CarND-Path-Planning-Project

Udacity Self-Driving Car Nanodegree - Path Planning Project

Goal

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 avoid hitting other cars at all costs 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.

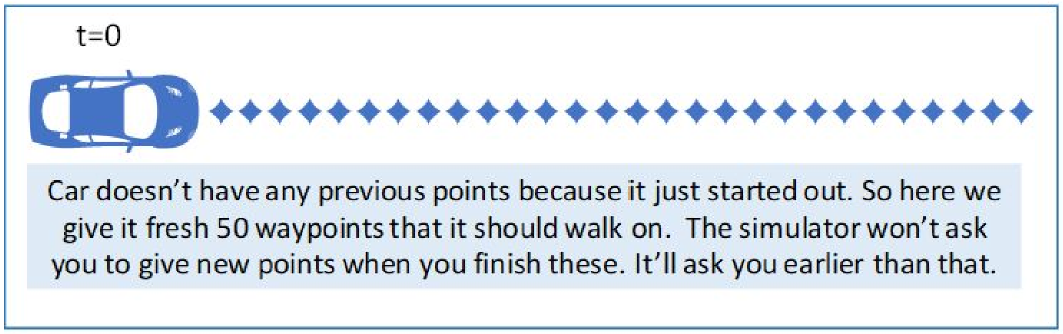
Reflection

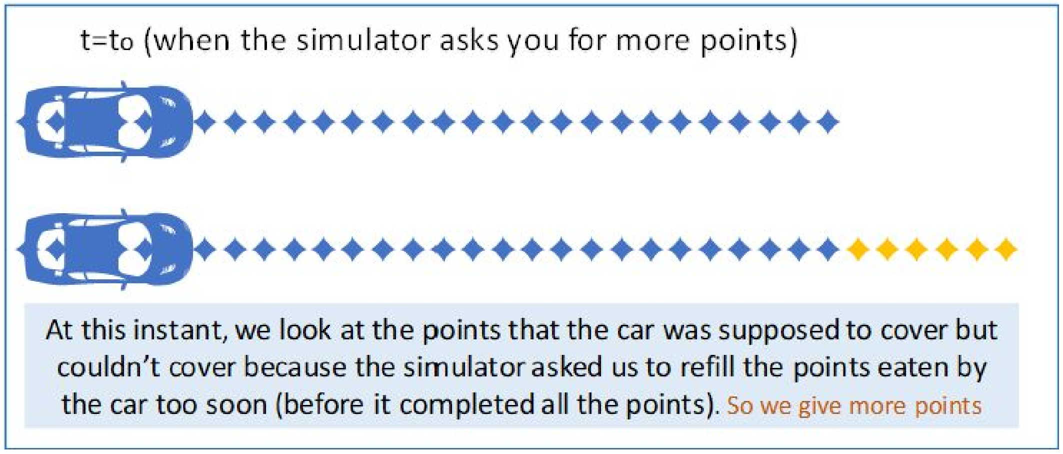
- Based on the provided code from the seed project, the path planning algorithms start at src/main.cpp
- The code could be separated into different functions to show the overall process in a more structured way but I preferred to keep it mostly in a single place and sometimes using helpers.h (especially for behavioral planning)

Overview of the project:

The idea of the project is to generate waypoints for the ego vehicle to follow. These waypoints are eventually fed to the simulator to follow. The simulator asks the ego vehicle to keep a record of 50 points that it wants to follow in the immediate future. As soon as the ego vehicle supplies some points to the simulator, it starts following them but soon comes back (without even completing the previous points to ask the ego to refill the points to make the count to 50 again). This basically gives the software to account for the latest changes in the surroundings and plan accordingly.

This is explained in the following image:


t=0
Car doesn't have any previous points because it just started out. So here we give it fresh 50 waypoints that it should walk on. The simulator won't ask you to give new points when you finish these. It'll ask you earlier than that.


t=to (when the simulator asks you for more points)
At this instant, we look at the points that the car was supposed to cover but couldn't cover because the simulator asked us to refill the points eaten by the car too soon (before it completed all the points). So we give more points

Now, the whole job is to smartly generate these 50 points.

Three processes are happening over here one by one in the code to achieve this.

- **Prediction:** This module takes in the sensor fusion data of other vehicles and localization data of the ego vehicle to generate knowledge about the surroundings. This data can help us to get information like the closest vehicle in ego vehicle's lane, position of vehicles in the adjacent lanes so that we can make lane change decisions later by calculating the space available to make a lane change.

  In this project, these predictions for all the vehicles have been made assuming uniform velocity as an approximation.

- **Behaviour Planning**: This module helps us answer the following two questions:
  - Should the ego vehicle change lane?
  - What should be the velocity of the ego vehicle for this iteration?

These decisions are taken by considering the following:

- Are we too close to a vehicle in front of us?
    - If yes, should we change lanes? What are the lane options available to us? Do we have enough space to safely shift to a new lane? If we have more than one choice, how do we pick one lane to be in?
    - If no, slow down!

- If there’s no vehicle in front of us, do we want to stay in our lane or change lanes? Can we speed up if we are not moving close to the speed limit?

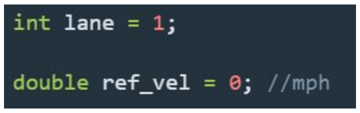
- How do we avoid simultaneous lane changes? (We don’t want to be a rude driver)

- Do we need to apply emergency brakes?

○ **Trajectory Planning**: This module takes the input from the behavior planner, Then we try to generate 5 points that the car would pass through in the future. Using these 5 points, we generate a smooth curve using “src/spline.h”. Once we have the equation of our curve, we can generate waypoints based on the reference velocity for this iteration that the behavior planner decides for us. We finally supply these waypoints back to the simulator.

Initializing variables:

- Initialization at the beginning of the program based on initial conditions:



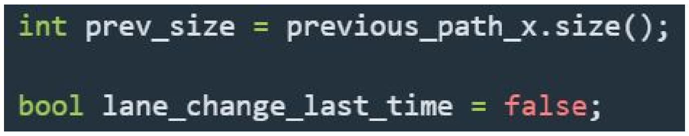
```
int lane = 1;

double ref_vel = 0; //mph
```

The ego vehicle started in the middle lane. The following lane convention was used:

Left lane	0
Middle lane	1
Right Lane	2

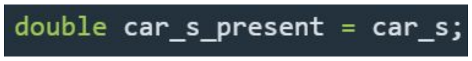
The ref_vel denotes the velocity that the car is supposed to follow at the current instant. We begin this value by 0 and later on adjust it in the code.

- Initializations at the beginning of each iteration (The simulator keeps sending data to the ego vehicles and keeps seeking the next waypoints to follow)



```cpp
int prev_size = previous_path_x.size();

bool lane_change_last_time = false;
```

prev_size contains the number of waypoints still present from the 50 points that the ego vehicle had given it in the last iteration. (The rest of the points were eaten by the car)
Lane_change_last_time will keep a record of whenever the car changed lanes last time and will prevent the change of lanes again in this iteration if it is true.



```cpp
double car_s_present = car_s;
```

This contains the “s” value of the ego vehicle at present (localization data)

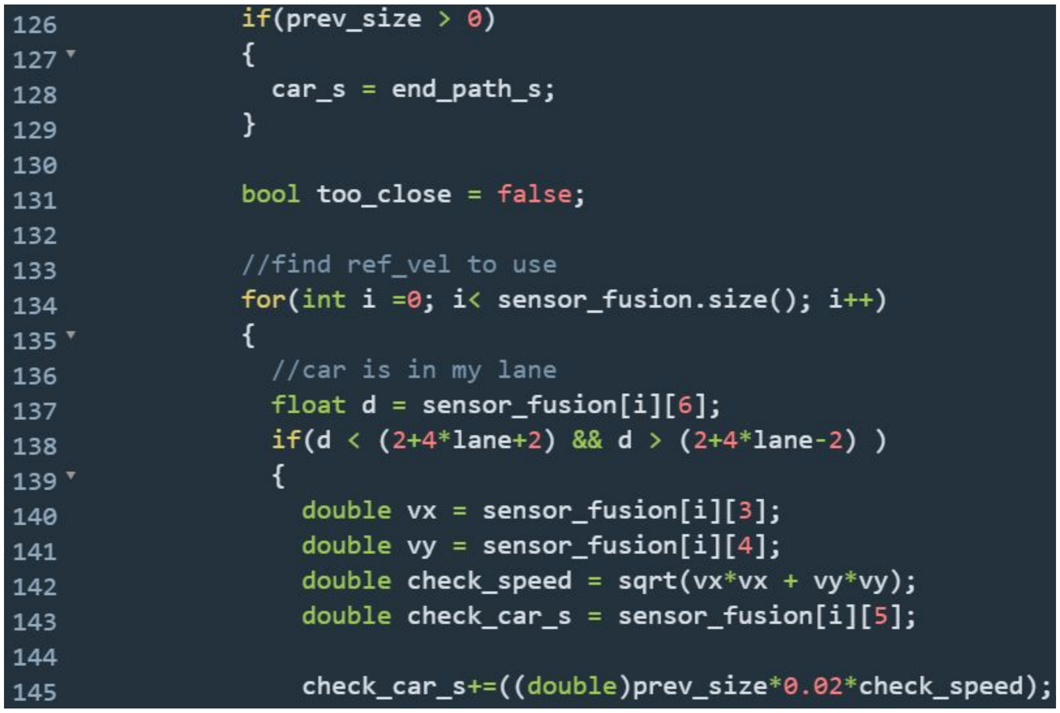
1. Prediction:

The sensor fusion data provided details about all the other traffic vehicles on the road. This data was used to achieve the following tasks within various parts of the code as when required:

- Predicting future positions of all the vehicles by implementing a basic constant velocity kinematic velocity model
- Finding vehicle in front of ego vehicle in the same lane within some threshold distance
- Checking if vehicles are present on either side of the ego vehicle when it intends to change lanes. This is done by taking the future position of the ego vehicle when it would want to change lanes, then using kinematic models, an approximate position of each traffic vehicle is obtained. Then a gap threshold is applied on the front and backside of the ego vehicle’s future position in the desired lane to check if there is enough space to make the change

In this project we make two kinds of predictions:

- **Prediction in the future:**
  As the simulator asks us to refill points and add up to the waypoints given last time, we only have control over what happens after the car completed the waypoints it was supposed to cover from the last iterations and then we add on to it. But by the time the car reaches the last point from its previous points, sensor fusion values will be outdated. So here we try to estimate the s coordinate value of other vehicles assuming a uniform motion model.
  prev_size*0.02 gives the time it would take the ego vehicle to complete pending points from the last time.

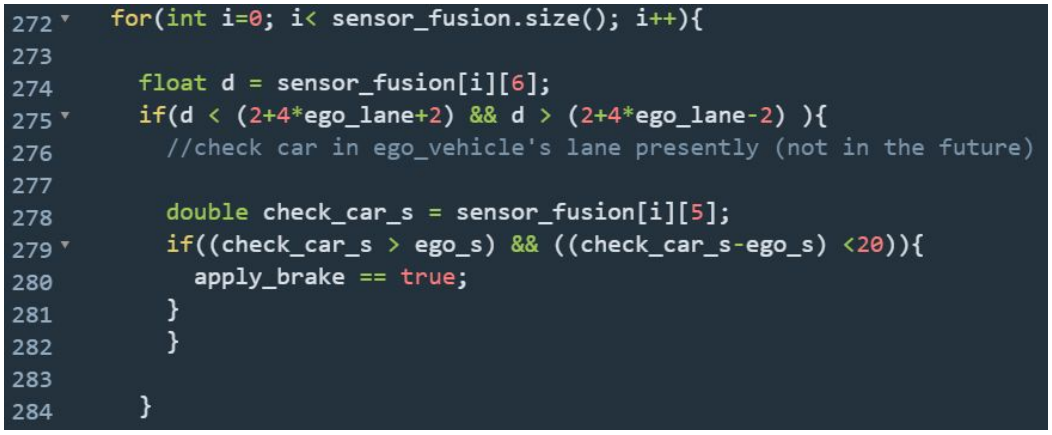


```
          if(prev_size > 0)
          {
            car_s = end_path_s;
          }

          bool too_close = false;

          //find ref_vel to use
          for(int i =0; i< sensor_fusion.size(); i++)
          {
            //car is in my lane
            float d = sensor_fusion[i][6];
            if(d < (2+4*lane+2) && d > (2+4*lane-2) )
            {
              double vx = sensor_fusion[i][3];
              double vy = sensor_fusion[i][4];
              double check_speed = sqrt(vx*vx + vy*vy);
              double check_car_s = sensor_fusion[i][5];

              check_car_s+=((double)prev_size*0.02*check_speed);
```

(main.cpp)

❖ **Prediction at the current time:**
For situations where conditions could suddenly change if a car changes lanes and comes in front of the ego vehicle or a car in front applies brakes suddenly, we have to do something with that information. For such cases, we just take the sensor fusion data.

Here, we are checking if any vehicle is too close to the ego vehicle:



```
for(int i=0; i< sensor_fusion.size(); i++){

  float d = sensor_fusion[i][6];
  if(d < (2+4*ego_lane+2) && d > (2+4*ego_lane-2) ){
    //check car in ego_vehicle's lane presently (not in the future)

    double check_car_s = sensor_fusion[i][5];
    if((check_car_s > ego_s) && ((check_car_s-ego_s) <20)){
      apply_brake == true;
    }
    }

  }
```

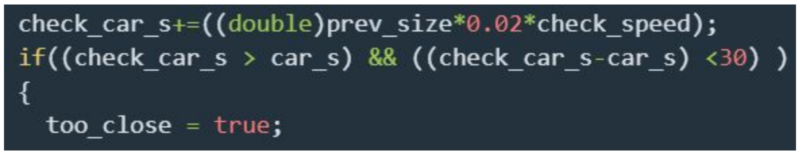
(helpers.h)

2. Behaviour Planner:

The goal of this module is to output the lane in which we want our ego vehicle to be. It could be a new lane or the same lane in which it is currently moving in.
It also updates the ref_vel which the car is supposed to follow for this iteration while laying waypoints.

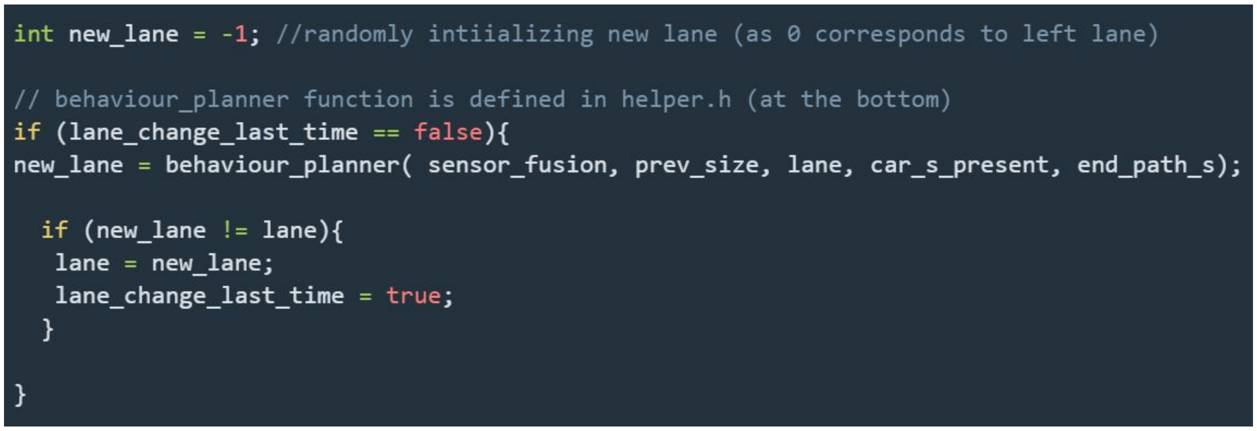
PART A: Changing lane to move to a faster lane

First, we check for a condition where a car is in front of the ego vehicle and is too close to the ego vehicle which hinders the speed of our ego vehicle



```
check_car_s+=((double)prev_size*0.02*check_speed);
if((check_car_s > car_s) && ((check_car_s-car_s) <30) )
{
  too_close = true;
```

If the vehicle is too close. We look for the possibility of lane change:



```
int new_lane = -1; //randomly intiializing new lane (as 0 corresponds to left lane)

// behaviour_planner function is defined in helper.h (at the bottom)
if (lane_change_last_time == false){
new_lane = behaviour_planner( sensor_fusion, prev_size, lane, car_s_present, end_path_s);

  if (new_lane != lane){
   lane = new_lane;
   lane_change_last_time = true;
  }

}
```

We first check if the lane was changed in the last iteration. If yes, we don't attempt to change the lane and choose to slow down. Otherwise, we call the behaviour_planner function from the helpers.h which outputs the lane for this iteration that the ego vehicle should follow. It takes in the following parameters:

- sensor_fusion: sensor fusion data containing details about traffic vehicles
- prev_size: the number of waypoints remaining from the last iteration
- lane: current lane of the ego vehicle
- car_s_present: the current "s" value of ego vehicle
- end_path_s: the future "s" value of the ego vehicle when it reaches the last waypoint that it was supposed to reach when it would eat up the waypoints from the last iteration (we would be appending new waypoints after this point so it would be useful to make predictions at this instant)
  Note: If there were no waypoints from the previous iteration i.e car is starting out,

all the predictions happen at the current instant and NOT the future as prev_size becomes 0.

Here’s the function:

```
int behaviour_planner( vector<vector<double>> sensor_fusion, int prev_size, int ego_lane, double ego_s, double
ego_prev_path_end_s)
```

We begin by defining the possible options that the car would have in each lane.

```
vector <vector<string>>options = { {"straight", "right"}, {"left", "straight", "right"}, {"left", "straight"} };
```

Eg. Left most lane will have options of either staying there or moving towards right

Then we start looping over all the possible option the ego vehicle has at the moment:

```
    vector <int> lane_val;
    vector <double> cost_lane;

    for (int i=0; i<options[ego_lane].size(); i++){
```

Our intent is to check for all the possible options if that’s possible to execute and then associate a cost to that option. In the end, we would just consider picking up the cheapest option.

For each option we consider, we update our lane to get a target lane:

```
      string direction = options[ego_lane][i];
      int target_lane;    //resulting lane whe

      if(direction=="left"){
        target_lane = ego_lane-1;
      }
      else if (direction=="right") {
        target_lane = ego_lane+1;
      }
      else if (direction=="straight") {
        target_lane = ego_lane;
      }
```

Then we start looping over all vehicles:

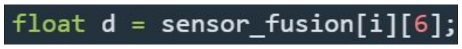
```
      for(int i =0; i< sensor_fusion.size(); i++)
      {
```

We only consider the vehicles that are in 150m range of the ego vehicle

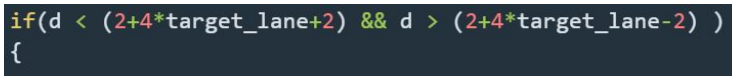
```
        double check_car_s = sensor_fusion[i][5];
        if ( abs(check_car_s - ego_s) > 150 ){
         continue;
```

For each valid vehicle, we obtain the d value.
The d value is the distance from the middle of the road. A typical lane would have a range of d values.

We further filter out vehicles based on their lane. We only consider the vehicles in the desired lane, the ego vehicle is considering to change lanes to if possible

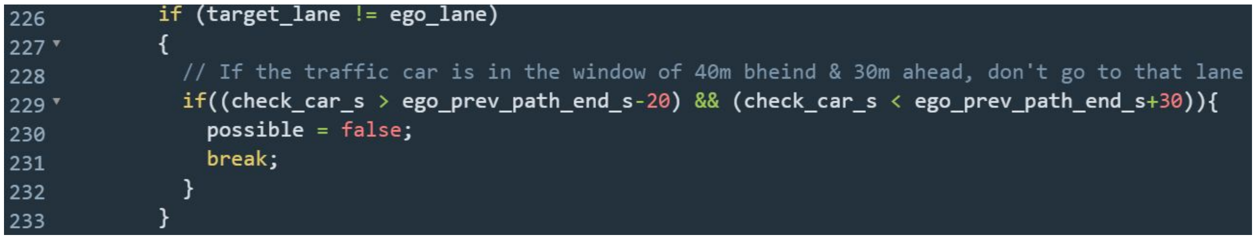


```
float d = sensor_fusion[i][6];
```



```
if(d < (2+4*target_lane+2) && d > (2+4*target_lane-2) )
{
```

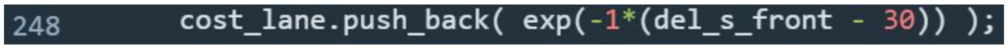
Then for these vehicles, we predict their position in the future when the ego vehicle will complete its previously pending waypoints. If the lane in consideration (target lane) is different from what the ego vehicle is moving in, we check the gap in that lane.
If there are no vehicles 20m behind and 30m ahead of ego vehicle’s s coordinate, we consider that target lane, otherwise, we remove that particular target lane from consideration and break out of the loop to look at the next lane option that the car has:



```
      if (target_lane != ego_lane)
      {
        // If the traffic car is in the window of 40m bheind & 30m ahead, don't go to that lane
        if((check_car_s > ego_prev_path_end_s-20) && (check_car_s < ego_prev_path_end_s+30)){
          possible = false;
          break;
        }
      }
```

Next, for the feasible lane options (including the current lane of the ego vehicle), we calculate the distance between the ego and the closest vehicle in front of it.
We call it: del_s_front

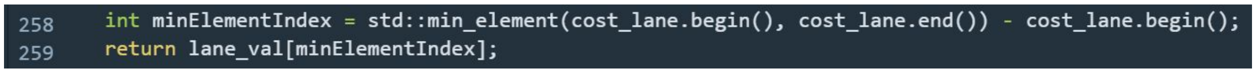
Finally, for each possible lane, we store its lane value, and assign a cost to that lane:



```
        cost_lane.push_back( exp(-1*(del_s_front - 30)) );
```

If the vehicle in front comes closer than 30m, the cost rises exponentially. If it is 30m apart, the cost would be 1. As it gets farther away, the cost function decreases exponentially.

We complete the bbehaviour_planner function by finding the lane which has the least cost associated with it:

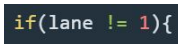


```
    int minElementIndex = std::min_element(cost_lane.begin(), cost_lane.end()) - cost_lane.begin();
    return lane_val[minElementIndex];
```

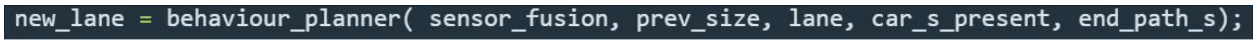
So now we have the lane value for the ego vehicle if it is too close to a vehicle in front of it.

PART B: Moving to the middle lane whenever possible

If the vehicle has a lot of free space in front of it and it is not moving in the middle lane (which is also almost empty for some distance ahead), the ego vehicle should consider changing lanes to move to the middle lane. This will decrease its chance of getting stuck in the leftmost or rightmost lane in case of high traffic.



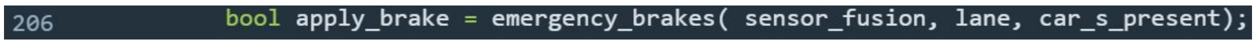
```
if(lane != 1){
```



```
new_lane = behaviour_planner( sensor_fusion, prev_size, lane, car_s_present, end_path_s);
```

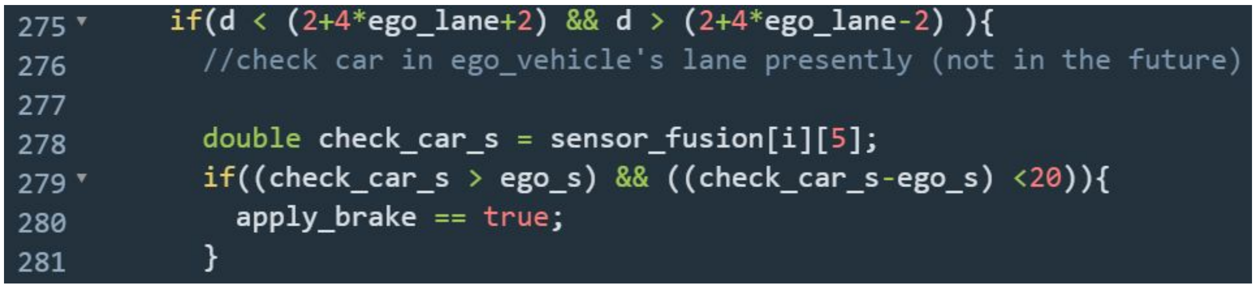
PART C: Emergency brakes

To avoid sudden collision with cars that change lanes in front of the ego vehicle or suddenly apply brakes, I tried to implement emergency brakes:



```
        bool apply_brake = emergency_brakes( sensor_fusion, lane, car_s_present);
```

The function is defined in helpers.h as follows:

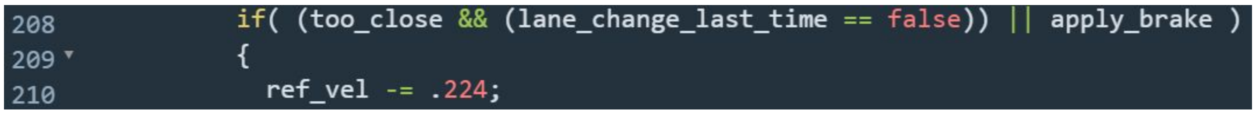


```
    if(d < (2+4*ego_lane+2) && d > (2+4*ego_lane-2) ){
      //check car in ego_vehicle's lane presently (not in the future)

      double check_car_s = sensor_fusion[i][5];
      if((check_car_s > ego_s) && ((check_car_s-ego_s) <20)){
        apply_brake == true;
      }
```

(If some other vehicle is in the same lane as that of ego vehicle and the distance is lesser than 20m, emergency brakes should be applied)

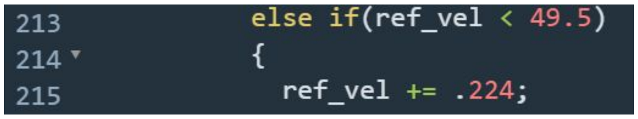
PART D: Modifying velocity of the ego vehicle for this iteration

If the ego vehicle is too close to a vehicle in front of it and it did not change lane in this iteration (the lane_change_last_time variable gets updated in every iteration to true if the car decides to change lanes)
OR
If it has to apply emergency brakes,
We decrease its velocity:



```
          if( (too_close && (lane_change_last_time == false)) || apply_brake )
          {
            ref_vel -= .224;
```

If the vehicles moving slower than the speed limit and there are no boundations on it to decrease its speed, it should increase its speed:

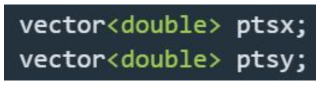


```
else if(ref_vel < 49.5)
{
  ref_vel += .224;
```

3. Trajectory Planner

The trajectory is based on the speed and lane output from the behavior, car coordinates and past path points.

We will take 5 points which will lie on the future path that our ego vehicle should walk on. We will store them in vectors as shown below.



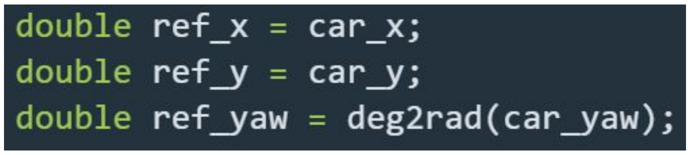
```
vector<double> ptsx;
vector<double> ptsy;
```

After that, we'll interpolate these waypoints with a spline. Once we have the x and y coordinate relation based on the spline we get, we'll take waypoints lying on that spline based on the velocity we need to maintain.

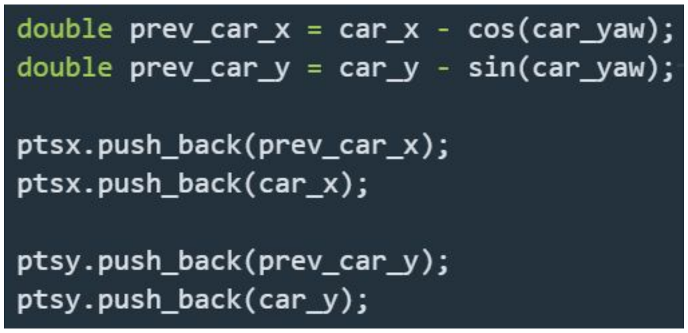
First, the last two points of the previous trajectory (or the car position if there are no previous trajectory) should be used in conjunction three points at a far distance:

- First two points:
  (These will help us in a smooth transition to the new way points that we'll create later on)

1. Case I: We DON’T have previous points



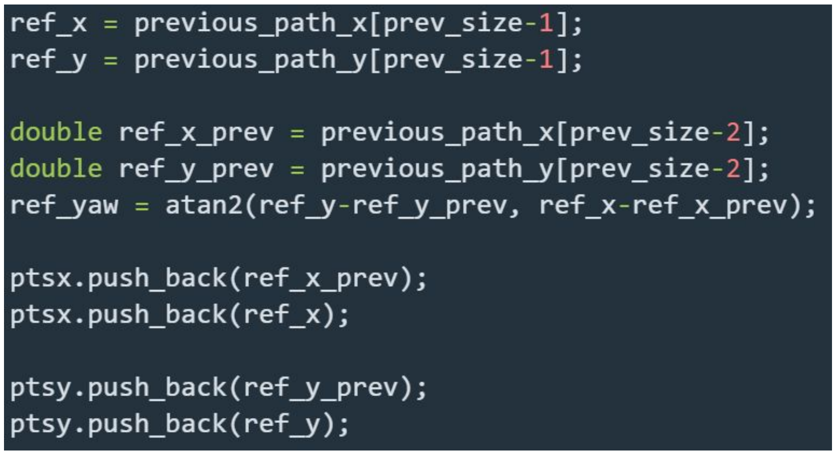
```
double ref_x = car_x;
double ref_y = car_y;
double ref_yaw = deg2rad(car_yaw);
```



```
double prev_car_x = car_x - cos(car_yaw);
double prev_car_y = car_y - sin(car_yaw);

ptsx.push_back(prev_car_x);
ptsx.push_back(car_x);

ptsy.push_back(prev_car_y);
ptsy.push_back(car_y);
```

   We use the current location and orientation of the ego vehicle. Using that we’ll generate one more pair or and y coordinates, extrapolating back in time using ego’s current yaw. Then we’ll store these two points.

2. Case II: We have previous points



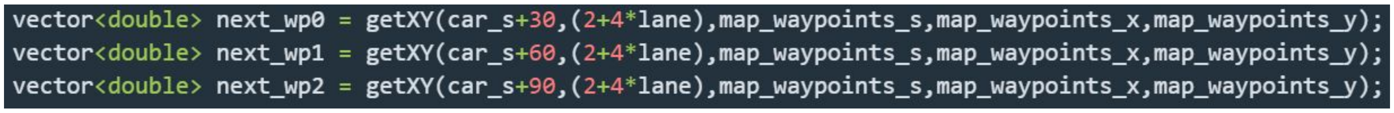
```
ref_x = previous_path_x[prev_size-1];
ref_y = previous_path_y[prev_size-1];

double ref_x_prev = previous_path_x[prev_size-2];
double ref_y_prev = previous_path_y[prev_size-2];
ref_yaw = atan2(ref_y-ref_y_prev, ref_x-ref_x_prev);

ptsx.push_back(ref_x_prev);
ptsx.push_back(ref_x);

ptsy.push_back(ref_y_prev);
ptsy.push_back(ref_y);
```

   Here, we are using the last two points from the points present in the previous path (that the ego vehicle is yet to cover but the simulator came back to it to ask for more points before it could complete those points). The last point is stored in ref_x and ref_x_prev (for x coordinates). We add them to our set of 5 points that we are going to collect to fit a spline.

Note1: Observe that ref_x is going to be the car's present position if previous points are not present. But if previous points are present, ref_x would be equal to the last point in the car’s previous point.

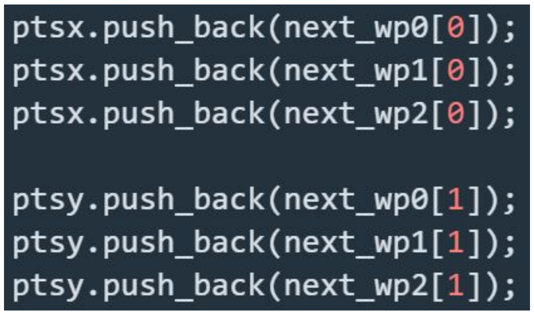
Note2: ref_x and ref_y are important coordinates for us because this is the last waypoint that car has. All the new waypoints are going to be added after this waypoint. So this coordinate would be used for coordinate transformation (which we'll come to in a bit).

Now to generate a continuous and differentiable function, we have the past two points. To this, we'll add three new points that we want the ego vehicle to pass through in the future. Here we are taking 3 points 30 meters apart from each other:



```
vector<double> next_wp0 = getXY(car_s+30,(2+4*lane),map_waypoints_s,map_waypoints_x,map_waypoints_y);
vector<double> next_wp1 = getXY(car_s+60,(2+4*lane),map_waypoints_s,map_waypoints_x,map_waypoints_y);
vector<double> next_wp2 = getXY(car_s+90,(2+4*lane),map_waypoints_s,map_waypoints_x,map_waypoints_y);
```

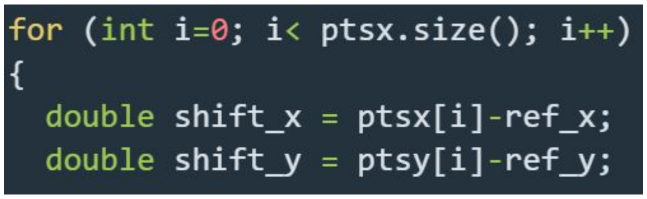
Then we'll add these 3 points to our collection of 5 points.



```
ptsx.push_back(next_wp0[0]);
ptsx.push_back(next_wp1[0]);
ptsx.push_back(next_wp2[0]);

ptsy.push_back(next_wp0[1]);
ptsy.push_back(next_wp1[1]);
ptsy.push_back(next_wp2[1]);
```

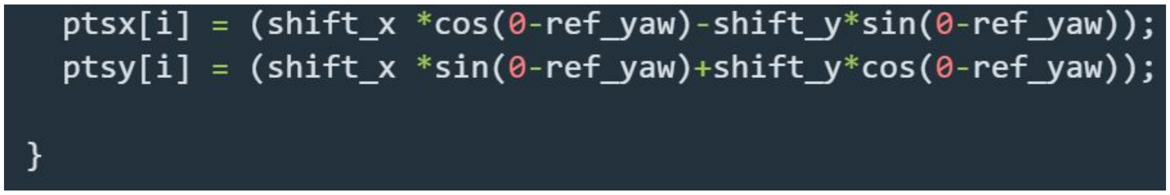
Now we'll perform Coordinate transformation to go from global frame of reference to car's local frame of reference. This is where ref_x and ref_y will be used because this is where the car would be positioned when it is supposed to add new points to it's path.

- We'll begin by shifting the origin to the local car's coordinates: ref_x and ref_y



```
for (int i=0; i< ptsx.size(); i++)
{
  double shift_x = ptsx[i]-ref_x;
  double shift_y = ptsy[i]-ref_y;
```

- Then we'll rotate the x and y axes to align it with the ego car's yaw at that point

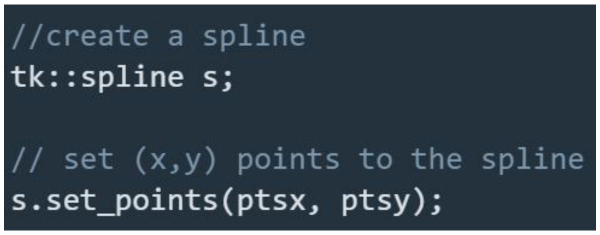


```
   ptsx[i] = (shift_x *cos(0-ref_yaw)-shift_y*sin(0-ref_yaw));
   ptsy[i] = (shift_x *sin(0-ref_yaw)+shift_y*cos(0-ref_yaw));


 }
```

To do the rotation, we use the homogeneous transformation formula.
The catch here is the value of angle that goes in cos and sin theta.

According to the theory of homogeneous transformation, we transform the points from a frame which is rotated by α to the ground frame (default frame). Here, the car's frame is rotated by ref_yaw, but our points are not expressed in terms of this frame. Our points are expressed in terms of the default frame which is present at -α with respect to the ego vehicle's frame. Now we're simply expressing points from the default frame to car frame. This is why we use -α (where α=ref_yaw). Refer to the theory of homogeneous transformation for more details.

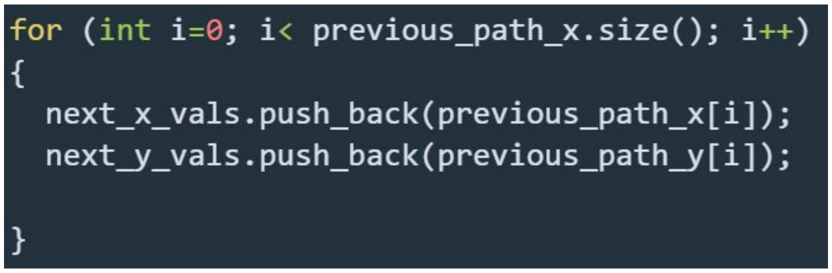
Now, we initialise spline:



```
//create a spline
tk::spline s;

// set (x,y) points to the spline
s.set_points(ptsx, ptsy);
```

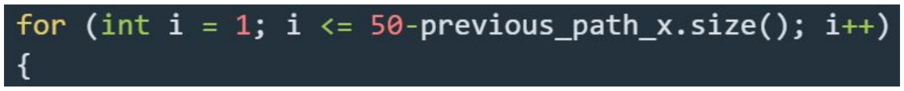
(Refer spline's documentation to understand this syntax)

Before we feed new waypoints to the simulator, we have to again add the previous points that the ego vehicle is supposed to follow but haven't walked on yet.



```
for (int i=0; i< previous_path_x.size(); i++)
{
  next_x_vals.push_back(previous_path_x[i]);
  next_y_vals.push_back(previous_path_y[i]);

}
```

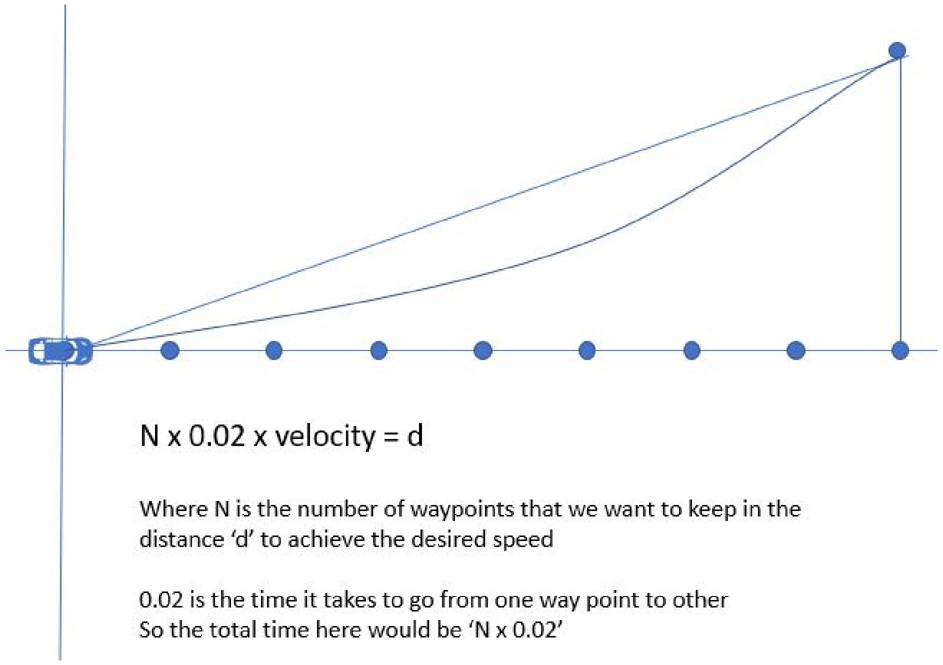
The simulator asks for 50 waypoints in each iteration. So for example, if the previous points amount to 35 waypoints, we just have to add 15 new waypoints. So we initiate a loop to get those remaining points:



```
for (int i = 1; i <= 50-previous_path_x.size(); i++)
{
```

Now the only thing that remains is finding those remaining waypoints that lie on this spline. We have a neat trick to do that as described as follows:

1. Our coordinate system will look as follows after coordinate transformation to car's local coordinate (with our car sitting on the origin facing X axis)



$N \times 0.02 \times velocity = d$

Where N is the number of waypoints that we want to keep in the distance 'd' to achieve the desired speed

0.02 is the time it takes to go from one way point to other
So the total time here would be 'N x 0.02'

2. The car would go in a curved path as shown in the image above, but we'll make some approximations. We have the spline function expresses in this coordinate frame (which'll look like this curve)
3. We'll start by taking a point on this spline function: $\Delta x = 30$, $\Delta y = s(30)$
4. Then we'll find the distance of our ego vehicle to this above point that we've taken using pythagoras theorem. We'll call this distance 'd' as shown in the image above.

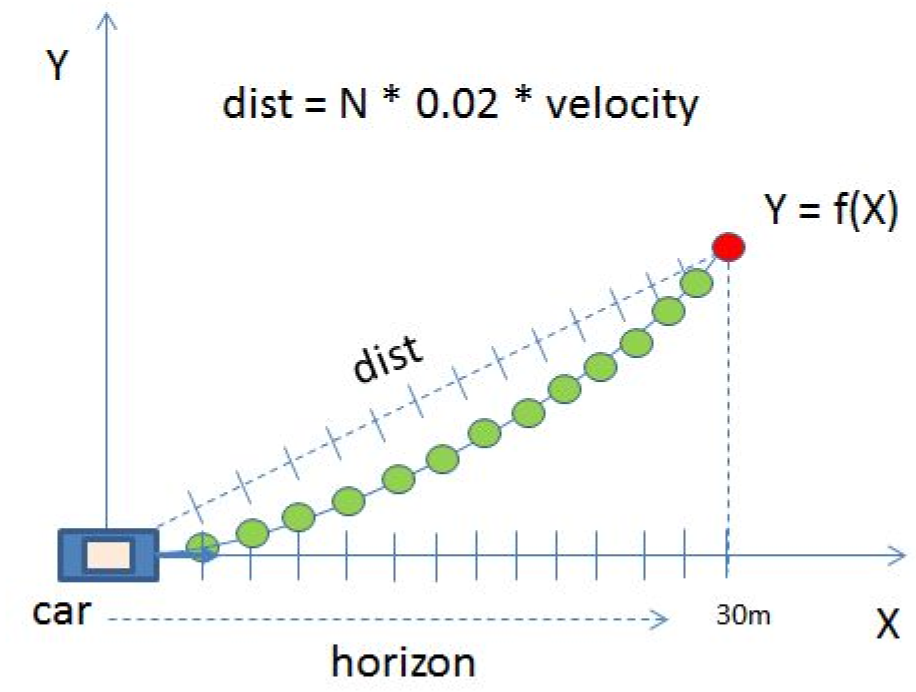
   The above two steps will be coded as follows:

```
double target_x = 30.0;
double target_y = s(target_x);

double target_dist = sqrt((target_x)*(target_x)+(target_y)*(target_y));

double x_add_on = 0;
```

5. Now we'll find the number of waypoints that'll lie on this portion of the curve that we're looking at. The simulator moves from one waypoint to another in 0.02 seconds. (0.02 x velocity) will give us the distance travelled between two consecutive waypoints. If we divide by the length of the curve, we'll get the number of waypoints that'll lie in this segment of the curve. Here we are approximating the length of the section of curve with 'd' that we calculated using pythagoras theorem in point number 4 above.

6. So now we'll have the number of points that we can accommodate approximately in this curve (as shown in green color in the image below).

   So we'll divide ∆x by N to get the x coordinates of all these waypoints. We'll put them in the equation of the spline curve to find the y coordinate



   We write this code in the for loop to achieve this:

```
double N = (target_dist/(0.02*ref_vel/2.24));
double x_point = x_add_on+(target_x)/N;
double y_point = s(x_point);

x_add_on = x_point;
```

   The x_add_on keeps track of where we were last time on the x axis shown in the picture shown above as we advance along the x direction.

7. Each waypoint we achieve in this process would still be expressed in the local car's coordinates. Now we need to transform it back to the global coordinate.

   We'll call these new coordinates of the new waypoints **x_ref, y_ref**

```
double x_ref = x_point;
double y_ref = y_point;
```

Then we’ll transform it back to global coordinate by using +ve α (where α will go in sin and cos theta as theta).   α=ref_yaw here. (Reason: We’re transforming back to the global coordinate system from the one that’s rotated by α).

We also have to add the x and y coordinate values of the origin of the car's local coordinate system (expressed in the global system) i.e. **ref_x and ref_y**

```
x_point = ref_x + (x_ref *cos(ref_yaw)) - (y_ref *sin(ref_yaw));
y_point = ref_y + (x_ref *sin(ref_yaw)) + (y_ref *cos(ref_yaw));
```

Note: Please don’t confuse yourself between x_ref and ref_x.

8. We finally add these new waypoints (transformed back to global coordinate) to our container of 50 items:

```
next_x_vals.push_back(x_point);
next_y_vals.push_back(y_point);
```

Further scope of improvement:

Improvements can be made in all the three modules:

1. Prediction

   - Instead of taking constant velocity model based prediction, we can maintain a vehicle object for each traffic vehicle. This way we’ll be able to keep a track of their velocity and hence will be able to find acceleration. Then we can use equations of motion to find their motion parameters

   - For the traffic vehicles close to the ego vehicle, we can consider:
     > Straight lane trajectory
     > Lane change trajectories

     We can generate the lane change trajectory in the same way we’re doing for the ego vehicle. Then we can track position and velocity details of these traffic vehicles from sensor fusion data. Finally, we can use Naive Bayes classifier to know where exactly are the neighbouring cars moving. This will be helpful in handling collisions due to sudden lane changes or sudden brakes applied by vehicles in front lane

2. Behaviour Planning
    - We can introduce Finite state machines: particularly prepare lane change. This would help the car adjust it's speed to get in a particular gap in some other lane if it is stuck in traffic
    - We can make better lane change decisions using more cost functions. We can consider cost function which capture information like traffic density in lanes
    - Introduce a better emergency brake system

3. Trajectory Planning
   One of the possible trajectories is given to us by our spline in this project. What we can do is: use the result from this spline to get parameters for Jerk Minimisation trajectory generation technique and then plan more trajectories. The benefit would be that we'll be able to take in account more cost functions which would help us in