PROJECT

Kidnapped Vehicle

A part of the Self Driving Car Engineer Nanodegree Program

PROJECT REVIEW

CODE REVIEW 3

NOTES

particle_filter.cpp 3

```
/*
 * particle_filter.cpp
 *
 *  Created on: Dec 12, 2016
 *      Author: Tiffany Huang
 */

#include <random>
#include <algorithm>
#include <iostream>
#include <numeric>
#include <math.h>
#include <iostream>
#include <sstream>
#include <string>
#include <iterator>

#include "particle_filter.h"

using namespace std;

// declare a random engine to be used across multiple and various method calls
static default_random_engine gen;

void ParticleFilter::init(double x, double y, double theta, double std[]) {
    // TODO: Set the number of particles. Initialize all particles to first position (based on estimates of
    //   x, y, theta and their uncertainties from GPS) and all weights to 1.
    // Add random Gaussian noise to each particle.
    // NOTE: Consult particle_filter.h for more information about this method (and others in this file).

    // Tried 'num_particles' for - 2, 3, 30, 50, 100
    // Increasing 'num_particles' beyond 50 has less impact on 'Cumulative mean weighted error'
    // However, 'Runtime' continues to increase linearly
    num_particles = 50;

    // Standard deviations for x, y, and theta
    double std_x, std_y, std_theta;

    // TODO: Set standard deviations for x, y, and theta.
    std_x = std[0];
    std_y = std[1];
    std_theta = std[2];

    // Creates normal (Gaussian) distribution centered around GPS location
    normal_distribution<double> dist_x(x, std_x);
    normal_distribution<double> dist_y(y, std_y);
    normal_distribution<double> dist_theta(theta, std_theta);

    for (int i = 0; i < num_particles; ++i) {

        // Sample  from these normal distrubtions
        // where "gen" is the random engine initialized earlier

        Particle p;
        p.id = i;
        p.x = dist_x(gen);
        p.y = dist_y(gen);
        p.theta = dist_theta(gen);
        p.weight = 1.;
        particles.push_back(p);

        // initializing weights of all particles to 1
        weights.push_back(1.);

    }
```

```
    is_initialized = true;
}

// To predict each particle's state for the next time step using control inputs - velocity and yaw_rate
// Also account for sensor noise by adding Gaussian noise by sampling from a gaussian distribution
void ParticleFilter::prediction(double delta_t, double std_pos[], double velocity, double yaw_rate) {
    // TODO: Add measurements to each particle and add random Gaussian noise.
    // NOTE: When adding noise you may find std::normal_distribution and std::default_random_engine useful.
    //  http://en.cppreference.com/w/cpp/numeric/random/normal_distribution
    //  http://www.cplusplus.com/reference/random/default_random_engine/

    double std_x, std_y, std_theta; // Standard deviations for x, y, and theta

    // TODO: Set standard deviations for x, y, and theta.
    std_x = std_pos[0];
    std_y = std_pos[1];
    std_theta = std_pos[2];

    for (auto& p : particles) {

        if (fabs(yaw_rate) < 0.00001) {
            p.x += velocity * delta_t * cos(p.theta);
            p.y += velocity * delta_t * sin(p.theta);
        }
        else {
            double a = sin(p.theta + yaw_rate*delta_t) - sin(p.theta);
            double b = -cos(p.theta + yaw_rate*delta_t) + cos(p.theta);

            p.x += (velocity/yaw_rate)*a;
            p.y += (velocity/yaw_rate)*b;
            p.theta += yaw_rate*delta_t;
        }
```

AWESOME

Great job avoiding division by zero.

```
        // Creates a normal (Gaussian) distribution with mean x, y, theta
        normal_distribution<double> dist_x(p.x, std_x);
        normal_distribution<double> dist_y(p.y, std_y);
        normal_distribution<double> dist_theta(p.theta, std_theta);

        // Sample  from these normal distrubtions
        // where "gen" is the random engine initialized earlier
        p.x = dist_x(gen);
        p.y = dist_y(gen);
        p.theta = dist_theta(gen);

    }
}


void ParticleFilter::dataAssociation(std::vector<LandmarkObs> predicted, std::vector<LandmarkObs>& observations) {
    // TODO: Find the predicted measurement that is closest to each observed measurement and assign the
    //   observed measurement to this particular landmark.
    // NOTE: this method will NOT be called by the grading code. But you will probably find it useful to
    //   implement this method and use it as a helper during the updateWeights phase.

    for (auto& obs : observations) {

        // landmark id to be associated with the observation
        int map_id = -1;

        //double min_dist = 1000000.;
        double min_dist = std::numeric_limits<double>::max();

        for (auto landmark : predicted) {

            double d = dist(landmark.x,landmark.y,obs.x,obs.y);

            if (d<min_dist) {
                min_dist = d;
                map_id = landmark.id;
            }
        }
        obs.id = map_id;
    }
}


void ParticleFilter::updateWeights(double sensor_range, double std_landmark[],
        std::vector<LandmarkObs> observations, Map map_landmarks) {
    // TODO: Update the weights of each particle using a mult-variate Gaussian distribution. You can read
    //   more about this distribution here: https://en.wikipedia.org/wiki/Multivariate_normal_distribution
    // NOTE: The observations are given in the VEHICLE'S coordinate system. Your particles are located
    //   according to the MAP'S coordinate system. You will need to transform between the two systems.
    //   Keep in mind that this transformation requires both rotation AND translation (but no scaling).
    //   The following is a good resource for the theory:
    //   https://www.willamette.edu/~gorr/classes/GeneralGraphics/Transforms/transforms2d.htm
    //   and the following is a good resource for the actual equation to implement (look at equation
    //   3.33. Note that you'll need to switch the minus sign in that equation to a plus to account
    //   for the fact that the map's y-axis actually points downwards.)
    //   http://planning.cs.uiuc.edu/node99.html

    for (int i = 0; i < num_particles; ++i) {
```

```
        // get the particle x, y coordinates
        double px = particles[i].x;
        double py = particles[i].y;
        double ptheta = particles[i].theta;
```

SUGGESTION

One could consider using the const keyword to define values like these which do not change throughout for loop. See this stackoverflow thread for usage of `const` keyword.

```

        //1. Make list of all landmarks within sensor range of particles
        // Prediction measurements between one particular particle and
        // all of the map landmarks within sensor range
        std::vector<LandmarkObs> pred_landmarks ;

        for (auto landmark : map_landmarks.landmark_list) {

            // get id and x,y coordinates
            int lm_id = landmark.id_i;
            float lm_x = landmark.x_f;
            float lm_y = landmark.y_f;

            if (fabs(lm_x - px) <= sensor_range && fabs(lm_y - py) <= sensor_range) {
                pred_landmarks.push_back(LandmarkObs{ lm_id, lm_x, lm_y });
            }

        }

        // observations: Actual landmark measurements (in local coordinate system) gathered from LIDAR
        //2. Convert all observations from local to global frame
        std::vector<LandmarkObs> transformed_obs ;

        for (auto obs : observations) {
            LandmarkObs t_obs;
            t_obs.id = obs.id;
            t_obs.x = obs.x*cos(ptheta) - obs.y*sin(ptheta) + px;
            t_obs.y = obs.x*sin(ptheta) + obs.y*cos(ptheta) + py;
            transformed_obs.push_back(t_obs);
        }


        //3. Perform nearest neighbour `dataAssociation`.
        // Find the nearest landmark (landmark with the minimum euclidian distance)
        // This will put the index of the `predicted_lm` nearest to each
        // `transformed_obs` in the `id` field of the `transformed_obs` element.
        dataAssociation(pred_landmarks, transformed_obs);


        //4. Loop through all the `transformed_obs`.
        //Use the saved index in the `id` to find the associated landmark and compute the gaussian.

        double ONE_OVER_2PI_std = 1/(2*M_PI*std_landmark[0]*std_landmark[1]) ;
```

AWESOME

Great work calculating this constant value out of the for loop.

```

        double w = 1.;
        double w_i = 1.;

        for (auto t_obs : transformed_obs) {

            for (auto landmark : pred_landmarks) {

                if (t_obs.id == landmark.id) {
                    double x = t_obs.x;
                    double y = t_obs.y;
                    double mx = landmark.x;
                    double my = landmark.y;

                    double a = ((x-mx)*(x-mx))/(2*std_landmark[0]*std_landmark[0]);
                    double b = ((y-my)*(y-my))/(2*std_landmark[1]*std_landmark[1]);

                    // weight for each observation in `transformed_obs`
                    w_i = ONE_OVER_2PI_std*exp(-0.5*(a+b));
                    w *= w_i;
                }
            }
        }
        //5. Multiply all the gaussian values together to get total probability of particle (the weight).
        // Posterior probability for  each  particle
        particles[i].weight = w;

        // used to normalize weights
        // Used to create discrete distribution for resampling
        weights[i] = w;

    }

    // Normalize weights
```

```
    for (int i = 0; i < num_particles; ++i) {
        double sum = std::accumulate(weights.begin(), weights.end(), 0.);
        if (sum != 0) {
            particles[i].weight = weights[i]/sum ;
        }
    }
}

// To sample particles in proportion to their weights
void ParticleFilter::resample() {
    // TODO: Resample particles with replacement with probability proportional to their weight.
    // NOTE: You may find std::discrete_distribution helpful here.
    //   http://en.cppreference.com/w/cpp/numeric/random/discrete_distribution

    discrete_distribution<> dist(weights.begin(), weights.end());

    std::vector<Particle> resampled_particles;
    resampled_particles.resize(num_particles);

    for (int i = 0; i < num_particles; ++i) {
        resampled_particles[i] = particles[dist(gen)];
    }
    particles = resampled_particles;
}

Particle ParticleFilter::SetAssociations(Particle particle, std::vector<int> associations, std::vector<double> sense_x, std:
{
    //particle: the particle to assign each listed association, and association's (x,y) world coordinates mapping to
    // associations: The landmark id that goes along with each listed association
    // sense_x: the associations x mapping already converted to world coordinates
    // sense_y: the associations y mapping already converted to world coordinates

    //Clear the previous associations
    particle.associations.clear();
    particle.sense_x.clear();
    particle.sense_y.clear();

    particle.associations= associations;
    particle.sense_x = sense_x;
    particle.sense_y = sense_y;

    return particle;
}

string ParticleFilter::getAssociations(Particle best)
{
    vector<int> v = best.associations;
    stringstream ss;
    copy( v.begin(), v.end(), ostream_iterator<int>(ss, " "));
    string s = ss.str();
    s = s.substr(0, s.length()-1);  // get rid of the trailing space
    return s;
}
string ParticleFilter::getSenseX(Particle best)
{
    vector<double> v = best.sense_x;
    stringstream ss;
    copy( v.begin(), v.end(), ostream_iterator<float>(ss, " "));
    string s = ss.str();
    s = s.substr(0, s.length()-1);  // get rid of the trailing space
    return s;
}
string ParticleFilter::getSenseY(Particle best)
{
    vector<double> v = best.sense_y;
    stringstream ss;
    copy( v.begin(), v.end(), ostream_iterator<float>(ss, " "));
    string s = ss.str();
    s = s.substr(0, s.length()-1);  // get rid of the trailing space
    return s;
}
```

RETURN TO PATH

Rate this review