Path Planner

Write up of work done for Udacity CarND Term 3, Path Planner Project, by ***Mani Srinivasan***

Project Objectives

The goal of this project is the following:

> *To design a path planner that is able to create smooth, safe paths for the car to follow along a 3-lane highway with traffic. A successful path planner will be able to keep inside its lane, avoid hitting other cars, and pass slower moving traffic all by using localization, sensor fusion, and map data.*

Included in the submission is this write-up that details how the project was completed as well as the GitHub repository containing the source code of the files, a link to the YouTube video recording.

This write-up includes the rubric points with a description of how I addressed each point. I have included a snippet of the code used in each step (with line-number references and code snippets where appropriate) and links to other supporting documents or external references.

Acknowledgements:

For completion, I have applied the key concepts in the course as well as some of the ideas from the solution walk thru' by Udacity instructors Aaron Brown and David Silver, especially for the generation of waypoints (YouTube video reference here: [https://www.youtube.com/watch?v=3QP3hJHm4WM&feature=youtu.be)](https://www.youtube.com/watch?v=3QP3hJHm4WM&feature=youtu.be)

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Compilation

The code compiles correctly.

Code must compile without errors with cmake and make.

See the output of the cmake and make steps below.

```
Manis-MacBook-Pro:Term3 srnimani$ cd CarND-PP-Final
Manis-MacBook-Pro:CarND-PP-Final srnimani$ mkdir build
Manis-MacBook-Pro:CarND-PP-Final srnimani$ cd build
Manis-MacBook-Pro:build srnimani$ cmake .. && make
-- The C compiler identification is AppleClang 8.1.0.8020042
-- The CXX compiler identification is AppleClang 8.1.0.8020042
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /Library/Developer/CommandLineTools/usr/bin/c++
-- Check for working CXX compiler: /Library/Developer/CommandLineTools/usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/srnimani/Desktop/Udacity/CarND/Term3/CarND-PP-Final/build
Scanning dependencies of target path_planning
[ 50%] Building CXX object CMakeFiles/path_planning.dir/src/main.cpp.o
[100%] Linking CXX executable path_planning
[100%] Built target path_planning
Manis-MacBook-Pro:build srnimani$
```

Valid Trajectories

The Criteria

The car is able to drive at least 4.32 miles without incident.

The top right screen of the simulator shows the current/best miles driven without incident. Incidents include exceeding acceleration/jerk/speed, collision, and driving outside of the lanes. Each incident case is also listed below in more detail.

The Car is able to drive without any incident. I have tested the path planner on the simulator provided on my laptop for more than **35 minutes** in which the car covered **more than 28 miles** without any incident.

The car drives according to the speed limit

The car doesn't drive faster than the speed limit. Also, the car isn't driving much slower than speed limit unless obstructed by traffic.

The car's speed is limited to a maximum of 49 MPH and the waypoint generation ensures that the speed limit is not exceeded. The car starts from 0 and ramp ups to a maximum speed gradually (@ acceleration of 9 m/sec2 against the maximum of 10 m/sec2, *ref_velocity of 0.4*). When there is a car

ahead within a distance of 30 meters and there is no available lane to switch to, the car slows down to match the speed of the car ahead. Otherwise, the car drives at the maximum speed set.

Lines 424 thru' 581 in the main file main.cpp, implement a finite state machine that goes thru' the following states:

-STARTING, -KL, -PLCL, -LCR, -PLCR, -LCL

The following lines/ code snippets takes care of the speed

State: -STARTING

(lines 434 thru' 454 of main.cpp)

```
case _STARTING: // Car just starting
{
    if (car_ahead_is_close(car_s, current_lane, prev_size, velocity_of_car_ahead))
    {
        ref_velocity -= 0.3;
        PP_next_state = _STARTING ; // Maintain same state
    }
    else if (ref_velocity < max_velocity) // No car close, so ramp up speed
    {
        ref_velocity += 0.4 ;
        PP_next_state = _STARTING ; // Maintain same lane
    }
    else
    {
        ref_velocity  = max_velocity ; // maximum speed
        PP_next_state = _KL ; // switch to next state
    }
    next_lane = current_lane ;
    cout << "Starting up.. new speed:\t " << ref_velocity << endl;
    break;
}
```

The above snippet of code implements the starting state of the car till the maximum speed is reached. If there is any obstruction during this period the car slows down to ensure that there is no collision.

State: -KL

(lines 498 thru' 509 of main.cpp)

```
case -1: // No free lanes, maintain lane at reduced speed
{
    if (ref_velocity > velocity_of_car_ahead)
    {
        ref_velocity -= 0.3 ; // reduce speed
        if (ref_velocity < velocity_of_car_ahead)
            ref_velocity = velocity_of_car_ahead; // no need to reduce below the speed of the car ahead
        cout << "Can't switch lanes now.. staying in the same lane at the same speed of the car ahead\t" << "Lane :\t
            " << current_lane << endl;
    }
    PP_next_state = _KL ;
    break;
}
```

The above snippet of code implements the deceleration of the car (@ 6.7 m/sec2), when there is no lane to switch to, till the car speed reaches the one in front.

The waypoints generation code snippet below ensures that the spacing of the waypoints (which controls the speed of the car) the car does not accelerate or speed beyond limits.

```
// Calculate how to break up spline points so that the car travels @ the desired speed

double target_x     = 30;
double target_y     = s(target_x);
double target_dist  = sqrt((target_x * target_x) + (target_y * target_y));

double x_add_on = 0;

// Now fill up the rest of the points for our path planner after filling up with the previous points

for (int i = 0; i <= 50 - previous_values; i++)
{
    double N = (target_dist / (0.02 * ref_velocity/ 2.24)); // MPH to m/s
    double x_point = x_add_on + (target_x) / N;
    double y_point = s(x_point);

    x_add_on = x_point;

    double x_ref = x_point;
    double y_ref = y_point;

    // Rotate back to normal after rotating it earlier
    x_point = (x_ref * cos(ref_yaw) - y_ref*sin(ref_yaw));
    y_point = (x_ref * sin(ref_yaw) + y_ref*cos(ref_yaw));

    x_point += ref_x;
    y_point += ref_y;

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

Max Acceleration and Jerk are not Exceeded

The car does not exceed a total acceleration of 10 m/s^2 and a jerk of 10 m/s^3.

As explained in the code snippets above, the car acceleration (9 m/sec2) and deceleration (6.9 m/sec2) are set within limits and ensure that there is no jerk or over acceleration/ deceleration.

Car does not have collisions

The car must not come into contact with any of the other cars on the road.

The finite-state machine (lines 424 thru' 581 in the main file main.cpp), implements a collision detection algorithm and ensures that the car is able to manoeuvre around cars in front as well as the ones in the back within a distance of 45 meters while trying to pass ahead of the cars ahead. The car switches lanes if possible or stay in the same lane by driving at a reduced speed.

```
cout << "Car ahead is going slow and we are closing in fast.. check if we can swich lanes.. " << endl;
free_lane = available_lane(car_s, current_lane);
cout << "Free lane :\t" << free_lane << endl;

switch (free_lane)
{
    case 0: // Left lane is free
    {
        PP_next_state = _PLCL ;
        cout << "Switching to left lane.. current_lane: \t " << current_lane << endl;
        break;
    }

    case 1:
    {
        if (last_lane == 0) // we are moving from left most lane
            PP_next_state = _PLCR ;
        else PP_next_state = _PLCL ; // we are moving from right lane
        break;
    }

    case 2:
    {
        PP_next_state = _PLCR ;
        cout << "Switching to right lane.. current_lane: \t " << current_lane << endl;
        break;
    }

    case -1: // No free lanes, maintain lane at reduced speed
    {
        if (ref_velocity > velocity_of_car_ahead)
        {
            ref_velocity -= 0.3 ; // reduce speed
            if (ref_velocity < velocity_of_car_ahead)
                ref_velocity = velocity_of_car_ahead; // no need to reduce below the speed of the car ahead
            cout << "Can't switch lanes now.. staying in the same lane at the same speed of the car ahead\t" << "Lane :\t
                " << current_lane << endl;
        }
        PP_next_state = _KL ;
        break;
    }
    default: // Maintain lane at reduced speed
    {
        if (ref_velocity > velocity_of_car_ahead)
        {
            ref_velocity -= 0.3 ; // reduce speed
            if (ref_velocity < velocity_of_car_ahead)
                ref_velocity = velocity_of_car_ahead; // no need to reduce below the speed of the car ahead
            cout << "Default state .. how did we reach here???....." << endl;
        }
        PP_next_state = _KL ;
        break;
    }
```

The code in the function (*car_ahead_is_close*) below checks for car in front at a distance less than 30 meters.

```
// Car ahead of us, within 'collision distance'..

bool car_ahead_is_close(double our_car_s, int our_lane, int prev_size, double &velocity_of_car_ahead)
{
    int number_of_cars  = cars.size();

    bool too_close      = false;

    for(int i = 0; i < number_of_cars; ++i)
    {
        if(our_lane == get_lane_number(cars[i].d))
        {
            double check_speed  = cars[i].get_car_velocity(); // get the speed of the car ahead
            double check_car_s  = cars[i].s;

            check_car_s += ((double)prev_size * 0.02 * check_speed);

            // check s values greater than our car and s gap

            if ((check_car_s > our_car_s) && (check_car_s - our_car_s) < safe_distance)
            {
                too_close = true;
                velocity_of_car_ahead = check_speed;
                return too_close;
            }
        }
    }
    return too_close ;
}
```

The car stays in its lane, except for the time between changing lanes

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.

The finite state machine implemented above ensures that the car stays within the 3 lanes and stays in the same lane at the maximum speed possible (49 MPH), unless there is a car in front at less than 30 meters and switching of lanes if possible to pass the slow car ahead.

The car is able to change lanes

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.

As mentioned earlier, the finite state machine (FSM) is able to sense the presence of car at a close distance and switch lanes if there is no chance of a collision. The additional function given below (lines 240 thru 281) looks for free space or gap in adjacent lanes and informs the FSM that it is safe to change lanes if there is no chance of a collision either with the cars ahead or behind.

```
// Check available lane to switch to, return -1 if no lane available
int available_lane(double car_s, const int our_lane)
{
    int free_lane           = -1 ; // Means no lane free
    double left_free_space  =  0 ; // Assume no space to start with
    double right_free_space =  0 ; // Assume no space to start with

    cout << "Safe Distance: \t" << safe_distance << endl;

    switch (our_lane)

    {
        case 0 : // We are in the left most lane
        {
            right_free_space = check_space(car_s, our_lane + 1); // returns the distance to the closest car on right
            if (right_free_space > safe_distance) free_lane = our_lane + 1;
            break;
        }

        case 1 : // we are in the middle lane lane
        {
            right_free_space = check_space(car_s, our_lane + 1); // returns the distance to the closest car on right
            left_free_space = check_space(car_s, our_lane - 1) ; // returns the distance to the closest car on left
            if ((right_free_space > safe_distance * 1.5) && (right_free_space > left_free_space)) free_lane = our_lane + 1;
            if ((left_free_space > safe_distance * 1.5) && (left_free_space >= right_free_space)) free_lane = our_lane - 1;
            break;
        }

        case 2 : // We are in the right most lane
        {
            left_free_space = check_space(car_s, our_lane - 1); // returns the distance to the closest car on left
            if (left_free_space > safe_distance) free_lane = our_lane - 1;
            break;
        }

        default:
        {
            break;
        }
    }
    return free_lane;
}
```

As can be seen in the snippet of code above, the car switches to a lane that has the maximum gap if it is stuck in the middle lane behind a slow car. In extreme lanes, the car looks for a clear gap (of 45 meters) in the adjacent lane before switching.

The function below returns the maximum available gap to facilitate the selection of the right lane.

```
double check_space(double car_s, double lane_to_check)
{
    double freespace = 1000000 ; // assume free space to start with
    double current_gap = 0;

    double s1, s2;  // segment from s1 to s2 to check

    s1 = car_s - safe_distance;
    s2 = car_s + safe_distance;

    if ((s1 < 0) || (s2 > track_length)) // currently not allowing any passes around the zero crossing
    {
        return 0 ; // declare no free space or gap
    }

    for(int i = 0; i < cars.size(); ++i)
    {
        if (lane_to_check == get_lane_number(cars[i].d))
        {
            current_gap = abs(cars[i].s - car_s) ;
            if (current_gap < safe_distance) // lane is unsafe
            {
                freespace = current_gap ;
                cout << "Car_id:\t" << i << "\tlane:\t" << lane_to_check << "\tFree Sapce:\t " << freespace << endl;
                return freespace; // Exit as we have found a close car, no need to continue checking
            }
            else
            {
                if (freespace > current_gap) freespace = current_gap;
            }
        }
    }
    return freespace;
}
```

Reflection

Details on how to generate paths.

The generation of waypoints is based on the required speed of the car and the lane selected. The waypoints use the previous path points as well as new points based on the car's current position and yaw (or angle). I ensured the smoothness of the trajectory by using spline, instead of using polynomials for trajectory generation, due to spline's ease of use and effectiveness.

Shifting of the coordinates to the car's position and angle made the math simpler.

The use of previous waypoints ensured a smooth transition without any jerk or over acceleration. As mentioned at the beginning, the walk thru' video on the project helped in arriving at the right mix of new and previous points and a smooth trajectory.

The code snippet (lines 590 thru' 710 in main.cpp) covers the waypoint generation of the path planner.

```
vector<double> ptsx;
vector<double> ptsy;

// Refernce x, y, yaw states

double ref_x    = car_x             ;
double ref_y    = car_y             ;
double ref_yaw  = deg2rad(car_yaw)  ;

if (prev_size < 2)
{
    // Use 2 points that make the path tangent to the car

    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);

}
// use the previous path's points as starting reference
else
{
    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];

    // Use 2 points that make the path tangent to the previous path's end points

    ptsx.push_back(ref_x_prev);
    ptsx.push_back(ref_x);

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

}
```

```
vector<double> next_wp0 = getXY(car_s + 30, (2 + 4 * next_lane), map_waypoints_s, map_waypoints_x, map_waypoints_y);
vector<double> next_wp1 = getXY(car_s + 60, (2 + 4 * next_lane), map_waypoints_s, map_waypoints_x, map_waypoints_y);
vector<double> next_wp2 = getXY(car_s + 90, (2 + 4 * next_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]);


for (int i = 0; i < ptsx.size(); i++)
{
    // Shift the car reference angle to 0 degrees.. simplifies the math..

    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));

}

// Create a spline path with the points generated

tk::spline s ;

s.set_points(ptsx, ptsy);


// Define the actual points that will be used by the planner

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

int previous_values = previous_path_x.size() ;

// Start with all of the previous points from last time, not discarding..

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

}
```

Finally, the generation of the way points:

```
// Calculate how to break up spline points so that the car travels @ the desired speed

double target_x     = 30;
double target_y     = s(target_x);
double target_dist  = sqrt((target_x * target_x) + (target_y * target_y));

double x_add_on = 0;

// Now fill up the rest of the points for our path planner after filling up with the previous points

for (int i = 0; i <= 50 - previous_values; i++)
{
    double N = (target_dist / (0.02 * ref_velocity/ 2.24)); // MPH to m/s
    double x_point = x_add_on + (target_x) / N;
    double y_point = s(x_point);

    x_add_on = x_point;

    double x_ref = x_point;
    double y_ref = y_point;

    // Rotate back to normal after rotating it earlier
    x_point = (x_ref * cos(ref_yaw) - y_ref*sin(ref_yaw));
    y_point = (x_ref * sin(ref_yaw) + y_ref*cos(ref_yaw));

    x_point += ref_x;
    y_point += ref_y;

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

The following is the link to the YouTube video showing 2 mts of the car on the simulator.

https://youtu.be/cPHqe0hpLWY
