This project was one of the most interesting SDC projects. I really enjoyed working on it. I have detailed in this documentation the approach I took.

STEP 1: Analyze the sensor fusion data and categorize it meaningfully

From the simulator, we get data for other vehicle in the simulator. This data includes the car_id, car_x, car_y, car_vx, car_vy, car_s and car_d

I took the data and divided all cars in the simulator into either my own lane, my left lane or my right lane if those lanes exist. Please see code line numbers 460-505.

Then I calculate the important measures I needed for making decisions. These were
i) Closest car in front of me in my lane
ii) Velocity of this car in front of me
iii) Closest cars in front and back in the left lane
iv) Closest cars in front and back in the right lane

Please see code lines 506-610. I also ensured a few things:

- If I am in the left most lane then I set violate_left = 1 to make sure I don't turn left. Similarly if I am in the right most lane then I set violate_right = 1
- If my lane is empty or if any ofher lanes is empty then the closest car distance was set to 999

STEP 2: Calculate cost of decisions and select the action with minimal cost

I thought of 4 possible decisions, my car could make at any instance:
1. Continue in my lane with max velocity
2. Continue in my lane but slow down to about the speed of the car in front
3. Change to the left lane
4. Change to the right

I decided to assign costs for all these different functions. The cost functions are designed in the code lines 210 to 315.

Below is an explanation of how the cost functions were designed

1. Continue in my lane

- If there is not traffic in my lane, then cost = 0
- If there is traffic but closest car in front > buffer (Set to difference in s = 50) then cost = 0
- If the closest car in front is getting closer than buffer (diffeence in s = 50), then collision is likely hence cost increases to 500

3/4. Change to left lane/right lane

- If lane in which we can turn doesn't exist then collision is likely hence cost = 500
- If lane exists and safe to make the turn then cost = 150
- (Safe to make a turn defined as closest car front is about an s of 50 away and closest car back is an s of 12 away. I decided these numbers by playing around in the simulator)
- If lane exists but unsafe to turn then collision likely. So cost = 500

- One extra condition, I added is if the closest car in front is s of 200+ away (lane emptyish in front), then that lane has a slightly lower cost. I did this so that the program can pick a lane if both are safe to turn

2. Cost of slow down
- I set the cost of slow down = 200. This was done so that the action is less preferred than lane change but better than collision

At every instance, I calculate the cost of all 4 decisions, put them in a vector and then choose the one with the lowest cost. This is done in the code line numbers 618-686

STEP 3: Once decision has been made, decide the target speed and lane for that decisions

To drive the car in the simulator, I took the approach of deciding the target lane using its d-value. Lane 1 has a target d-value of 2, lane 2 has a target d-value of 6 and lane 3 has a target d-value of 10. So depending on the decision (stay in lane, or turn left or right) I decided the target d- value

To control speed in the simulator, I used a s increment. I found that the closer s increments slowed down the car whereas farther speeded the car. To drive at max speed, I chose the s increment of 0.415. When I needed to brake in my lane, I reduced s increment based on the speed of the car in front of me. You can see the code in line numbers 700-822.

STEP 4: Create Jerk minimizing trajectories

The final step was to create jerk minimizing trajectories in the simulator. The first step was to use the previous path in the new path. I followed the code suggested in the lecture. This is important to deal with latency in the simulator.

I took the previous path length and appended to it the new path so that the total steps are 50. See the code in line numbers 825 – 960.

I did the following to solve some of the issues encountered when driving in the simulator

1. Minimize acceleration and Jerk when the car first starts – When the car first starts, we want it to slowly increased speed to maximum. I did this by looking at the difference between current pos_s and prev_s (delta_prev_s) and increasing pos_s slowly as a function of delta_prev_s. The parameters below were tuned till I got jerk free start. See code line 930

```
pos_s += min(target_s, (delta_prev_s*(1.005)+0.002))
```

2. Smooth acceleration and braking
Whenever target_s changes, the car accelerates or brakes. To smooth that I made sure that the change in the target_s happens slowly. I stored previous target_s in a variable prev_s_inc and then smoothed the change in target_s over 80 steps. See code lines 894-895 and 923.

```
//smooth acceleration and braking
double s_smoothing_steps = 80;
double delta_s = (target_s_inc – prev_s_inc)/s_smoothing_steps;
target_s += min(delta_s, 0.01);
```

3. Smooth change of lanes

For lane changes, again I tried the logic above. Essentially do the lane change over 80 steps with small increments. See code lines 886-891 and 933-945

```
double d_smoothing_steps = 80;
double delta_d = (target_d - prev_d) / d_smoothing_steps;
pos_d += min(delta_d, 0.02);
```

The parameters were chosen by experimenting with values and noting performance in the simulator

STEP 5: Fit splines to the waypoint and generate trajectories

I fitted splines to x, y, dx and dy based on pos_s. This is an idea I got from slack channel and led to very smooth lane following. See code lines 354-384.

Once splines are fit, pos_x and pos_y are calculated based on pos_s and pos_d as shown in code line numbers 948-955

Improvements for the future

My code works in the simulator most of the time however there are some opportunities for improvement

1. Include behavior prediction in the program. Can we predict what other cars in the simulator are planning to do by looking at their d relative to the width of the lane?

2. Sometimes my car is in the leftmost lane, the middle lane has traffic and is unsafe to turn into but the rightmost lane is empty. A better logic would identify that and perform a series of quick lane changes to get to the rightmost lane instead of slowing down in its current lane. Essentially the logic currently only looks at the immediate adjacent lane in making its decisions

3. I would like to have better ways to design jerk minimizing trajectories using the methods suggested in the lecture than playing around with parameters