UDACITY

‹ Return to "Self-Driving Car Engineer" in the classroom

DISCUSS ON STUDENT HUB

Kidnapped Vehicle

REVIEW

CODE REVIEW 2

HISTORY

▾ src/particle_filter.cpp 2

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

#include "particle_filter.h"

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

#include "helper_functions.h"

using std::string;
using std::vector;


void ParticleFilter::init(double x, double y, double theta, double std[]) {
  /**
```

```
   * Set the number of particles. Initialize all particles's position and weig
   */

  // The number of particles was selected based on suggestion from project
  // overview by Udacity:
  // https://www.youtube.com/watch?v=-3HI3Iw3Z9g&feature=youtu.be
  // I found, however, that as small as 10 and as large as 1000+ number would
  // With smallest value, the error grows, but can still pass the acceptable t
  // very small value of particles such as 5, some lucky initialization can pa
  // With larger number of particles, such as 1000, the program becomes too sl
  num_particles = 100;

  // Random engine for particle pose noise generation.
  std::default_random_engine gen;
```

SUGGESTION

This is actually pseudo random, we are always generating the same particles during every run of ParticleFi

we can create a truly random generator with something like below

```
random_device rd;
default_random_engine gen(rd());
```

```

  // Sensor noise distributions and associated variables.
  std::normal_distribution<double> x_dist(0, std[0]);
  std::normal_distribution<double> y_dist(0, std[1]);
  std::normal_distribution<double> theta_dist(0, std[2]);
  double x_noise = 0.0, y_noise = 0.0, theta_noise = 0.0;

  // Initializing all particles.
  for (int i = 0; i < num_particles; i++) {
    Particle p;
    x_noise = x_dist(gen);
    y_noise = y_dist(gen);
    theta_noise = theta_dist(gen);

    p.id = i;
    p.x = x + x_noise;
    p.y = y + y_noise;
    p.theta = theta + theta_noise;
    // All initial weights are one per assignment requirement.
    p.weight = 1.0;

    particles.push_back(p);
  }

  is_initialized = true;
}

void ParticleFilter::prediction(double delta_t, double std_pos[],
                                double velocity, double yaw_rate) {
  /**
   * Use measurement to calculate prediction for particular state.
   * Adds random Gaussian noise to each particle state.
   */
```

```
  std::default_random_engine gen;
  std::normal_distribution<double> x_dist(0, std_pos[0]);
  std::normal_distribution<double> y_dist(0, std_pos[1]);
  std::normal_distribution<double> theta_dist(0, std_pos[2]);
  double x_noise, y_noise, theta_noise = 0;

  for (int i = 0; i < num_particles; i++) {
    x_noise = x_dist(gen);
    y_noise = y_dist(gen);
    theta_noise = theta_dist(gen);

    // Considering two cases: when the car drives straight (yaw_rate is close
    // and car is turning. Similarly to Kalman filter project, we do not direc
    // yaw rate to zero angle of steering, but rather check zero's vicinity.
    // Interestingly, I found the noise to be very important. Without
    // adding noise the main particle will steer off the trajectory of the car
    if (fabs(yaw_rate) < 0.001) {
      particles[i].x += velocity * delta_t * cos(particles[i].theta) + x_noise
      particles[i].y += velocity * delta_t * sin(particles[i].theta) + y_noise
      particles[i].theta += theta_noise;
    }
    else {
      particles[i].x += velocity / yaw_rate
          * (sin(particles[i].theta + yaw_rate*delta_t) - sin(particles[i].the
      particles[i].y += velocity / yaw_rate
          * (cos(particles[i].theta) - cos(particles[i].theta + yaw_rate*delta
      particles[i].theta += yaw_rate * delta_t + theta_noise;
    }
  }
}

void ParticleFilter::dataAssociation(vector<LandmarkObs> predicted,
                                     vector<LandmarkObs>& observations) {
  /**
   * Find the predicted measurement that is closest to each observed measureme
   * and assign the observed measurement to this particular landmark.
   */

  for (unsigned int i = 0; i < observations.size(); i++) {

      // Observations are already in global coordinate system, thus can be dir
      LandmarkObs curr_observation = observations[i];

      // Initializing the minimum distance and matching index of the landmark.
      double min_distance = std::numeric_limits<double>::max();
      int match_index = std::numeric_limits<int>::min();;

      // Going through each observable landmark and getting the best match to
      // Best match is found through nearest neighbor algorithm. Assuming the
      // observable landmarks is greater than the actual number of observation
      for (unsigned int j = 0; j < predicted.size(); j++) {
        LandmarkObs curr_landmark = predicted[j];

        double curr_distance = dist(curr_observation.x, curr_observation.y,
                                    curr_landmark.x, curr_landmark.y);

        if (curr_distance < min_distance) {
          min_distance = curr_distance;
          match_index = curr_landmark.id;
        }
      }
```

```

      // Recording the match between observation and landmark.
      observations[i].id = match_index;
  }
}

void ParticleFilter::updateWeights(double sensor_range, double std_landmark[],
                                   const vector<LandmarkObs> &observations,
                                   const Map &map_landmarks) {
  /**
   * Update the weights of each particle using a multi-variate Gaussian
   * distribution.
   *
   * The observations are given in the VEHICLE'S coordinate system.
   * The particles are located according to the MAP'S coordinate system.
   * Thus, the transform between the two systems is performed.
   */
  for (int i = 0; i < num_particles; i++) {
    Particle p = particles[i];
    // Product of multi-variate Gaussian.
    double multivar_prod = 1;

    // The subset of landmarks that fall within the sensor range.
    vector<LandmarkObs> landmarks_p;

    // Populating the observable subset of landmarks.
    for (unsigned int j = 0; j < map_landmarks.landmark_list.size(); j++) {
      Map::single_landmark_s l = map_landmarks.landmark_list[j];

      //The subset is defined by radial distance between observation and a lan
      if (dist(l.x_f, l.y_f, p.x, p.y) <= sensor_range){
        landmarks_p.push_back(LandmarkObs{l.id_i, l.x_f, l.y_f});
      }
    }

    // Transforming observations into global coordinate system.
    vector<LandmarkObs> transformed_os;
    for (unsigned int j = 0; j < observations.size(); j++) {
      LandmarkObs o = observations[j], t;

      t.id = o.id;
      t.x = cos(p.theta)*o.x - sin(p.theta)*o.y + p.x;
      t.y = sin(p.theta)*o.x + cos(p.theta)*o.y + p.y;
      transformed_os.push_back(t);
    }

    // Match subset of landmarks to their observations.
    dataAssociation(landmarks_p, transformed_os);

    for (unsigned int j = 0; j < transformed_os.size(); j++) {
      LandmarkObs o = transformed_os[j];
      // Recording the matched x, y, and id in the following var.
      LandmarkObs m;
      m.id = o.id;

      // Scanning the subset to get the donor for coordinates.
      for (unsigned int k = 0; k < landmarks_p.size(); k++) {
        LandmarkObs p = landmarks_p[k];
        if (p.id == m.id) {
          m.x = p.x;
          m.y = p.y;
```

```
        }
      }

      // Obtaining new weight for this observation with multivariate Gaussian.
      double std_x = std_landmark[0];
      double std_y = std_landmark[1];
      double new_weight = ( 1/(2*M_PI*std_x*std_y)) * exp( -( pow(m.x-o.x,2)/(
                      + (pow(m.y-o.y,2)/(2*pow(std_y, 2))) ) );

      // Updating multivariate product.
      multivar_prod *= new_weight;
    }
    // Weight is updated on actual particle from the list as it is used downst
    particles[i].weight = multivar_prod;
  }
}

void ParticleFilter::resample() {
  /**
   * Re-sample particles with replacement with probability proportional
   *   to their weight.
   */
  vector<Particle> new_particles;
  // Using library implementation of discrete distribution generator
  // with weights. Replaces the re-sample wheel algorithm.
  std::default_random_engine gen;

  // Need to create iterable for required distribution parameter.
  vector<double> weights;
    for (int i = 0; i < num_particles; i++) {
      weights.push_back(particles[i].weight);
  }

  for (int i = 0; i < num_particles; ++i) {
      std::discrete_distribution<> discrete_dist(weights.begin(), weights.end(
      new_particles.push_back(particles[discrete_dist(gen)]);
  }

  particles = new_particles;
```

SUGGESTION

We can avoid the deep copy of vector data by using move semantics like below. See here for more details.

```
particles = std::move(new_particles);
```

```
}

void ParticleFilter::SetAssociations(Particle& particle,
                                     const vector<int>& associations,
                                     const vector<double>& sense_x,
                                     const vector<double>& sense_y) {
  // particle: the particle to which assign each listed association,
  //   and association's (x,y) world coordinates mapping
  // associations: The landmark id that goes along with each listed associatio
  // sense_x: the associations x mapping already converted to world coordinate
  // sense_y: the associations y mapping already converted to world coordinate
  particle.associations= associations;
```

```
  particle.sense_x = sense_x;
  particle.sense_y = sense_y;
}

string ParticleFilter::getAssociations(Particle best) {
  vector<int> v = best.associations;
  std::stringstream ss;
  copy(v.begin(), v.end(), std::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::getSenseCoord(Particle best, string coord) {
  vector<double> v;

  if (coord == "X") {
    v = best.sense_x;
  } else {
    v = best.sense_y;
  }

  std::stringstream ss;
  copy(v.begin(), v.end(), std::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

▸ data/map_data.txt

▸ cmakepatch.txt

▸ README.md

▸ CMakeLists.txt

RETURN TO PATH