PROJECT

Path Planning

A part of the Self-Driving Car Engineer Program

PROJECT REVIEW

CODE REVIEW 5

NOTES

src/main.cpp 4

```
#include <fstream>
#include <math.h>
#include <uWS/uWS.h>
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>
#include "Eigen-3.3/Eigen/Core"
#include "Eigen-3.3/Eigen/QR"
#include "json.hpp"
#include "spline.h"

using namespace std;

// for convenience
using json = nlohmann::json;

// For converting back and forth between radians and degrees.
constexpr double pi() { return M_PI; }
double deg2rad(double x) { return x * pi() / 180; }
double rad2deg(double x) { return x * 180 / pi(); }

// Checks if the SocketIO event has JSON data.
// If there is data the JSON object in string format will be returned,
// else the empty string "" will be returned.
string hasData(string s) {
  auto found_null = s.find("null");
  auto b1 = s.find_first_of("[");
```

```
  auto b2 = s.find_first_of("}");
  if (found_null != string::npos) {
    return "";
  } else if (b1 != string::npos && b2 != string::npos) {
    return s.substr(b1, b2 - b1 + 2);
  }
  return "";
}

double distance(double x1, double y1, double x2, double y2)
{
    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}
int ClosestWaypoint(double x, double y, const vector<double> &maps_x, const vector<dou
{

    double closestLen = 100000; //large number
    int closestWaypoint = 0;

    for(int i = 0; i < maps_x.size(); i++)
    {
        double map_x = maps_x[i];
        double map_y = maps_y[i];
        double dist = distance(x,y,map_x,map_y);
        if(dist < closestLen)
        {
            closestLen = dist;
            closestWaypoint = i;
        }

    }

    return closestWaypoint;

}

int NextWaypoint(double x, double y, double theta, const vector<double> &maps_x, const
{

    int closestWaypoint = ClosestWaypoint(x,y,maps_x,maps_y);

    double map_x = maps_x[closestWaypoint];
    double map_y = maps_y[closestWaypoint];

    double heading = atan2((map_y-y),(map_x-x));

    double angle = fabs(theta-heading);
  angle = min(2*pi() - angle, angle);

  if(angle > pi()/4)
  {
    closestWaypoint++;
  if (closestWaypoint == maps_x.size())
  {
    closestWaypoint = 0;
  }
  }

  return closestWaypoint;
}

```

```
// Transform from Cartesian x,y coordinates to Frenet s,d coordinates
vector<double> getFrenet(double x, double y, double theta, const vector<double> &maps_
{
    int next_wp = NextWaypoint(x,y, theta, maps_x,maps_y);

    int prev_wp;
    prev_wp = next_wp-1;
    if(next_wp == 0)
    {
        prev_wp  = maps_x.size()-1;
    }

    double n_x = maps_x[next_wp]-maps_x[prev_wp];
    double n_y = maps_y[next_wp]-maps_y[prev_wp];
    double x_x = x - maps_x[prev_wp];
    double x_y = y - maps_y[prev_wp];

    // find the projection of x onto n
    double proj_norm = (x_x*n_x+x_y*n_y)/(n_x*n_x+n_y*n_y);
    double proj_x = proj_norm*n_x;
    double proj_y = proj_norm*n_y;

    double frenet_d = distance(x_x,x_y,proj_x,proj_y);

    //see if d value is positive or negative by comparing it to a center point

    double center_x = 1000-maps_x[prev_wp];
    double center_y = 2000-maps_y[prev_wp];
    double centerToPos = distance(center_x,center_y,x_x,x_y);
    double centerToRef = distance(center_x,center_y,proj_x,proj_y);

    if(centerToPos <= centerToRef)
    {
        frenet_d *= -1;
    }

    // calculate s value
    double frenet_s = 0;
    for(int i = 0; i < prev_wp; i++)
    {
        frenet_s += distance(maps_x[i],maps_y[i],maps_x[i+1],maps_y[i+1]);
    }

    frenet_s += distance(0,0,proj_x,proj_y);

    return {frenet_s,frenet_d};

}

// Transform from Frenet s,d coordinates to Cartesian x,y
vector<double> getXY(double s, double d, const vector<double> &maps_s, const vector<d
{
    int prev_wp = -1;

    while(s > maps_s[prev_wp+1] && (prev_wp < (int)(maps_s.size()-1) ))
    {
        prev_wp++;
    }

    int wp2 = (prev_wp+1)%maps_x.size();

```

```
    double heading = atan2((maps_y[wp2]-maps_y[prev_wp]),(maps_x[wp2]-maps_x[prev_wp]
    // the x,y,s along the segment
    double seg_s = (s-maps_s[prev_wp]);

    double seg_x = maps_x[prev_wp]+seg_s*cos(heading);
    double seg_y = maps_y[prev_wp]+seg_s*sin(heading);

    double perp_heading = heading-pi()/2;

    double x = seg_x + d*cos(perp_heading);
    double y = seg_y + d*sin(perp_heading);

    return {x,y};

}

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;

  //Global variable placeholder for intended velocity of car based on constraints, pr
  double intended_velocity = 0.0;
```

AWESOME

well done setting this value to `0.0`. This accounts for the safe kickoff of the car during simulation.

```

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

  string line;
  while (getline(in_map_, line)) {
    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);
  }

  h.onMessage([&intended_velocity, &map_waypoints_x,&map_waypoints_y,&map_waypoints_s
                     uWS::OpCode opCode) {
    // "42" at the start of the message means there's a websocket message event.
    // The 4 signifies a websocket message
    // The 2 signifies a websocket event
    //auto sdata = string(data).substr(0, length);
    //cout << sdata << endl;
    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 r
            auto sensor_fusion = j[1]["sensor_fusion"];

            json msgJson;

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

            //Constants for current simulator environment
            // Width of lane in meters
            const double lane_width = 4.0;
            // Time taken by simulator to travel from current to next waypoint - 20 m
            const double simulator_reach_time = 0.02;
            //Converter to convert velocity from mph to m/s
            const double velocity_mph_to_ms_conv = 1609.344 / 3600;

            //Speed limit constraints
            //Speed limit
            const double safe_speed_limit = 48 * velocity_mph_to_ms_conv;
            //Minimum speed to ensure path smoother spline library gets coordinates i
            const double minimum_speed_limit = 3 * velocity_mph_to_ms_conv;

            //Safe distance between cars constraints
            //Safe distance ahead of our car
```

```
const int safe_range_ahead = 30;
//Safe distance behind our car. This is used in lane shift
const int safe_range_behind = 15;

//Static variable for intended lane for car.
//1. 0 for leftmost lane
//2. 1 for middle lane
//3. 2 for rightmost lane
static float lane_id = 1.0;

//Number of waypoints left for previous set of path planner points
int previous_size = previous_path_x.size();

//Looking forward in time
if (previous_size > 0) {
  car_s = end_path_s;
}

/** PREDICTION COMPONENT
  Detects presence of cars ahead in current and ahead and behind in other
  determined distance range
*/
//Flag for prediction of cars in current lane of car and other lanes
bool is_car_ahead = false;
bool is_car_left = false;
bool is_car_right = false;

//Loop in sensor fusion data which has information on location and velocit
for (int i = 0; i < sensor_fusion.size(); i++) {
  double o_car_vx = sensor_fusion[i][3];
  double o_car_vy = sensor_fusion[i][4];
  double o_car_s = sensor_fusion[i][5];
  double o_car_d = sensor_fusion[i][6];
  float o_car_lane;

  if (o_car_d > 0 && o_car_d < lane_width) {
    o_car_lane = 0.0;
  } else if (o_car_d > lane_width && o_car_d < (lane_width * 2)) {
    o_car_lane = 1.0;
  } else if (o_car_d > (lane_width * 2) && o_car_d < (lane_width * 3)) {
    o_car_lane = 2.0;
  } else {
    o_car_lane = -1.0;
  }
```

AWESOME

Nice job here. Impressive logic here to come about smart lane change by ego car in this section.

```
  //Not interested if cars are not on the same side of road/divider
    if (o_car_lane == -1) {
        continue;
    }

  //Calculate the velocity and predicted Frenet s coordinate of car
  double o_car_vel = sqrt(pow(o_car_vx, 2) + pow(o_car_vy, 2));
  double o_car_s_ahead = o_car_s + (o_car_vel * simulator_reach_time * pre

  //If other car is in the same lane
```

```
            if (o_car_lane == lane_id) {
              //If car is getting closer than the safe range
              if ((o_car_s_ahead > car_s) && ((o_car_s_ahead - car_s) < safe_range_a
                is_car_ahead = true;
              }
            } //If other car is the lane right of our car
            else if ((o_car_lane - lane_id) == 1) {
              //If car is getting closer than the safe range either from behind or i
                  if (((o_car_s_ahead > car_s) && ((o_car_s_ahead - car_s) < safe_ra
                      ((car_s > o_car_s_ahead) && ((car_s - o_car_s_ahead) < safe_ra
                is_car_right = true;
              }
              } //If other car is the lane left of our car
            else if ((o_car_lane - lane_id) == -1) {
              //If car is getting closer than the safe range either from behind or i
                  if (((o_car_s_ahead > car_s) && ((o_car_s_ahead - car_s) < safe_ra
                  ((car_s > o_car_s_ahead) && ((car_s - o_car_s_ahead) < safe_range_
                is_car_left = true;
              }
            }
          }
```

AWESOME

Impressive speed control logic in this section. It was well perceived in this project

```

          /** BEHAVIOR PLANNER COMPONENT
            Deducts the correct behavior the car should follow. Following are the de
            1. Continue in current lane and accelerate reaching speed limit
            2. Slow down in current lane in order to avoid collision with car ahead
            3. Change lane to left with current speed if not in leftmost lane
            4. Change lane to right with current speed if not in rightmost lane
          */

          //Car ahead is getting closer
          if (is_car_ahead) {
            //If right lane shift is possible and our car is not in rightmost lane
              if ((!is_car_right) && (lane_id != 2)) {
                  lane_id += 1;
              } //If left lane shift is possible and our car is not in leftmost lane
            else if ((!is_car_left) && (lane_id != 0)) {
                  lane_id -= 1;
              } //No lane change is possible, decelerate by 0.5mph or 0.22 m/s
            else {
                  intended_velocity -= 0.5 * velocity_mph_to_ms_conv;
              }
          } //No car is ahead and the road is clear in current lane, accelerate at (
            else {
              intended_velocity += 0.5 * velocity_mph_to_ms_conv;
          }

          //Cap the speed of car to safe speed limit slightly less than speed limit
          if (intended_velocity >= safe_speed_limit) {
            intended_velocity = safe_speed_limit;
          }
          //Minimum speed of car is ensured to avoid spline library exception
          if (intended_velocity <= minimum_speed_limit) {
            intended_velocity = minimum_speed_limit;
```

```
          }

          /** PATH SMOOTHER ALGORITHM
            Derives path of car with waypoints ahead in time. This path is then fed
            1. Create anchor points for spline library. Spline takes the anchor poi
            2. Feed anchor points to spline and derive waypoints for lookahead dist
            3. Feed the waypoints to simulator
          */

          //Anchor points for spline in global coordinates
          std::vector<double> anchor_x;
          std::vector<double> anchor_y;

          //Anchor points for spline in local coordinates
          std::vector<double> anchor_x_local;
          std::vector<double> anchor_y_local;

          //Step 1 - Start point is car's current position or previous path
          double current_yaw_rad;
          double tmp_x_1;
          double tmp_y_1;
          double tmp_x_2;
          double tmp_y_2;

          //If more than 2 waypoints passed to simulator in previous iteration are
          // use them to ensure smooth transition to next set of waypoints
          if (previous_size > 2) {
            tmp_x_2 = previous_path_x[previous_size - 2];
            tmp_y_2 = previous_path_y[previous_size - 2];
            tmp_x_1 = previous_path_x[previous_size - 1];
            tmp_y_1 = previous_path_y[previous_size - 1];

            anchor_x.push_back(tmp_x_2);
            anchor_y.push_back(tmp_y_2);
            anchor_x.push_back(tmp_x_1);
            anchor_y.push_back(tmp_y_1);

            current_yaw_rad = atan2(tmp_y_1 - tmp_y_2, tmp_x_1 - tmp_x_2);
          } //Car has almost travelled every waypoint from previous iteration,
          // use car's current location and one waypoint backward in time for smoot
          else {
            anchor_x.push_back(car_x - cos(car_yaw));
            anchor_y.push_back(car_y - sin(car_yaw));
            anchor_x.push_back(car_x);
            anchor_y.push_back(car_y);
            current_yaw_rad = deg2rad(car_yaw);
          }

          //Step 2 - Set lookahead distance and anchors
          //This is 30 meters
          double lookahead_weight = 30;
          int num_lookahead_steps = 3;

          //Step 3 - Use car's frenet coordinates to get lookahead frenets and conv
          double tmp_lookahead_s = 0.0;
          double tmp_lookahead_d = 0.0;
          std::vector<double> tmp_global_xy;
          for (int i = 0; i < num_lookahead_steps; i++) {
              tmp_lookahead_s = car_s + ((i + 1) * lookahead_weight);
              tmp_lookahead_d = (lane_id * lane_width) + (lane_width/2);
              tmp_global_xy = getXY(tmp_lookahead_s, tmp_lookahead_d, map_waypoints_
```

```
        anchor_x.push_back(tmp_global_xy[0]);
        anchor_y.push_back(tmp_global_xy[1]);
      }

      //Step 4 - Convert anchor points to local coordinates in order to feed it
      //generate waypoints along the path to anchor

      double tmp_diff_x;
      double tmp_diff_y;
      double tmp_local_x;
      double tmp_local_y;

      for (int i = 0; i < anchor_x.size(); i++) {
        //Shift axes
        tmp_diff_x = anchor_x[i] - anchor_x[0];
        tmp_diff_y = anchor_y[i] - anchor_y[0];

        //Rotate axes
        tmp_local_x = tmp_diff_x * cos(-current_yaw_rad) - tmp_diff_y * sin(-cu
        tmp_local_y = tmp_diff_x * sin(-current_yaw_rad) + tmp_diff_y * cos(-cu

        anchor_x_local.push_back(tmp_local_x);
        anchor_y_local.push_back(tmp_local_y);
      }

      //Step 5 - Initialize a spline and set local anchor points to it
      tk::spline sp;
      sp.set_points(anchor_x_local, anchor_y_local);

      //Step 7 - Create waypoints in local coordinate  system
      // i. Determine the number of waypoints that can fit between 2 anchor poi
      //         using velocity and the lookahead distance
      // ii. Generate x value on the same straight line as vehicle x
      // iii. Determine y value from the spline curve

      double minimum_distance_simulator = intended_velocity * simulator_reach_t
      int num_waypoints = sqrt(pow(lookahead_weight, 2) + pow(sp(lookahead_weig
      int waypoint_steps = 30;

      std::vector<double> waypoints_x_local;
      std::vector<double> waypoints_y_local;
      double waypoint_x;
      double waypoint_y;

      for (int i = 0; i < waypoint_steps - previous_size; i++) {
        //Rotate axes
        waypoint_x = anchor_x_local[1] + (i + 1) * lookahead_weight / num_waypo
        waypoint_y = sp(waypoint_x);

        //Shift axes
        waypoints_x_local.push_back(waypoint_x);
        waypoints_y_local.push_back(waypoint_y);
      }

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

      //Step 8 - Convert waypoints from local to global coordinates

```

```
          for (int i = 0; i < waypoint_steps - previous_size; i++) {
            waypoint_x = waypoints_x_local[i] * cos(current_yaw_rad) - waypoints_y_
            waypoint_y = waypoints_x_local[i] * sin(current_yaw_rad) + waypoints_y_
            waypoint_x += anchor_x[0];
            waypoint_y += anchor_y[0];

            next_x_vals.push_back(waypoint_x);
            next_y_vals.push_back(waypoint_y);
          }

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

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

          //this_thread::sleep_for(chrono::milliseconds(1000));
          ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);

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

// We don't need this since we're not using HTTP but if it's removed the
// program
// doesn't compile :-(
h.onHttpRequest([](uWS::HttpResponse *res, uWS::HttpRequest req, char *data,
                   size_t, size_t) {
  const std::string s = "<h1>Hello world!</h1>";
  if (req.getUrl().valueLength == 1) {
    res->end(s.data(), s.length());
  } else {
    // i guess this should be done more gracefully?
    res->end(nullptr, 0);
  }
});

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();
```

AWESOME

Great documentation in this section. It goes a long way to improve code readability. Nice practice.

```
554 }
555
```

README.md 1

src/Eigen-3.3/unsupported/Eigen/CXX11/src/Tensor/README.md

src/Eigen-3.3/test/bug1213_main.cpp

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/README.md

src/Eigen-3.3/demos/opengl/README

src/Eigen-3.3/demos/mix_eigen_and_c/README

src/Eigen-3.3/bench/btl/README

RETURN TO PATH

Student FAQ