Design of Path Planning for Self Driving Car

Yu Shen

October 23, 2017

1 Introduction

This document provides a design and implementation of path planning for self-driving-car. It's a project for Udacity Nano-Degree of Self-Driving-Car.

The document uses literate programming approach to present the design and implementation. It presents software construction as prose of writing natural language narrative. The key distinction is that all text are comments to code, while code are presented in special code block. For details of it, please check the link for it. With this style of narrative, key chunks of code relevant to higher level design are presented with much more thought process presented. This would make the source code much more understandable, removing the challenges of providing updated and appropriate comments to the source code.

The full source code can be generated from this literate programming document: ./design-workbook.org

In this design document, "my_car" refers to the car being controlled by the path planner.

2 Performance

The path planner can drive the car to run multiple full lapses of the track. Most of the time the car runs steady. But sometimes, it has problem of exceeding the speed limits, and may exceed the jerk limit sometimes. Occasionally, it might even have collision.

Here is a link on Youtube for a capture of a running session: `https://youtu.be/VNHvzWy84GE`

3 The definition of data

The following is the definition of the input and output data. They serves as the assumption of the design.

They are copied from the project requirements with some my own digestion.

3.1 Definition of waypoint data

- Waypoint Data

Each waypoint has an (x,y) global map position, and a Frenet s value and Frenet d unit normal vector (split up into the x component, and the y component).

The s value is the distance along the direction of the road. The first waypoint has an s value of 0 because it is the starting point.

The d vector has a magnitude of 1 and points perpendicular to the road in the direction of the right-hand side of the road. The d vector can be used to calculate lane positions. For example, if you want to be in the left lane at some waypoint just add the waypoint's (x,y) coordinates with the d vector multiplied by 2. Since the lane is 4 m wide, the middle of the left lane (the lane closest to the double-yellow diving line) is 2 m from the waypoint.

If you would like to be in the middle lane, add the waypoint's coordinates to the d vector multiplied by 6 = (2+4), since the center of the middle lane is 4 m from the center of the left lane, which is itself 2 m from the double-yellow diving line and the waypoints.

3.2 Definition of sensor_fusion data

The data format for each car is: [id, x, y, v_x, v_y, s, d]. The id is a unique identifier for that car. The x, y values are in global map coordinates, and the v_x, v_y values are the velocity components, also in reference to the global map. Finally s and d are the Frenet coordinates for that car.

The v_x, v_y values can be useful for predicting where the cars will be in the future. For instance, if you were to assume that the tracked car kept moving along the road, then its future predicted Frenet s value will be its current s value plus its (transformed) total velocity (m/s) multiplied by the time elapsed into the future (s).

4 The interface to the simulator

The path planning module should send to the simulator, the x, y coordinates in an interval to drive the car to move to the given coordinates.

The interval that the simulator moves the car along is fixed. The value is 0.02 seconds.

The spacing between the points with the fixed interval determines the speed and acceleration of the car.

The code responsible for interacting and driving the simulator as follows:

`driving-simulator`:

```
if (event == "telemetry")
  {
    // j[1] is the data JSON object
    // My car's localization Data
    double car_x = j[1]["x"];
    double car_y = j[1]["y"];
    double car_s = j[1]["s"];
    double car_d = j[1]["d"];
    double car_yaw = j[1]["yaw"]; // in degree
    double car_speed = j[1]["speed"]; // in mile per hour

    // Previous path data given to the Planner
    // actually they are the remaining points of trajectory not yet visited by the car
    // they are issued by the path planner to the car in the previous control time
    auto remaining_path_x = j[1]["previous_path_x"];
    auto remaining_path_y = j[1]["previous_path_y"];
    // Previous path's end s and d values
    double remaining_path_end_s = j[1]["end_path_s"];
    // not yet used, keep for documentation purpose
    double remaining_path_end_d = j[1]["end_path_d"];
    // not yet used, keep might be needed

    // Sensor Fusion Data, a list of all other cars on the same side of the road.
    auto sensor_fusion = j[1]["sensor_fusion"];

    <<path_plan>>

    json msgJson;
    msgJson["next_x"] = trajectory.x_vals;
    msgJson["next_y"] = trajectory.y_vals;

    auto msg = "42[\"control\","+ msgJson.dump()+"]";

    //this_thread::sleep_for(chrono::milliseconds(1000));
    ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
  }
```

In the above code listing, "`«path_plan»`" is the notation to indicate that a code block named "path_plan" will be inserted at the place in the eventual source code generation.

The code block is the implementation of path planning integrated into the telemetry processing loop.

5 Top level design

5.1 Majors Modules

The path planner are consists of the following modules:

- Kinematics: collect and analysis of my_car's kinematics information required for maneuver decision in relationship to the other cares
- Maneuvers: determines the appropriate maneuver
- Trajectory: create the smooth trajectory
- Data-model: the common data structures to support the other modules

I presents the design from top to bottom for ease of review.

5.2 Path Planner Data Model

Here are the main data structures for the construction of the path planner. It's the main body of concepts and vocabulary of the degsin.

From the sensor fusion data, we need to find the nearest vehicles in each lane in front of my_car and behind, and also the projected nearest distance to the nearest vehicle. The nearest distance is further processed into congestion characterization.

The congestion with the nearest vehicle would be used to compute the cost of collision and buffer.

The distance would be calculated based on the time horizon when the new trajectory would start to be used, till the end of the new trajectory.

`path-planner-data-model-declaration`:

```
enum DIRECTION {LEFT = 1, RIGHT = 2};

enum MANEUVER {KL=1, LCL=2, LCR=3, PLCL=4, PLCR=5};

// Parse the sensor_fusion data
string state_str(MANEUVER state) {
  switch(int(state)) {
  case int(KL):
    return "KL";
  case int(LCL):
    return "LCL";
  case int(LCR):
    return "LCR";
  case int(PLCL):
    return "PLCL";
  case int(PLCR):
    return "PLCR";
  default:
    return "Invalid";
  }
}
struct KINEMATIC_DATA {
  double a;
  double v;
  double gap_front;
  double gap_behind;
  double horizon; // evaluation horizon
};

struct Decision {
```

```
  int    lane_index_changed_to; // note, for prepare to change lane, it's not changed actually
  MANEUVER maneuver;
  // double velocity_delta;
  double cost;
  KINEMATIC_DATA projected_kinematics; // for key: "velocity", and "acceleration"
};

struct Car {
  double id;
  double x;
  double y;
  double yaw;
  double v_x;
  double v_y;
  double s;
  double d;
  double v;
  double remaining_path_end_s;
  double remaining_path_end_d;
  double a;
  double jerk;
  int    lane_index;
  bool   empty;
};

struct LaneData {
  Car nearest_front;
  Car nearest_back;
  // double         car_density_front;
  double gap_front; // the projected smallest distance with the car in front, depreciated
  double gap_behind; // the projected smallest distance with the car behind, depreciated
  double congestion_front;      // the congestion with the car in front
  double congestion_behind;     // the congestion with the car behind
};

struct DATA_LANES {
  map<int, LaneData> lanes;
  //double projected_duration;
  bool car_to_left = false;
  bool car_to_right = false;
  bool car_crashing_front_or_behind = false;
};
struct TRAJECTORY {
  vector<double> x_vals;
  vector<double> y_vals;
};

typedef vector< vector<double> > SENSOR_FUSION;
```

parse-fusion-data:

```
void update_surronding(Car my_car, double congestion, int lane,
                       DATA_LANES *data_lanes)
{
  /*
    Based on the distance between the car in front, and that behind,
    congestion to determine the car's status,
    represented in the fields of DATA_LANES:
```

```
    car_crashing_front_or_behind, car_to_left, car_to_right.
   */
  data_lanes-> car_crashing_front_or_behind = false;
  data_lanes-> car_to_left                  = false;
  data_lanes-> car_to_right                 = false;
  if (0.899 < congestion)
    {
    switch (my_car.lane_index - lane) {
    case 0:
      data_lanes->car_crashing_front_or_behind = true;
      break;
    case 1:
      data_lanes->car_to_left = true;
      break;
    case -1:
      data_lanes->car_to_right = true;
    default:
      break;
    }} else
    {
      // cout <<"car_{right, left, ahead}: " << data_lanes->car_to_right << ", "
      // << data_lanes->car_to_left << ", "
      // << data_lanes->car_crashing_front_or_behind;
    }
}

DATA_LANES parse_sensor_data(Car my_car, SENSOR_FUSION sensor_fusion,
                             double start_time, double end_time)
{ /* find the nearest car in front, and behind, and
     find the congestion conditions in front of my_car, and behind
     for the time period of start_time and end_time.
  */

  DATA_LANES data_lanes;
  for (int i = 0; i < NUM_LANES; i++)
    { // initialize the data structure with default values
    LaneData lane_data;
    data_lanes.lanes[i] = lane_data; // assume copy semantics
    data_lanes.lanes[i].nearest_back.empty = true;
    data_lanes.lanes[i].nearest_front.empty = true;
    data_lanes.lanes[i].gap_front  = SAFE_DISTANCE;
    data_lanes.lanes[i].gap_behind = SAFE_DISTANCE;
    data_lanes.lanes[i].congestion_front  = 0.0;
    data_lanes.lanes[i].congestion_behind = 0.0;
    }

  Car a_car;
  for (auto data:sensor_fusion)
    { // find the nearest in front and behind
    a_car.d     = data[6];
    if ((a_car.d < 0) || (lane_width*NUM_LANES < a_car.d))
      {
      continue;                 // ignore invalid record
      }
    a_car.id    = data[0];
    a_car.x     = data[1];
```

```
  a_car.y     = data[2];
  a_car.v_x   = data[3];
  a_car.v_y   = data[4];
  a_car.s     = data[5];

  a_car.lane_index = d_to_lane_index(a_car.d);
  a_car.v     = sqrt(pow(a_car.v_x, 2) +
                     pow(a_car.v_y, 2));
  a_car.empty = false;

  // cout << "a car at lane: " << a_car.lane_index;
  // Find the nearest cars in front of my_car, and behind:
  if (a_car.s <= my_car.s) {// there is a car behind
    if (data_lanes.lanes[a_car.lane_index].nearest_back.empty) {
      // cout << ", first registration for nearest_back ";
      data_lanes.lanes[a_car.lane_index].nearest_back        = a_car;
    } else {
      if (data_lanes.lanes[a_car.lane_index].nearest_back.s < a_car.s) {
        data_lanes.lanes[a_car.lane_index].nearest_back      = a_car;
        // cout << ", update for nearest_back ";
      }}}
  if (my_car.s <= a_car.s) { // there is a car in front
    if (data_lanes.lanes[a_car.lane_index].nearest_front.empty) {
      // cout << ", first registration for nearest_front ";
      data_lanes.lanes[a_car.lane_index].nearest_front       = a_car;
    } else {
      if (a_car.s < data_lanes.lanes[a_car.lane_index].nearest_front.s) {
        // cout << ", update for nearest_back ";
        data_lanes.lanes[a_car.lane_index].nearest_front     = a_car;
      }}}}

// For only the legal lanes adjacent to my_car.lane_index,
int left_lane  = my_car.lane_index -1;
int right_lane = my_car.lane_index +1;
// cout << "candidates_{left | right}_lane: " << left_lane << " | "
// << right_lane << "; ";
vector<int> lanes_interested = {my_car.lane_index};
if (0 <= left_lane)         lanes_interested.push_back(left_lane);
if (right_lane < NUM_LANES) lanes_interested.push_back(right_lane);
for (auto lane:lanes_interested) {
  cout << "interested lane: " << lane << "; ";
  if (!data_lanes.lanes[lane].nearest_back.empty)
    {
      cout << " back congestion: ";
      double congestion = congestion_f(my_car, data_lanes.lanes[lane].nearest_back,
                                       start_time, end_time);
      data_lanes.lanes[lane].congestion_behind = congestion;
      update_surronding(my_car, congestion, lane, &data_lanes);
    }
  if (!data_lanes.lanes[lane].nearest_front.empty)
    {
      cout << " front congestion: ";
      double congestion = congestion_f(data_lanes.lanes[lane].nearest_front, my_car,
                                       start_time, end_time);
      data_lanes.lanes[lane].congestion_front = congestion;
      update_surronding(my_car, congestion, lane, &data_lanes);
```

```
      }
  }
  return data_lanes;
}
```

5.3 path_plan

path_plan code is the top most level program for path planning. It ingests update of my_car's status while performing necessary transformation. Especially, it bases on the current location of my_car, improves the accuracy of the waypoint maps in order to achieve higher accuracy in estimation of locations.

Then it calls `trajectory_f` to generate the new trajectory in trajectory data structure, which will be fed to the simulator for subsequent control.

path_plan:

```
// Assemble information to call trajectory_f:
my_car.id = -1; // hopefully impossible id of the other cars
my_car.x  = car_x;
my_car.y  = car_y;
my_car.yaw = deg2rad(car_yaw);

double old_v = my_car.v;
my_car.v  = mph_2_meterps(car_speed);
my_car.s  = wrap_around(car_s);
my_car.d  = car_d;
my_car.lane_index = d_to_lane_index(car_d);

double old_a = my_car.a;
my_car.a = (my_car.v - old_v)/UPDATE_INTERVAL;

my_car.jerk = (my_car.a - old_a)/UPDATE_INTERVAL;

my_car.remaining_path_end_s = wrap_around(remaining_path_end_s);
my_car.remaining_path_end_d = remaining_path_end_d;

<<debug:my_car_states>>
TRAJECTORY remaining_trajectory;
<<debug:remaining_trajectory>>
for (auto x:remaining_path_x) {
  remaining_trajectory.x_vals.push_back(x);
  // cout << setw(6) << x << ", ";
 }

//cout << endl;
//cout << "remaining y: ";
for (auto y:remaining_path_y) {
  remaining_trajectory.y_vals.push_back(y);
  // cout << setw(6) << y << ", ";
 }

// cout << endl;

// Fix and refine the waypoint maps to improve the resolution of computing

WAYPOINTS_MAP refined_maps = refine_maps_f(my_car,
                                           map_waypoints_x, map_waypoints_y, map_waypoints_s,
                                           map_waypoints_dx, map_waypoints_dy);
TRAJECTORY trajectory
```

```
= trajectory_f(my_car, sensor_fusion, remaining_trajectory, refined_maps);
```

5.4 trajactory

Produces the next trajectory to control the car based on my_car's sates, sensor fusion data of the other cars in the roads.

For trajectory generations,

- it first parses the sensor data producing the lane congestion status in relationship to the position

and current speed of `my_car`. This is implemented by `parse_sensor_data`. The congestion information is returned in `data_lanes`

- based on the congestion information, it considers maneuver options including keep the current lane, or change to adjacent lanes. It selects the option with the least cost. This is implemented by `maneuver`. The decision is returned in `decision`.
- the rest of the code, generate the trajectory for the selected decision.

The decision is defined in terms of the lane to change to, the speed to use, etc.

The new trajectory is formed by combing the first portion of the remaining trajectory that had been fed to the simulator, and additional trajectory points. Special caution is made to make smooth transition from the previously planned trajectory and the additional points.

The smoothness is realized using spline routine with two points from the end of adopted previous trajectory and two points dictated by the maneuver decision.

With the 4 seeding points, two spline lines are generated in terms of functions from frenet s coordinate value to the corresponding x, and y coordinate values respectively.

The purpose of the above scheme is to generate speed control spaced trajectory points. Another series of s coordinate values corresponding to the number of additional trajectory points is generated such that the length between consecutive s δs value would satisfy with the following:

$$\delta s \quad = \quad v \cdot \delta t \tag{1}$$

where v is the expected velocity for my_car for the segment on frenet coordinate, and δt is the time interval which simulator updates the car's position.

The expected velocity for segment is calculated based the starting velocity from at the start of the adopted remaining trajectory, in the series of segment, each segment would increment a delta. The value of the delta is empirically determined to avoid increasing too abruptly, not as to generate the feeling of jerk to the passengers in the car. The increment would become zero when the speed reaches the target speed. The target speed is the speed at the end of the new trajectory.

This algorithm may not be accurate. Sometimes, it may still lead to jerk, and often it may be drive the car too slow.

The following approximation might work better:

$$\delta v \quad = \quad \frac{(v_{end} - v_{start})}{n} \tag{2}$$

where δv is the increment in velocity, v_{start} the starting speed of my_car at the start of the adopted remaining trajectory. v_{end} the expected speed of my_car at the end of the new trajectory.

v_{start} value can be rather reliably estimated based on the remaining trajectory data with waypoints.

It's v_{end} that I may not have confidence. The current approximation may be too large often. I may revisit the related kinematics modeling.

We use the generated s sequence access the two spline functions to get the corresponding x, and y values respectively, then the x, y value pair would be the expected additional trajectory.

Finally, we join the adopted portion of the remaining trajectory and the newly generated trajectory.

`trajectory`:

```
TRAJECTORY trajectory_f(Car my_car, SENSOR_FUSION sensor_fusion,
                        TRAJECTORY remaining_trajectory,
                        WAYPOINTS_MAP waypoints_maps)
{
  TRAJECTORY trajectory; // the return value

  int remaining_path_adopted_size = min((int)remaining_trajectory.x_vals.size(),
                                        NUM_ADOPTED_REMAINING_TRAJECTORY_POINTS);

  int new_traj_size = PLANNED_TRAJECTORY_LENGTH - remaining_path_adopted_size;
  // cout << " new_traj_size: " << new_traj_size << "; ";

  double start_time = remaining_path_adopted_size * UPDATE_INTERVAL;
  double end_time   = start_time + new_traj_size  * UPDATE_INTERVAL;

  DATA_LANES data_lanes = parse_sensor_data(my_car, sensor_fusion,
                                            start_time, end_time);

  Decision decision = maneuver_f(my_car, data_lanes);

  // default values for the start of the new trajectory,
  // applicable when there is not enough remaining_trajectory
  double start_s   = my_car.s;
  double start_x   = my_car.x;
  double start_y   = my_car.y;
  double start_yaw = my_car.yaw;
  double start_v   = my_car.v;
  double start_d   = my_car.d;

  // modulate the start values of trajectory by the remaining trajectory:
  if (2 <= remaining_path_adopted_size) {
    // consider current position to be last point of previous path to be kept
    start_x          = remaining_trajectory.x_vals[remaining_path_adopted_size-1];
    start_y          = remaining_trajectory.y_vals[remaining_path_adopted_size-1];
    double start_x2  = remaining_trajectory.x_vals[remaining_path_adopted_size-2];
    double start_y2  = remaining_trajectory.y_vals[remaining_path_adopted_size-2];
    double start_yaw = atan2(start_y-start_y2,
                             start_x-start_x2);
    vector<double> frenet = getFrenet(start_x, start_y, start_yaw,
                                      waypoints_maps._x, waypoints_maps._y,
                                      waypoints_maps._s);
    start_s = frenet[0];
    start_s = wrap_around(start_s); // maybe needed
    start_d = frenet[1];

    // determine dx, dy vector from set of interpoated waypoints,
    // with start_x, start_y as reference point;
    // since interpolated waypoints are ~1m apart and
    // path points tend to be <0.5m apart,
    // these values can be reused for previous two points
    // (and using the previous waypoint data may be more accurate)
    // to calculate vel_s (start_v), vel_d (start_d_dot),
    // acc_s (s_ddot), and acc_d (d_ddot)
    int next_interp_waypoint_index = NextWaypoint(start_x, start_y, start_yaw,
                                                  waypoints_maps._x, waypoints_maps._y);
    double dx = waypoints_maps._dx[next_interp_waypoint_index - 1];
```

```
  double dy = waypoints_maps._dy[next_interp_waypoint_index - 1];
  // sx,sy vector is perpendicular to dx,dy
  double sx = -dy;
  double sy = dx;

  // calculate start_v & start_d_dot
  double vel_x1 = (start_x - start_x2) / UPDATE_INTERVAL;
  double vel_y1 = (start_y - start_y2) / UPDATE_INTERVAL;
  // want projection of xy velocity vector (V) onto S (sx,sy) and D (dx,dy) vectors,
  // and since S and D are unit vectors this is simply the dot products
  // of V with S and V with D
  start_v = vel_x1 * sx + vel_y1 * sy;
}

// ********************* PRODUCE NEW PATH ***********************
// begin by pushing the last and next-to-last point
// from the previous path for setting the
// spline the last point should be the first point in the returned trajectory,
// but because of imprecision, also add that point manually

double prev_s = wrap_around(start_s - start_v * UPDATE_INTERVAL);
int smallest_start_index = 0; // default 0
if (start_s < prev_s)
  {
    smallest_start_index = 1;
    cout << "start_s <= prev_s start_s | prev_s: "
         << start_s << "|" << prev_s << "; ";
  }
double prev_x, prev_y;

// first two points of coarse trajectory, to ensure spline begins smoothly
if (2 <= remaining_path_adopted_size) {
  prev_x = (remaining_trajectory.x_vals[remaining_path_adopted_size-2]);
  prev_y = (remaining_trajectory.y_vals[remaining_path_adopted_size-2]);
} else {
  prev_s = wrap_around(start_s - 1);
  prev_x = start_x - cos(start_yaw);
  prev_y = start_y - sin(start_yaw);
}

// last two points of coarse trajectory, use target_d and current s + 30,60
double target_1_s = (start_s + 30);
if (MAX_S <= target_1_s)
  {
    smallest_start_index = 2;
    target_1_s -= MAX_S;
  }
double target_d1 = lane_center_d(decision.lane_index_changed_to);
vector<double> target_xy1 = getXY(target_1_s, target_d1,
                                  waypoints_maps._s,
                                  waypoints_maps._x,
                                  waypoints_maps._y);
double target_1_x = target_xy1[0];
double target_1_y = target_xy1[1];
double target_2_s = (target_1_s + 30);
if (MAX_S <= target_2_s)
```

```
  {
    smallest_start_index = 3;
    target_2_s -= MAX_S;
  }
double target_d2 = target_d1;
vector<double> target_xy2 = getXY(target_2_s, target_d2,
                                  waypoints_maps._s,
                                  waypoints_maps._x,
                                  waypoints_maps._y);
double target_2_x = target_xy2[0];
double target_2_y = target_xy2[1];
vector<double> coarse_s_traj, coarse_x_traj, coarse_y_traj;

// arrange the seeding trajectory points to ensure coarse_s_traj has
// increasing order
map<string, map<string, double> > seeds =
  {
    {"prev",     {{"s", prev_s},     {"x", prev_x},     {"y", prev_y}}},
    {"start",    {{"s", start_s},    {"x", start_x},    {"y", start_y}}},
    {"target_1", {{"s", target_1_s}, {"x", target_1_x}, {"y", target_1_y}}},
    {"target_2", {{"s", target_2_s}, {"x", target_2_x}, {"y", target_2_y}}}
  };
map<string, vector<double>* > trajs =
  {
    {"s", &coarse_s_traj},
    {"x", &coarse_x_traj},
    {"y", &coarse_y_traj}
  };

for (string sxy: {"s", "x", "y"})
  {
    // cout << "case : " << smallest_start_index << "; ";

    // cout << "re-arranged: ";
    switch (smallest_start_index)
      {
      case 0:
        for (string p: {"prev", "start", "target_1", "target_2"})
          {
            trajs[sxy]->push_back(seeds[p][sxy]);
          }
        break;
      case 1:
        for (string p: {"start", "target_1", "target_2", "prev"})
          {
            // cout << seeds[p][sxy] << ", ";
            trajs[sxy]->push_back(seeds[p][sxy]);
          }
        break;
      case 2:
        for (string p: {"target_1", "target_2", "prev", "start"})
          {
            trajs[sxy]->push_back(seeds[p][sxy]);
          }
        break;
      case 3:
```

```
        for (string p: {"target_2", "prev", "start", "target_1"})
          {
            trajs[sxy]->push_back(seeds[p][sxy]);
          }
        break;
      default:
        cout << "Illegal index of the smallest s value. ";
      }
  }
// cout << " coarse_s_traj.size(): " << coarse_s_traj.size() << "; " << endl;

// next s values
vector<double> interpolated_s_traj, interpolated_x_traj, interpolated_y_traj;
double target_v = decision.projected_kinematics.v; // best_target[0][1];
double next_s = start_s;
// double prev_updated_s = -MAX_S; // impossibly small

double next_v = start_v;
// cout << " next_v: ";
for (int i = 0; i < new_traj_size; i++) {
  double v_incr = 0;
  next_s += next_v * UPDATE_INTERVAL;
  // prevent non-increasing s values:
  next_s = wrap_around(next_s);
  // if (next_s <= prev_updated_s)
  //   break;
  // prev_updated_s = next_s;
  // cout << setw(5) << next_v << ", ";
  interpolated_s_traj.push_back(next_s);
  if (fabs(target_v - next_v) < 2 * VELOCITY_INCREMENT_LIMIT) {
    v_incr = 0;
  } else {
    // arrived at VELOCITY_INCREMENT_LIMIT value empirically
    v_incr =
      (target_v - next_v)/(fabs(target_v - next_v)) * VELOCITY_INCREMENT_LIMIT;
  }
  next_v += v_incr;
}
// cout << " coarse_s_traj: ";
// for (auto s: coarse_s_traj)
//   {
//     cout << s << ", ";
//   }
// cout << endl;

interpolated_x_traj =
  interpolate_points(coarse_s_traj, coarse_x_traj, interpolated_s_traj);
interpolated_y_traj =
  interpolate_points(coarse_s_traj, coarse_y_traj, interpolated_s_traj);

// add previous path, if any, to next path
// Start with the adopted portion of the previous path points from last time
for (int i = 0; i < remaining_path_adopted_size; i++) {
  trajectory.x_vals.push_back(remaining_trajectory.x_vals[i]);
  trajectory.y_vals.push_back(remaining_trajectory.y_vals[i]);
}
```

```
  // add xy points from newly generated path
  // Fill up the rest of the points for the planner
  for (int i = 0; i < interpolated_s_traj.size(); i++) {
    trajectory.x_vals.push_back(interpolated_x_traj[i]);
    trajectory.y_vals.push_back(interpolated_y_traj[i]);
  }
  return trajectory;
}
```

5.5 congestion characterization

This models the congestion condition between two cars, the front and the behind, on the same lane, supposing if my_car would be in that lane.

The function returns the congestion coefficient between the two cars.

Here is more motivation discussion:

Simpply considering the shortest distance between two car is not enough. The time to reach the low limit of distance also matter. The sooner to reach, the worst. So in terms of cost, I can expression the cost inversely proportional to the time reaching the low limit, and the distance at the time.

For the case, when the front car is faster, then the time is at the start of the trajectory, and the distance is at the time of the trajectory start.

For the case, when the front car is slower, the distance is going to reduce over time further. So I can only measure when the time the distance becomes not acceptable.

`congestion`:

```
double start_distance_congestion(double dist_start)
{
  return exp(-max(dist_start/SAFE_DISTANCE, 0.0) );
}

const double SAFE_DISTANCE_CONGESTION = start_distance_congestion(SAFE_DISTANCE);
double threshold_congestion(double time_threshold, double start_time)
{
  double damper = SAFE_DISTANCE_CONGESTION/exp(-start_time);
  // adjust the congestion for this case,
  // to be comparable with that computed by start_distance_congestion
  // if time_threshold == start_time,
  // then the congestion would be equal to start_distance_congestion(SAFE_DISTANCE)
  double c = damper * exp(-time_threshold);
  return c;
}

double punished_start_distance_congestion(double dist_start)
{
  double punish_weight = 1.01; // punish further this case
  double c = punish_weight * start_distance_congestion(dist_start);
  return c;
}

double congestion_f(Car front, Car behind, double start_time, double end_time)
{ // returns the congestion coefficient between the two cars.
  // To simplify, assume they have zero acceleration
  double c = 0.0;
  double dist_start = (front.s - behind.s) + (front.v - behind.v)*start_time;
  if (behind.v <= front.v)
    {
      c = start_distance_congestion(dist_start);
```

```
      cout << " start_time: " << setw(5) << start_time
           << ", front faster, dist_start: "
           << setw(7) << dist_start << " c: " << setw(7) << c << "; ";
    } else
    { // behind.v > front.v
      if (dist_start <= SAFE_DISTANCE)
        {
          c = punished_start_distance_congestion(dist_start);
          cout <<  " start_time: " << setw(5) << start_time
               << ", front slower and start with less safe distance, dist_start: "
               << setw(7) << dist_start <<  " c: " << setw(7) << c <<"; ";
        } else
        { // dist_start > SAFE_DISTANCE
          // with equation:
          // dist = (front.s - behind.s) + (front.v - behind.v)* t = SAFE_DISTANCE
          // time_threshold should be when the projected distance
          // between the front and the behind would equal to SAFE_DISTANCE
          double time_threshold =
            (SAFE_DISTANCE - (front.s - behind.s)) / (front.v - behind.v);
          cout <<
            "front slower, and start wtih more than safe distance, time_threshold: "
               << setw(7) << time_threshold << " c: " << setw(7) << c <<"; ";
          assert(start_time <= time_threshold); // by the model's reasoning
          c = threshold_congestion(time_threshold, start_time);
        }
    }
  return c;
}
```

5.6 maneuver

This is the top level program to consider applicable options and select the one with the lowest cost. It outputs in terms of a structure `Decision`.

The structure Decision represent all the consequence of a maneuver decision including

- the targeted velocity, acceleration,
- the target lane changed into, etc.

It only considers that the lanes that are consider to be safe at the time of the beginning of the new planned trajectory.

For each plausible maneuver options, `evaluate_decision` performs the analysis and outputs the details of the decision, and also provides the cost of the decision.

Then the decision with the lowest cost will be selected as the decision.

`maneuver`:

```
Decision maneuver_f(Car my_car, DATA_LANES data_lanes) {
  vector<MANEUVER> states;
  if (!data_lanes.car_crashing_front_or_behind) {
    states.push_back(KL);
  }
  // starting from 0, from the left most to the right most
  if (0 < my_car.lane_index) {// change to left lane possible
    if (!data_lanes.car_to_left) {
      states.push_back(LCL);
    }
  }
  if (my_car.lane_index < NUM_LANES-1) { // change to right lane possible
```

```
    if (!data_lanes.car_to_right) {
      states.push_back(LCR);
    }
  }
  map<MANEUVER, Decision> decisions;
  for (auto proposed_maneuver:states) {
    // Decision a_decision = evaluate_decision(proposed_maneuver, my_car, data_lanes);
    Decision decision = project_maneuver(proposed_maneuver, my_car, data_lanes);
    decision.cost = calculate_cost(decision, my_car, data_lanes);

    cout << setw(5) << state_str(proposed_maneuver) << ", cost: "
         << setw(5) <<  decision.cost << " | ";
    decisions[proposed_maneuver] = decision;
  }

  Decision decision = min_map_element(decisions)->second;
  cout << "Sel. man.: "  << setw(5) << state_str(decision.maneuver);
  // << ", cost: " << setw(7) << decision.cost << " ";
  cout << endl; // end of displaying cost evaluations
  return decision;
}
```

5.7 project_maneuver:

Compute the decision should the maneuver is performed.
project_maneuver:

```
Decision project_maneuver(MANEUVER proposed_maneuver, Car my_car,
                          DATA_LANES data_lanes) {
  Decision decision;
  int changed_lane = my_car.lane_index;

  switch(int(proposed_maneuver)) {
  case int(KL):
    decision.projected_kinematics =
      kinematic_required_in_front(my_car, data_lanes, my_car.lane_index);
    decision.lane_index_changed_to = my_car.lane_index;
    break;
  case int(LCL):
    changed_lane = my_car.lane_index-1;
    decision.projected_kinematics =
      kinematic_required_in_front(my_car, data_lanes, changed_lane);
    decision.lane_index_changed_to = changed_lane;
    break;
  case int(LCR):
    changed_lane = my_car.lane_index+1;
    decision.projected_kinematics =
      kinematic_required_in_front(my_car, data_lanes, changed_lane);
    decision.lane_index_changed_to = changed_lane;
    break;
  case int(PLCL):
    decision.lane_index_changed_to = my_car.lane_index;
    // no lane change yet, but evaluate with the proposed change
    decision.projected_kinematics =
      kinematic_required_behind(my_car, data_lanes, my_car.lane_index -1);
    break;
  case int(PLCR):
```

```
    decision.lane_index_changed_to = my_car.lane_index;
    // no lane change yet, but evaluate with the proposed change
    decision.projected_kinematics =
      kinematic_required_behind(my_car, data_lanes, my_car.lane_index +1);
    break;
  default:
    cout << "Not supported proposed state: " << proposed_maneuver << endl;
    break;
  };
  decision.maneuver = proposed_maneuver;
  cout // <<  "prop. man.: "
       << setw(5) << state_str(decision.maneuver) << ", " << " to: "
       << decision.lane_index_changed_to << ", ";
  return decision;
}
```

5.8 kinematic_required_in_front

Calculate at the start of new trajectory, the required and permitted (maximum) velocity and acceleration and speed.

All the expected kinematic data of interests are stored in the structure KINEMATIC. It's the type of the return value.

The target velocity will be computed. It's needed as the target speed to adjust my_car's speed in the new trajectory generation in `trajectory_f`, when my_car change lane.

Experiment to make the allowed speed to have tighter condition: only when the car in front is slower, and close enough to adopt it's speed.

It's not reasonable to expect the car to accelerate/deacceleration within one update interval. This might be the root cause of the car jerks too often. It's reasonable to assume that a car would be able to adjust the speed in a few seconds. I'd experiment with 5 seconds. I call this the planning horizon. I should use consistently wherever applicable. This is the time period that a reasonable car should be adjust its speed to the range desirable.

This is an experimental design. I have not found better approximation yet given the time limit.

`kinematic_required_in_front`:

```
KINEMATIC_DATA kinematic_required_in_front
(Car my_car, DATA_LANES data_lanes, int lane_changed_to) {
  KINEMATIC_DATA kinematic;
  kinematic.v = SPEED_LIMIT; // assuming there is no car in front.
  kinematic.horizon = 200*UPDATE_INTERVAL; // 4 seconds
  double critical_congestion = punished_start_distance_congestion(SAFE_DISTANCE);
  if (critical_congestion < data_lanes.lanes[lane_changed_to].congestion_front)
    {
      kinematic.v = min(SPEED_LIMIT, data_lanes.lanes[lane_changed_to].nearest_front.v);
    }
  kinematic.a = (kinematic.v - my_car.v)/kinematic.horizon;
  return kinematic;
}
```

`projected_gap`:

```
double projected_gap_front(double front_s, double front_v,
                           double behind_s, double behind_v, double behind_a,
                           double delta_t)
{
  double gap = front_s - behind_s + (front_v - behind_v)*delta_t +
    - 0.5*behind_a*(delta_t * delta_t) - VEHICLE_LENGTH;
  return gap;
```

```
}

double projected_gap_behind(double behind_s, double behind_v,
                            double front_s, double front_v, double front_a,
                            double delta_t)
{
  double gap = front_s - behind_s + (front_v - behind_v)*delta_t +
    + 0.5*front_a*(delta_t * delta_t) - VEHICLE_LENGTH;
  return gap;
}
void update_gaps_in_kinematic(Car front, Car my_car, Car behind,
                              double horizon, KINEMATIC_DATA *kinematic)
{
  kinematic->horizon = horizon;
  if (behind.empty) {
    kinematic->gap_behind = SAFE_DISTANCE; // extremely large
  } else {
    kinematic->gap_behind = projected_gap_behind
      (behind.s, behind.v, my_car.s,
       kinematic->v, kinematic->a, kinematic->horizon);
  }
  if (front.empty) {
    kinematic->gap_front = SAFE_DISTANCE; // extremely large
  } else {
    kinematic->gap_front = projected_gap_front
      (front.s, front.v, my_car.s,
       kinematic->v, kinematic->a, kinematic->horizon);
  }
}
```

5.9 kinematic_required_behind

This calculates the minimum acceleration and velocity required in order to be crashed by the nearest car behind my_car.

It's currently no being used, as the maneuvers of PLCL and PLCR (prepare change lane left/right) are not being considered.

kinematic_required_behind:

```
//map<string, double>
KINEMATIC_DATA kinematic_required_behind
(Car my_car, DATA_LANES data_lanes, int lane_index) {
  KINEMATIC_DATA kinematic;
  if (data_lanes.lanes[lane_index].nearest_back.empty) {
    kinematic.a = my_car.a;
    kinematic.v = my_car.v;
  } else {
    double gap_behind =
      my_car.s - data_lanes.lanes[lane_index].nearest_back.s;
    if (gap_behind <= 0)
      { // invalid with assumption that the other car is behind
      kinematic.a = my_car.a;
      kinematic.v = my_car.v;
    } else {
      double delta_v =
        my_car.v - data_lanes.lanes[lane_index].nearest_back.v;
      double min_acceleration_pushed_by_nearest_back =
```

```
        (delta_v*delta_v)/(2*gap_behind);
      kinematic.a =
        min(min_acceleration_pushed_by_nearest_back,
            my_car.a +
            MAX_ACCELERATION_DELTA_METERS_PER_UPDATE_INTERVAL);
      kinematic.v = min(data_lanes.lanes[lane_index].nearest_front.v,
                        my_car.v + kinematic.a * UPDATE_INTERVAL);
      // kinematic.v is used per UPDATE_INTERVAL
    }}
  update_gaps_in_kinematic(data_lanes.lanes[lane_index].nearest_front,
                           my_car,
                           data_lanes.lanes[lane_index].nearest_back,
                           10*UPDATE_INTERVAL, &kinematic);
  return kinematic;
}
```

5.10 calculate_cost

For a maneuver, the following costs show in the code are evaluated. Considering all possible costs:

- collision
- buffer_cost
- inefficiency_cost
- not_middle_cost: encourage to be in the middle lane
- lane_change_extra_cost: model the extra risk and inconvenience in changing lane

Add all of them together.
The data required:

- projected speed of my_car with the maneuver, based on the projected acceleration/speed
- distance to the car in front, or behind (closest_approach), based on data_lanes data structure
- the time to collision, based on the projected acceleration and data_lanes (not yet fully considered successfully)

calculate_cost:

```
double calculate_cost(Decision decision, Car my_car, DATA_LANES data_lanes) {
  // cout << " lane: " << decision.lane_index_changed_to;
  double collision_cost
    = COLLISION_C *   collision_cost_f(decision, my_car, data_lanes);
  double inefficiency_cost
    = EFFICIENCY_C *  inefficiency_cost_f(decision, my_car, data_lanes);
  double buffer_cost
    = DANGER_C *      buffer_cost_f(decision, my_car, data_lanes);
  double not_middle_cost
    = NOT_MIDDLE_C *  not_middle_cost_f(decision, my_car, data_lanes);
  double lane_change_extra_cost
    = LANE_CHANGE_C * lane_change_extra_cost_f(my_car, decision);
  double cost = collision_cost + buffer_cost + inefficiency_cost
    + not_middle_cost + lane_change_extra_cost;
  cout << "coll. c: " << setw(3) << collision_cost << " buf. c: " << setw(3) << buffer_cost
       << " ineff. c: " << setw(3) << inefficiency_cost << ", ";
  return cost;
}
```

5.11 collision_cost

Use the current acceleration and velocity of my_car to asses collision risk in more realistic than using those the projected ones.

collision_cost:

```
double collision_cost_f(Decision decision, Car my_car, DATA_LANES data_lanes)
{
  if (data_lanes.car_crashing_front_or_behind)
    {
      return 1.0;
    } else
    {
      return 0.0;
    }
}
```

5.12 inefficiency_cost

Model the extent how much my_car's velocity can reach the speed limit. It calculates the difference between the speed limit and my_car's projected speed by the maneuver.

inefficiency_cost:

```
double inefficiency_cost_f(Decision decision, Car my_car, DATA_LANES data_lanes) {
  double projected_v = decision.projected_kinematics.v;
  // expect the speed can match SPEED_LIMIT in 1 UPDATE_INTERVAL seconds
  // just relatively compare
  double cost = pow((SPEED_LIMIT - projected_v)/SPEED_LIMIT, 2);
  return cost;
}
```

5.13 buffer_cost

It models the degree of congestion with the nearest car behind and in front.

buffer_cost:

```
double buffer_cost_f(Decision decision, Car my_car, DATA_LANES data_lanes)
{ // express the requirements that both the gap_front and gap_behind should be
  // larger or equal to SAFE_DISTANCE.

  double cost_front  = data_lanes.lanes[decision.lane_index_changed_to].congestion_front;
  double cost_behind = data_lanes.lanes[decision.lane_index_changed_to].congestion_behind;
  return cost_front + 1.0 * cost_behind;
}
```

5.14 not-middle-cost

not-middle-cost:

```
double not_middle_cost_f(Decision decision, Car my_car, DATA_LANES data_lanes) {
  // favor the middle lane, to have more options to change lane when needed
  return logistic(fabs(decision.lane_index_changed_to - 2));
}
```

5.15 lane_change_extra_cost_f

model the observation that the ease of changing lane is proportional to the speed of my_car.

I might want to considered some "inertia" factor for my_car to stay in a lane for a while after changing into the lane.

lane_change_extra_cost_f:

```
double lane_change_extra_cost_f(Car my_car, Decision decision) {
  if ((decision.maneuver == LCL) || (decision.maneuver == LCR))
    return exp(-fabs(my_car.v));
  else
    return 0;
}
```

5.16 car-constants

Here are the parameters for the path planner.

car-constants:

```
#ifndef PARAMETERS
#define PARAMETERS

/*
  parameters.h
  The parameters for path planning design.

*/
const double METERS_PER_SECOND_IN_MPH = 1609.344/3600;
double mph_2_meterps(double mph) {
  double meter_per_seconds = mph*METERS_PER_SECOND_IN_MPH;
  return meter_per_seconds;
}
const double SPEED_LIMIT = mph_2_meterps(49.0); // mph the top speed allowed
const int NUM_LANES = 3;
// The max s value before wrapping around the track back to 0
const double MAX_S = 6945.554;

const double VEHICLE_LENGTH = 3.0; // meters,
//23 meters is the maximum vehicle length, according to California highway standard
// const double BUFFER_ZONE = 10*VEHICLE_LENGTH;
const double NEARBY = 1*VEHICLE_LENGTH; // metres, very near to my_car

const double UPDATE_INTERVAL = 0.02; // seconds,
// the interval to update maneuver decision

const int PLANNED_TRAJECTORY_LENGTH = 50; // 3;
// the length of the planned trajectory fed to the simulator
const int NUM_ADOPTED_REMAINING_TRAJECTORY_POINTS = 50; // 3, 30;
// the length of the first portion of the remaining trajectory (previous_path)
// from experiment, it seems 25 might be too few when the CPU is busy.

const double VELOCITY_INCREMENT_LIMIT = 0.125; // 0.07 not stable

const double MAX_ACCELERATION_METERS_PER_SECOND_SQUARE = 10; // meter/s^2
const double MAX_VELOCITY_DELTA_PRE_UPDATE_INTERVAL
= MAX_ACCELERATION_METERS_PER_SECOND_SQUARE * UPDATE_INTERVAL;

const double MAX_JERK_METERS_PER_SECOND_CUBIC = 10; // meter/s^3
const double MAX_ACCELERATION_DELTA_METERS_PER_UPDATE_INTERVAL
= MAX_JERK_METERS_PER_SECOND_CUBIC * UPDATE_INTERVAL;
const double COLLISION_C  = .1E6f;
const double DANGER_C     = .1E7f;
const double EFFICIENCY_C = .1E3f;
const double NOT_MIDDLE_C = .1E1f;
```

```
const double LANE_CHANGE_C= .1E4f;
// const double NEAR_ZERO = .1E-1f;
// const double DESIRED_TIME_BUFFER = 10; // seconds,
// according to http://copradar.com/redlight/factors/ ;
// change from 30 to 10 for better differentiation
const double SAFE_DISTANCE = 90.0; // meters,
// large enough to conisder to be safe to drive at top speed

const double LANE_CHANGE_INERTIA_C = 1000.0;

#endif
```

6 Further investigation

1. Need to explore how to have better algorithm of the speed control to avoid the problems of exceeding jerk limit. Maybe, the criterion whether to adopt the velocity of the car in front can be typed to certain threshold of the congestion in the lane between my_car and the one in front. For example, the congestion value when the front car is slower and the distance at the start_time of the new trajectory to be equal to SAFE_DISTANCE.
2. The adjustment increment of velocity VELOCITY_INCREMENT is very sensitive to the stability of the my_car driving. More study is needed.

7 Supporting implementation

The following are lower level supporting functions and the actual main program construction.

7.1 Kinematic modules

Computes kinematics with my_car and in relationship to the other cars
kinematics-module:

```
<<projected_gap>>
<<congestion>>
<<parse-fusion-data>>
<<kinematic_required_in_front>>
<<kinematic_required_behind>>
```

7.2 maneuvers modules

maneuvers-module:

```
<<project_maneuver>>
<<collision_cost>>
<<buffer_cost>>
<<inefficiency_cost>>
<<not-middle-cost>>
<<lane_change_extra_cost_f>>
<<calculate_cost>>
<<maneuver>>
```

7.3 trajectory modules

trajectory-module:

```
<<refine_maps>>
<<trajectory>>
```

7.4 Assembly Decorations

Here it assembles the required modules of kinematics, maneuvers, and trajectory includes:

```
#include <assert.h> // #include <assert> does not work, why?

#include <iomanip>

#include <fstream>

#include <iostream>

#include <math.h>
#include <uWS/uWS.h>
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>
#include "Eigen-3.3/Eigen/Core"
#include "Eigen-3.3/Eigen/QR"
#include "json.hpp"

#include "spline.h"
#include "parameters.h"
#include "utils.h"
#include "data_model.h"
#include "kinematics.h"
#include "maneuvers.h"
#include "trajectory.h"

using namespace std;

// for convenience
using json = nlohmann::json;
```

7.5 persistent-car-declaration

persistent-car-declaration:

```
// double ref_val = MAX_VELOCITY_DELTA_PRE_PLANNING_INTERVAL; // initial
Car my_car;
my_car.a = 0;
my_car.jerk = 0;
```

7.6 main

7.6.1 load-waypoint-data

Here are the data from the map file:

- vector<double> map_waypoints_x;
- vector<double> map_waypoints_y;
- vector<double> map_waypoints_s;
- vector<double> map_waypoints_dx;
- vector<double> map_waypoints_dy;

7.6.2 refine_maps

Improve the resolution of waypoint maps.

7.6.3 onHttpRequest

7.6.4 Connection and Disconnection Handling

7.6.5 main

main.cpp:

```
<<includes>>

int main() {
  <<load-waypoint-data>>
  <<persistent-car-declaration>>

  int update_count = 0; // used to debug to capture the first trace
  uWS::Hub h;
  h.onMessage([&map_waypoints_x, &map_waypoints_y, &map_waypoints_s, &map_waypoints_dx,
               &map_waypoints_dy, &my_car, &update_count]
              (uWS::WebSocket<uWS::SERVER> ws, char *data, size_t length, uWS::OpCode opCode) {
    // "42" at the start of the message means there's a websocket message event.
    // The 4 signifies a websocket message
    // The 2 signifies a websocket event
    //auto sdata = string(data).substr(0, length);
    //cout << sdata << endl;
    if (length && length > 2 && data[0] == '4' && data[1] == '2') {
      auto s = hasData(data);
      if (s != "") {
        auto j = json::parse(s);
        string event = j[0].get<string>();
        <<driving-simulator>>
      } else {
        // Manual driving
        std::string msg = "42[\"manual\",{}]";
        ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
      }
    }
  });

  <<onHttpRequest>>
  <<on_connection_handling>>
  int port = 4567;
  if (h.listen(port)) {
    std::cout << "Listening to port " << port << std::endl;
  } else {
    std::cerr << "Failed to listen to port" << std::endl;
    return -1;
  }
  h.run();
}
```

7.7 Other design decisions

- Use meter per seconds to measure speed As the measure of distance and speed from fusion data is in meters, and the update interval is in seconds (0.02 seconds). The exception is the measurement of the speed of "my_car" (the car being controlled),

it's speed is in mph (mile per hour).

7.8 Control Parameters

There are mainly two control choices at the each interval (UPDATE_INTERVAL seconds):

- lane
- velocity/acceleration/deceleration

Changing lane would be desirable if the controlled car have to severely slowdow or even being crashed in the current line. The acceleration/deceleration should be adjusted to be safe, fast and comfortable.

It seems that changing lane is more fundamental maneuver, I'll focus on it while assuming a constant acceleration/deceleration for now. Given the short interval of UPDATE_INTERVAL second control interval, it may be OK to assume small constant acceleration/deceleration. The assumption has been partially confirmed in experiment. The acceleration/deceleration is assumed to be (+2.24m/sˆ2 or -2.24/sˆ2).

7.9 Selection of Lane

Assuming the acceleration/deceleration controlled to maximize the speed within legal limit, the major consideration of selection of a lane, is to avoid collision without too much slowing down. Given other considerations being equal, changing lane may involve additional collision risk, and overhead.

Therefore, the control problem would be modeled by cost function, and the control solution should have the lowest cost among all the legal lane choices. The cost function would have the following components:

- collision cost
- changing lane cost

7.9.1 Collision Cost

The collision cost reflects the risks of collision. The risk of collision has 4 scenarios:

- Longitudinal collision:
  - collision with the car in front
  - collision by the car in the back
- Lateral collision:
  - collision by the car in the left
  - collision by the car in the right

The longitudinal collision can be characterized the overlapping of vehicles' body from the moment of evaluation to the foreseeable future.

7.9.2 Changing lane cost

Changing lane cost may have one major components and one minor component. The major components is the lateral collision risk. It will be proportional to the collision cost then.

The minor component is the overhead and discomfort caused. This is hard to estimate. It will be assumed as a constant for now.

7.10 Avoiding lateral collision and interference

It's not desirable to be next to another car in the adjacent lane. This problem can only solved by adjusting the acceleration/deceleration. Thus, this is a case that should be considered with adjustment of acceleration/deceleration.

It will be less likely, and will be a refinement to do in the future.

Currently, it only tries to avoid changing lane when there is a car next to in the adjacent lane.