UDACITY

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

DISCUSS ON STUDENT HUB

Path Planning

REVIEW

CODE REVIEW 5

HISTORY

src/main.cpp 4

```
#include <uWS/uWS.h>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "Eigen-3.3/Eigen/Core"
#include "Eigen-3.3/Eigen/QR"
#include "helpers.h"
#include "spline.h"
```

AWESOME

Good job using splines for path generation

```
#include "json.hpp"
#include <math.h>

// for convenience
using nlohmann::json;
using std::string;
using std::vector;

int main() {
  uWS::Hub h;
```

```
  // Load up map values for waypoint's x,y,s and d normalized normal vectors
  vector<double> map_waypoints_x;
  vector<double> map_waypoints_y;
  vector<double> map_waypoints_s;
  vector<double> map_waypoints_dx;
  vector<double> map_waypoints_dy;

  // Waypoint map to read from
  string map_file_ = "../data/highway_map.csv";
  // The max s value before wrapping around the track back to 0
  double max_s = 6945.554;

  std::ifstream in_map_(map_file_.c_str(), std::ifstream::in);

  string line;
  while (getline(in_map_, line)) {
    std::istringstream iss(line);
    double x;
    double y;
    float s;
    float d_x;
    float d_y;
    iss >> x;
    iss >> y;
    iss >> s;
    iss >> d_x;
    iss >> d_y;
    map_waypoints_x.push_back(x);
    map_waypoints_y.push_back(y);
    map_waypoints_s.push_back(s);
    map_waypoints_dx.push_back(d_x);
    map_waypoints_dy.push_back(d_y);
  }

  int lane = 1; // Middle lane within the Frenet space.

  // Reference velocity and acceleration for the car.
  double ref_vel = 0.0; // In miles per hour.
```

AWESOME

Good work initializing the vehicle at 0 mph and increasing gradually to the target speed.👏

```
  double ref_acc = 0.6; // In miles per hour.

  h.onMessage([&map_waypoints_x,&map_waypoints_y,&map_waypoints_s,
               &map_waypoints_dx,&map_waypoints_dy, &lane, &ref_vel, &ref_acc]
              (uWS::WebSocket<uWS::SERVER> ws, char *data, size_t length,
               uWS::OpCode opCode) {
    // "42" at the start of the message means there's a websocket message even
    // The 4 signifies a websocket message
    // The 2 signifies a websocket event
    if (length && length > 2 && data[0] == '4' && data[1] == '2') {

      auto s = hasData(data);

      if (s != "") {
```

```
      auto j = json::parse(s);

      string event = j[0].get<string>();

      if (event == "telemetry") {
        // j[1] is the data JSON object

        // Main car's localization Data
        double car_x = j[1]["x"];
        double car_y = j[1]["y"];
        double car_s = j[1]["s"];
        double car_d = j[1]["d"];
        double car_yaw = j[1]["yaw"];
        double car_speed = j[1]["speed"];

        // Previous path data given to the Planner
        auto previous_path_x = j[1]["previous_path_x"];
        auto previous_path_y = j[1]["previous_path_y"];
        // Previous path's end s and d values
        double end_path_s = j[1]["end_path_s"];
        double end_path_d = j[1]["end_path_d"];

        // Sensor Fusion Data, a list of all other cars on the same side
        //   of the road.
        auto sensor_fusion = j[1]["sensor_fusion"];

        json msgJson;

        /** PROJECT CONTRIBUTION BEGIN */

        /*
          List of widely separated waypoints to fit spline trajectory onto.
          Initially, they are spaced at PLANNING_HORIZON_DISTANCE meters apa
        */
        vector<double> ptsx;
        vector<double> ptsy;

        int prev_size = previous_path_x.size();

        if (prev_size > 0)
        {
          // For planning purposes, we go to the place in time where the las
          // predicted path point would be.
          car_s = end_path_s;
        }

        // Sensor fusion and prediction and trajectory generation.
        bool too_close = false;

        // The following two variables are `poor's man` cost functions for c
        // lane to the right or to the left. The larger the distance to the
        // of those lanes, the more preferable such direction is for a lane
        double closest_car_left_lane = 10000; // Distance to the closest car
        double closest_car_right_lane = 10000;  // Distance to the closest c
        for (int i = 0; i < sensor_fusion.size(); ++i) {
            float d = sensor_fusion[i][6];
            // Assign the car to the lane space.
            int car_lane = getLane(d);
            if (car_lane == -1)
              continue;

```

```
          double vx = sensor_fusion[i][3]; // vx component of the other ca
          double vy = sensor_fusion[i][4]; // vy component of the other ca
          double check_speed = sqrt(vx * vx + vy * vy);
          double check_car_s = sensor_fusion[i][5];

          // Using previous points project s value out in time for the
          // whole duration of 'previous_path' points.
          check_car_s += static_cast<double>(prev_size) * PLANNING_TICK_IN

          if (lane - car_lane == 0) {  // Checking ego lane.
            if (check_car_s > car_s && check_car_s - car_s < PLANNING_HORI
              too_close = true;
          } else if (lane - car_lane == 1 ) { // Checking closest vehicle
            if (check_car_s - car_s < closest_car_left_lane &&
                check_car_s - car_s > -0.5 * PLANNING_HORIZON_DISTANCE){
              closest_car_left_lane = check_car_s - car_s;
            }
          } else if (lane - car_lane == -1 ) { // Checking closest vehicle
            if (check_car_s - car_s < closest_car_right_lane  &&
                check_car_s - car_s > -0.5 * PLANNING_HORIZON_DISTANCE){
              closest_car_right_lane = check_car_s - car_s;
            }
          }
          else{
            continue ; // Ignore cars two lane apart from the ego car.
          }
      }

      #ifdef DEBUG_MODE
      std::cout << "Closest vehicle in left lane: " <<  closest_car_left_l
      std::cout << "Closest vehicle in right lane: " <<  closest_car_right
      #endif


      if (too_close) { // Car ahead
        // Considering left and right change based on the distance to the
        // vehicle in those lanes.
        if (closest_car_left_lane > PLANNING_HORIZON_DISTANCE &&
            (closest_car_left_lane > closest_car_right_lane || lane == 2)
            lane != 0) {
          lane--; // Change lane left.
        } else if (closest_car_right_lane > PLANNING_HORIZON_DISTANCE &&
             (closest_car_right_lane > closest_car_left_lane || lane == 0)
             lane != 2) {
          lane++; // Change lane right.
        } else {
          ref_vel -= ref_acc; // Adjust speed with the speed of the vehicl

          if (fabs(ref_acc) > MAX_ACCELERATION)
            ref_acc -= MAX_JERK;
        }
      } else {
        // Adjust the speed to right below speed limit. The adaptive adjus
        // is needed to avoid oscilliation behavior when tail-gating a veh
        if (ref_vel < MAX_SPEED) {
          ref_vel += ref_acc;
          ref_acc += MAX_JERK;
          if (fabs(ref_acc) < MAX_ACCELERATION)
            ref_acc -= MAX_JERK;
```

SUGGESTION

Check out the following code for lane changing.

```
//---------------------------Lane Changer-------------------------------------//
//About: Call function "check_lane" to flag when it is legal to change lane.
            vector<double> change_left {0, 0};
            vector<double> change_right {0, 0};
            double diff_abs = 0.0;
            if(change_lane == true){
              if (host_lane == 0){
                change_right = check_lane(cars, car_s, dist_to_front, 1);
                if(change_right[0] == true){host_lane=1;}
              }
              else if (host_lane == 1){
                change_left = check_lane(cars, car_s, dist_to_front, 0);
                change_right = check_lane(cars, car_s, dist_to_front, 2);

                if(change_left[0] == true && change_right[0] == false){
                  host_lane=0;
                }
                else if(change_left[0] == false && change_right[0] == true){
                  host_lane=2;
                }
                else if (change_left[0] == true && change_right[0] == true){
                  //To avoid uncertainty & wabbling when changing lanes & Distanc
                  //to front cars  are similar.
                  diff_abs = abs(change_left[1]-change_right[1]);
                  if(diff_abs>1.5){
                    if(change_left[1] >= change_right[1]){host_lane=0;}
                    else{host_lane=2;}
                  }
                  else{host_lane=1;}
                }
              }
              else if (host_lane==2){
                change_left = check_lane(cars, car_s, dist_to_front, 1);
                if(change_left[0] == true){host_lane=1;}
              }
            }
```

193 }

SUGGESTION

Below is some code that avoids collisions, you can employ the logic from it.

```
// Decide on Behavior : Let's see what to do.
            double speed_diff = 0;
            const double MAX_SPEED = 49.5;
            const double MAX_ACC = .224;
            if ( car_ahead ) { // Car ahead
              if ( !car_left && lane_num > 0 ) {
                // if there is no car left and there is a left lane.
                lane_num--; // Change lane left.
```

```
          } else if ( !car_right && lane_num != 2 ){
            // if there is no car right and there is a right lane.
            lane_num++; // Change lane right.
          } else {
            speed_diff -= MAX_ACC;
          }
        } else {
          if ( lane_num != 1 ) { // if we are not on the center lane.
            if ( ( lane_num == 0 && !car_right ) || ( lane_num == 2 && !car_l
              lane_num = 1; // Back to center.
            }
          }
          if ( ref_vel < MAX_SPEED ) {
            speed_diff += MAX_ACC;
          }
        }
```

```
          }

          // Reference x, y, and yaw for the car.
          double ref_x = car_x;
          double ref_y = car_y;
          double ref_yaw = deg2rad(car_yaw);

          // If previous size is almost empty, use the car as starting referen
          if (prev_size < 2){
            // Calculate single previous point by tracing ego-motion back
            // by unit-vector, making the two-point path tangent to the ego ca
            double prev_car_x = car_x - cos(car_yaw);
            double prev_car_y = car_y - sin(car_yaw);

            ptsx.push_back(prev_car_x);
            ptsx.push_back(car_x);

            ptsy.push_back(prev_car_y);
            ptsy.push_back(car_y);
          }
          else{
            // Use previous path's end point as starting reference.
            ref_x = previous_path_x[prev_size - 1];
            ref_y = previous_path_y[prev_size - 1];

            double ref_x_prev = previous_path_x[prev_size - 2];
            double ref_y_prev = previous_path_y[prev_size - 2];
            ref_yaw = atan2(ref_y - ref_y_prev, ref_x - ref_x_prev);

            // Use the previous two points as the begining of the points list
            // on which to calculate the spline.
            ptsx.push_back(ref_x_prev);
            ptsx.push_back(ref_x);

            ptsy.push_back(ref_y_prev);
            ptsy.push_back(ref_y);
          }

          // In Frenet coordinate space add evenly spaced by PLANNING_HORIZON_
          // three points ahead of the starting reference point.
          vector<double> next_wp0 = getXY(
```

```
          car_s + PLANNING_HORIZON_DISTANCE * 1,
          2 + 4 * lane,
          map_waypoints_s,
          map_waypoints_x,
          map_waypoints_y
          );
        vector<double> next_wp1 = getXY(
          car_s + PLANNING_HORIZON_DISTANCE * 2,
          2 + 4 * lane,
          map_waypoints_s,
          map_waypoints_x,
          map_waypoints_y
          );
        vector<double> next_wp2 = getXY(
          car_s + PLANNING_HORIZON_DISTANCE * 3,
          2 + 4 * lane,
          map_waypoints_s,
          map_waypoints_x,
          map_waypoints_y
          );

        ptsx.push_back(next_wp0[0]);
        ptsx.push_back(next_wp1[0]);
        ptsx.push_back(next_wp2[0]);

        ptsy.push_back(next_wp0[1]);
        ptsy.push_back(next_wp1[1]);
        ptsy.push_back(next_wp2[1]);

        // Shift reference points to ego-car own coordinate system.
        // Shift car heading to 0 degrees (for everything being in ego coodi
        for (int i = 0; i < ptsx.size(); ++i){
          double shift_x = ptsx[i] - ref_x;
          double shift_y = ptsy[i] - ref_y;

          ptsx[i] = (shift_x * cos(0 - ref_yaw) - shift_y * sin(0 - ref_yaw)
          ptsy[i] = (shift_x * sin(0 - ref_yaw) + shift_y * cos(0 - ref_yaw)
        }

        tk::spline spl; // Spline object.
        // Calculate the resulting spline in ego-car coordinate system.
        // This way it minimizes the instant velocity and acceleration.
        spl.set_points(ptsx, ptsy);

        vector<double> next_x_vals;
        vector<double> next_y_vals;

        for(int i = 0; i < prev_size; ++i){
          next_x_vals.push_back(previous_path_x[i]);
          next_y_vals.push_back(previous_path_y[i]);
        }

        // Calculate how to break up spline points so that we travel at our
        // desired reference velocity.
        double target_x = PLANNING_HORIZON_DISTANCE;
        double target_y = spl(target_x);
        double target_dist = sqrt(target_x * target_x + target_y * target_y)

        double x_add_on = 0;

        // Fill up the rest of our path planner after filling it with prevoi
```

```
        // points, here we will always output PLANNING_NUM_INTEVALS points
        for (int i = 1; i <= PLANNING_NUM_INTEVALS - prev_size; ++i){
          double N = (target_dist / (PLANNING_TICK_INTEVAL * ref_vel / MPH2M

          double x_point = x_add_on + (target_x) / N;
          double y_point = spl(x_point);

          x_add_on = x_point;

          double x_ref = x_point - 0;
          double y_ref = y_point - 0;

          // Rotate back to global coordinate system after rotating it earli
          x_point = (x_ref * cos(ref_yaw - 0) - y_ref * sin(ref_yaw - 0));
          y_point = (x_ref * sin(ref_yaw - 0) + y_ref * cos(ref_yaw - 0));

          x_point += ref_x;
          y_point += ref_y;

          next_x_vals.push_back(x_point);
          next_y_vals.push_back(y_point);
        }

        /** PROJECT CONTRIBUTION END */

        msgJson["next_x"] = next_x_vals;
        msgJson["next_y"] = next_y_vals;

        auto msg = "42[\"control\","+ msgJson.dump()+"]";

        ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
      }  // end "telemetry" if
    } else {
      // Manual driving
      std::string msg = "42[\"manual\",{}]";
      ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
    }
  }  // end websocket if
}); // end h.onMessage

h.onConnection([&h](uWS::WebSocket<uWS::SERVER> ws, uWS::HttpRequest req) {
  std::cout << "Connected!!!" << std::endl;
});

h.onDisconnection([&h](uWS::WebSocket<uWS::SERVER> ws, int code,
                       char *message, size_t length) {
  ws.close();
  std::cout << "Disconnected" << std::endl;
});

int port = 4567;
if (h.listen(port)) {
  std::cout << "Listening to port " << port << std::endl;
} else {
  std::cerr << "Failed to listen to port" << std::endl;
  return -1;
}

h.run();
}
```

- README.md 1
- src/Eigen-3.3/unsupported/Eigen/CXX11/src/Tensor/README.md
- src/Eigen-3.3/demos/opengl/README
- src/Eigen-3.3/demos/mix_eigen_and_c/README
- src/Eigen-3.3/demos/mandelbrot/README
- src/Eigen-3.3/bench/tensors/README
- src/Eigen-3.3/bench/btl/libs/ublas/main.cpp
- src/Eigen-3.3/bench/btl/libs/tvmet/main.cpp
- src/Eigen-3.3/bench/btl/libs/mtl4/main.cpp
- src/Eigen-3.3/bench/btl/libs/gmm/main.cpp
- src/Eigen-3.3/bench/btl/libs/blaze/main.cpp
- src/Eigen-3.3/bench/btl/libs/STL/main.cpp
- src/Eigen-3.3/bench/btl/libs/BLAS/main.cpp
- src/Eigen-3.3/bench/btl/README
- src/Eigen-3.3/README.md

RETURN TO PATH