part of the exercise resides in the correct implementation of the particle_filter, which we are given as blueprint.

We will then analyze the main completed code of this exercise, particle_filter.cpp.

The first parameter we choose is the number of particles we will be using. By trial and error we have found that a range between 800 and 1200 is adequate. Having more or less makes the filter fail, when running the tests.

The first function we use is init. We use a gaussian distribution to initialize particle.x, particle.y, and particle.theta. Each particle in this filter, has a 2d position, made of x,y, and angle(theta).

The second function we have to implement is prediction, which contains measurements for each particle and added noise. We run a for loop for all particles(auto &p: particles), and we consider two cases. If the particle has a yaw angle, we will have to consider it in the equations we are going to use. Each point is computed taking into account, the yaw_angle, the current angle, and delta t. We create two variables, velocity_by_yaw_rate which is the velocity divided by the speed and next_theta, which gives us next angle. Equations are explained in the comments in the function. If angle is close to 0, p.x and p.y are just velocity*by time change *by cosine time or sine time. Finally, we add noise, using white noise with a gaussian distribution, dist_x, dist_y, dist_y.

The third function we use is the dataAssociation one. We find the predicted measurement that is closest to each observed measurement and assign the observed measurement to this particular landmark. For each of the predicted Landmark observations, we calculate the distance from the landmark to the code using this function:

double distance = dist(obs.x, obs.y, pred.x, pred.y);

Now if the distance is less than than the shortest we assign to the shortest that distance. Therefore that is the closest prediction and we assign pred to closest. Finally we do this for all predictions, as we are in a for loop. Finally we add all landmarks using:

associated_landmarks.push_back(closest);

Next we will review the updateWeights function. In this function we update the weights using a multi-variant gaussian distribution. What we do is first we iterate over all the particles using a for loop, and for all the recorded observations, we transform them. Using the following formula:

transformed_observation.x = p.x + observation.x * cos(p.theta) - observation.y * sin(p.theta);
transformed_observation.y = p.y + observation.x * sin(p.theta) + observation.y * cos(p.theta);

Now we get all the landmarks associated with a given particle, and if the distance to the landmark is less than the sensor_range, this is the landmark we will choose. We will assign it to the predicted landmark with the following statement.

predicted.push_back(one_landmark);

Now finally we will associate the nearest landmark to each observation of the particles. First we create a vector which associates the closest landmark to the transformed observations.

vector associated_landmarks = dataAssociation(predicted, transformed_observations);

For all associated landmarks we calculate the probability and add to the weight of the particle as follows. The differential distance in x, corresponds to the difference between the x point in the transformed observations and that of the landmark. The y differential point corresponds to the difference between the y point and the associated landmark. The probability calculation is done as we have seen in the lessons. Finally for each particle we add the weight using :
p.weight = probability;

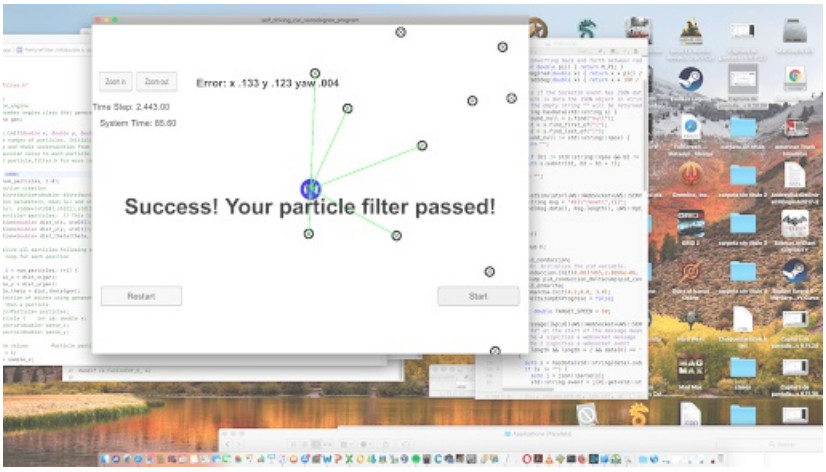
And to the weight vector we add the probability of each weight.
weights[i] = probability;

The five last functions are helper functions: resample, associations, getSenseX, getSenseY, and setAssociations.

It is recommended or suggested that if all the functions in the particle filter are correctly implemented we will not need to verify the code in main.cpp. So we will follow this procedure and see how our results perform.

PERFORMANCE.

With the number of particles set to 1000, we have obtained several results as shown:



We see than it is very close to the limit of 100 seconds.Another iteration and we get the following result