Path Planning Project

Readme Explanation

You're reading it now!

Project Explanation

This project is to show a simple algorithm going around a track in a simulated world. Fused sensor inputs are taken in to the algorithm and used to define what lanes are safe, what reasonable speeds of the vehicle is, etc.

Code documentation

Setup

```
1  #include <fstream>
2  #include <math.h>
3  #include <uWS/uWS.h>
4  #include <chrono>
5  #include <iostream>
6  #include <thread>
7  #include <vector>
8  #include "Eigen-3.3/Eigen/Core"
9  #include "Eigen-3.3/Eigen/QR"
10 #include "json.hpp"
11 #include "spline.h"
12
13
14 using namespace std;
15
16 // for convenience
17 using json = nlohmann::json;
18
19 // For converting back and forth between radians and degrees.
20 constexpr double pi() { return M_PI; }
21 double deg2rad(double x) { return x * 0.0174532925199433; } //pi/180
22 double rad2deg(double x) { return x * 57.2957795130823; } //180/pi
23
24 // Checks if the SocketIO event has JSON data.
25 // If there is data the JSON object in string format will be returned,
26 // else the empty string "" will be returned.
27 string hasData(string s) {
28   auto found_null = s.find("null");
29   auto b1 = s.find_first_of("[");
30   auto b2 = s.find_first_of("}");
31   if (found_null != string::npos) {
32     return "";
33   } else if (b1 != string::npos && b2 != string::npos) {
34     return s.substr(b1, b2 - b1 + 2);
35   }
36   return "";
37 }
38
39 double distance(double x1, double y1, double x2, double y2)
40 {
41   return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
42 }
```

Lines 1 through 42 are fairly straightforward, defining libraries, namespaces, and the json function. Lines 21 and 22 convert between degrees and radians. Lines 27 through 37 are a check to determine whether a string has data, and the function distance is defined.

Lines 42 through 55 show a function ‘ClosestWaypoint’ which is fairly self explanatory, but for the sake of being verbose I’ll say it loops through the waypoints on the map and finds the waypoint that is closest to the x and y input into the function.

```
42 int ClosestWaypoint(double x, double y, vector<double> maps_x, vector<double> maps_y) {
43   double closestLen = 100000; //large number
44   int closestWaypoint = 0;
45   for(int i = 0; i < maps_x.size(); i++) {
46     double map_x = maps_x[i];
47     double map_y = maps_y[i];
48     double dist = distance(x,y,map_x,map_y);
49     if(dist < closestLen) {
50       closestLen = dist;
51       closestWaypoint = i;
52     }
53   }
54   return closestWaypoint;
55 }
```

Lines 57 through 67 show the function ‘NextWaypoint’ which calls closest waypoint and then searches for the waypoint following that one.

```
57 int NextWaypoint(double x, double y, double theta, vector<double> maps_x, vector<double> maps_y) {
58   int closestWaypoint = ClosestWaypoint(x,y,maps_x,maps_y);
59   double map_x = maps_x[closestWaypoint];
60   double map_y = maps_y[closestWaypoint];
61   double heading = atan2( (map_y-y),(map_x-x) );
62   double angle = abs(theta-heading);
63   if(angle > 0.785398163) {
64     closestWaypoint++;
65   }
66   return closestWaypoint;
67 }
```

Lines 70 through 102 are the function getFrenet, which returns frenet s and d from inputs of x, y, theta, maps_x and maps_y.

```
69 // Transform from Cartesian x,y coordinates to Frenet s,d coordinates
70 vector<double> getFrenet(double x, double y, double theta,
71                          vector<double> maps_x, vector<double> maps_y) {
72   int next_wp = NextWaypoint(x,y, theta, maps_x,maps_y);
73   int prev_wp;
74   prev_wp = next_wp-1;
75   if(next_wp == 0)  {
76     prev_wp  = maps_x.size()-1;
77   }
78   double n_x = maps_x[next_wp]-maps_x[prev_wp];
79   double n_y = maps_y[next_wp]-maps_y[prev_wp];
80   double x_x = x - maps_x[prev_wp];
81   double x_y = y - maps_y[prev_wp];
82   double proj_norm = (x_x*n_x+x_y*n_y)/(n_x*n_x+n_y*n_y);   // find the projection of x onto n
83   double proj_x = proj_norm*n_x;
84   double proj_y = proj_norm*n_y;
85   double frenet_d = distance(x_x,x_y,proj_x,proj_y);
86   double center_x = 1000-maps_x[prev_wp];
87   double center_y = 2000-maps_y[prev_wp];
88   double centerToPos = distance(center_x,center_y,x_x,x_y);
89   double centerToRef = distance(center_x,center_y,proj_x,proj_y);
90
91   if(centerToPos <= centerToRef) {
92     frenet_d *= -1;
93   }
94
95   // calculate s value
96   double frenet_s = 0;
97   for(int i = 0; i < prev_wp; i++) {
98     frenet_s += distance(maps_x[i],maps_y[i],maps_x[i+1],maps_y[i+1]);
99   }
100  frenet_s += distance(0,0,proj_x,proj_y);
101  return {frenet_s,frenet_d};
102 }
```

Lines 105 to 121 transform a set of coordinates from the Frenet s and d (distance along the path and distance lateral to the path) into the x and y Cartesian coordinates. It is the inverted function from the prior function.

```
104 // Transforms from Frenet s,d coordinates to Cartesian x,y
105 vector<double> getXY(double s, double d, vector<double> maps_s,
106                      vector<double> maps_x, vector<double> maps_y)
107 {
108   int prev_wp = -1;
109   while(s > maps_s[prev_wp+1] && (prev_wp < (int)(maps_s.size()-1) ))   {
110     prev_wp++;
111   }
112   int wp2 = (prev_wp+1)%maps_x.size();
113   double heading = atan2((maps_y[wp2]-maps_y[prev_wp]),(maps_x[wp2]-maps_x[prev_wp]));
114   // the x,y,s along the segment
115   double seg_s = (s-maps_s[prev_wp]);
116   double seg_x = maps_x[prev_wp]+seg_s*cos(heading);
117   double seg_y = maps_y[prev_wp]+seg_s*sin(heading);
118   double perp_heading = heading-pi()/2;
119   double x = seg_x + d*cos(perp_heading);
120   double y = seg_y + d*sin(perp_heading);
121   return {x,y};
122 }
```

Lines 123 to 157 start setting up the main function, loading up the map, and starting to define functions and waypoints.

```
123 int main()
124 {
125   uWS::Hub h;
126 
127   // Load up map values for waypoint's x,y,s and d normalized normal vectors
128   vector<double> map_waypoints_x;
129   vector<double> map_waypoints_y;
130   vector<double> map_waypoints_s;
131   vector<double> map_waypoints_dx;
132   vector<double> map_waypoints_dy;
133 
134   string map_file_ = "../data/highway_map.csv";     // Waypoint map to read from
135   double max_s = 6945.554;                          // The max s value before wrapping around the track back to 0
136 
137   ifstream in_map_(map_file_.c_str(), ifstream::in);
138   std::cout << round(2.3) ;
139   string line;
140   while (getline(in_map_, line))
141   {
142     istringstream iss(line);
143     double x;
144     double y;
145     float s;
146     float d_x;
147     float d_y;
148     iss >> x;
149     iss >> y;
150     iss >> s;
151     iss >> d_x;
152     iss >> d_y;
153     map_waypoints_x.push_back(x);
154     map_waypoints_y.push_back(y);
155     map_waypoints_s.push_back(s);
156     map_waypoints_dx.push_back(d_x);
157     map_waypoints_dy.push_back(d_y);
158   }
```

Lines 161 to 172 define some variables – lane, reference velocity, some times (useful to calculate acceleration and jerk, but less useful than one might think - the calculation I was doing for accel and jerk is not perfectly the same as the one done by the simulator).

```
int lane = 1;
double ref_vel = 4;
time_t startTime = time(0);
time_t elapsedTime = time(0);
time_t deltaTime = time(0);
time_t now = time(0);
int     counter =  0;
double accel = 0;
double oldAccel = 0;
double oldVel = 0;
double jerk = 0;
bool    verbose = false;
```

Lines 174 to 205 define the lambda function and variables passed to the lambda function, read in the first data message, and define some of the variables that we'll be using, like the car's X, Y , S, and D parameters, Yaw, speed, velocity, etc.

```
174  h.onMessage([&now, &startTime, &elapsedTime, &deltaTime, &accel, &oldAccel,
175               &oldVel, &jerk, &counter,
176               &ref_vel, &map_waypoints_x,&map_waypoints_y,
177               &map_waypoints_s,&map_waypoints_dx,&map_waypoints_dy,
178               &lane, &verbose] (uWS::WebSocket<uWS::SERVER> ws, char *data,
179               size_t length, uWS::OpCode opCode) {
180
181    if (length && length > 2 && data[0] == '4' && data[1] == '2') {  // "42" is the answer to everything.
182      auto s = hasData(data);
183      if (s != "") {
184        auto j = json::parse(s);
185        string event = j[0].get<string>();
186        if (event == "telemetry") {
187          // j[1] is the data JSON object
188          // Our vehicle's Data
189          double carX = j[1]["x"];
190          double carY = j[1]["y"];
191          double carS = j[1]["s"];
192          double carD = j[1]["d"];
193          double carYaw = j[1]["yaw"];
194          double carSpeed = j[1]["speed"];
195          double vel = 0.44704*carSpeed; //.44704 = m/s in a MPH
196          double desFollowDist = 12;
197          double maxAccelRef = 10;
198          double desSpeed;
199          double trailModeSpeed;
200          bool    trailMode = false;
201          bool    tailgating = false;
202          bool    lane0IsOK = true;
203          bool    lane1IsOK = true;
204          bool    lane2IsOK = true;
205          bool    lane3IsOK = false;    // driving on the berm works!
```

Please keep in mind that the car driving on the berm works!

Lines 207 to 217 start calculating my own calculations fo jerk and acceleration. These are less useful than originally desired in part because the time function only gives information in seconds, which is not useful. Other functions could be used but even ignoring these problems the calculations weren't lining up with the simulator – various scale values had to be used – so other techniques were used to avoid having problems with acceleration or velocity.

```
207          deltaTime = time(0)-now;    //less useful than desired.  This function only gives seconds, we need ms.
208          now = time(0);
209          elapsedTime = now-startTime;
210          counter++;
211          accel = 15*(vel-oldVel);  // about 15 iterations per second
212          jerk =  15*(accel-oldAccel);
213          oldVel = vel;
214          oldAccel = accel;
215
216          std::cout << std::fixed;
217          std::cout << std::setprecision(2);
```

Lines 220 through 225 define some previous path data…

```
// Previous path data
auto previousPathX = j[1]["previous_path_x"];// Previous paths X values
auto previousPathY = j[1]["previous_path_y"];// Previous paths Y values
double endPathS = j[1]["end_path_s"];        // Previous path's end s values
double endPathD = j[1]["end_path_d"];        // Previous path's end d values
auto sensor_fusion = j[1]["sensor_fusion"];  // Sensor Data, with all other cars on this side of road.
int prev_size = previousPathX.size();
```

Lines 228 to 243 define carS, and then starts to loop through each sensor's data of vehicle detection and starts to define the other cars variables – like x velocity, y velocity, s, d, etc.

```
228  if (prev_size > 0) {
229    carS = endPathS;
230  }
231
232  for (int i=0; i<sensor_fusion.size(); i++) { // cycle through each car on the road, check each lane to s
233
234    double otherVx = sensor_fusion[i][3];
235    double otherVy = sensor_fusion[i][4];
236    double otherS = sensor_fusion[i][5];
237    double otherD = sensor_fusion[i][6];
238    double otherVel = sqrt(otherVx*otherVx+otherVy*otherVy);
239    double relativeVel = otherVel - vel;        // this would come in handy for some slightly more sophist
240    int    otherLane = round(.25*otherD-.5);
241    bool   startThinkingAboutOtherLanes = abs(otherS-carS);
242    bool   tooClose = abs(otherS-carS) < 30;    // 22 is tailgating distance, abs(s-carS) is distance betw
243    bool   otherCarInFront = otherS-carS > -15;
```

Lines 245 to 268 go through what happens if a car that we're following is too close – first we set the flag that we're tailgating, then we set whatever lane that vehicle is in as off limits. Finally we output some debugging data if the verbose flag is set to true.

```
245    if(otherLane == lane && tooClose  ) {
246      tailgating = true;
247      trailModeSpeed = 2.5*otherVel;  //2.237 MPH in a m/s.  But otherVel / carSpeed seems to be about 2.
248      if (verbose) {
249        std::cout << "Tailgating:  ";
250      }
251    }
252
253    if( otherLane == 0 && tooClose ) {  // if the car is one lane to my left
254      lane0IsOK = false;
255    }
256    else if( otherLane == 1 && tooClose) {
257      lane1IsOK = false;
258    }
259    else if( otherLane == 2 && tooClose) {
260      lane2IsOK = false;
261    }
262
263    if(verbose)  {
264      std::cout << i << " " << lane << " " << lane0IsOK << " " << lane1IsOK << " " << lane2IsOK << " "
265              << carD << " " << carS << " " << ref_vel << " "<< otherS << " " << otherD << " " << otherLa
266              << relativeVel << " " << otherVel << " " << vel << " " << trailModeSpeed << " "
267              << carSpeed << " " << tooClose << " " << otherCarInFront << "\n";
268    }
269  }
```

Liens 270 to 286 ask, first, if we're tailgating, and if we are, then goes through some simple logic to check what other lanes might be possible, and if any lanes are possible, move into one of them.

```
            if (tailgating) {                                  //if the previous for loop found a car in the lane, do somet
              if (lane == 0 && lane1IsOK ) {
                lane = 1;
              }
              else if (lane == 1 && lane0IsOK) {
                lane = 0;
              }
              else if (lane == 1 && lane2IsOK) {
                lane = 2;
              }
              else if (lane == 2 && lane1IsOK) {
                lane = 1;
              }
              else {
                trailMode = true;
              }
            }
```

Lines 289 to 324 set the speed of the vehicle. In the first few iterations of the simulation, we sest the reference velocity to 6 and then add in some other terms. After 20 iterations, we go to a max acceleration scheme. As we get closer to the speed limit we slow down. Separately, if trailmode is active, we slow down.

```
289             if (trailMode == false) {
290               if(counter < 2) {
291                   ref_vel = 6;
292               }
293               else if(counter < 20) {
294                   ref_vel = 6 + counter*0.15 + counter*counter*.02;
295               }
296
297               else if(ref_vel < 38) {
298                 //double coefficient = 2.23694*deltaTime; //this is what it should be... m/s to MPH times deltaTime
299                 double coefficient = .08;
300                 ref_vel = coefficient * maxAccelRef + ref_vel; //should be actual velocity but that only gets update
301               }
302               else if (ref_vel < 49.7) {
303                 ref_vel = ref_vel + 0.1;
304               }
305               else if (ref_vel > 49.9 ) {
306                 ref_vel = 49.8;
307               }
308             }
309
310             if (trailMode == true)  {
311               if(verbose) {
312                 std::cout << "trailMode: " << trailModeSpeed << " " << carSpeed << " " << ref_vel ;
313
314               }
315               if (carSpeed>(trailModeSpeed - .2)) { // slow down to follow the leading car
316                 //ref_vel = 0.5*(carSpeed + (trailModeSpeed-.2));
317                 ref_vel = ref_vel - .2;
318                 std::cout << " Slow to " << ref_vel;
319               }/*
320               else {  //could mean acceleration in some corner cases
321                 ref_vel = trailModeSpeed-.1;
322                 std::cout << "\n Match " << ref_vel << "\n";
323               }            */
324             }
325
```

Lines 326 through 328 are primarily used to debug the algorithm, outputting several intermediate steps to the author.

Lines 330 to 361 establish ptsx and ptsy, some of the referenc x, y, and zy, and start assembling the variables to send to the simulator.

Lines 364, 365, and 366 get the X and Y from the car S.

```
326         std::cout << counter << " " << elapsedTime << " " << deltaTime << " " << lane << " "
327                   << carD << " " << carS << " " << trailMode << " " << tailgating << " "
328                   << vel << " " << accel << " " << jerk << " " << ref_vel << "\n";
329
330         // create a list of widely spaced (x,y) waypoints, evenly spaced at 30m
331         vector<double> ptsx;
332         vector<double> ptsy;
333
334         // reference x, y, yaw states
335         double ref_x = carX;
336         double ref_y = carY;
337         double ref_yaw = deg2rad(carYaw);
338
339         if(prev_size < 2) {
340           double prev_carX = carX - cos(carYaw);
341           double prev_carY = carY - sin(carYaw);
342
343           ptsx.push_back(prev_carX);
344           ptsx.push_back(carX);
345           ptsy.push_back(prev_carY);
346           ptsy.push_back(carY);
347         }
348         else {
349           ref_x = previousPathX[prev_size-1];
350           ref_y = previousPathY[prev_size-1];
351
352           double ref_x_prev = previousPathX[prev_size-2];
353           double ref_y_prev = previousPathY[prev_size-2];
354           ref_yaw = atan2(ref_y-ref_y_prev, ref_x-ref_x_prev);
355
356           ptsx.push_back(ref_x_prev);
357           ptsx.push_back(ref_x);
358
359           ptsy.push_back(ref_y_prev);
360           ptsy.push_back(ref_y);
361         }
362
363         // append another 3 points at the end of previous path
364         vector<double> next_wp0 = getXY(carS+30, (2+4*lane),map_waypoints_s, map_waypoints_x, map_waypoints_y);
365         vector<double> next_wp1 = getXY(carS+60, (2+4*lane),map_waypoints_s, map_waypoints_x, map_waypoints_y);
366         vector<double> next_wp2 = getXY(carS+90, (2+4*lane),map_waypoints_s, map_waypoints_x, map_waypoints_y);
```

Lines 368 to 397 are equivalent to ones given in the video and start assembling the values to give back to the simulator.

```
368         ptsx.push_back(next_wp0[0]);
369         ptsx.push_back(next_wp1[0]);
370         ptsx.push_back(next_wp2[0]);
371
372         ptsy.push_back(next_wp0[1]);
373         ptsy.push_back(next_wp1[1]);
374         ptsy.push_back(next_wp2[1]);
375
376         for (int i=0; i<ptsx.size(); i++) { // move car reference to zero degrees
377           double shift_x = ptsx[i]-ref_x;
378           double shift_y = ptsy[i]-ref_y;
379           ptsx[i] = (shift_x*cos(0-ref_yaw)-shift_y*sin(0-ref_yaw));
380           ptsy[i] = (shift_x*sin(0-ref_yaw)+shift_y*cos(0-ref_yaw));
381         }
382
383         tk::spline s;                                   // spline curve
384         s.set_points(ptsx, ptsy);                       // fit spline
385         vector<double> next_x_vals;
386         vector<double> next_y_vals;
387
388         for (int i=0; i<previousPathX.size(); i++) {    // add the previuos path for a smooth transition
389           next_x_vals.push_back(previousPathX[i]);
390           next_y_vals.push_back(previousPathY[i]);
391         }
392
393         // keep extending the previous path
394         double target_x = 30.0; // m
395         double target_y = s(target_x);
396         double target_dist = sqrt(target_x*target_x + target_y*target_y);
397         double x_add_on = 0;
```

Lines 400 to 430 finish up adding the points and send the message to the simulator. Then the lambda function is concluded (line 430).

```
399             // 50 more waypoints
400             for (int i=1; i<=50-previousPathX.size();i++) {
401               double N = target_dist/(.02*ref_vel/2.24); // number of intervals
402               double x_point = x_add_on+target_x/N;
403               double y_point = s(x_point);
404               double x_ref = x_point;
405               double y_ref = y_point;
406 
407               x_add_on = x_point;
408               // need to translate x and y back to original coordinates
409               x_point = x_ref*cos(ref_yaw) - y_ref*sin(ref_yaw);
410               y_point = x_ref*sin(ref_yaw) + y_ref*cos(ref_yaw);
411               x_point += ref_x;
412               y_point += ref_y;
413               next_x_vals.push_back(x_point);
414               next_y_vals.push_back(y_point);
415             }
416 
417           json msgJson;
418           msgJson["next_x"] = next_x_vals;
419           msgJson["next_y"] = next_y_vals;
420           auto msg = "42[\"control\","+ msgJson.dump()+"]";
421           //this_thread::sleep_for(chrono::milliseconds(1000));
422           ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
423           }
424         }
425         else {     // Manual driving
426           std::string msg = "42[\"manual\",{}]";
427           ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
428         }
429       }
430     });
```

The remainder of the file is verbatim from examples given and is not included here.

Conclusions and Further Thoughts

The car drives around the track.

There are some corner cases that could be improved. The acceleration allowed is not calculated for all conditions and instead is dealt with empirically in some cases. Finding the best lane is not done in all circumstances and in fact should be dealt with earlier as basically all cars' position on the road is know early. Even further, we could predict some simple movements of other cars and then calculate out what we want to do from there.

All in all it works but it's fairly ugly.