Model Documentation P7: Path Planning

Introduction

This document is the model documentation for path planning project. I detail the code structure and finite state machine used to implement the project.
I ran the code without red flags on my local machine using the following configuration:
Graphics: **800 x 600**
Quality: **simple**

Parameters

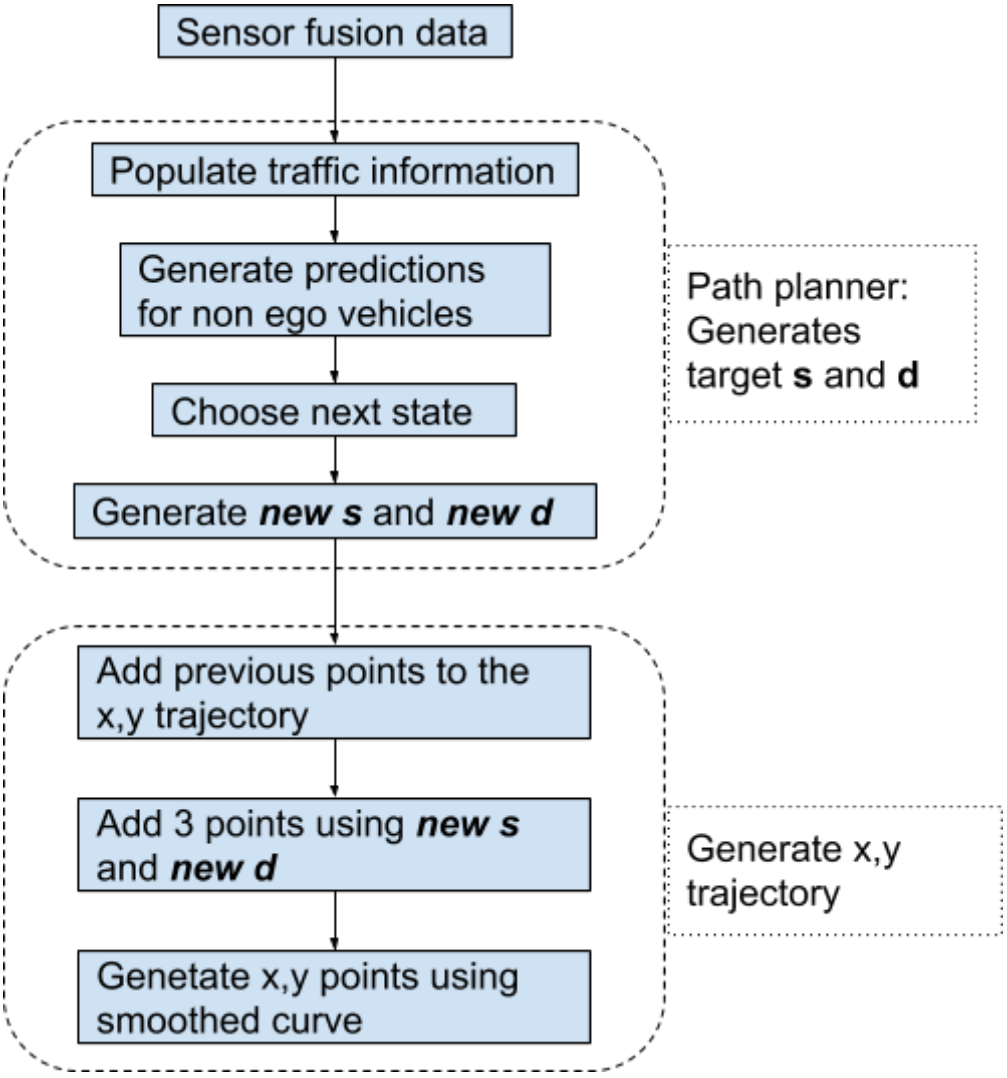
```
//Init road parameters
double REF_VEL=49.0;
double GOAL_S=6945.554;
vector<float> LANE_SPEEDS = {REF_VEL,REF_VEL,REF_VEL};
int LANE_WIDTH=4;
int MAX_ACCEL = 1;
float TIME_HORIZON=2;
float MPH_CONVERT=0.447;
int NUM_LANES = LANE_SPEEDS.size();
double acc=0;
int TARGET_LANE=1;//the lane we want to reach after each new manoeuvre
int car_state=0;//0=="KL",1=="LCR",-1=="LCL"

//init road & car config
Road road = Road(REF_VEL, LANE_SPEEDS,LANE_WIDTH,TIME_HORIZON,MPH_CONVERT);
vector<float> ego_config = {REF_VEL*MPH_CONVERT,NUM_LANES,GOAL_S,MAX_ACCEL};
```

Classes

I use 3 classes:
Road: creates the road + fills traffic information + collects traffic predictions from vehicle class
Vehicle: Creates vehicle instances + Select next state + generate trajectory
Cost: Computes trajectory's cost

Code structure



Traffic planner

I started with the behavior planner provided in the behavior planning course.
A lot of modifications were introduced, I will mention the most important ones:

Populate traffic information

```
void Road::populate_traffic2(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_accelera
    int lane_num=car_data[3]/this->lane_width;
    this->add_ego2(lane_num,car_data[2],car_data[3],car_data[4],car_data[5],car_data[6],car_data[7],ego_conf);
}
```

Each iteration:

- Reset the list of vehicles
- Add all non ego vehicles using sensor fusion data
- Add ego vehicle using self localised data

Advance vehicles

```
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;
                /*double delta_d=1;
                if(mycar.target_lane<mycar.lane) delta_d=-2;
                it->second.d+=delta_d;*/
            }
        }else{
            it->second.s_increment(time_horizon);
        }
        it++;
    }
}
```

This is the “bulk” of the behavior planner:

- Generates position and speed predictions for non ego cars.This information will be used to generate traffic information for ego car.

- Choose next state
- Get new s and new d

Finite state machine

```
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;
}
```

For reasons of simplicity, I ignored the states PLCL and PLCR. The default state is KL, the FSM will not check Left turns if it's to leftmost lane. Similarly, it will not check right turn if it's in rightmost lane.

Lane change decision

```
vector<Vehicle> Vehicle::lane_change_trajectory(string state, map<int, vector<Vehicle>> predictions,float time_
    /*
    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.l
        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.la
        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_lan
    vector<float> kinematics = get_kinematics(predictions, new_lane,time_window);
    float new_d = new_lane*4+2;
    //float new_d = this->d +lane_direction[state];
    trajectory.push_back(Vehicle(new_lane, kinematics[0],new_d, kinematics[1], kinematics[2],state,new_lane));
    return trajectory;
}
```

We consider a lane change if there's no vehicle ahead and no vehicle behind.

Cost function

```
float goal_distance_cost(const Vehicle & vehicle, float dist) {
    /*
    Cost increases based on distance of intended lane (for planning a lane change) and final lane of trajector
    Cost of being out of goal lane also becomes larger as vehicle approaches goal distance.
    */
    float cost;
    float distance = dist-vehicle.s;
    if (distance > 0) {
        cost = distance;
    } else {
        cost = 1;
    }
    return cost;
}

float safety_cost(const Vehicle & vehicle, float dist) {
    /*
    Cost becomes higher for trajectories with intended lane and final lane that have traffic slower than vehic
    */

    float cost;
    float delta = vehicle.target_speed-vehicle.v_s;
    if (delta > 0) {
        cost = 1 - exp(-delta);
    } else {
        cost = 100000;
    }
    return cost;
}

float comfort_cost(const Vehicle & vehicle, float dist) {
    /*
    Cost becomes higher for trajectories with intended lane and final lane that have traffic slower than vehic
    */

    float cost=0;
    if(vehicle.state.compare("LCR")==0||vehicle.state.compare("LCL")==0) cost=1;
    return cost;
}
```

We consider 2 cost functions:

- Distance to goal
- Comfort: avoid lane change if lead to similar s values

Improvement areas

1. Address the case of lateral displacement from non ego cars
2. Consider more complex scenarios such as avoid lanes that contain cars in 100m ahead for example