**Project Highway Driving:**

**Project Description:**

The goal of this project is to build a path planner that creates smooth, safe trajectories for the car to follow. The highway track has other vehicles, all going different speeds, but approximately obeying the 50 MPH speed limit.

The car transmits its location, along with its sensor fusion data, which estimates the location of all the vehicles on the same side of the road.

**Point Paths:**

The path planner should output a list of x and y global map coordinates. Each pair of x and y coordinates is a point, and all of the points together form a trajectory. You can use any number of points that you want, but the x list should be the same length as the y list.

Every 20 ms the car moves to the next point on the list. The car's new rotation becomes the line between the previous waypoint and the car's new location.The car moves from point to point perfectly, so you don't have to worry about building a controller for this project.

The velocity of the car depends on the spacing of the points. Because the car moves to a new waypoint every 20ms, the larger the spacing between points, the faster the car will travel. The speed goal is to have the car traveling at (but not above) the 50 MPH speed limit as often as possible. But there will be times when traffic gets in the way.

**Solution:**

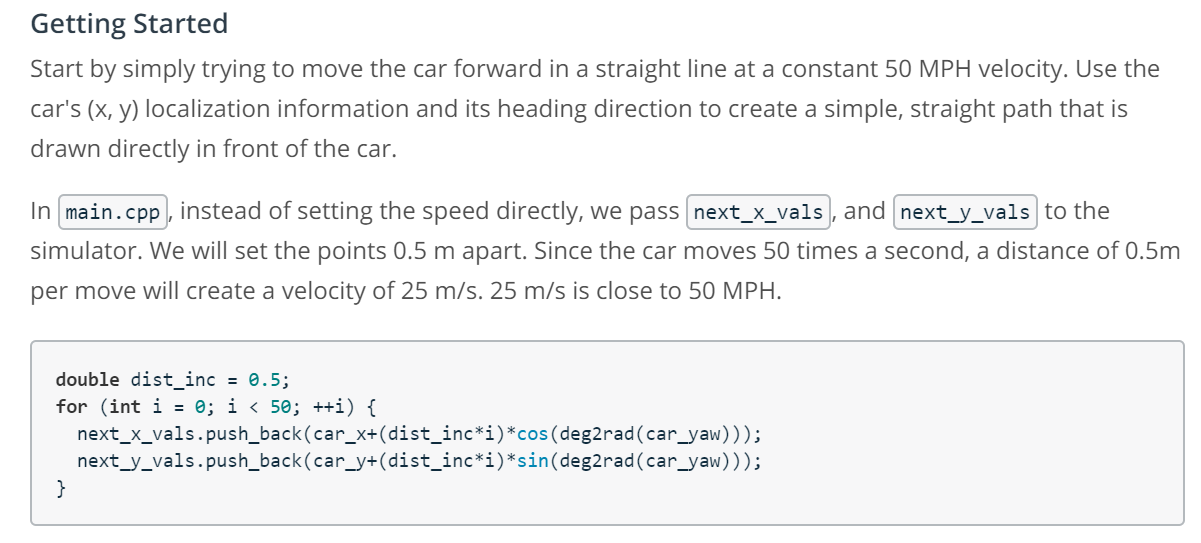
**General Info:**

* The ProjectQ&A video in the Highway Driving lessons in the course is of great help in getting a solution.
* In the code also , the steps are explained in detail in the form of comments.
* Downloaded spline.h, a single file spline library, from the following location <https://github.com/ttk592/spline> and included it in the main.cpp

**Get the car running:**

When we run the startup code in the simulator the car doesn’t move.This is because the future path was not getting populated as we have an empty (x,y) list. In the “Getting Started” lesson the instructor has given a code snippet to get the car moving. This is very helpful in checking whether the code is getting compiled properly, by seeing the car moving in the simulator.

Following is the description and code change that would be done. When we check in simulator, we see that even though the car is moving, it violates almost all the rules. So now we will look into solving these problems one by one. Some of these issues which we are going to address in coming sections are - car doesn’t stay in its lane, cold start problem, maximum jerk issue, collision with other cars and lane change.



**Solution to keep the car in its own lane :**

The code above uses (x,y) coordinates. We can address the issue of keeping the car in its lane by using the frenet coordinates i.e (s,d) .

**Solution for smooth trajectory generation:**

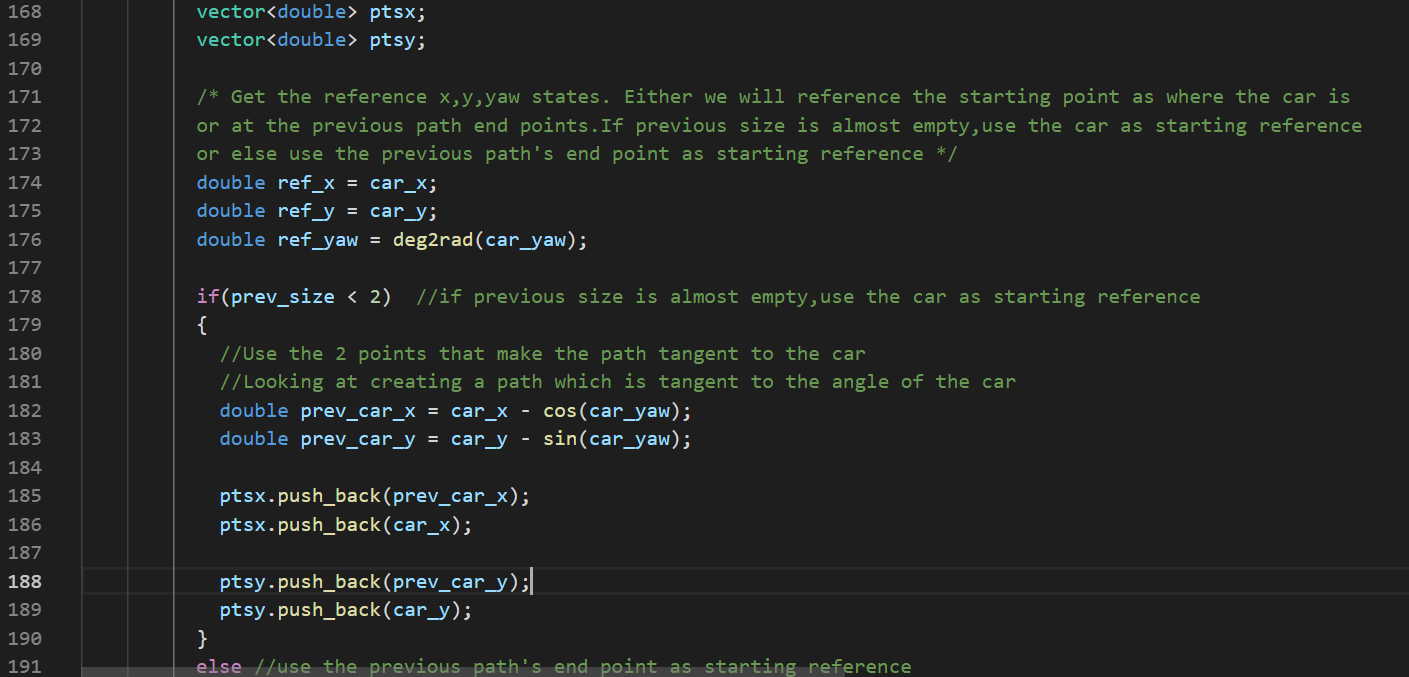
Map is made up of sparse way points.At the corners and in some line segments way points gets crowded and that causes acceleration which leads to jerk issues and speed violation.

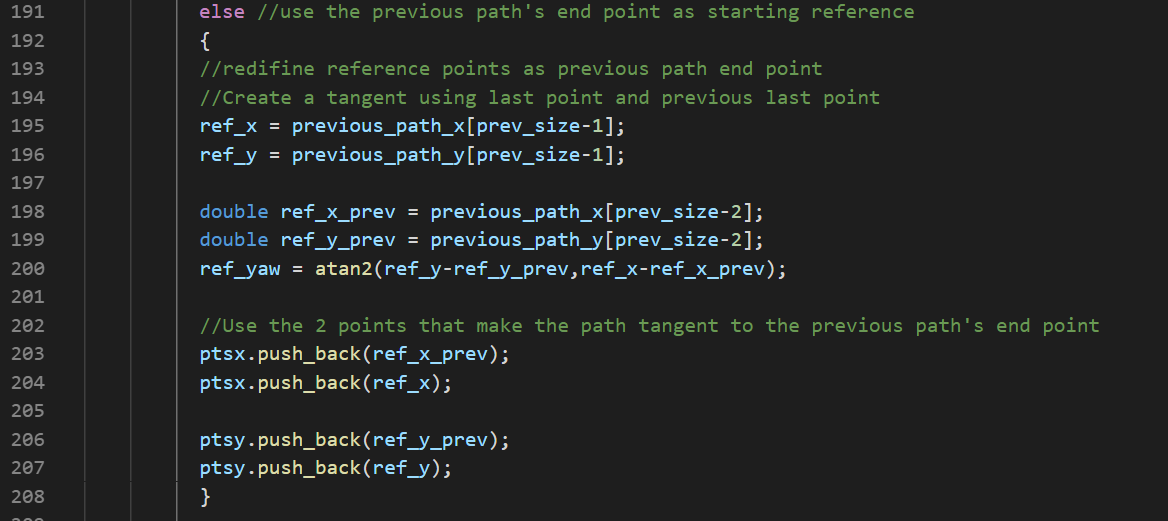
To keep the path smooth we can use spline. Spline is better than polynomial as it's guaranteed to go through all the points and it's a piecewise functions of polynomials. I have downloaded spline.h, a single file spline library, from the following location <https://github.com/ttk592/spline> and included it in the main.cpp

One approach is to take 2 points far apart , may be 30 mts, and then fit a spline and use the points in between. Create a list of widely placed spaced(x,y) waypoints (evenly spaced at 30m). Later we will interpolate these waypoints with a spline and fill it in with more points that control spline.

Get the reference x,y,yaw states. Either we will reference the starting point as where the car is or at the previous path end points.If previous size is almost empty,use the car as starting reference or else use the previous path's end point as starting reference.

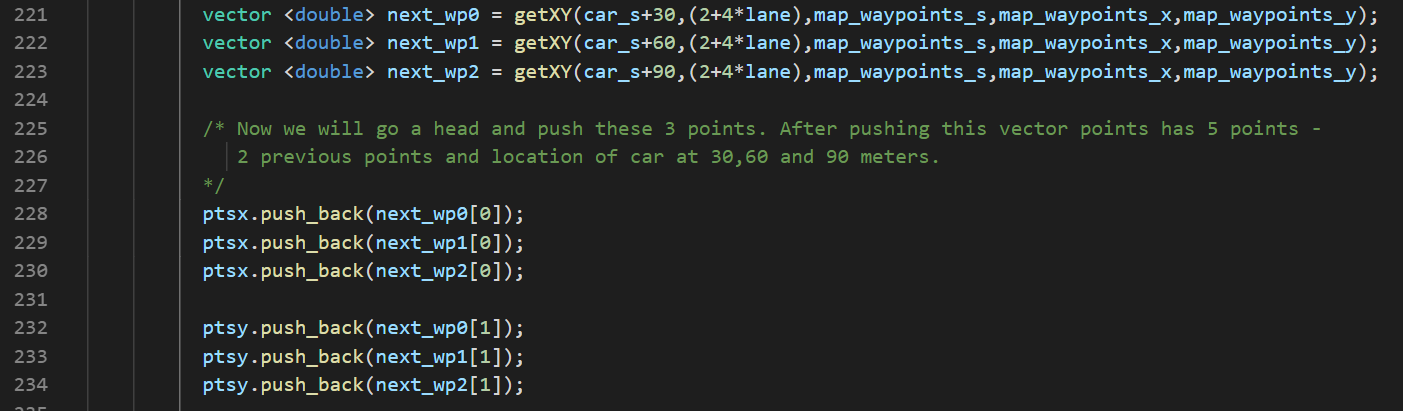
Following is the code snippet which implements this :





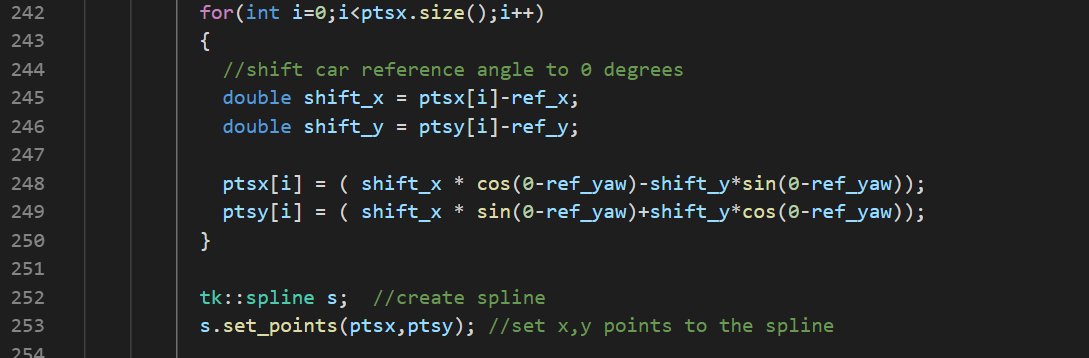
In summary, in the above code , we took last couple of points from the previous path the car was following and then calculated what angle the car was heading using those last couple of points.Then we pushed them on to list of previous points.So far we have pushed 2 x and 2 y. So we have 2 points basically and this is our starting reference

In Frenet coordinates we will add , evenly 30 mts spaced points, ahead of the starting reference. Earlier we used 50 points, now we are using 3 points. Instead of being just .5 meters space now they are all the way to 30m. Actually 30mts is good enough, as during lane change it would to be smooth and lesser value can make lane change little aggressive. The following code snippet implements this logic.



To make sure the car or the last point of the previous path is at (0,0), the origin, and its angle at zero degrees we do a transformation to car's local coordinate system. We are basically shifting to look into the car reference frame.This transformation is basically shift and rotation. Then we are creating a spline and adding all the reference points created.

The following code snippet implements this change.



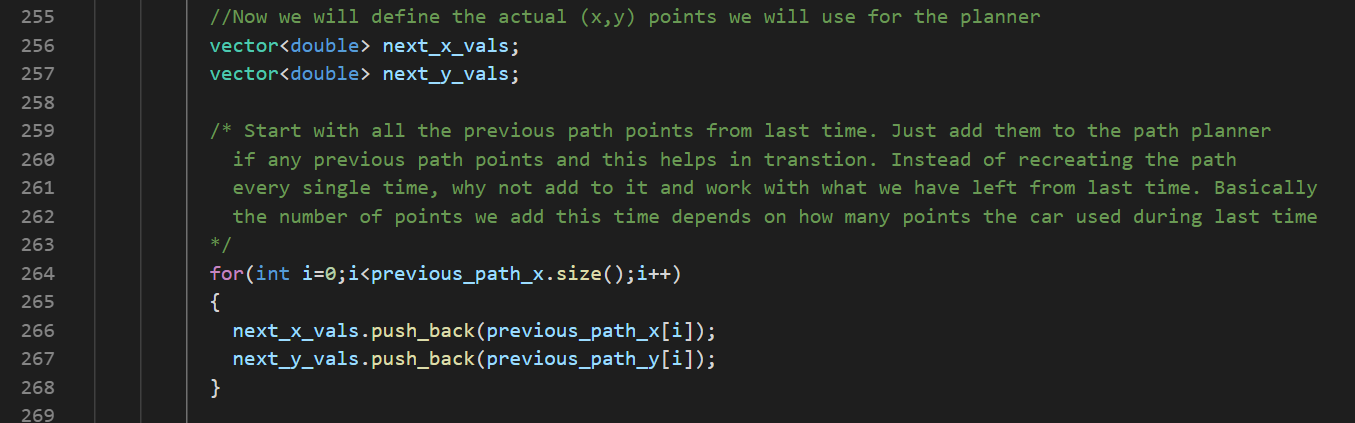
**Solution for getting actual (x,y) path for the planner:**

Until now we have used the previous points and reference points, but we need to identify the actual (x,y) paths that would be used for path planning.

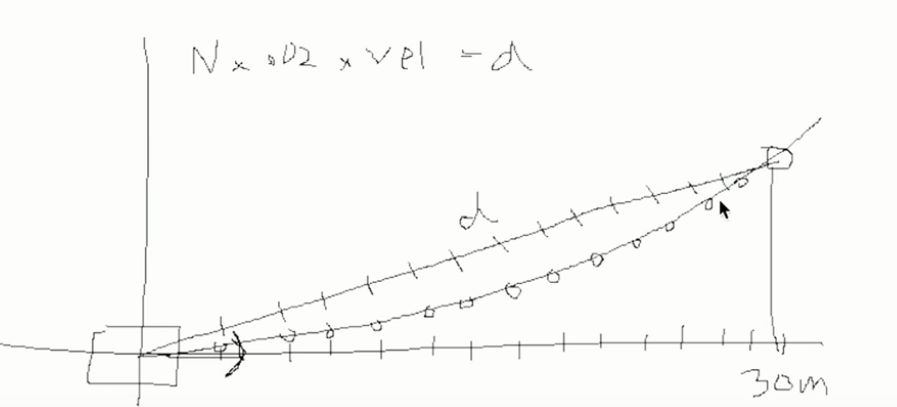
Using information from the previous path ensures that there is a smooth transition from cycle to cycle. But the more waypoints we use from the previous path, the less the new path will reflect dynamic changes in the environment.Ideally, we might only use a few waypoints from the previous path and then generate the rest of the new path based on new data from the car's sensor fusion information.

The simulator runs a cycle every 20 ms (50 frames per second), but our C++ path planning program will provide a new path at least one 20 ms cycle behind. The simulator will simply keep progressing down its last given path while it waits for a new generated path.This means that using previous path data becomes even more important when higher latency is involved. Imagine, for instance, that there is a 500ms delay in sending a new path to the simulator. As long as the new path incorporates a sufficient length of the previous path, the transition will still be smooth.A concern, though, is how accurately we can predict other traffic 1-2 seconds into the future. An advantage of newly generated paths is that they take into account the most up-to-date state of other traffic.

The next path (x,y) values are initiated in the below code



Following is the approach we will use to generate the new path. The diagram below is car’s local coordinate system. The goal is to space out points in the spline in such a way that car travels at the desired speed ( 50mph).

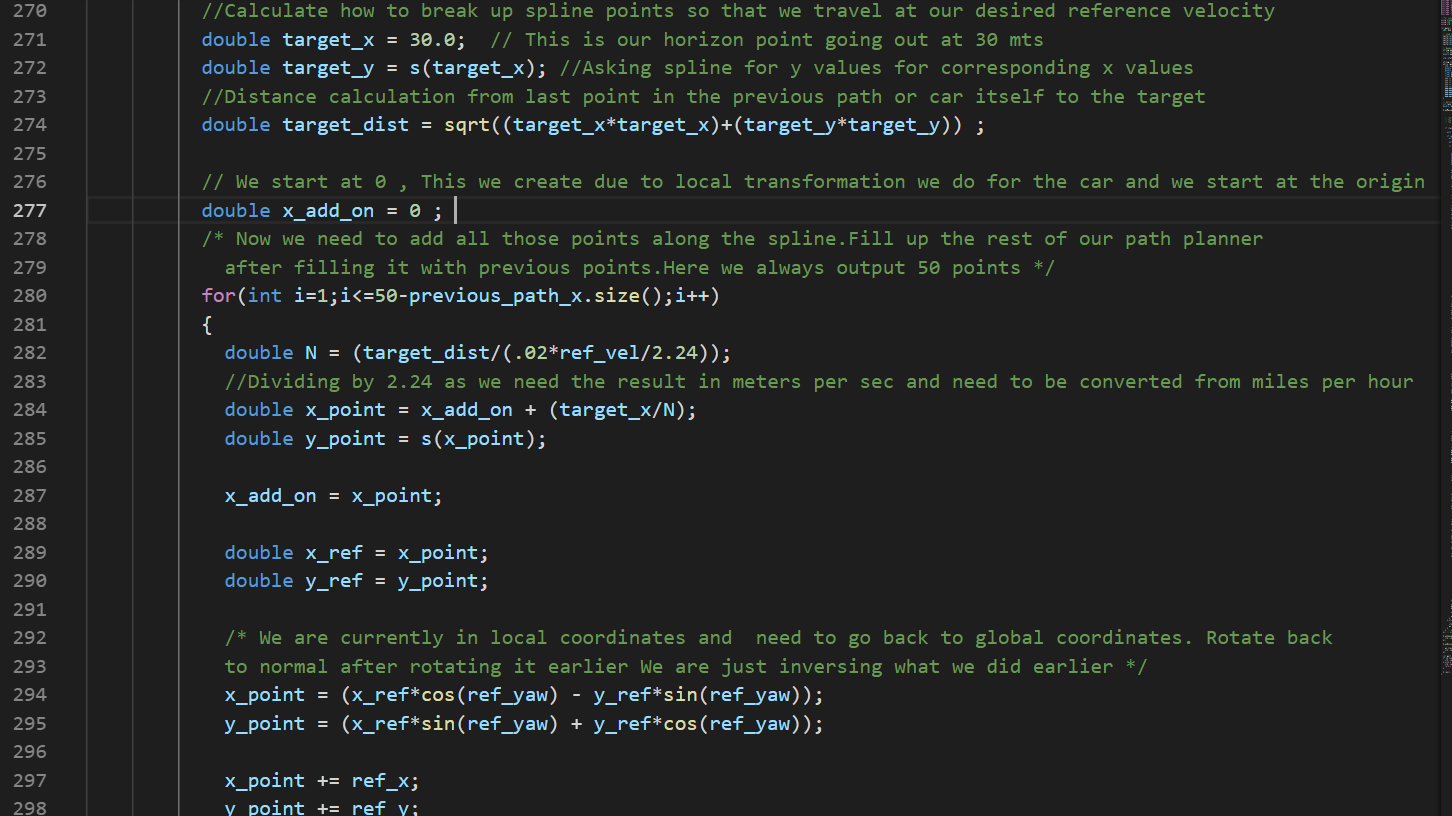


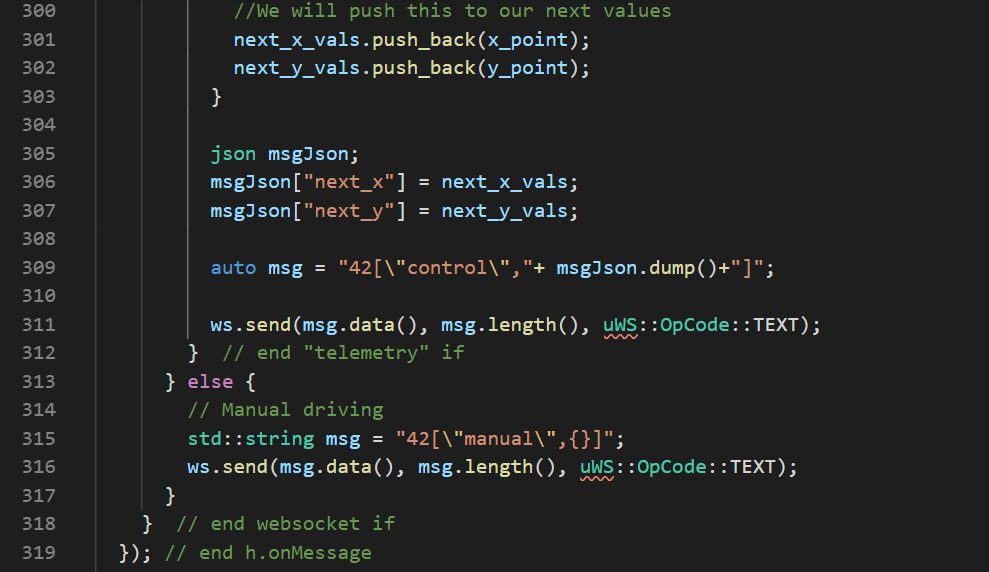
First we look at some horizon value(target) , in our implementation its 30 mts, and we will figure out where this point lies in the spline ( using S function). We then calculate the ‘d’ value i.e the distance between the car and the target. This ‘d’ need to be split into ‘n’ pieces and this value is obtained using the following equation:

D = n \* 0.02 \* velocity

By doing this we have calculated what n should be. And since this is a right angled triangle, we can break up x also into n pieces like we've broken up hypotenuse into n pieces.And then, once we have these x values we can get the corresponding y values using the spline S.

The following code implements the approach explained above.





**Collision problem Solution:**

The simulator ( sensor fusion) reports other car’s values like vx,vy, s & d.We will make use of this information in finding out whether any car are going in front of us, how collision can be prevented and also use this info during lane change.

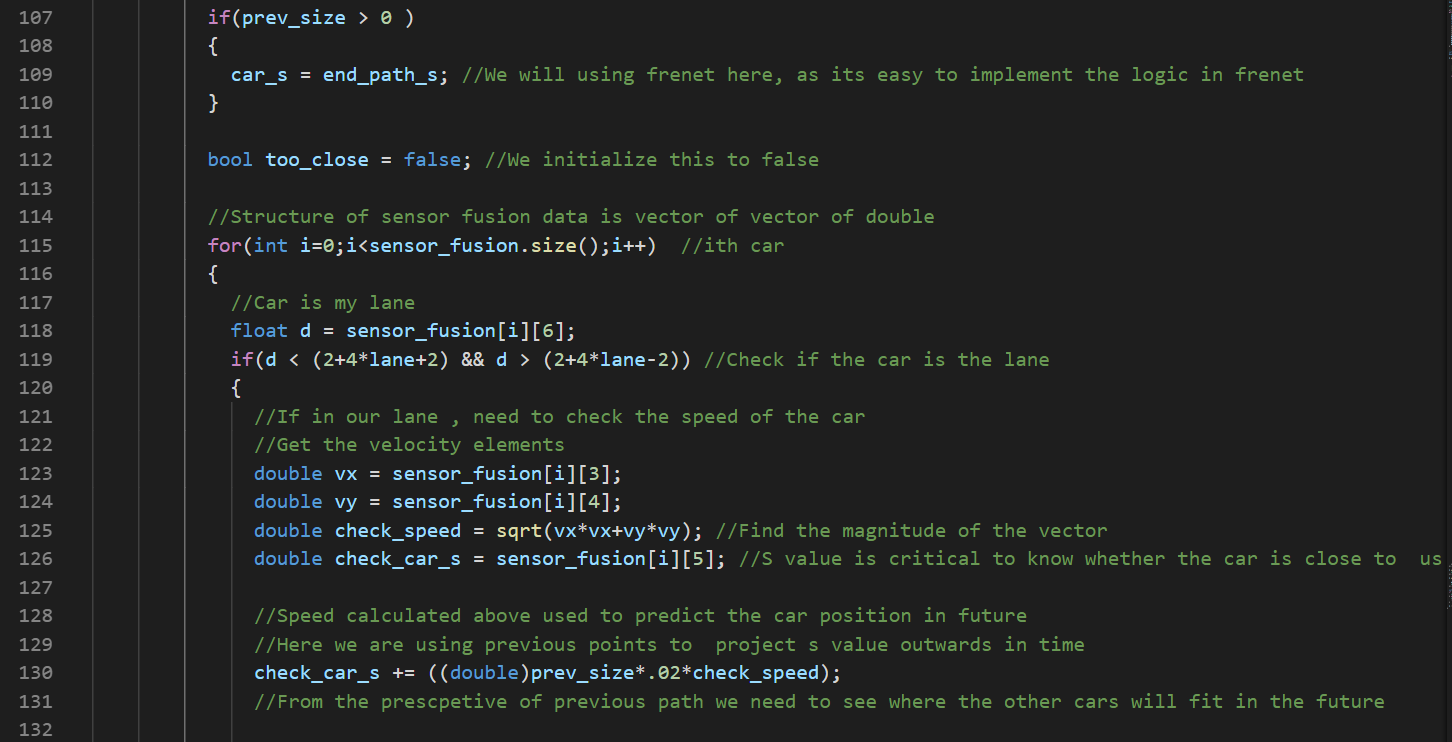
Need to take appropriate actions, If we see that some other car is going slow in our lane and there is a chance of collision.

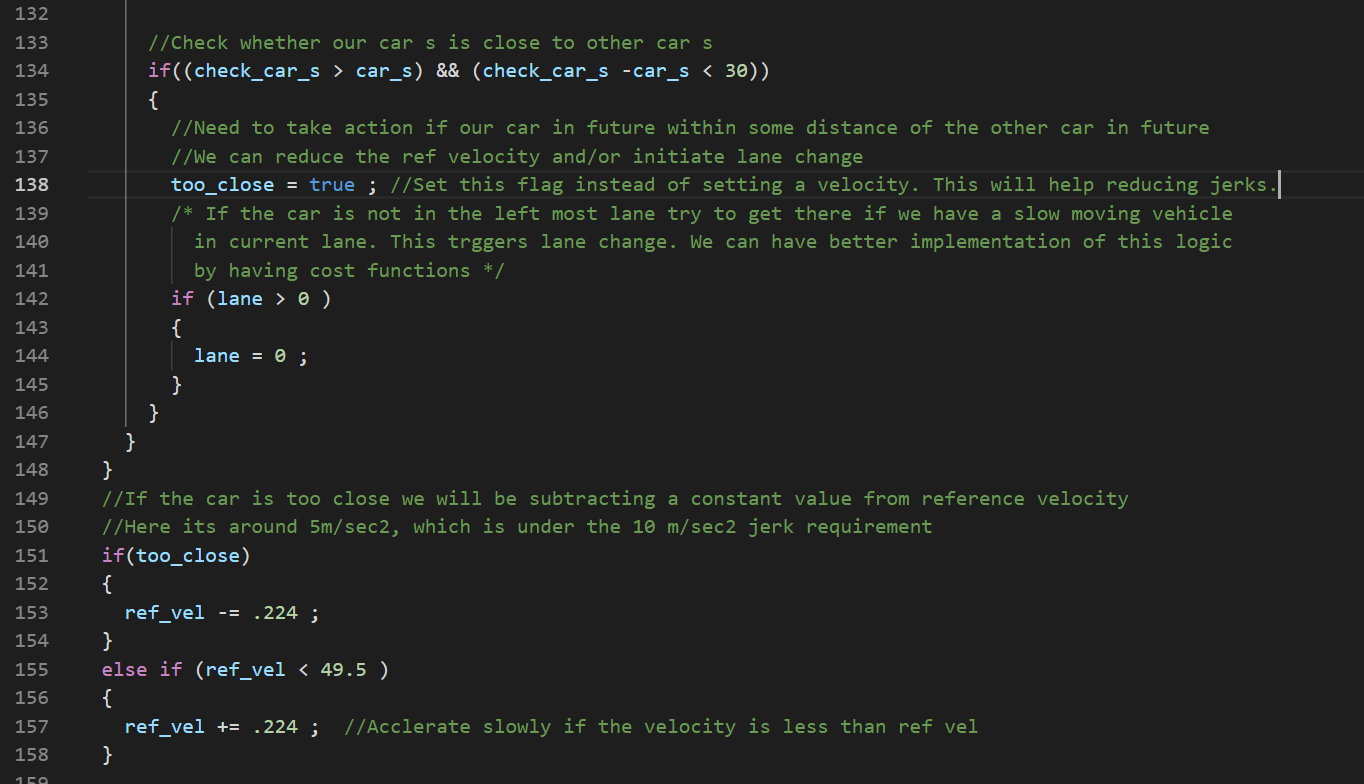
We will iterate through each car in the sensor fusion data and check whether that car is in our lane and check the speed. Using the speed we can predict the future position of the car.

If there is another car in our lane and in future if its within a certain distance, then we need to take some action. In our implementation we reduce the reference velocity by a fixed amount.Here its around 5m/sec2, which is under the 10 m/sec2 jerk requirement.

When we implemented this, we realized that once the car slows down on identifying a slow moving vehicle, it doesn’t get back to its desired speed but continue with the reduced speed.So we added additional condition that if there are no slow moving vehicle now in our lane and our vehicle is going slower than the reference velocity we will increment the reference velocity by a constant speed (5m/sec2). This also helps in code start problem.

The other action we can take is lane change also, to avoid following a slow moving vehicle. In our implementation we do the lane change once to lane 0 from 1. We have used a simple logic. We can make it even better by implementing a cost function and make lane change with the objective of reducing speed by following all the constraints. But in the following code we have not implemented this. The solution for this can be found in the course videos.

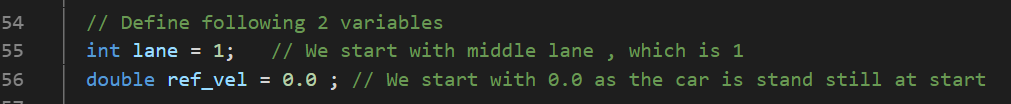




**Cold Start problem solution:**

We have seen that when the simulator starts the car accelerates to 50mph from 0 mph within first iteration. This causes jerk and maximum acceleration limit. The code above takes care of this in most of the cases, as we accelerate or decelerate slowly in most of the scenarios, other than during start.

The solution to the cold start problem is simple one, we initiate the reference velocity as 0.0 in the start of the code.



With this we have addressed all the issues and now the car completes a lap without violating any conditions.