Path Planning Project

The goal of this project is developing a code to safely navigate a simulated highway. Safety assumes:

1) The car doesn't drive faster than the speed limit of 50 mph. Also the car isn't driving much slower than speed limit unless obstructed by traffic.
2) The car does not exceed a total acceleration of 10 m/s^2 and a jerk of 10 m/s^3.
3) The car must not come into contact with any of the other cars on the road.
4) The car doesn't spend more than a 3 second length outside the lane lanes during changing lanes, and every other time the car stays inside one of the 3 lanes on the right hand side of the road.
5) The car is able to smoothly change lanes when it makes sense to do so, such as when behind a slower moving car and an adjacent lane is clear of other traffic.

Simulator

The simulator allows driving the vehicle by setting a path as a list of points in Cartesian coordinates. Each point is a location of vehicle in 20 ms from previous point. The simulator also provides current vehicle coordinates in Cartesian and Frenet coordinates, as well as unprocessed points from the previously set path.

Map

Project template comes with a map of the simulator's highway. The map is represented by a list of waypoints on the middle of the highway (line between highway directions). Each waypoint has an (x,y) global map position, a Frenet s value and Frenet d unit normal vector (split up into the x component, and the y component).

I have chosen to perform path planning in Frenet coordinates. As simulator requires points in Cartesian coordinates I use the map to convert points in Frenet coordinates from planner to Cartesian coordinates. The number of waypoints is insufficient for smooth conversion thus I approximate the waypoints by a polynomial spline, map.cpp:

```
52      //
53      // Set splines
54      //
55      x_spline.set_points(waypoint_s, waypoint_x);
56      y_spline.set_points(waypoint_s, waypoint_y);
57      dx_spline.set_points(waypoint_s, waypoint_dx);
58      dy_spline.set_points(waypoint_s, waypoint_dy);
59
```

```cpp
/**
 * Convert Frenet coordinates to Cartesian
 *
 * @param s     - s Frenet coordinate
 * @param d     - d Frenet coordinate
 * @param x     - output Cartesian x coordinate
 * @param y     - output Cartesian y coordinate
 */
void Map::get_cartesian(double s, double d, double &x, double &y)
{
    x = x_spline(s) + d * dx_spline(s);
    y = y_spline(s) + d * dy_spline(s);
}
```

Path Planning

I perform path planning in Frenet coordinates because using Frenet coordinates simplifies planning as path does not depend on the road curvature.

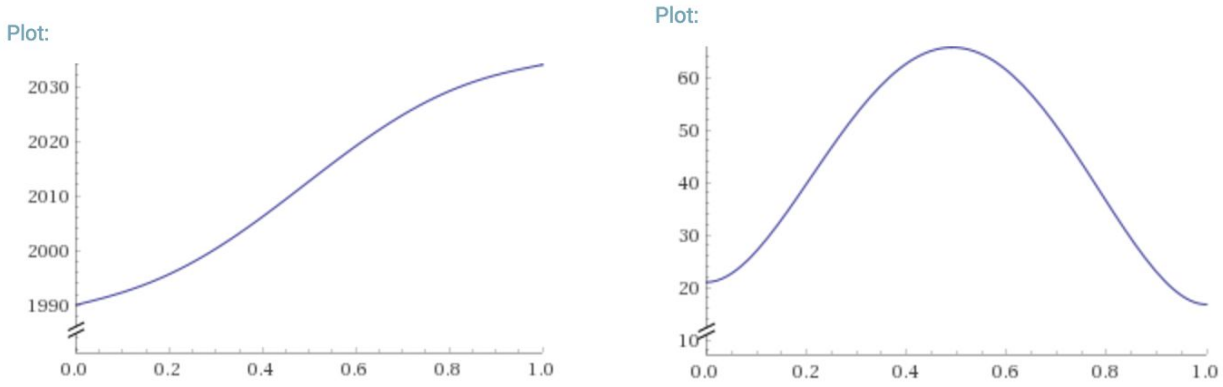
The planner is implemented as Planner class (planner.h/cpp) and uses current vehicle's state, such as position, velocity and acceleration, to define the target state (state at one second ahead). The target state is selected to meet constraints such as speed limit, acceleration limit, avoiding collisions, driving within lanes, changing lanes to overtake obstacles.

The target state is used to generate jerk-minimized trajectories by 5th order polynomial approximation (trajectory.h/cpp). The approximated trajectories are used to generate trajectory points that complement previous trajectory points, main.cpp:

```cpp
//
// Plan a new trajectory in Frenet
//
planner.update_state(s, v_s, a_s, d, v_d, a_d, objects);
Trajectory s_trajectory = planner.get_s_trajectory();
Trajectory d_trajectory = planner.get_d_trajectory();

//
// Add new trajectory points
//
for (size_t i = 0; i < 50 - points_from_prev_trajectory; i++) {
    //
    // get Frenet coordinates
    //
    double s_i = s_trajectory((i + 1) * tic_time);
    double d_i = d_trajectory((i + 1) * tic_time);
    s_i = map.normalize_s(s_i);

    //
    // Convert to Cartesian
    //
    double x, y;
    map.get_cartesian(s_i, d_i, x, y);

    //
    // Add points to the lists
    //
    next_x_vals.push_back(x);
    next_y_vals.push_back(y);

    current_s.push_back(s_i);
    current_d.push_back(d_i);
}
```

Using few previous trajectory points is required to deal with variable latency of controls as well as for smoothing the path. For example, I tried using reported current car position and it led to not evenly distributed path points for desired velocity and many issues with exceeding speed and acceleration limits.

Initially, I considered to use least previous path points but it would not work with jerk-minimized trajectories because they normally require full trajectory duration. For example, below is the figure of trajectory, left - position, right - velocity.



As you may see velocity raises first half of the trajectory. If I use only small part of this trajectory, e.g. 0.3 seconds of it, then velocity will only increase, assuming I use 0.3 seconds of the next trajectories as well. Such trajectories usually generated when a hard break is required due to small distance from the leading vehicle. Hence, instead of speed drop vehicle only accelerates and collides with an obstacle.

To avoid this I use all points from the previous trajectory. When less than 15 points left, a new trajectory is generated, main.cpp:

```
153                     //
154                     // if previous trajectory size is greater or equal to 15 then use previous trajectory
155                     //
156                     if ( points_from_prev_trajectory < 15 ) {
157                         //
158                         // if previous trajectory size is less than 15 then plan new trajectory
159                         //
```

Using previous path requires computing current position, velocity and acceleration from the latest previous path points, as illustrated by the main.cpp code fragment below.

```
184                     //
185                     // Compute latest positions, velocity and accelerations
186                     //
187                     double s = car_s, v_s = prev_v_s, a_s = prev_a_s;
188                     double d = car_d, v_d = prev_v_d, a_d = prev_a_d;
189                     //
190                     // If previous trajectory size exceeds 3 then we can compute postion, velocity and acceleration
191                     // otherwise use previous values
192                     //
193                     if ( points_from_prev_trajectory >= 3 ) {
194                         //
195                         // s position, velocity, and acceleration
196                         //
197
198                         // position
199                         s = prev_s[points_from_prev_trajectory - 1];
200                         double s_1 = prev_s[points_from_prev_trajectory - 2];
201                         double s_2 = prev_s[points_from_prev_trajectory - 3];
202
203                         // velocity
204                         v_s = map.diff_s(s, s_1) / tic_time;
205
206                         // acceleration
207                         double vs_perv = map.diff_s(s_1, s_2) / tic_time;
208                         a_s = (v_s - vs_perv) / tic_time;
209
210                         //
211                         // d position, velocity, and acceleration
212                         //
213
214                         // position
215                         d = prev_d[points_from_prev_trajectory - 1];
216                         double d_1 = prev_d[points_from_prev_trajectory - 2];
217                         double d_2 = prev_d[points_from_prev_trajectory - 3];
218
219                         // velocity
220                         v_d = ( d - d_1 ) / tic_time;
221
222                         // acceleration
223                         double vd_perv = ( d_1 - d_2 ) / tic_time;
224                         a_d = ( v_d - vd_perv ) / tic_time;
```

Prior to calling the path planner, I retrieve information on other vehicles from the sensor fusion data. The other vehicles are treated generally as “objects” assuming there could be non-vehicle objects, such as pedestrians and etc.

```
161                     //
162                     // Retrieve objects from sensor fusion data
163                     //
164                     vector<Object> objects;
165                     for (size_t i = 0; i < sensor_fusion.size(); i++) {
166                         auto &entry = sensor_fusion[i];
167
168                         Object::params params;
169                         params.d = entry[6];
170                         // ignore all objects outside of the road
171                         if ( map.is_road(params.d) ) {
172
173                             params.type = Object::type_vehicle;
174                             params.id = entry[0];
175                             params.s = entry[5];
176                             double vx = entry[3];
177                             double vy = entry[4];
178                             map.get_frenet_velocity(params.s, vx, vy, params.vs, params.vd);
179                             params.s += params.vs * points_from_prev_trajectory * tic_time;
180                             objects.push_back(Object(params));
181                         }
182                     }
```

The Object class (object.cpp/h) is used to maintain object’s parameters and generate predictions - future positions of the objects, object.cpp:

```cpp
/**
 * Predict future states of an object
 *
 * @param horizon       - prediction horizon, number of time steps
 * @param delta_t       - the duration of a time step
 *
 * @return              - list of future states, size equal to prediction horizon
 */
vector<Object::state> Object::predict(size_t horizon, double delta_t)
{
    vector<state> result;

    double s = current_position.s;
    double d = current_position.d;
    double vs = current_position.vs;
    double vd = current_position.vd;
    //
    // first element - is the current state
    //
    result.push_back({s, vs, d});

    //
    // Compute object's states up to prediction horizon
    //
    for ( size_t i = 0; i < horizon; i++ ) {
        //
        // assume constant velocity
        //
        s += vs * delta_t;
        d += vd * delta_t;
        //
        // add updated state to the list
        //
        result.push_back({s, vs, d});
    }

    return result;
}
```

Lane Changing

The planning is essentially selecting a lane and defining the next target state. In order to select a lane I use a simple algorithm which enumerates over the three options, such as (1) keep lane, (2) change lane to the left, (3) change lane to the right. For each action I compute a score and then select an action with the highest score, planner.cpp:

```cpp
        //
        // select an action with maximum score
        //
        double max_score = 0;
        //
        // By default - keep lane
        //
        best_action = act_keep_lane;
        //
        // Enumerate all actions
        //
        for ( int action = 0; action < act_max; action++ ) {
```

In order to evaluate the score I validate action applicability and initialize target state by calling Planner::set_action() method in planner.cpp:

```
72                  //
73                  // initialize score and vehicle state
74                  //
75                  double score = 0;
76                  vehicle_state vehicle = current_state;
77                  //
78                  // set action target state, such as s, d, velocities and accelerations
79                  //
80                  if ( set_action(action, vehicle, 0) ) {
```

Action may not be applicable if lane change leads to outside of the road (target lane number is less than 1 and greater than 3); or if it leads to a collision. For an applicable action I compute the score based on distance the vehicle may proceed with the target velocity, planner.cpp:

```
81                  //
82                  // No collision detected at the current step, proceed with next time steps up to prediction horizon
83                  //
84                  vehicle.s = target_s;
85                  vehicle.vs = target_vs;
86                  for ( size_t i = 1; i <= prediction_horizon; i++ ) {
87                      //
88                      // find a closest object in front
89                      //
90                      Object::state front_object;
91                      if ( get_closest_object_state(vehicle, i, front_object) ) {
92                          //
93                          // object found, check for collision via distance between our vehicle and object
94                          //
95                          double distance = map->diff_s(front_object.s, vehicle.s);
96                          if (distance >= -vehicle_length && distance < vehicle_length) {
97                              //
98                              // treat vehicle size distance as a collision
99                              //
100                             //
101                             // in case of collision reset score to zero
102                             //
103                             score = 0;
104                             break;
105                         }
106                     }
107                     //
108                     // Compute next stat of vehicle
109                     //
110                     vehicle.s += vehicle.vs;
111                     //
112                     // Increase score by distance that vehicle traveled
113                     //
114                     score += vehicle.vs;
115                 }
```

If a collision is detected then the score is reset to zero. To check for collisions I use predicted positions of other vehicles initialized earlier in planner.cpp:

```
45      size_t prediction_horizon = 3;
46      for ( size_t i = 0; i < objects.size(); i++ ) {
47          predictions.push_back(objects[i].predict(prediction_horizon, 1));
48      }
```

A closest object in front is selected and its position is verified against the planned position of our vehicle assuming all drive with the constant speed. The score is evaluated for the prediction horizon of three seconds with a step of one second.

To avoid unnecessary lane changes I penalize it unless score difference exceeds vehicle size, planner.cpp:

```
117                 //
118                 // penalize lane change
119                 //
120                 if ( action != act_keep_lane )
121                     score -= vehicle_length;
```

I also disallow lane changing while yet in transition to a new lane and when vehicle is in dangerously short distance to its lead, planner.cpp:

```
54      if ( last_action != act_keep_lane && current_state.lane != target_lane ) {
55          //
56          // keep action while in transition to a new lane
57          //
58          best_action = last_action;
```

Defining Target State

Planner::set_trajectory_targets() method sets s-axis target position, velocity and acceleration at one second in future. Initially the method adjusts maximum velocity along s-axis to take in account maximum velocity along d-axis (multiplied by two), planner.cpp:

```
271     //
272     // Adjust maximum velocity to account d axis velocity
273     //
274     double vd = get_d_trajectory().get_max_velocity(1) * 2;
275     double max_velocity_s = sqrt(max_velocity * max_velocity - vd * vd);
```

Then it looks up for closest front object, and sets higher possible velocity if no object is detected, planner.cpp:

```
277     //
278     // Retrieve closest object in front
279     //
280     if ( get_closest_object_state(vehicle, time_step, front_object) == false ) {
281         //
282         // no obstacles, go with maximum velocity
283         //
284         target_vs = min(vehicle.vs + max_acceleration, max_velocity_s);
285         target_as = target_vs - vehicle.vs;
286         target_s = vehicle.s + vehicle.vs + target_as / 2;
```

Target velocity is a minimum of velocity increase by maximum acceleration and maximum velocity. Acceleration is computed as difference of target and current velocities. The target position is: current_position + velocity + target_acceleration/2.

If an object is detected ahead I compute time to a collision if it suddenly stops while ours moves with the constant current velocity (sudden_stop_time_distance), planner.cpp:

```
288         //
289         // A front object is found, measure distance in time
290         // This is time to collision if front object suddenly stops while ours moves with constant velocity
291         //
292         double sudden_stop_time_distance;
293         double distance = front_object.s - vehicle.s;
294         sudden_stop_time_distance = vehicle.vs > 1e-3 ? distance / vehicle.vs : 10;
```

Then I proceed with defining the target state depending on that time to collision, planner.cpp:

```
295         //
296         // Update targets according to the time distance
297         //
298         if ( sudden_stop_time_distance < 0.75 ) {
299             //
300             // Keep distance by dropping speed
301             //
302             target_as = min(max_velocity_s, front_object.vs) - 2 - vehicle.vs;
303             target_vs = vehicle.vs + target_as;
304             target_s = vehicle.s + vehicle.vs + target_as / 2;
305         }
306         else if ( sudden_stop_time_distance < 1 ) {
307             //
308             // Set speed smaller than lead
309             //
310             target_as = ( min(max_velocity_s, front_object.vs) - 1 - vehicle.vs );
311             target_vs = vehicle.vs + target_as;
312             target_s = vehicle.s + vehicle.vs + target_as / 2;
313         }
314         else if ( sudden_stop_time_distance < 1.5 ) {
315             //
316             // Adjust speed to match lead
317             //
318             target_as = ( min(max_velocity_s, front_object.vs) - vehicle.vs );
319             target_vs = vehicle.vs + target_as;
320             target_s = vehicle.s + vehicle.vs + target_as / 2;
321         }
322         else {
323             //
324             // Obstacle is far enough - go as fast as can
325             //
326             target_vs = min(vehicle.vs + max_acceleration, max_velocity_s);
327             target_as = target_vs - vehicle.vs;
328             target_s = vehicle.s + vehicle.vs + target_as / 2;
329         }
```

First, I compute acceleration from the desired velocity difference and then velocity and position. In case if object is farther than 1.5 seconds to a collision, I proceed with the maximum speed as if there are no objects ahead.

For d-axis I set target position to the lane center, while the velocity and acceleration to zero. The trajectory time is adjusted assuming 1m/s velocity, planner.cpp:

```
368         //
369         // set target velocity and acceleration to zero
370         //
371         double target_vd = 0;
372         double target_ad = 0;
373
374         //
375         // get d by target lane (target lane is in current state as it gets updated in update_state())
376         //
377         target_d = map->get_lane_center(current_state.lane);
378         //
379         // set trajectory duration assuming maximum velocity of 1 m/s
380         //
381         double diff = target_d - current_state.d;
382         double time = max(1.0, fabs(diff) / 1);
```

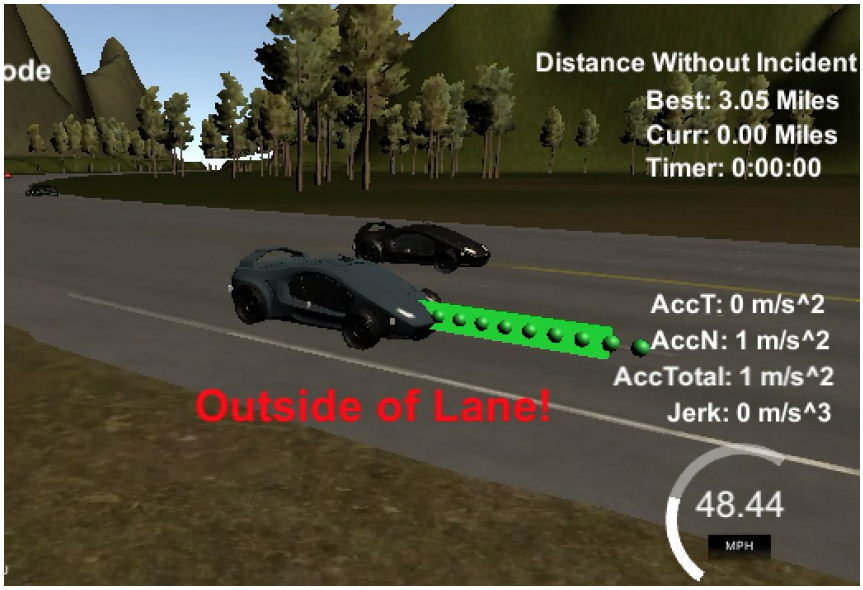
Issues

1) Due to limited number of waypoints and spline approximation, my Frenet to Cartesian conversion is not precise and resulting x,y coordinates do not exactly match actual x,y. This may create an issue when previous Frenet trajectory is not available and speed is low as at start. At low speed a slight discrepancy leads to quick yaw change. I fixed that by adding a new waypoint for the initial vehicle location at the corresponding position of CSV map file:
909.480 1134.834833 124.8336 0.0 -1.0

Another solution is not using converted x,y explicitly but fitting a 3d order polynomial with fixed initial and final coordinates. This solution should also resolve other issues with Frenet to Cartesian conversion errors. This should be further improvement of this project.

2) I observed speed limit violation which I don't measure in Frenet coordinates. Small variations are possible due to jerk-minimization polynomial but I didn't measure it either. I assume this could also be due to errors in Frenet to Cartesian conversions or imperfection of the simulator controller, rather former. I fixed this issue by setting planner's speed limit to 46 mph instead of 50 mph.

3) I regularly observed "out of lane" violations at 3.05 mile while vehicle is at third lane.



I have recorded the video and could not see the vehicle is actually out of lane. It appears to be Frenet to Cartesian error as well. I fixed it by slightly adjusting the d coordinate of the third lane by setting it to 9.8 instead of 10, map.h:

```
81      /**
82       * Get d coordinate of lane center
83       *
84       * @param lane   - number of lane (1..3)
85       *
86       * @return       - d coordinate of the lane
87       */
88      double get_lane_center(int lane)
89      {
90          return lane == 3 ? 9.8 : get_lane_width() * ( lane - 0.5 );
91      }
```

4) To simplify handling of the map edge I have added to CSV map file a waypoint for the initial position but with s coordinate equal to the track length:
784.6001 1135.571 6945.554 -0.02359831 -0.9997216

5) The lane changing algorithm does not select from the all available lanes, only to the left or right from the current lane. This may lead to overlooked “free” lane which is farther from the current lane. For example, current lane is 3 and “free” lane is 1. Considering changing to all lanes should fix the issue. This is planned as further improvement.