# Self-Driving Car Engineer – Path Planning Project (Highway Driving)

1. Introduction

   In this project your goal is to safely navigate around a virtual highway with other traffic that is driving +-10 MPH of the 50 MPH speed limit. You will be provided the car's localization and sensor fusion data, there is also a sparse map list of waypoints around the highway. The car should try to go as close as possible to the 50 MPH speed limit, which means passing slower traffic when possible, note that other cars will try to change lanes too. The car should always avoid hitting other cars at all cost as well as driving inside of the marked road lanes, unless going from one lane to another. The car should be able to make one complete loop around the 6946m highway. Since the car is trying to go 50 MPH, it should take a little over 5 minutes to complete 1 loop. Also, the car should not experience total acceleration over 10 m/s^2 and jerk that is greater than 10 m/s^3.

2. Simulation results

   The code executed successfully, and the simulator was set on following settings:

   a) Graphics: 800 by 600
   b) Quality: Good

   The following is a screenshot after successfully travelling 4.32 miles without any incidents:


3. Code Structure

I have structured my code in a modular way. The main.cpp is not cluttered with allot of functions in one place, however, I have created separated files for vehicle class, road class and cost function, as it was taught in the behavior planning class. In order to execute my code on simulator, I also made the necessary change in the CMakeLists.txt file to link all the ".h" and ".cpp" files just in time as its required to run.

Here I explain my code in main.cpp in a step by step manner:

a) Necessary files: The necessary files are listed below:
   i. helpers.h
   ii. spline.h
   iii. road.h
   iv. vehicle.h

The required functions in road.h and vehicle.h are implemented in their respective cpp files. I followed this approach of modular programming as taught in the behavior planning class.

b) Necessary Parameters: In main.cpp, line 20 to line 33 show all the necessary parameters initialized.
c) Update car state: In main.cpp, line 119 to 142 updates the state of the car
d) Transformation and spline creation: In main.cpp, line 144 to 196 does the transformation to Frenet coordinates and initializes a spline using spline.h

## 4. Projects Rubrics

1. **Compilation**: Yes, the code compiles correctly.
2. **Valid Trajectories**:

| Criteria | Meets specifications |
|---|---|
| The car is able to drive at least 4.32 miles without incident. | yes |
| The car drives according to the speed limit. | yes |
| Max Acceleration and Jerk are not Exceeded. | yes |
| Car does not have collisions. | No collisions observed |
| The car stays in its lane, except for the time between changing lanes. | yes |
| The car is able to change lanes | yes |

3. **Reflection**:

The path planning code is present in main.cpp file, it starts at line 119 to 216. This part of the code deals with sensor fusion data and telemetry event. The lines from 119 to 142 observe the behavior of the car, updates the state of the car and prints the new position value. The lines from 143 to 216 are first of all making sure the transformation of coordinates is in Frenet system and then adjusts position in this coordinate system. This helps in changing the speed and not exceeding the limit and also deciding in when to change the lane and come back to the target lane without jerks or collisions.

The modular functions are briefly discussed below:

1. Populate traffic information: In road.cpp, populate_traffic() is called in each iteration to reset list of vehicles, add all non-ego vehicles using sensor fusion data and add all ego vehicles using localized data.

```
void Road::populate_traffic(vector<vector<double>> sf_data, vector<double> car_data) {

    Vehicle mycar = this->get_ego();
    this->vehicles_added = 0;
    this->vehicles.clear();
    for (int i = 0; i < sf_data.size(); i++){
        vector<double> car = sf_data[i];
        double x = car[1];
        double y = car[2];
        double vx = car[3];
        double vy = car[4];
        double s = car[5];
        double d = car[6];
        int lane = d/lane_width;
        double speed = sqrt(vx*vx+vy*vy);
        Vehicle vehicle = Vehicle(lane,s,d,speed,0,"CS");
        this->vehicles_added += 1;
        this->vehicles.insert(std::pair<int, Vehicle>(vehicles_added,vehicle));
    }
    vector<float> ego_conf = {this->speed_limit*this->mph_convert, this->num_lanes, mycar.goal_s, mycar.max_acceleration};
    int lane_num = car_data[3]/this->lane_width;
    this->add_ego(lane_num,car_data[2], car_data[3], car_data[4], car_data[5], car_data[6], car_data[7], ego_conf);
}
```

2. Advance vehicles: In road.cpp, advance() predicts position of the non-ego cars.

```
void Road::advance() {

    map<int ,vector<Vehicle> > predictions;
    map<int, Vehicle>::iterator it = this->vehicles.begin();
    float current_speed = 0;
    while(it != this->vehicles.end()){
        int v_id = it->first;
        if(v_id != ego_key){
            vector<Vehicle> preds = it->second.generate_predictions(time_horizon);
            predictions[v_id] = preds;
        }
        it++;
    }
    it = this->vehicles.begin();
    while(it != this->vehicles.end()){
        int v_id = it->first;
        if(v_id == ego_key){
            Vehicle mycar = this->get_ego();
            if(mycar.lane==mycar.target_lane){
                vector<Vehicle> trajectory = it->second.choose_next_state(predictions, time_horizon);
                it->second.realize_next_state(trajectory);
            }else{
                vector<float> kinematics = mycar.get_kinematics(predictions, mycar.target_lane, time_horizon);
                it->second.s = kinematics[0];
                it->second.v_s = kinematics[1];
                it->second.a_s = kinematics[2];
                it->second.lane = mycar.target_lane;
                it->second.d = 4*mycar.target_lane+2;
            }
        }else{
            it->second.s_increment(time_horizon);
        }
        it++;
    }
}
```

3. Estimate next possible state: This is done using a finite state machine. The code is in vehicle.cpp and the function is:

```
vector<string> Vehicle::successor_states() {

    // Provides the possible next states given the current state for the FSM
    //   discussed in the course, with the exception that lane changes happen
    //   instantaneously, so LCL and LCR can only transition back to KL.
    vector<string> states;
    states.push_back("KL");
    string state = this->state;
    if(state.compare("KL") == 0 && this->lane < 2){
        states.push_back("LCR");
    }
    if(state.compare("KL") == 0 && this->lane > 0){
        states.push_back("LCL");
    }

    //If state is "LCL" or "LCR", then just return "KL"
    return states;
}
```

4. Decide lane changes: The decision to change lane is done when it is safe enough and the code is present in vehicle.cpp file and the function is:

```
vector<Vehicle> Vehicle::lane_change_trajectory(string state, map<int, vector<Vehicle>> predictions, float time_window) {

    // Generate a lane change trajectory.
    int new_lane = this->lane + lane_direction[state];
    float future_s = this->s_position_at(time_window);
    bool car_ahead = false;
    bool car_behind = false;
    vector<Vehicle> trajectory;
    Vehicle next_lane_vehicle;

    // Check if a lane change is possible (check if another vehicle occupies that spot).
    for (map<int, vector<Vehicle>>::iterator it = predictions.begin(); it != predictions.end(); ++it) {
        next_lane_vehicle = it->second[0];
        car_ahead = (next_lane_vehicle.s - future_s) < 5 && next_lane_vehicle.s > future_s && next_lane_vehicle.lane == new_lane;
        if (car_ahead) break;
    }
    for (map<int, vector<Vehicle>>::iterator it = predictions.begin(); it != predictions.end(); ++it) {
        next_lane_vehicle = it->second[0];
        car_behind = (future_s-next_lane_vehicle.s) < 5 && next_lane_vehicle.s < future_s && next_lane_vehicle.lane == new_lane;
        if (car_behind) break;
    }
    if(car_behind || car_ahead){
        return trajectory;
    }
    trajectory.push_back(Vehicle(this->lane, this->s, this->d, this->v_s, this->a_s, this->state, this->target_lane));
    vector<float> kinematics = get_kinematics(predictions, new_lane,time_window);
    float new_d = new_lane*4+2;
    trajectory.push_back(Vehicle(new_lane, kinematics[0],new_d, kinematics[1], kinematics[2], state,new_lane));

    return trajectory;
}
```

Similarly, vehicle.cpp also has functions to decide whether to stay in the same lane and when to prepare to change the lane.