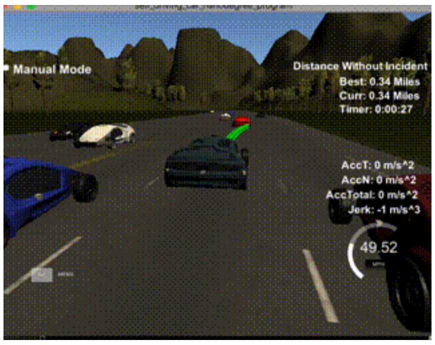
Path Planning



The goal of this project is to build the path planning for a car in a highway simulator, so that the car can smoothly navigate through the traffic.

Project setup and goals

Simulator

The simulator is based on the Unity engine and can be downloaded here: https://github.com/udacity/self-driving-car-sim/releases/tag/T3_v1.2).

Every 0.02s it will create a callback to the path planning programm in order to get a pearl chain of waypoints that the car will precisely follow in the simulator. Thus, the spacing between the points of this path affect the speed of the vehicle and the x,y coordinate the actual path.

Goals

In this project the goal is to safely navigate around a virtual highway with other traffic that is driving +-10 MPH of the 50 MPH speed limit. The simulator provides 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. The car should avoid hitting other cars at all cost as well as driving inside of the marked road lanes at all times, 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.

Prediction and Data from Sensor Fusion

As soon as new data is received from the simulator, first, the sensor fusion data is extracted and evaluated. In particular, the sensor fusion data delivers information about each other car detected in the proximity of our car. The information contains:

- x,y coordinates
- vx, vy
- s and d (Frenet representation of x,y)

Additional information can be computed from these values: - v magnitude (variable called check_speed) - lane (variable called lane)

The magintude is simply computed via Pythagoras: `cpp double check_speed = sqrt(vx*vx + vy*vy);`

In order to compute the lane, we must recap that `d=0` means the center yellow line of the highway and each of the 3 lanes has a width of 4 meters: `cpp int lane = (int) floor(d/4);`

All other cars in the simulator except for ego do not change lanes. For that reason, the prediction can be limited lane following movement. The final prediction for each detected car is done as follows:

```
double check_car_s = sensor_fusion[i][5];
check_car_s += ((double) 0.02 * check_speed * previous_path.size());
```

What acutally is computed here is the predicted position of that car in future time `t`, where `t` is the number of ticks that are still left on the previous path for ego to go times the `delta_t` of each tick. Thus, the predicted position of other cars is exactly at that time in the future, where we will start to add new points to ego`s path.

Planning the behaviour

By iterating over all cars detected by the sensor fusion along with their predicted future position, overall three new information can be computed:

1. for each of the 3 lanes, whether there is a car in front of ego that is closer than the minimum safety distance, defined by the array `bool road_blocked_ahead[]`.
2. for each of the 3 lanes, whether all minimum safety distances would be obeyed if ego changed to that lane, defined by the array `bool safe_for_lane_change[]`.
3. for each of the 3 lanes the speed that could be driven by ego at maximum on that lane given legal regulations and other cars, defined by the array

`double speed_limits[]` .

Note, the indices map to the lane id that the index is referring to, i.e. index 0 refers the information regarding lane 0, which is the left most lane.

The arrays are initialized before iterating over the sensor fusion data:
`bool road_blocked_ahead[] = {false, false, false}; bool safe_for_lane_change[] = {true, true, true}; double speed_limits[] = {49.5, 49.5`
These values are now adapted by iterating over the predictions for the other cars.

Detecting blocked roads ahead

The safety distance of a car driving with speed `v` to the car infront of it should be at least `v/2` . Here, this information is used from ego`s perspective regardless the lane the detected car is driving in:

```
if ((check_car_s > end_s) && (abs(check_car_s - end_s) < car.speed/2.)) {
      std::cout << "found blocking car on lane " << lane  << " distance " << check_car_s << std::endl;
      road_blocked_ahead[lane] = true;
  }
```

The first condition checks whether the other car is infront of ego and the second condition checks whether ego violates the safety distance to that car in the longitudinal `s` dimension. If yes, the entry of `road_blocked_ahead` matching with the detected cars position is set to `true` . This information will later be used to decide if the current lane needs to be left, i.e. a lane change needs to be perfomres

Detecting safe lane change opportunities

Safe lane change opportunities are those that obey the safety distance every point of time. However, here, the problem needs to be looked at from both point of views - ego and the detected cars. The lane change can only be executed if ego and all other cars could keep their safety distance `v/2.0` during the maneuver. Before looping over the detected cars, each lane is marked as a safe lane to change to. By iterating over the detected cars, violations are detected and the lane change will be prohibited:

```
if (((check_car_s > end_s) && (abs(check_car_s - end_s) < car.speed/2.0)) ||
                  ((check_car_s < end_s) && (abs(check_car_s - end_s) < check_speed/2.0))) {
    safe_for_lane_change[lane] = false;
}
```

The first condition checks, whether the detected car is infront of ego and ego violates the longitudinal safety distance. That would render the lane change unsafe for ego. The second condition checks whether the detected car is behind of ego and violates the longitudinal safety distance at the future time stamp t. If any of these conditions is true, the lane where the detected car is driving on is prohibited for any lane change.

Detecting possible maximum speeds for lanes

At last, the maximum possible driving speed for each of the lanes is detected. Given a free lane, this matches the legally allowed maximum speed of `50 mp/h` with a safety buffer of `0.5mp/h` to prevent the car from exceeding the limit. However, if the longitudinal distance of a detected car on lane `l` is closer than safety distance, ego will not be able to drive faster on that lane than the detected car in front of it. Consequently, the speedlimit for this road is adjusted:

```
if ((check_car_s > end_s) && (abs(check_car_s - end_s) < car.speed/2.0)) {
      // smallest speed ahead will be the reference speed
      if(check_speed < speed_limits[lane]) {
        // Smoothly adapt the speed limit when approaching cars
        speed_limits[lane] = fmin(check_speed * (abs(check_car_s - end_s)/30.), 49.5);
      }
}
```

Note, here we are again using the predicted future position of the detected cars and the last point in the previous path of ego, as this is the time where our new behaviour will be appended to ego`s next path.

By iterating over all cars infront of ego, this yields the maximimum future driving speed of ego for each lane.

The `fastest_lane` is then evaluated by iterating over the 3 detected lane speeds and selecting the minimum. In case of ambiguities, the rightmost lane is chosen:

```
int fastest_lane = 2;
double highest_speed_limit = speed_limits[fastest_lane];
for (int i = 2; i >= 0; --i) {
    if(speed_limits[i] > highest_speed_limit) {
      highest_speed_limit = speed_limits[i];
      fastest_lane = i;
    }
}
```

The detected `fastest_lane` defines the desired target lane of the behaviour planner that it wants ego to drive on. If possible, according lane changes will be conducted to bring ego on this fastest lane.

Deciding for a lane change

A new lane change shall only take place if the old one was completed. This prevents the car from changing two lanes at once. A lane change is completed if the car is `+-1m` around the center of a lane. If ego is in the center and a new lane change would be possible, it is checked whether the fastest lane is left of the current lane `ref_lane` and whether it would be safe to conduct a left change. If yes, the left change is conducted. If the left change was not possible, it is checked whether the fastest lane is on the right side of ego. Again, it is also checked whether it would be safe for ego to execute the right change. If yes, the right change is conducted. Note, that if `fastest_lane` anyway matches ego's current lane `ref_lane`, no lane change needs to be performed, i.e. the value of `ref_lane` must not be changed.

```
// check if car arrived in the center of its target lane - only then look for new lines.
if((car.d < ref_lane*4 + 3.5) && (car.d > ref_lane*4 + 1.5)) {
    // change towards the best lane - standard is lane
    if(safe_for_lane_change[ref_lane-1] && fastest_lane < ref_lane) {
        ref_lane -=1;
    } else if(safe_for_lane_change[ref_lane+1] && fastest_lane > ref_lane){
        ref_lane +=1;
    }
}
```

Technically, the lane change is conducted by changing the value of `ref_lane` to the lane id that ego should be driving on. This lane id will later be used to expand the trajectory of ego with the right points.

Adjusting the speed of ego

After it is clear on which lane ego shall drive, its velocity is adapted to match the maximum possible driving speed of the new target lane. Two cases are distinguished. Case 1, ego is already violating the longitudinal safety distance. Here, the velocity of ego is reduced by "0.244" starting from the first point that is appended to the previous path. Case 2, the road is free and ego has no yet reached the legally allowed maximum speed. In that case, the reference velocity is increased by that factor.

```
if (road_blocked_ahead[ref_lane] && ref_vel > 0.244) {
    ref_vel -= 0.244;
} else if(ref_vel < speed_limits[ref_lane]) {
    ref_vel += 0.244;
}
```

Path planning: Creating the path

After analyis of the execution of the behaviour planning it the new reference velocity (`ref_vel`) and new reference lane (`ref_lane`) have been defined. Along with the previous path and the map data, this is passed to the path planning module:

```
Path res_path = GeneratePath(car, previous_path, end_s, ref_lane, ref_vel, map);
```

The `previous_path` contains all waypoints that ego did not manage to reach in his previous tick. Those points are added to the path again plus additional points so that the path of this tick has an overall length of 50, which matches exactly a time horizon of 1s, given the interval of 0.02s between the ticks.

The new points are created by defining a spline using the spline library, just like described in the tutorial video.

Basically this works by defining overall 5 points: - the first anchor point is either the point before the last in the previous path, or if no previous path exists, a back projection of the current car position by 1s. - the second anchor point is either the last point of the previouspath, or if no previous paths exists, just the current position of the car. - three new anchor points that have a longitudinal distance of 30, 60 and 90 from the second anchor point.

These anchor points are now shifted so that the second anchor point is at `(0,0)` and rotated so that the yaw angle at anchor point two matches the x-axis after the rotation.

Based on that, a spline curve can be created:

```
tk::spline s;
s.set_points(transf_anchor_pts.getX(), transf_anchor_pts.getY());
```

A target point is sampled from that spline: `cpp Point target_point = {30.0, s(30.0)};`

After that, the goal is to extract new waypoints for ego from that spline. For that, the x-coordinate on the spline is linearized by computing the distance traveled between two of the new waypoints:

`double N = d/(delta_t * desired_speed * CONST_MPH_TO_MPS);`

In detail, `N` is the linearized distance on the x axis between two neighboring points that will be extracted from the spline. For all points to be extracted, the x distance is the x of the last sampled point plus this delta distance:

```
double x_val = x_add_on + target_point.x/N;
double y_val = s(x_val);
x_add_on = x_val;
```

The y value is then extracted from the spline. The generated points are added to the output path and returned. This path is finally reported back to the simulator, as the cars next trajectory.

Limitations of the approach

The approach has several limitations that are based on the simplifications made in the simulation:

- Other cars in the simulation do not change lanes. The approach would need to be extended to support such a scenario,
- As the previous path points are re-used to create the new path, there is an bound an upper bound on the reaction time of ego. Ego cannot react faster then `prev_path_size * 0.02s` to new events in the traffic.

Basic Build Instructions

1. Clone this repo.
2. Make a build directory: `mkdir build && cd build`
3. Compile: `cmake .. && make`
4. Run it: `./path_planning`.

Dependencies

- cmake >= 3.5
  - All OSes: click here for installation instructions

- make >= 4.1
  - Linux: make is installed by default on most Linux distros
  - Mac: install Xcode command line tools to get make
  - Windows: Click here for installation instructions

- gcc/g++ >= 5.4
  - Linux: gcc / g++ is installed by default on most Linux distros
  - Mac: same deal as make - [install Xcode command line tools]((https://developer.apple.com/xcode/features/)
  - Windows: recommend using MinGW

- uWebSockets

  - Run either `install-mac.sh` or `install-ubuntu.sh`.
  - If you install from source, checkout to commit `e94b6e1`, i.e. `git clone https://github.com/uWebSockets/uWebSockets cd uWebSockets git checkout e94b6e1` ## Editor Settings

This project was configured for Visual Studio Code. Debug funtionality is available after running `ccmake` and setting `Debug`.