Self-driving cars program - project 7: Path planner

Francesco Boi

Contents

1 Content of the project

Here is the content of the project:

- *writeup.pdf* (this file): report of the project;
- *writeup.tex*: source tex file;
- *src*: folder containing the C++ source code of the project;
- *README.md*: file giving a general description of the project.

2 Project goals

The final goal of the project is to autonomously drive the car in the simulator for a minimum distance of 4.32 miles respecting the following constraints:

- the car stays in its lane, except for the time between changing lanes;
- the speed limit must not be exceeded;
- maximum acceleration and jerk must not be exceeded;

- the car does not have any collision with other cars;
- the car is able to change lane;

3 Instruction

Instructions to compile and run the code are included in the README.md file.

4 General workflow

At each iteration, some sparse waypoints are chosen, which are then interpolated using splines to get a path with a total of 50.

Generating such waypoints completely anew at each iteration might result in discontinuity that generates high jerks, large steering angles and so on. To avoid this, the extra path points the car did not go through at the previous iteration are partly reused: the last two points of the path are taken as the first two waypoints, generating in this way a continuous path across different iterations. In contrast, if at least two points of the previous path are not available, two points are used: the latter is chosen as the current car point, whereas the former is a point chosen as `car_x-cos(ref_yaw)` and `car_y-sin(ref_yaw)`. The goal of this is to generate a path that is tangent to the current path of the car: without this workaround infeasible paths might be generated, for example one that is perpendicular or opposite to the current heading of the car. Other three waypoints equally spaced are chosen. Once we have all the waypoints we interpolate them using spline to get the final path.

5 Keep lane or change lane

Driving a car through a highway most of the time consists of keeping the current lane. Achieving this goal with classical Cartesian coordinates it is difficult due to the curves in the road. It is much better to use Frenet coordinates presented in lecture. At each iteration we check if a more suitable lane is available or not. To do that we iterate over each car in the `sensor_fusion` vector, and extract the vehicle's information (lines 107-114 of the `main.cpp` file):

```
1 for (size_t i=0; i<sensor_fusion.size(); ++i)
2 {
3   float d = sensor_fusion[i][6];
4   double vx = sensor_fusion[i][3], vy=sensor_fusion[i][4];
5   double check_speed = sqrt(pow(vx,2)+pow(vy,2));
6   double check_car_s = sensor_fusion[i][5];
```

d is the the distance from the median line of the road: from its value we can deduct the current lane of the other vehicle. Three cases are considered: the vehicle is in our current lane, it is in the right lane next to ours or in the left one.

5.1 Car in the same lane

The first case is implemented in the lines 115-132:

```
if ((d<2+4*lane+2) && (d>2+4*lane-2)) //car in the same lane
{
    if((check_car_s>car_s)&&(check_car_s-car_s<30.))
    {
        //slow down
        too_close = true;
        propose_lane = std::max(lane-1, 0);
    }
    else if((check_car_s<car_s)&&(car_s-check_car_s<25.)&&
      (propose_lane==lane))//if we have decided to overtake,
      //we stick to that decision, otherwise we move to the
          right
      // lane because there's a faster car behind us
    {
      propose_lane = std::min(lane+1, 2);
    }
}
```

The first nested condition checks if we are too close to a vehicle ahead of us in the same lane; if so, we raise the `too_close` flag to reduce the speed and propose a lane change to the left. In contrast, the second condition checks if a car behind us is coming too close: if so, a lane change to the right is put forward in order to let the other car go. Note the extra boolean condition `(propose_lane==lane)` in the second block: this is because if in the previous iterations we proposed a left line change, then we stick to that decision. The goal of this control is to avoid changing the decision at each iteration. In fact, when we get too close to a car ahead the planner slows the car down and propose a left change. By slowing down, a car from behind might get too close, so that a right lane change seems reasonable to the algorithm, overwriting in this case the previous decision of changing to the left lane. This control avoids this situation.

5.2 Car in the left lane

Lines from 137 to 149 checks if a left lane change can be performed safely:

```
else if((d<2+4*(lane-1)+2) && (d>2+4*(lane-1)-2)) //car in the
    left lane
{
  //if there is a car ahead in the left lane with slower speed
   if ((check_car_s>car_s)&& (check_car_s-car_s<40.*ref_vel/
       car_speed) && (check_speed<car_speed))
  {
    safe_left_change = false;
  }
  // if the car from behind is coming with higher speed
  else if ((check_car_s<car_s)&&(car_s-check_car_s<40.*ref_vel/
      car_speed))
  {
    safe_left_change = false;
  }
}
```

The first nested condition checks if there is a car ahead in the left lane that is too close (within $40m$) and if so a flag, initialised to `true` is set to `false`. In the same way, the flag is set to false even when there is a car in the left lane that is close to us. Instead of using a rough distance threshold, `ref_vel/car_speed` has been used: this is because when we are going at lower speed, for example when there is a car in front of us, the manoeuvre takes more time to be performed.

5.3 Car in the right lane

Lines from 155 to 168 checks if a right lane change can be performed safely:

```
1 else if ((d<2+4*(lane+1)+2) && (d>2+4*(lane+1)-2)) //car in the
      right lane
2 {
3   //if the car is ahead with slower speed than ours
4   if ((check_car_s>car_s)&&(check_car_s-car_s<40.*ref_vel/
        car_speed))
5   {
6     safe_right_change = false;
7   }
8   // if the car from behind is coming with higher speed
9   else if ((check_car_s<car_s)&&(car_s-check_car_s<40.*ref_vel/
        car_speed))
10  {
11    safe_right_change = false;
12  }
13 }
```

The subconditions are the same of the ones in the left change.

5.4 Extra condition at the starting of the simulator

Before checking the left or right lane change are feasible, an extra condition is added before those in lines 132-236, that if true, it excludes the formers:

```
1 else if (car_s<200)
2 {
3   safe_left_change = false;
4   safe_right_change = false;
5 }
```

At the beginning of the simulation the car has 0 speed and it accelerates quite slowly. In this case, a lane change, especially to the right, is very easily proposed, but due to the slow speed of our vehicle, a collision might happen with car coming from behind. For this, if the car has not travelled at least $200m$, no lane change is considered.

5.5 Conifrm the lane change

After the iteration over all the vehicles, the lane change is confirmed or denied by the following code:

```
if ((safe_right_change && (propose_lane==lane+1)) ||
 (safe_left_change && (propose_lane==lane-1)))
{
  lane = propose_lane;
}
```

Here, the proposed change is compared to the current lane and the flags for a safe transition are checked too. If no lane change is proposed or if it is not safe then the current lane is kept.

6 Speed limit must not be exceeded

The speed limit is hardly coded in the lines from 171 to 178:

```
if (too_close)
{
  ref_vel -= 0.2;
}
else
{
  ref_vel = fmin(ref_vel+0.5, 49.9);
}
```

`too_close` is the flag that is raised when the vehicle is getting too close with a car ahead in the same lane. When it is raised, the desired velocity of the vehicle is reduced to avoid collision. Otherwise, it is incremented up to $49.9mph$, which is a little less than the highway speed limit of 50. (using 50. results in exceeding the speed limit). The maximum increment of velocity is calculated using the constraint of having a maximum acceleration of $10\frac{m}{s^2}$ and a time interval of ... The `ref_vel` is ensured to not exceed the speed limit by using the minimum value between the incremented `ref_vel` and the `speed_limit`.

7 Path generation

Two sets of points are used: one, consisting of the vectors `next_x_vals` and `next_y_vals` represents the real path points obtained by interpolation, the other, consisting of the vectors `ptsx` and `ptsy` are the sparse waypoints.

7.1 Sparse waypoints

Lines from 191 to 213 sets the first two waypoints.

```
size_t prev_size = previous_path_x.size();
...
double ref_x=car_x, ref_y=car_y, ref_yaw=deg2rad(car_yaw);
/* If prev_size is (almost) empty (all waypoints consumed), use
    the car
as starting reference*/
if (prev_size<2)
{
    //use 2 points that make the path tangent to the car
```

```
9       std::cout<<"No prev points\n";
10      double prev_car_x = car_x - cos(ref_yaw);
11      double prev_car_y = car_y - sin(ref_yaw);
12      ptsx.push_back(prev_car_x);
13      ptsx.push_back(car_x);
14      ptsy.push_back(prev_car_y);
15      ptsy.push_back(car_y);
16  }
17  else //use the remaining points of the prev path
18  {
19      ref_x = previous_path_x[prev_size-1];
20      ref_y = previous_path_y[prev_size-1];
21      double prev_ref_x = previous_path_x[prev_size-2];
22      double prev_ref_y = previous_path_y[prev_size-2];
23      ref_yaw = atan2(ref_y-prev_ref_y, ref_x-prev_ref_x);
24      ptsx.push_back(prev_ref_x);
25      ptsx.push_back(ref_x);
26      ptsy.push_back(prev_ref_y);
27      ptsy.push_back(ref_y);
28  }
```

In the second block, if at least two path points are available from the previous path, then the last and the last but one are chosen as the first two waypoints, the reference point given by `ref_x` and `ref_y` is updated to the the last point and `ref_yaw` is calculated. In this way, discontinuities caused by generating the whole path anew are avoided.

If there are not enough points from the previous path the actual car position is used together with an arbitrary previous point chosen to make the corresponding segment tangent to the current path. In this way, the path planner generates always feasible paths that do not cause the car to exceed the maximum jerk.

Other three way points spaced at $30m$ are added by the code in the lines from 215 to 221:

```
1  for (size_t i=1; i<4; ++i)
2  {
3    vector<double> next_wp = getXY(car_s+30.*i, 2+4*lane,
         map_waypoints_s,
4      map_waypoints_x, map_waypoints_y);
5    ptsx.push_back(next_wp[0]);
6    ptsy.push_back(next_wp[1]);
7  }
```