carnd_t2_p3_kidnapped_vehicle

Carnd - term 2 - project 3 - kidnapped vehicle

Overview

The goal of this project is to build an Paticle Filter (PF) model to process a series of sensor coordinates (x,y,theta) to exercise the localization of vehicle with respect to global position system(GPS) coordinates. The PF model predicts the vehicle location base on the random generated particles which update their weights by finding their shortest distances from each landmarks in the global map coordinates. Similar to Porject 1 and 2, the PF model is connected to a simulator via uWebSocketIO with predefined landmarks coordinates.

Project Repository

All resource are located in Udacity's project repository CarND-Kidnapped-Vehicle-Project (https://github.com/udacity/CarND-Kidnapped-Vehicle-Project)

Project Submission

All modified code including results are committed to my personal github page carnd_t2_p3_kidnapped_vehicle (https://github.com/chriskcheung/carnd_t2_p3_kidnapped_vehicle)

Key Files

main.cpp

establishes communication between simulator and PF model using uWebSocketIO, and reads in data during set time interval and send sensor measurements to ParticleFilter::init(), ParticleFilter::prediction(), ParticleFilter::updateWeights(), and ParticleFilter::resample() in particle_filter.cpp for processing

ukf.cpp

contains 4 main functions: ParticleFilter::init(), ParticleFilter::prediction(), ParticleFilter::updateWeights(), ParticleFilter::resample().

ParticleFilter::init() generates a number (_num*particles*) of particles, then use normal_guassian distribution algorithm to create x,y coordinates offseting from the provided GPS coordinates of the vehicle. Each particles will have a slightly different x,y coordinates to each others. All particles's weight are initialized to 1 at the beginning.

ParticleFilter::prediction() is similar to ParticleFilter::init() that it uses normal_guassian distribution algorithm to update all particles's coordinate. Instead of using GPS coordinates of the vehicle, it uses sensor provided data like, vehicle velocity, yaw rate, sensor error deviation, and delta time of each measurements, to calculate the new vehicle location at delta time using trigonometry equations. Once new vehicle location is calculated, normal_guassian distribution is used to introduce new noise to each particles's coordinates as prediction.

ParticleFilter::updateWeights() is split into 4 steps: 1) update the list of landmarks each particles should include for calculating particle weights, 2) transform the vehicle observations from vehicle's prespective coordinates to global positioning coordinates, 3) pick the closest landmark to each transformed observation coordinates by finding the shortest distance in between them, 4) apply Multivariate-Gaussian's standard deviation algorithm to transformed observations coordinates and their closest landmark coordinates to calculate their new weights, which is in term the probability of the particle to be picked for determining the vehicle's whereabout in the global position coordinate, 5) finally normalize all particle weights.

ParticleFilter::resample() uses discrete distribution algorithm to pick/sample the particles so that particles with the higher weights/probability will be more likely to be picked/saved as a new set of pacticles for representing the prediction of the vehicle's local position.

side note

Due to the predefined input parameter of dataAssociation() is not flexible to use, I decided to handle data association in updateWeight() instead of using dataAssociation() in my implemenation.

Implementation Challenge

Initialization

Applies normal distribution algorithm directly to GPS x,y coordinates and angle theta in radian directly and use for loop to create new particles with different offset within the standard deviation of which x,y,theta.

```
    void ParticleFilter::init(double x, double y, double theta, double std
[]) {
        ...
        default_random_engine gen;
        normal_distribution<double> dist_x(x, std[0]);
        normal_distribution<double> dist_y(y, std[1]);
        normal_distribution<double> dist_theta(theta, std[2]);
        ...
```

Prediction

First, new x,y,theta are calculating from the vehicle velocity and yawrate.

```
        if (yaw_rate != 0.0f){
            x = x + velocity/yaw_rate*( sin(theta + yaw_rate*delta_t) - sin(
theta));
            y = y + velocity/yaw_rate*(-cos(theta + yaw_rate*delta_t) + cos(
theta));
            theta = theta + yaw_rate*delta_t;
        }
        else {
            x = x + velocity*cos(theta)*delta_t;
            y = y + velocity*sin(theta)*delta_t;
            theta = theta;
        }
```

Prediction is a bit different than initialization which it comes to using normal distribution algorithm. The algorithm is used to generate noise to the newly predicted x,y, and theta.

```
        default_random_engine gen;
        normal_distribution<double> dist_x(0, std_pos[0]);
        normal_distribution<double> dist_y(0, std_pos[1]);
        normal_distribution<double> dist_theta(0, std_pos[2]);
        ...
            particles[i].x = x +dist_x(gen);
            particles[i].y = y+dist_y(gen);
            particles[i].theta = theta+dist_theta(gen);
```

Update Weight

Since the sensor range is provided, there is no point to process landmarks that are outside or the sensor range. Therefore, reducing the landmarks size is recommended and it helps to improve the filter performance.

```
        for (int j=0; j<map_landmarks.landmark_list.size(); j++){
            if (dist(particles[i].x, particles[i].y, map_landmarks.landmark_list
    [j].x_f, map_landmarks.landmark_list[j].y_f) <= sensor_range){
                LandmarkObs lmObs;
                lmObs.id = map_landmarks.landmark_list[j].id_i;
                lmObs.x  = map_landmarks.landmark_list[j].x_f;
                lmObs.y  = map_landmarks.landmark_list[j].y_f;
                reduced_map.push_back(lmObs);
            }
        }
```

Next, transform vehicle's obsersation from vehicle's coordinates to global positioning coordinates so all datas are in the same coordination system.

```
        for (int j=0; j<observations.size(); j++){
            LandmarkObs tmp;
            tmp.x = particles[i].x + observations[j].x*cos(particles[i].theta) -
     observations[j].y*sin(particles[i].theta);
            tmp.y = particles[i].y + observations[j].y*cos(particles[i].theta) +
     observations[j].x*sin(particles[i].theta);
            transformed_obs.push_back(tmp);
            ...
```

To associate each transformed observations to their closest landmarks, find their shortest distance. To simplify the code, I used the id field in transformed observations to hold the corresponding landmark's id.

```
        for (int k = 0; k < reduced_map.size(); k++) {
            double diff = dist(tmp.x, tmp.y, reduced_map[k].x, reduced_map[k
].y);
            if (diff < shortest) {
                shortest = diff;
                transformed_obs[j].id = reduced_map[k].id;
                shortest_idx = k;
            }
        }
```

To further improve performance, drop the landmark from the reduced_map list once it is associated to a transformed observation assuming a one-to-one relationship between each observation and its corresponding landmark, and no more than one landmark will be paired with each observation. If an landmark is paired to a wrong observation, their big distance will result to smaller weight and lowering its probabity to be picked later on during resampling.

```
        // optimize association finding by illiminating the landmark that ar
e just associated to a transformed observation
        // about 15% improvement on timing
        if (shortest < std::numeric_limits<double>::infinity()){
            reduced_map.erase(reduced_map.begin() + shortest_idx);
        }
```

Use Multivariate-Gaussian's standard deviation algorithm to find the sum of product between particles and associated as weight update.

```
        for (int j=0; j<transformed_obs.size(); j++){
            double x = transformed_obs[j].x;
            double y = transformed_obs[j].y;
            //double ux = map_landmarks.landmark_list[transformed_obs[j].id
 - 1].x_f;
            //double uy = map_landmarks.landmark_list[transformed_obs[j].id
 - 1].y_f;
            double ux = map_landmarks.landmark_list[transformed_obs[j].id -
1].x_f;
            double uy = map_landmarks.landmark_list[transformed_obs[j].id -
1].y_f;
            double diffx2 = (x-ux) * (x - ux);
            double diffy2 = (y-uy) * (y - uy);
            double sqrt2pr= (2.0f*M_PI*std_landmark[0]*std_landmark[1]);
            new_weight *= exp(-(diffx2/(2.0f*stdx2)) - (diffy2/(2.0f*stdy2
)))/sqrt2pr;
        }
        weights[i] = new_weight;
        particles[i].weight = new_weight;
```

Lastly, normalized weights.

```
    // normalized all weights by dividing each weight element with sum of al
l weights
    double norm_weight = std::accumulate(weights.begin(), weights.end(), 0.0
f);
    if (norm_weight > 0) {
        for (int i =0; i<num_particles; i++){
            weights[i] /= norm_weight*1.0f;
            particles[i].weight /= norm_weight;
        }
    }
```

Resample

I originally use Sabastian's Resampling Wheel algorithm, it works but it is slower than using discrete distribution algorithm, which discrete distribution offers 15% faster. As a result, discrete distribution is used.

```
    default_random_engine gen;
    discrete_distribution<> ddist(weights.begin(), weights.end());

    std::vector<Particle> new_particles;
    for (int i = 0; i < num_particles; i++) {
        new_particles.push_back(particles[ddist(gen)]);
    }
    particles = new_particles;
```

Result

In the first running version, the particle was following the vehicle only at the first few time step and then it drifted away upward from the vehicle. It still followed the vehicle direction as it traveled, but eventually run outside of the max error requirement. It was due to a few typo to the Multivariate-Gaussian's standard deviation algorithm and observation transformation. After fixing the bugs, the particles were following the vehicle closely as it travel. I started using 100 particles, which was extremely slow. I dropped it to 15, and I saw huge improvement. I further reduced it to 8, 7, then 6 and started to notice the paticle would drift away at 6 particles. So I finalized to 7 particles as the most optimal size to use.

```
In [ ]:
```