UDACITY

‹ Back to Self-Driving Car Engineer

Kidnapped Vehicle

REVIEW

CODE REVIEW 3

HISTORY

▾ src/particle_filter.cpp 3

```
/*
 * particle_filter.cpp
 *
 *  Created on: Dec 12, 2016
 *      Author: Tiffany Huang (Udacity)
 * & Laura Le
 */

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

#include "particle_filter.h"

#define EPS 0.00001

using namespace std;

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 pos
    //   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
    num_particles = 100;
```

AWESOME

Great choice for the number of particle filter. 👍

```

    //normal disitbrution of sensor noise
    normal_distribution<double> N_x(0, std[0]);
    normal_distribution<double> N_y(0, std[1]);
    normal_distribution<double> N_theta(0, std[2]);

    //initiate particles
    for (int i = 0; i < num_particles; i++){
        Particle p;
        p.id = i;
        p.x = x;
        p.y = y;
        p.theta = theta;
        p.weight = 1.0;

        // add noise
        p.x += N_x(gen);
        p.y += N_y(gen);
        p.theta += N_theta(gen);

        particles.push_back(p);
    }

    is_initialized=true;

}


void ParticleFilter::prediction(double delta_t, double std_pos[], double veloc
    // TODO: Add measurements to each particle and add random Gaussian noise.
    // NOTE: When adding noise you may find std::normal_distribution and std::
    //  http://en.cppreference.com/w/cpp/numeric/random/normal_distribution
    //  http://www.cplusplus.com/reference/random/default_random_engine/

    // define normal distribution for sensor noise
    normal_distribution<double> N_x_sensor(0, std_pos[0]);
    normal_distribution<double> N_y_sensor(0, std_pos[1]);
    normal_distribution<double> N_theta_sensor(0, std_pos[2]);

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

        // get new state
        if (fabs(yaw_rate) < EPS){
            particles[i].x += velocity * delta_t * cos(particles[i].theta);
            particles[i].y += velocity * delta_t *sin(particles[i].theta);
            //yaw rate continue being the same
        }
        else{
            particles[i].x += velocity / yaw_rate * (sin(particles[i].theta + y
            particles[i].y += velocity / yaw_rate * (cos(particles[i].theta) -
            particles[i].theta += yaw_rate * delta_t;
        }

        // ad noise
        particles[i].x += N_x_sensor(gen);
        particles[i].y += N_y_sensor(gen);
        particles[i].theta += N_theta_sensor(gen);
    }

}


void ParticleFilter::dataAssociation(std::vector<LandmarkObs> predicted, std::v
    // TODO: Find the predicted measurement that is closest to each observed me
    //   observed measurement to this particular landmark.
    // NOTE: this method will NOT be called by the grading code. But you will p
    //   implement this method and use it as a helper during the updateWeights

    unsigned int nObservations = observations.size();
    unsigned int nPredictions = predicted.size();

    for (unsigned int i=0; i< nObservations; i++){

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

        // initiate id of landmark from map to be associated with the observati
        int map_id = -1;
        double min_distance = numeric_limits<double>::max();

        for (unsigned int j =0; j < nPredictions; j ++){
            LandmarkObs pred = predicted[j];

            // get distance between current observation and landmark
            double curr_distance = dist(obs.x, obs.y, pred.x, pred.y);

            // find the nearest predicted landmark to current observation
            if (curr_distance < min_distance){
                min_distance = curr_distance;
                map_id = pred.id;
            }
        }

        // set observation id to the nearest found preidcted landmark's id
        observations[i].id = map_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
    //   more about this distribution here: https://en.wikipedia.org/wiki/Multi
    // NOTE: The observations are given in the VEHICLE'S coordinate system. You
    //   according to the MAP'S coordinate system. You will need to transform b
    //   Keep in mind that this transformation requires both rotation AND trans
    //   The following is a good resource for the theory:
    //   https://www.willamette.edu/~gorr/classes/GeneralGraphics/Transforms/tr
    //   and the following is a good resource for the actual equation to implem
    //   3.33
    //   http://planning.cs.uiuc.edu/node99.html
```

```

  for (unsigned i = 0; i < num_particles; i++){
      double particle_x = particles[i].x;
      double particle_y = particles[i].y;
      double particle_theta = particles[i].theta;

      // create vector to store map landa=mark location within sensor range
      vector<LandmarkObs> predictions;

      for (unsigned int j = 0; j < map_landmarks.landmark_list.size(); j++){
          float landmark_x = map_landmarks.landmark_list[j].x_f;
          float landmark_y = map_landmarks.landmark_list[j].y_f;
          int landmark_id = map_landmarks.landmark_list[j].id_i;

          double dX = particle_x - landmark_x;
          double dY = particle_y - landmark_y;
          double sensor_range_2 = sensor_range * sensor_range;

          // consider only landmark within sensor range
          if( dX*dX + dY*dY <= sensor_range_2){

              // add prediction to vector
              predictions.push_back(LandmarkObs{landmark_id, landmark_x,land
          }
      }

          //transform observation coordinate from car's to map's
      vector<LandmarkObs> trans_observations;
      for (unsigned int j = 0; j < observations.size(); j++) {
          double transObs_x = cos(particle_theta)*observations[j].x - sin(pa
          double transObs_y = sin(particle_theta)*observations[j].x + cos(pa
          trans_observations.push_back(LandmarkObs{ observations[j].id, tran
      }

      // perform data association between particle and selected landmark
      dataAssociation(predictions, trans_observations);

      //reset weights
      particles[i].weight = 1.0;

      for (unsigned int j=0; j<trans_observations.size(); j++){
          double obs_x = trans_observations[j].x;
          double obs_y = trans_observations[j].y;

          double pred_x, pred_y;

          int associated_pred_id = trans_observations[j].id;

          bool found = false;
          unsigned int k =0;
          unsigned int pred_sz = predictions.size();
          while(!found && k < pred_sz){
              if(predictions[k].id == associated_pred_id){
                  found = true;
                  pred_x = predictions[k].x;
                  pred_y = predictions[k].y;
              }
              k++;
          }
          //calculate weight for this observation with multivariate gaussian
          double stdlm_x = std_landmark[0];
```

```
            double stdlm_y = std_landmark[1];
            double obs_weight = ( 1/(2*M_PI*stdlm_x*stdlm_y)) * exp( -( pow(pre
            if(obs_weight < EPS){
                obs_weight = EPS;
            }
            //update particle weight
            particles[i].weight *= obs_weight;
        }
    }
}
```

AWESOME

The `updateWeights` code structure is logical, neat and all the methods have been correctly implemented.

```


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

    //get weights
    vector<double> weights;
    double max_weight = numeric_limits<double>::min();
    for(int i=0; i< num_particles; i++){
        weights.push_back(particles[i].weight);
        if(particles[i].weight > max_weight){
            max_weight = particles[i].weight;
        }
    }

    // create disitribution
    uniform_real_distribution<double> dist_double(0.0, max_weight);
    uniform_int_distribution<int> dist_int(0, num_particles-1);

    int index= dist_int(gen);
    double beta = 0.0;

    // create the wheel
    vector<Particle> resampled_particles;
    for(int i=0; i<num_particles;i++){
        beta += dist_double(gen)*2.0;
        while(beta > weights[index]){
            beta -= weights[index];
            index=(index+1)% num_particles;
        }
        resampled_particles.push_back(particles[index]);
    }

    //update particles with the new list of resampled particles
    particles = resampled_particles;
}
```

AWESOME

Resampling algorithm is pythonic👍

```


Particle ParticleFilter::SetAssociations(Particle& particle, const std::vector<
                                     const std::vector<double>& sense_x, const
{
    //particle: the particle to assign each listed association, and associatio
    // associations: The landmark id that goes along with each listed associati
    // sense_x: the associations x mapping already converted to world coordina
    // sense_y: the associations y mapping already converted to world coordina

    //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;
}

```

▶ src/particle_filter.h

▶ src/map.h

- src/main.cpp
- src/helper_functions.h
- ide_profiles/xcode/CMakeFiles/TargetDirectories.txt
- ide_profiles/xcode/CMakeFiles/3.11.1/CompilerIdCXX/CMakeCXXCompilerId.cpp
- ide_profiles/xcode/CMakeCache.txt
- data/map_data.txt
- cmakepatch.txt
- README.md
- CMakeLists.txt

RETURN TO PATH

Rate this review