UDACITY

PROJECT

Kidnapped Vehicle

A part of the Self-Driving Car Engineer Program

PROJECT REVIEW

CODE REVIEW 8

NOTES

p3-kidnapped-vehicle/src/particle_filter.cpp 8

```
1 /*
2  * particle_filter.cpp
3  *
4  *  Created on: Dec 12, 2016
5  *      Author: Tiffany Huang
6  */
7
8 #include <random>
9 #include <algorithm>
10 #include <iostream>
11 #include <numeric>
12 #include <math.h>
13 #include <iostream>
14 #include <sstream>
15 #include <string>
16 #include <iterator>
17
18 #include "particle_filter.h"
19
20 using namespace std;
21
22 // declare a random number engine
23 default_random_engine gen;
24
25 void ParticleFilter::init(double x, double y, double theta, double std_pos[]) {
26     // TODO: Set the number of particles. Initialize all particles to first position (based on estimates of
27     //   x, y, theta and their uncertainties from GPS) and all weights to 1.
28     // Add random Gaussian noise to each particle.
29     // NOTE: Consult particle_filter.h for more information about this method (and others in this file).
30
31     num_particles = 100;
32
33   // define normal (Gaussian) distributions for sensor noise
34   normal_distribution<double> dist_x(0, std_pos[0]);
35   normal_distribution<double> dist_y(0, std_pos[1]);
36   normal_distribution<double> dist_theta(0, std_pos[2]);
```

SUGGESTION

You could centre these distributions around `x`, `y`, and `theta` respectively and simply assign the sampled values at lines 48-50 allowing you to avoid the duplicate assignments at lines 42-44. This is because the distributions have the same mean values for all the particles.

```
37
38   // initialize particles
39   for (int i = 0; i < num_particles; i++) {
40     Particle p;
41     p.id = i;
42     p.x = x;
43     p.y = y;
44     p.theta = theta;
45     p.weight = 1.0;
46
47     // add noise
48     p.x += dist_x(gen);
49     p.y += dist_y(gen);
50     p.theta += dist_theta(gen);
```

```

    particles.push_back(p);
        weights.push_back(p.weight);
  }

  is_initialized = true;
}

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/

    // define normal (Gaussian) distributions for sensor noise
    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]);
```

AWESOME

Great work centring the noise gaussian distributions at 0 so they could be defined outside of the particle loop and reused for each particle.

```

    // add measurements to each particle
      for (int i = 0; i < num_particles; i++) {

        // calculate new state
        if (fabs(yaw_rate) < 0.00001) {
          particles[i].x += velocity * delta_t * cos(particles[i].theta);
          particles[i].y += velocity * delta_t * sin(particles[i].theta);
        }
        else {
          particles[i].x += velocity / yaw_rate * (sin(particles[i].theta+yaw_rate*delta_t) - sin(particles[i].theta));
          particles[i].y += velocity / yaw_rate * (cos(particles[i].theta) - cos(particles[i].theta+yaw_rate*delta_t));
          particles[i].theta += yaw_rate * delta_t;
        }
```

AWESOME

Great job handling both zero and non-zero yaw rates!

```

        // add noise
        particles[i].x += dist_x(gen);
        particles[i].y += dist_y(gen);
        particles[i].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.

    // get the closest predicted measurement to each observed measurement
    for (int i = 0; i < observations.size(); i++) {

        // current observation
        LandmarkObs observation = observations[i];

        // init minimum distance to maximum possible
    double min_distance = numeric_limits<double>::max();

        // init id of landmark to associate with closest measurement
        int landmark_id = -1;

        for(int j=0; j < predicted.size(); j++) {

                    // get current prediction
                    LandmarkObs predicted_measurement = predicted[j];

                    // calculate distance between current vs. predicted landmarks
            double distance = dist(observation.x, observation.y, predicted_measurement.x, predicted_measurement.y);

                    // find the closest landmark
                    if (distance < min_distance) {
                min_distance = distance;
                landmark_id = predicted_measurement.id;
            }
        }
        // assign the observed measurement to specific landmark
        observations[i].id = landmark_id;
    }
}

```

```
void ParticleFilter::updateWeights(double sensor_range, double std_landmark[],
        const std::vector<LandmarkObs> &observations, const 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
    //   http://planning.cs.uiuc.edu/node99.html

    // iterate through each particle
    for (int i = 0; i < num_particles; i++) {
        double x = particles[i].x;
        double y = particles[i].y;
        double theta = particles[i].theta;

        // keep predicted landmark locations within sensor range of particle
        vector<LandmarkObs> predicted_landmarks;

        for (int j = 0; j < map_landmarks.landmark_list.size(); j++) {

                // get id and x,y coordinates
            int lm_id = map_landmarks.landmark_list[j].id_i;
            double lm_x = map_landmarks.landmark_list[j].x_f;
            double lm_y = map_landmarks.landmark_list[j].y_f;
            LandmarkObs curr_lm = {lm_id, lm_x, lm_y};

            // select landmark within range
            if (fabs(dist(lm_x, lm_y, x, y)) <= sensor_range){
```

SUGGESTION

Wrapping the distance in the `fabs` method isn't necessary as the `dist` method can't return a negative value.

```
                predicted_landmarks.push_back(curr_lm);
            }
        }
        // transform points
        vector<LandmarkObs> transformed_observations;

        for (int k = 0; k < observations.size(); k++) {
            double tr_x = observations[k].x * cos(theta) - observations[k].y * sin(theta) + x;
            double tr_y = observations[k].x * sin(theta) + observations[k].y * cos(theta) + y;

            LandmarkObs transformed_ob;
            transformed_ob.id = observations[k].id;
            transformed_ob.x = tr_x;
            transformed_ob.y = tr_y;

            transformed_observations.push_back(transformed_ob);
        }

        dataAssociation(predicted_landmarks, transformed_observations);

        double total_weight = 1.0;
        weights[i] = 1.0;
```

AWESOME

Well done resetting the weight with every iteration of the filter.

```
        vector<int> associations_vec;
        vector<double> sense_x_vec;
        vector<double> sense_y_vec;

        for (int l = 0; l < transformed_observations.size(); l++) {

                // placeholders for observation and prediction coordinates
            int obv_id = transformed_observations[l].id;
            double obv_x = transformed_observations[l].x;
            double obv_y = transformed_observations[l].y;
            double pred_x;
            double pred_y;

                // get coordinates of the prediction for the current observation
            for (int m = 0; m < predicted_landmarks.size(); m++) {
                if (predicted_landmarks[m].id == obv_id){
                    pred_x = predicted_landmarks[m].x;
                    pred_y = predicted_landmarks[m].y;
                }
            }
```

SUGGESTION

It would be best to add some defensive logic to handle the situation where none of the landmarks are inside the sensor range even if it doesn't occur in the simulation.

```

            // update weight for this observation
            double w = (1/(2*M_PI*std_landmark[0]*std_landmark[1])) *
                        exp(-(pow(pred_x-obv_x, 2)/(2*pow(std_landmark[0],2)) +
                            pow(pred_y-obv_y, 2)/(2*pow(std_landmark[1],2)) -
                            (2*(pred_x-obv_x)*(pred_y-obv_y)/(sqrt(std_landmark[0])*sqrt(std_landmark[1])))));
```

SUGGESTION

There are elements of this calculation (such as the denominator) that don't depend on the particle, landmark, or observation. I suggest calculating them separately outside of the particle loop and reusing these values.

```

            total_weight *= w;
            associations_vec.push_back(obv_id);
            sense_x_vec.push_back(obv_x);
            sense_y_vec.push_back(obv_y);
        }
        particles[i].weight = total_weight;
        weights[i] = total_weight;

        SetAssociations(particles[i], associations_vec, sense_x_vec, sense_y_vec);
        predicted_landmarks.clear();
    }
}

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

    // use discrete distribution for weights
    discrete_distribution<int> index(weights.begin(), weights.end());

    // placeholder for resampled particles
    vector<Particle> resampled_particles;

    // resample particles
    for (int i = 0; i < num_particles; i++) {
        resampled_particles.push_back(particles[index(gen)]);
    }
    particles = resampled_particles;
```

AWESOME

Nice job using `discrete_distibution` to resample the particles proportional to their weights.

Another option would be to implement the `resampling wheel` discussed by Sebastian Thrun in the Python Particle Filters lesson.

```
}

Particle ParticleFilter::SetAssociations(Particle& particle, const std::vector<int>& associations,
                                     const std::vector<double>& sense_x, const std::vector<double>& sense_y) {
    //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 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;
}

```

- p3-kidnapped-vehicle/src/particle_filter.h
- p3-kidnapped-vehicle/src/map.h
- p3-kidnapped-vehicle/src/main.cpp
- p3-kidnapped-vehicle/src/helper_functions.h
- p3-kidnapped-vehicle/cmakepatch.txt
- p3-kidnapped-vehicle/README.md
- p3-kidnapped-vehicle/CMakeLists.txt
- p2-unscented-kalman-filter/src/ukf.h
- p2-unscented-kalman-filter/src/ukf.cpp
- p2-unscented-kalman-filter/src/tools.h
- p2-unscented-kalman-filter/src/tools.cpp
- p2-unscented-kalman-filter/src/measurement_package.h
- p2-unscented-kalman-filter/src/main.cpp
- p2-unscented-kalman-filter/src/Eigen/src/plugins/MatrixCwiseUnaryOps.h
- p2-unscented-kalman-filter/src/Eigen/src/plugins/MatrixCwiseBinaryOps.h
- p2-unscented-kalman-filter/src/Eigen/src/plugins/CommonCwiseUnaryOps.h
- p2-unscented-kalman-filter/src/Eigen/src/plugins/CommonCwiseBinaryOps.h
- p2-unscented-kalman-filter/src/Eigen/src/plugins/CMakeLists.txt
- p2-unscented-kalman-filter/src/Eigen/src/plugins/BlockMethods.h
- p2-unscented-kalman-filter/src/Eigen/src/plugins/ArrayCwiseUnaryOps.h
- p2-unscented-kalman-filter/src/Eigen/src/plugins/ArrayCwiseBinaryOps.h
- p2-unscented-kalman-filter/src/Eigen/src/misc/blas.h
- p2-unscented-kalman-filter/src/Eigen/src/misc/SparseSolve.h
- p2-unscented-kalman-filter/src/Eigen/src/misc/Solve.h