Project 6: Highway Driving

The goal of this project is to design a path planning algorithm to drive car on a ~ 4-mile stretch of a highway on a Udacity simulator. The simulator sends position and speed coordinates of other cars on the road and in return, it expects a set of x,y points that represents car's trajectory spaced at 0.02s

Code Requirements fulfilled as per the Rubric

- **Code compilation without errors**
- **Car is able to drive at least 4.32 miles without errors** – I tested the algorithm on the simulator for more than 5 miles and recorded no errors
- **The car drives according to speed limit** – Max speed limit of car is set to 49.5 mph as suggested in Q&A video. The car drives around that speed when no slow moving traffic is in the vicinity. When slow moving vehicle is in front, the car slows down or changes lane depending on the cars in adjacent lanes
- **Max acceleration and jerk are not exceeded** – Car is set to increase speed by 0.224mph as suggested in the Q&A video, this keeps the acceleration of the car around 5m/s2 which is below the max acceleration limit of 10m/s2. As a result, max acceleration and jerk limits are not exceeded throughout the drive.
- **Car does not have collisions** – Car does not have any collision with any other vehicle on the road because a logic for lane changing or slowdown is implemented in the code that is dependent on where are the cars in adjacent lanes located
- **Car stays in the lane except for the time between changing lanes** – car stays in the lane as long as there is no vehicle within 30m in the front. If the front vehicle is too close, then it looks for an opportunity to change lanes. If lane changing is not possible, then the car slows down until vehicle in front is at a safe distance.
- **The car is able to change lanes** – the car changes lanes if there is a vehicle in front of the car and it is safe to change lanes.

Reflection

Path planning algorithm is implemented in the main.cpp from lines 114 to 339. Main logic to drive car on the highway while meeting criteria listed above is implemented in following steps

1. Cold start (*lines 63-71*) –
    - Initialize current lane of the car (*lane*) as 1 and initial velocity (*ref_val*) as zero. Then slowly increase speed of the car by 0.224mph until its speed is lower than 49.5mph.

```
//define lane
int lane = 1;

//define reference target velocity in mph
double ref_val = 0; // setting initial ref speed to zero to avoid initial jerk at cold start
if(ref_val < 49.5)
{
  ref_val += 0.224; //initial accelerate by 5m/s2
}
```

2. Detecting cars in the all three lanes (*lines 132-197*)
    - For every car in the sensor fusion list, define the car's lane (*car_lane*) depending upon distance d from the center

        - 0<d<4 = car_lane, 0
        - 4<d<8 = car_lane, 1
        - 8<d<12 = car_lane, 2
    - From the data in the sensor fusion list, extract x and y speed of the car (*vx* and *vy*), calculate absolute speed of every car and save it in variable *check_speed.* This is used in predicting where the car will be in future
    - Save Frenet coordinate s for every car in *check_car_s*. This is used to check if the car is too close to us in the same lane or when changing lanes. If using previous points, we can project s value outward in time.
    - In this section, we define three scenarios
        - If the car in the list is in my lane (this is identified based on the Frenet coordinate d of the car), and it is in front (this is done when *check_car_s >car_s*), and the distance is less than 30m. – then this car is too close to our car and we set the variable *too_close* to true.
        - If the car is in the left lane (identified based on d), and its within 30m of my car's position in either front or rear (i.e. distance between my car and the car in left lane is less than 30m) – then this *car_left_lane* flag is set to true
        - If the car is in the right lane (identified based on d), and its within 30m of my car's position in either front or rear (i.e. distance between my car and the car in left lane is less than 30m) – then this *car_right_lane* flag is set to true

```
bool too_close = false;
bool car_same_lane = false;
bool car_left_lane = false;
bool car_right_lane = false;
double check_speed;

//find ref_v to use
for (int i =0; i < sensor_fusion.size(); i++) //go through sensor fusion list
{
  //determine if car is in my lane
  float d = sensor_fusion[i][6];// sensor_fusion[i] is data about ith car and its 6th vaalue which is d

  // determine other cars' lane
  int car_lane = -1;
  if (d>0 && d<4)
  {
    car_lane =0;
  }
  else if (d>4 && d<8)
  {
    car_lane=1;
  }
  else if (d>8 && d<12)
  {
    car_lane = 2;
  }

  //every car's speed irrespective of its lane
  double vx = sensor_fusion[i][3];//check speed of the car if its in my lane
  double vy = sensor_fusion[i][4];
  check_speed = sqrt(vx*vx+vy*vy); // speed of the car. helpful for predicting where the car will be in future
  double check_car_s = sensor_fusion[i][5]; //check car's s value to see if it is close to use.

  check_car_s +=((double)prev_size*0.02*check_speed); //if using previous points, we can project s value outwards in time
  // because if we are using previous path points, our car is still little behind of where it is supposed to be so we want to see what the car will look like in future
```

```
  // check if the front car in the lane is too close
  if(d<(2+4*lane+2) && d>(2+4*lane-2)) //condition for other car in my lane
  {
    // check if our car's s is closer to other car's s
    if ((check_car_s > car_s)&&((check_car_s-car_s) <30)) // if car is within 30m and in front
    {
      //lower reference velosity so that we dont crash cars in front of us
      //we could also flag the car to change lanes - implement this
      too_close = true;
    }
  }
  //detect car in the left lane
  if (d<(2+4*(lane-1)+2) && d>(2+4*(lane-1)-2))
  {
    if (abs(check_car_s - car_s) < 30)
    {
      car_left_lane = true;
    }
  }

  //detect car in right lane
  if (d<(2+4*(lane+1)+2) && d>(2+4*(lane+1)-2))
  {
    if (abs(check_car_s - car_s) < 30)
    {
      car_right_lane = true;
    }
  }
}
```

3. Actions based on car's position –
   The logic is set such that the car will only change its planned trajectory if the flag *too_close* is true in the earlier part of the code.
   - If the car in front is too close in the current lane and there is not car within 30m in the right lane and the car is not in the rightmost lane – then the car is moved to adjacent right lane by incrementing lane by 1
   - If the car in front is too close in the current lane and there is not car within 30m in the left lane and the car is not in the leftmost lane – then the car is moved to adjacent left lane by decrementing lane by 1
   - If both the conditions are not true, then the car stays in the same lane but gradually decreases its speed so as to not collide with the car in front.
   - Irrespective of the lane, when the car's speed is less than 49.5mph reference speed, it is gradually increased so that the car drives at the recommended highway speed of 50mph for most part of the highway

```
          //if car in my current lane is too close in the front

          if(too_close)
          {
            if(car_right_lane == false && lane<2)
            {
              lane++;
            }
            else if(car_left_lane == false && lane>0)
            {
              lane--;
            }
            else
            {
              ref_val-=0.224;
            }

          }

          else if(ref_val < 49.5)
          {
            ref_val += 0.224; //accelerate by 5m/s2
          }
```

4. Trajectory planning *(lines 222 to 337)*
    - This part of the code forms a trajectory for the car to follow, using spline.h function, calculated lane and speed of the car calculated in the previous section of the code.
    - As a starting position, we take two points that are tangent to the car's current position or last two points from car's previous trajectory along with three points 30m apart in the distance to create a spline.
    - This calculation is performed by converting all Cartesian coordinates into car's local coordinates
    - Spline.h is used to calculate closely spaced points in these five points. These points are transformed back into Cartesian coordinates
    - The speed change calculated in the previous part of the code is used in this section to increase or decrease the speed of the car on every trajectory points
    - For a continuous trajectory, past trajectory points are continued to the new trajectory

```
222           //list of widely spaced waypoints at 30m to use spline function on and fill in more points
223           vector<double> ptsx;
224           vector<double>ptsy;
225
226           //reference x,y,yaw states
227           // starting point could either be current state of car or previous path's endpoint
228           double ref_x = car_x;
229           double ref_y = car_y;
230           double ref_yaw = deg2rad(car_yaw);
231
232           //if previous size is empty, we use car as starting point
233           if (prev_size<2)
234           {
235             //use two points that make go tangent to the car
236             double prev_car_x = car_x - cos(car_yaw);
237             double prev_car_y = car_y - sin(car_yaw);
238
239             ptsx.push_back(prev_car_x);
240             ptsx.push_back(car_x);
241
242             ptsy.push_back(prev_car_y);
243             ptsy.push_back(car_y);
244
245           }
246           else
247           {
248             //redefine reference state as previous path end points
249             ref_x = previous_path_x[prev_size-1];
250             ref_y = previous_path_y[prev_size-1];
251
252             double ref_x_prev = previous_path_x[prev_size-2];
253             double ref_y_prev = previous_path_y[prev_size-2];
254
255             ref_yaw = atan2(ref_y-ref_y_prev,ref_x-ref_x_prev);
256
257             //use two points that make tangent to previous paths endpoints
258             ptsx.push_back(ref_x_prev);
259             ptsx.push_back(ref_x);
260
261             ptsy.push_back(ref_y_prev);
262             ptsy.push_back(ref_y);
263
264           }
```

```
          //addint 30m spaced points in frenet distance on the road ahead of starting reference that we just calculated
          vector<double> next_wp0 = getXY(car_s+30,(2+4*lane),map_waypoints_s, map_waypoints_x, map_waypoints_y);
          vector<double> next_wp1 = getXY(car_s+60,(2+4*lane),map_waypoints_s, map_waypoints_x, map_waypoints_y);
          vector<double> next_wp2 = getXY(car_s+90,(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]);

          //at this point ptsx, ptsy has 5 points in it 2+3

          //shift car's reference point to origin, for math simplicity
          for (int i =0; i<ptsx.size();i++)
          {
            //shift car reference angle to zero degree
            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
          tk::spline s;

          //set x,y points to the spline
          s.set_points(ptsx,ptsy);

          //define the actual (x,y) points to be used for the path planner
          vector<double> next_x_vals;
          vector<double> next_y_vals;

          //start with all the previous path points from the last time
          for(int i=0; i < previous_path_x.size();i++)
          {
            next_x_vals.push_back(previous_path_x[i]); //actual points that path planner will use
            next_y_vals.push_back(previous_path_y[i]);
          }

          // calculate how to break up spline points so that we travel at our desired reference speed
           double target_x =30.0; //horizon x value
           double target_y = s(target_x); //this calculates corresponding y for every n point

           double target_dist = sqrt((target_x)*(target_x)+(target_y)*(target_y));
           double x_add_on =0;

           //fill up the rest of the path planner after filling it with previous points. here we will always output 50 points
           for (int i =1; i<= 50-previous_path_x.size();i++)
           {
             double N = (target_dist/(0.02*ref_val/2.24)); // mph--> 2.24 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 to car reference
             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);
           }



           //END
```