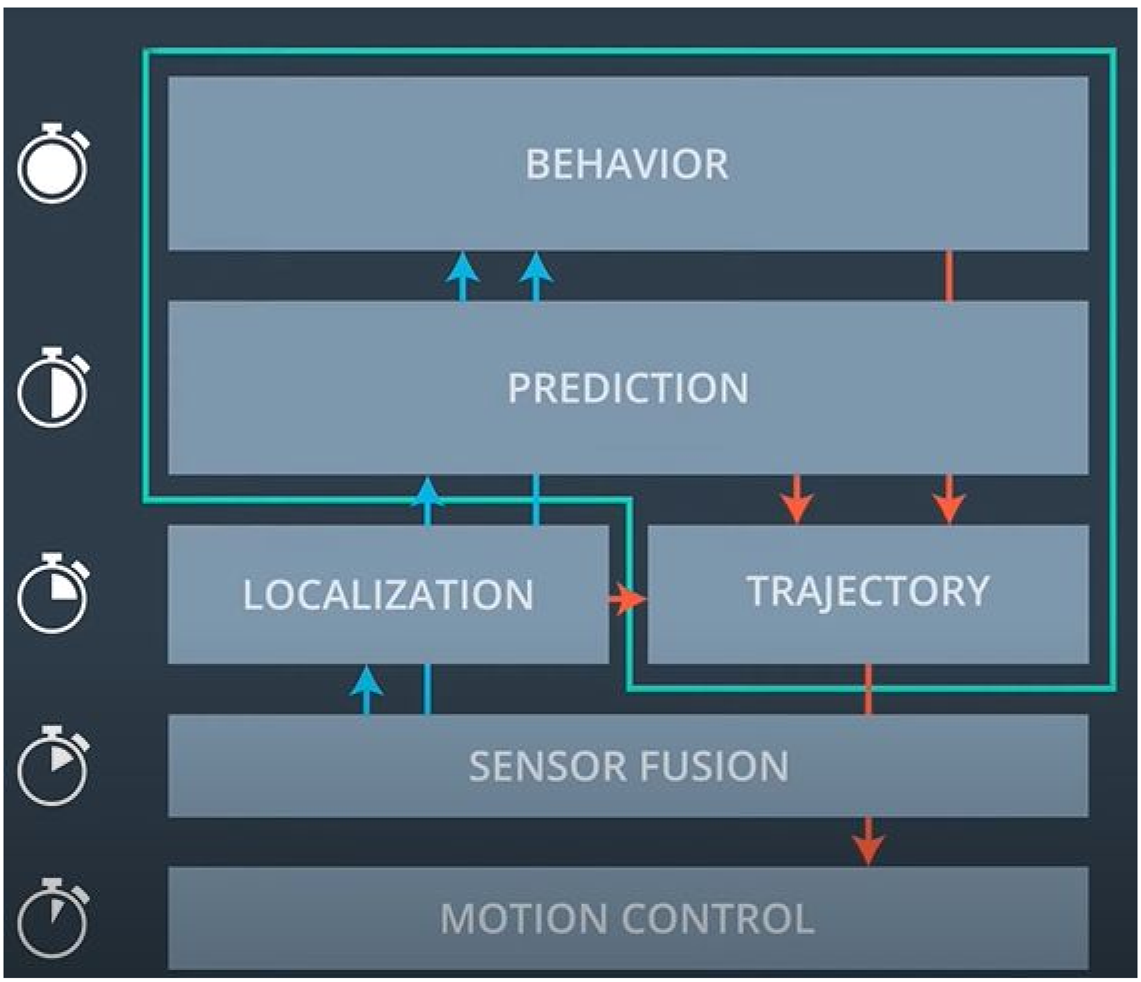
Highway Driving

Overview

"Highway Driving", this project involves making a toy car navigate autonomously avoiding potential obstacles, safely maneuvering around them avoiding collision, changing lanes safely by following a quintic polynomial with optimal cost and that minimizes jerk. "Sensor Fusion" data is provided from the simulator, predictions over the neighboring cars are made, then the behavior is being calculated, then choosing the optimal path (that is to keep the lane or to take left or to take right lanes) for a particular behavior.



Goals

- Collision Free maneuvers.
- Jerk minimized Lane changing trajectories.
- Doesn't exceed the maximum acceleration of 1g.
- Able to change lanes.
- Doesn't exceed the speed limit.

Brief

The code has to implement the following parts :

1. Predictions.
2. Behaviour Planning.
3. Trajectory Generation (using spline technique for numerical interpolation).

Predictions:

Based on the sensor fusion optimal estimate data we get useful data about the objects around us. For this, we get data about the neighboring cars around the ego vehicle, this data includes the car's id, x,y,s,d,vx,vy , representing identification number, map position, fernet coordinates, velocity components respectively. Using the following code we are able to find whether there are cars to the left, front, and right of the ego vehicle for modeling the behavior (143 - 210 lines from the main.cpp code).

```
for(int i =0;i<sensor_fusion.size();i++)
          {
            //get the current neighbouring car s , d , present lane
            double nei_car_s = sensor_fusion[i][5];
            double nei_car_d = sensor_fusion[i][6];
            int nei_car_lane  = get_lane(nei_car_d);
             if(nei_car_lane < 0)
             {
               continue;
             }
            double vx = sensor_fusion[i][3];
            double vy = sensor_fusion[i][4];
            double nei_car_speed = sqrt(vx*vx + vy*vy);
```

```
            /*check whether the neighbouring car is
            1.Too close(based on the preferred buffer distance between the
two)
            2.In the relative left lane
            3.In the relative right lane
            */

            nei_car_s+=((double)nei_car_speed * 0.02 * prev_size);
//observed car might have travelled further within the tema that we have
received the measurement .
            if(nei_car_lane == lane)
            {
              //car in the same lane as that of the neighbouring car
              if (nei_car_s>car_s)
              {
                //making sure that we are not calculating teh closeness to
the previous car .
                if(nei_car_s - car_s <=preferred_buffer)
                {
                  //infront car is too close to the ego car
                  car_front = true;
                }
              }
            }
            else
            {
              //if car is not in the same lane
              if(nei_car_lane  == lane-1)
              {
                /*
                inorder to flag a neighbouring car present to the left of
the car it has to satisfy the following condition
                the neighbouring car's s coordinates must be within the ego
car +/- preferred buffer
                if not this means that we have enough space to make a
transition to the left.
                  "car_s-preferred_buffer < nei_car_s <
car_s+preferred_buffer"
                */
                //neighbouring car is left to the current car
                if(car_s-preferred_buffer<nei_car_s &&
car_s+preferred_buffer>nei_car_s)
```

```
                {
                  //this means that there is clearly a car and we cannot
make a transition to the left.
                  car_left = true;
                }
              }else if(nei_car_lane == lane+1)
              {
                 /*
                inorder to flag a neighbouring car present to the left of
the car it has to satisfy the following condition
                the neighbouring car's s coordinates must be within the ego
car +/- preferred buffer
                if not this means that we have enough space to make a
transition to the left.
                  "car_s-preferred_buffer < nei_car_s <
car_s+preferred_buffer"
                */
                //neighbouring car is right to the current car
                if(car_s-preferred_buffer<nei_car_s &&
car_s+preferred_buffer>nei_car_s)
                {
                  //this means that there is clearly a car and we cannot
make a transition to the left.
                  car_right = true;
                }
              }
            }
          }
```

Behavior Planning :

After the above predictions, we might want to accelerate or wait for lane change either to the left or right lane. A programmed FSM's are being used for this. Following code is used for behavior planning (lines 212 - 250 in main. cpp are used for this).

```
 /*we now have enough knowledge on the cars around us , we plan the
behaviour
          behaviour include :
          1. if there is a car infront and too close then we have to change
the lanes either to the left or right based on avaialability , when none is
availabe we decrese our velocity inorder not to collide;
          2. if ther is no car infront and too far away then, we can
```

```
maintain the lane speeding up or for preference we can change to the center
lane based on the avialability
          */
          if(car_front)
          {
            if(!car_left && lane>0)
            {
              //there is a space to the left and we are not crossing the
highway yellow line
              lane-=1;
            }else if(!car_right && lane < 2)
            {
              //there is a space to the right and we are not crossing the
highway boundaries
              lane+=1;
            }
            std::cout<<"in changing acceleration slowly"<<std::endl;
            ref_vel-=0.3584;
          }
          else
          {
            if(lane !=1)
            {
              //car not in the middle lane ,
              if((lane  == 0 && !car_right) || (lane  == 2 && !car_left))
              {
                //then we can change to left lane that is lane-=1
                lane = 1;
              }
            }
if(ref_vel < 49.5)
            {
              ref_vel+=0.3584; //acelerating if the reference velocity is
less that some velocity ,,since velocity can read 49.5+/-5.00
            }
          }
```

Trajectory Generation:

After the above behavior planning, we need to plan a perfect trajectory from the current car position to the respective point in the same lane or in some other lane either to the left or right, a perfect trajectory minimizes the jerk involved in lane changing, avoids collision, smooth enough.

For a smooth transition function generation between the current following trajectory and the trajectory we intended to follow, we include the sample points from the previous trajectory and calculate the ego car yaw from these points. We generate a spline interpolating the waypoints from the current car position (transformed map coordinates from map reference frame to the car frame i.e., taking origin with respect to the car) to the waypoints on the intended lane. In order to maintain a constant 50MPH speed along the maneuver, we need to sample the points on the spline at a particular interval. We then transform the points to the map coordinate space again, in order to make the ego car follow this trajectory in the real simulation world.

Following code used for trajectory generation(lines from 255 - 356 main.cpp is used for this task).

```
 //variables to store the sample points to define a spline
          vector<double> point_x;
          vector<double> point_y;


          double reference_yaw = deg2rad(car_yaw);
          double reference_x = car_x;
          double reference_y = car_y;
          //including two previous points for smooth tansition

          if(!(prev_size < 2))
          {
            // we have enough points to define a yaw according to the slope
of the points

            reference_x = previous_path_x[prev_size - 1];
            reference_y = previous_path_y[prev_size - 1];
            double pre_prev_x = previous_path_x[prev_size - 2];
            double pre_prev_y = previous_path_y[prev_size - 2];
                  //getting the reference yaw
            reference_yaw = atan2(reference_y - pre_prev_y, reference_x -
pre_prev_x);

            point_x.push_back(pre_prev_x);
            point_x.push_back(reference_x);

            point_y.push_back(pre_prev_y);
            point_y.push_back(reference_y);
```

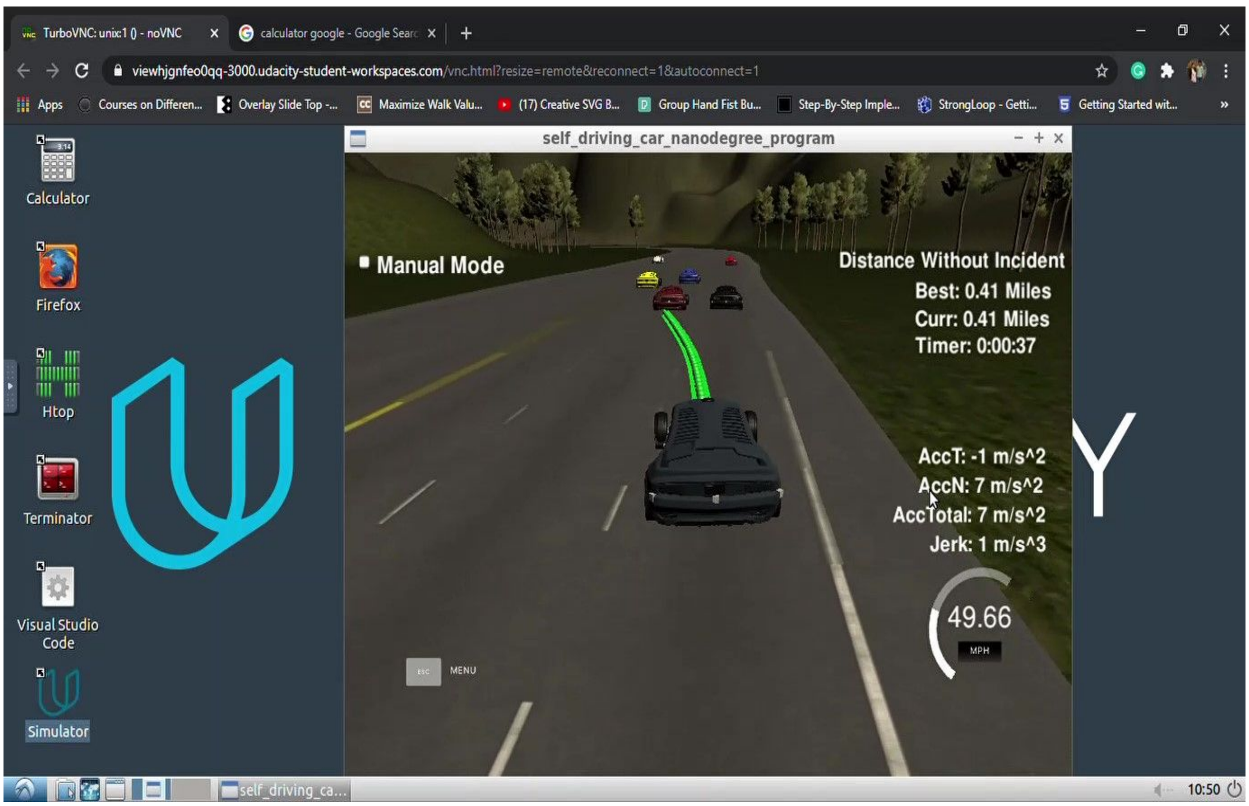
```
          }
          else
          {
            //there are not enough previous path points to include in the
points
            //so we extrapolate backwards based on the car present yaw
            double poin_x = car_x - cos(car_yaw);
            double poin_y = car_y - sin(car_yaw);
            point_x.push_back(poin_x);
            point_x.push_back(car_x);

            point_y.push_back(poin_y);
            point_y.push_back(car_y);

          }

          //next we have to generate some more points into future using the
lane changed and also the 30 gapped s values
          for(int i =30;i<=90;i+=30)
          {
            vector<double> wp = getXY(car_s+i, 2+4*lane,
map_waypoints_s,map_waypoints_x,map_waypoints_y);
          point_x.push_back(wp[0]);

          point_y.push_back(wp[1]);
          }

          /*now we got the way points(discretizing the trajectory) in map
space ...now have to transform them into car reference frame for easy
calculation of the spline points before we transform them again in mapspace
for plotting in the simulation*/

          for(int i =0;i<point_x.size();i++)
           {
                  double transl_x = point_x[i] - reference_x;
             double transl_y = point_y[i] - reference_y;
             point_x[i] = transl_x*cos(0-reference_yaw) -
transl_y*sin(0-reference_yaw);
             point_y[i] = transl_x*sin(0-reference_yaw) +
transl_y*cos(0-reference_yaw);
           }
```

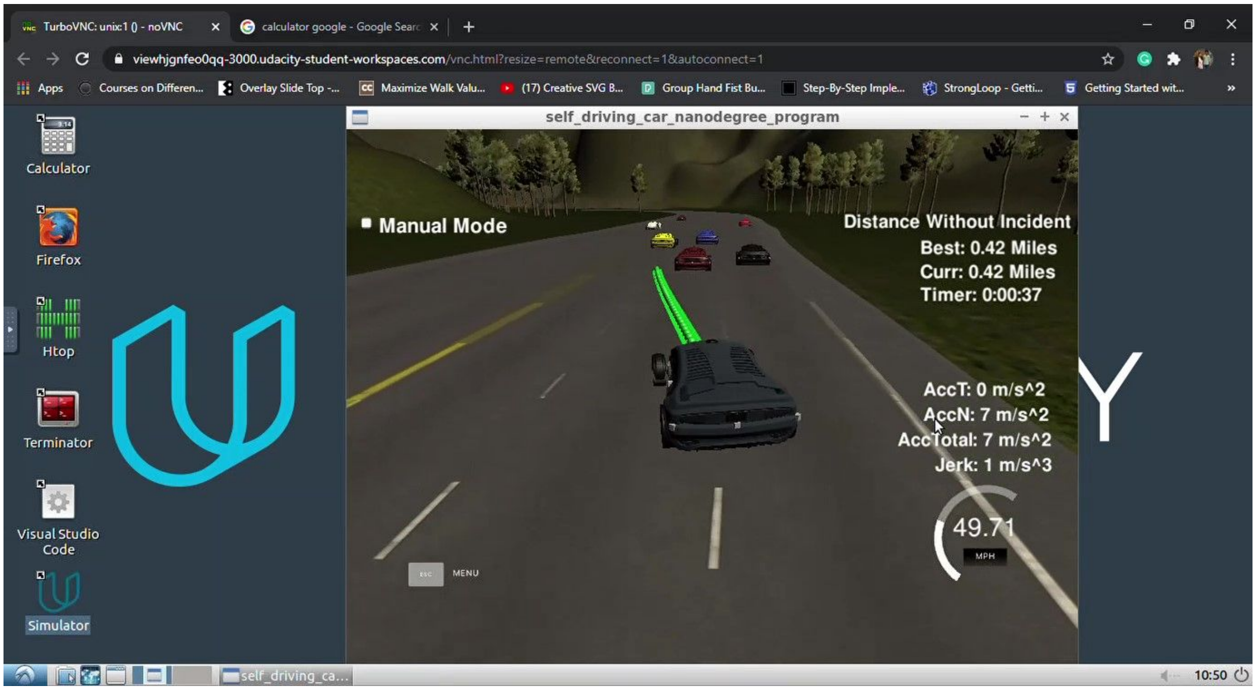
```
          //now we got the transformed coordinates of the waypoints on the
trajectory
          //now inorder to make them match the speed specification given
that is 50MPH ..
          //we need to get the sampled points on interpolated polynomial
at intervals
          //equal to total_end_dist/(0.02*target_speed(in m/s ..divide by
2.24))

          //first keep the previous path points for smooth transition
         for(int i = 0; i < prev_size; i++)
         {
           next_x_vals.push_back(previous_path_x[i]);
           next_y_vals.push_back(previous_path_y[i]);
         }

          //interpolating the waypoints using cubic spline technique
          // it is under tk namespace in spline.h header file
          tk::spline spl;
          spl.set_points(point_x,point_y);

          //now we have to sample the interpolated polynomial and include
enough of them so that we have 50 points.
          double x = 30;
          double y = spl(x);
          double dist = sqrt(x*x+y*y); //becuase we have shifted the
points to make the car position as origin
          double dummy_x = 0;
         double N = dist / (0.02 * ref_vel/2.24);
          for(int i=0;i<50 -prev_size;i++)
          {
           double x_point = dummy_x + x / N;
           double y_point = spl(x_point);

           dummy_x = x_point;

           double x_dum = x_point;
           double y_dum = y_point;

           // Rotating back to normal after rotating it earlier.
           x_point = x_dum * cos(reference_yaw) - y_dum *
sin(reference_yaw);
```

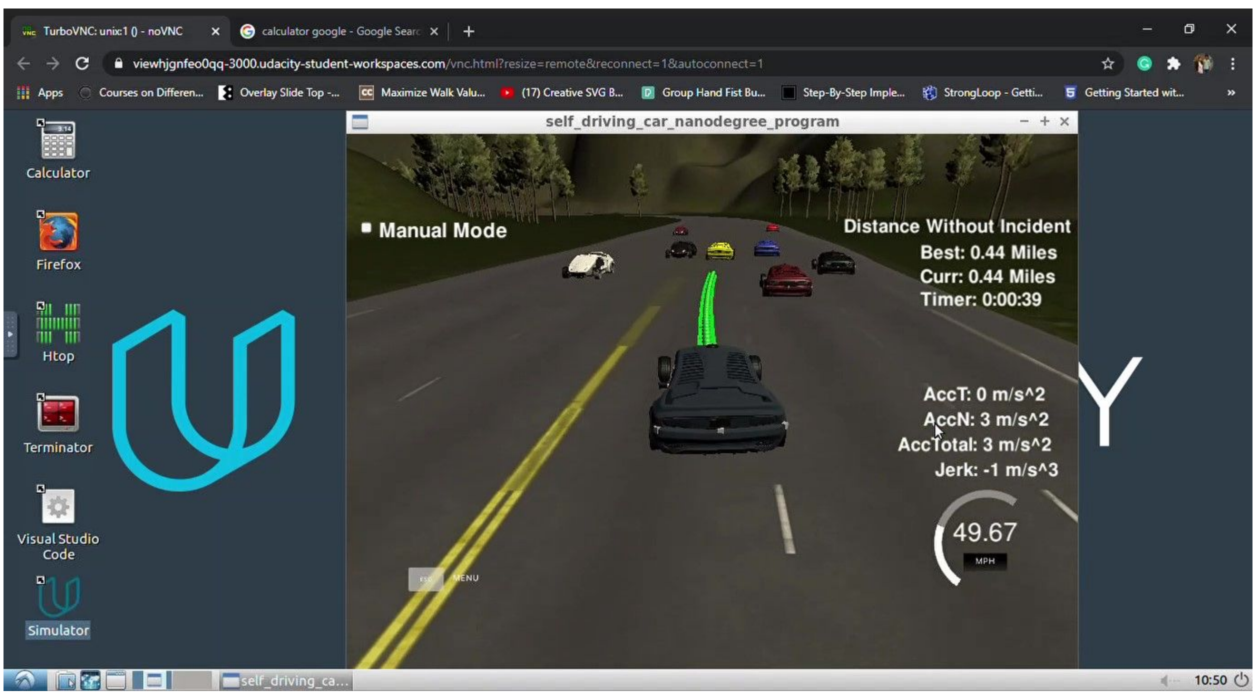
```
            y_point = x_dum * sin(reference_yaw) + y_dum *
cos(reference_yaw);

            x_point += reference_x;
            y_point += reference_y;
             next_x_vals.push_back(x_point);
             next_y_vals.push_back(y_point);
           }
```

Output :

- Output pictures of ego vehicle doing a tight maneuver




TurboVNC: unix:1 () - noVNC
calculator google - Google Searc
viewhjgnfeo0qq-3000.udacity-student-workspaces.com/vnc.html?resize=remote&reconnect=1&autoconnect=1
Calculator
Firefox
Htop
Terminator
Visual Studio Code
Simulator
self_driving_car_nanodegree_program
Manual Mode
Distance Without Incident
Best: 0.42 Miles
Curr: 0.42 Miles
Timer: 0:00:37
AccT: 0 m/s^2
AccN: 7 m/s^2
AccTotal: 7 m/s^2
Jerk: 1 m/s^3
49.71
MPH
MENU
self_driving_ca...
10:50


TurboVNC: unix:1 () - noVNC
calculator google - Google Searc
viewhjgnfeo0qq-3000.udacity-student-workspaces.com/vnc.html?resize=remote&reconnect=1&autoconnect=1
Calculator
Firefox
Htop
Terminator
Visual Studio Code
Simulator
self_driving_car_nanodegree_program
Manual Mode
Distance Without Incident
Best: 0.44 Miles
Curr: 0.44 Miles
Timer: 0:00:39
AccT: 0 m/s^2
AccN: 3 m/s^2
AccTotal: 3 m/s^2
Jerk: -1 m/s^3
49.67
MPH
MENU
self_driving_ca...
10:50

To Improve:

To improve the current algorithm :

- we can further add the cost optimization function that calculates the cost for speed, cost for acceleration limiting, cost for respective lane changing.
- In my opinion, we can use adaptive speed controlling (PID controllers) to control the speed continuously through an adaptive amount rather than a fixed decrease in the velocity.

References :

- Referred the Q/A video section code for developing the above algorithm.