Path Planning

A) Self Driving Car Path Planning Project

The goals / steps of this project are the following:

a) Code must compile without errors with cmake and make.

b) The top right screen of the simulator shows the current/best miles driven without incident. Incidents include exceeding acceleration/jerk/speed, collision, and driving outside of the lanes. Each incident case is also listed below in more detail.

1. The car doesn't drive faster than the speed limit. Also the car isn't driving much slower than speed limit unless obstructed by traffic.
2. The car does not exceed a total acceleration of 10 m/s^2 and a jerk of 10 m/s^3.
3. The car must not come into contact with any of the other cars on the road.
4. The car doesn't spend more than a 3 second length out side the lane lanes during changing lanes, and every other time the car stays inside one of the 3 lanes on the right hand side of the road.
5. The car is able to smoothly change lanes when it makes sense to do so, such as when behind a slower moving car and an adjacent lane is clear of other traffic.

Image References:

Image-1 : Source code structure of project & vehicle class (a Sample)
Image-2 : A spline of three cubic plonomial (example)
Image-3 : Hashdef file parameters
Image-4 : Smooth crossing of left vehicle
Image-5 : Smooth crossing of right side vehicle
Image-6 : Lane change in left side
Image-7 : Lane change while 2-lanes occupied by vehicle in parrellel
Image-8 : Sucessfull long drive without any incidence (10.40 miles)
Image-9 : Sucessfull long drive without any incidence (14.40 miles)

➢ Files Submitted & Code Quality

Submission includes all required files and can be used to run the simulator in autonomous mode, my project includes the following files:

1. Project Structure : SelfdrivingCar_pathPlanning repository structure has 3 folders src, data, build and independent files for compilation and setup
   src : this folder contains source code i.e header, CPP, JSON and Eigen3.3 Library
   data : this folder contails highway map csv file as a data source for simulation
   build : this folder is used to compile project and build executable (path_planning)
2. Compilation log : code compiled without any error and a log file maintained in build (compile.log file)
3. Video's (*.mp4) : Project execution is recorded in *.mp4 for demo, file as;
   - PathPlanning.mp4
4. examples (*.png) : containing two folders (outcome) with outcome images for, scenario's
   - *Smooth crossing of left vehicle*
   - *Smooth crossing of right vehicle*
   - *Lane change in left side*
   - *Smooth crossing of left vehicle*
   - *Smooth crossing of right side vehicle*
   - *Lane change in left side*
   - *Lane change while 2-lanes occupied by vehicle in parrellel*
   - *Sucessfull long drive without any incidence*
5. Pathplanning-writeup.pdf describe project, summarizing result and objectives achieved

- **Submission includes functional code & other files**
  SelfdrivingCar_pathPlanning project contains a fully functional code, also recorded video and images outcome as reference.

- **Submission code is usable and readable**
  The SelfdrivingCar_pathPlanning contains the code for functionalities mentined in goal/objectives. The file containes definition of various function, their execution outcome, visualisation for specific objectives.

  - This report describe each function how algorithm work and brief of code syntax.
  - Code has modularity, independent functions for specific purposes
  - No duplicacy of code syntax and mostly system defined parameters (to avoid hardcodings).

(B) Functionalies overview

Path Planning project has different sections to implement needed functionalities, described as;

- **main.cpp – project framework**

  Project defines a number of independent mathematical functions (trignometric, euclidean distance and waypoints calculations as taugh in course structure, as summarized;

  1. Trignomatric

```
constexpr double pi() { return M_PI; }
double deg2rad(double x) { return x * pi() / 180; }
double rad2deg(double x) { return x * 180 / pi(); }
double distance(double x1, double y1, double x2, double y2)
```

  2. Waypoints manipulation (Path, Trajectory & Co-ordinate transformation)

```
int ClosestWaypoint(double x, double y, vector<double> maps_x, vector<double> maps_y)
int NextWaypoint(double x, double y, double theta, vector<double> maps_x,
vector<double> maps_y)
vector<double> getFrenet(double x, double y, double theta, vector<double> maps_x,
vector<double> maps_y, vector<double> maps_s)
vector<double> getXY(double s, double d, vector<double> maps_s, vector<double>
maps_x, vector<double> maps_y)
```

  Standard message exchange/recieved on WebSocket (Simulator vs Project)

  In main() method call first activity is to recieve a message and decode same for further processing of path planning. OnMessage() method is standard function and architected for course to work quicly and move on with a sub attribute (event) “TELEMETRY” to do subsequent processing.

  Hereon, natural flow of program runs (on message recieve) as follows:

  1. Initialization & decleration of - local variable decleration, reading path planning CSV and setting up a log file
  2. Construct nearby waypoints of nearby area (with smoothing – taking in consideration previous and future points) and process for wrap by using spline interpolation
  3. Determine ego car and construct vehicle objects
  4. Determine interpolated waypoints, calculate speed/accelaration and differentiating trajectories
  5. Generate prediction frome sensor fusion data – read data format for each car, add ADAS warning capabilities (left or right car, collision or speed limitations ...etc)
  6. Identify best trajectory and produce new path
  7. Send newly generated data for simulation

- **Vehicle Identification (Position, motion – longitudnal/transverse and accelaration)**

1. Vehicle definition – class, constructor, destructor and functionalities algorithms
2. functionalities – get frenet trajectory, vehicle available states, lane data and actions taken
3. **Source code (in src)** : vehicle.cpp and vehicle.h

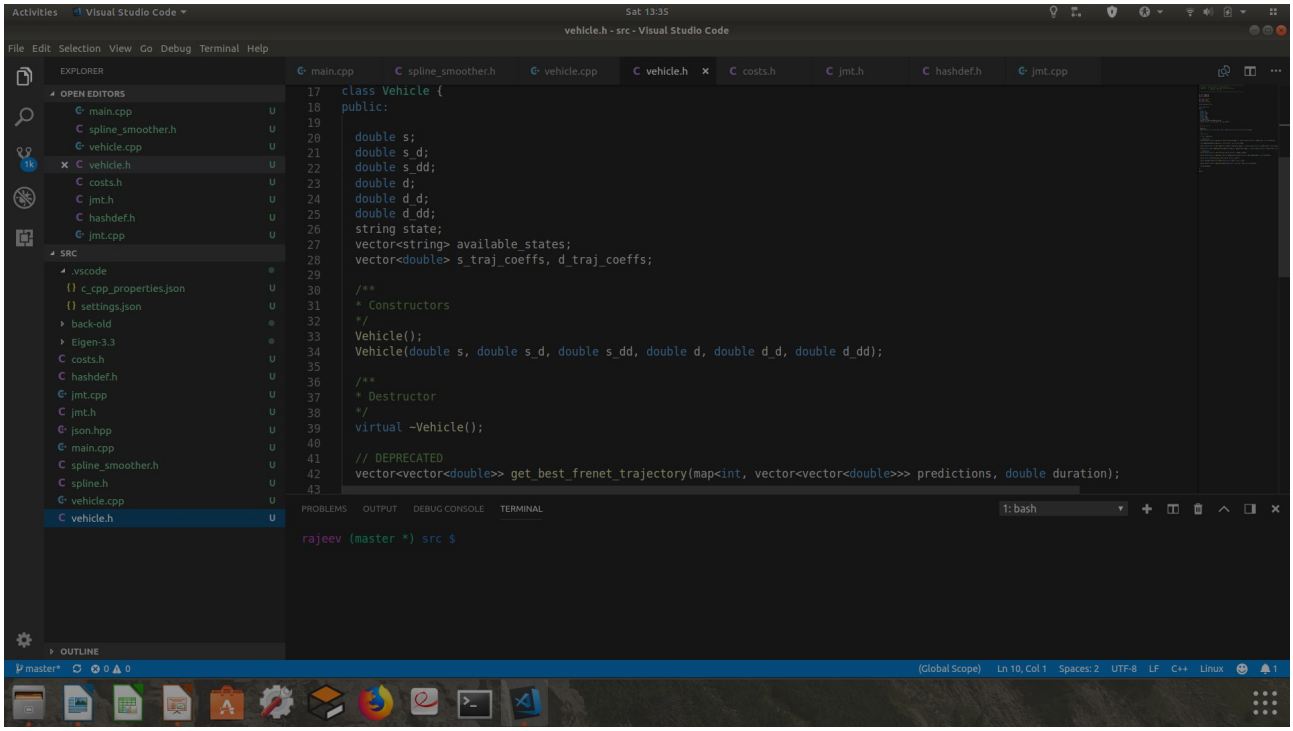


Image-1 : Source code structure of project & vehicle class (a Sample)

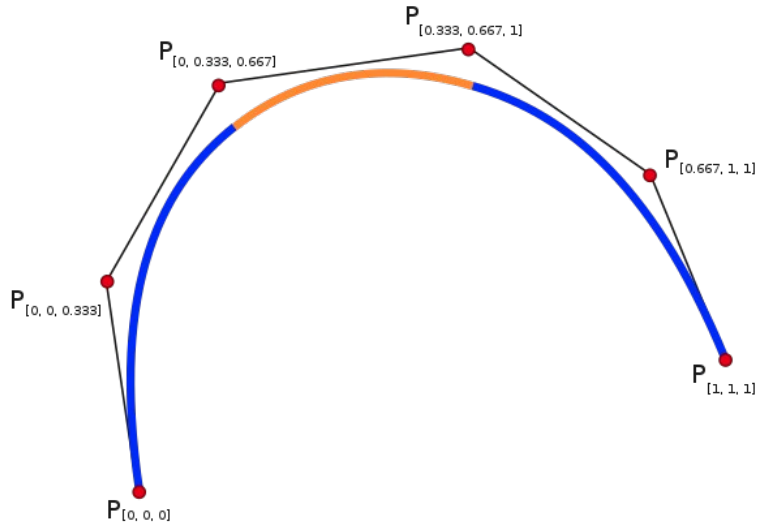
- **Library used - Eigen 3.3, spline & Json framework**

Eigen is a C++ library of template header for linear algebra, matrix and vector operations, geometrical transformation, numerical solver and related algorithms. It is a open source library and latest version 3.3 recieved in 2016.

Spline (mathematics & interpolation) is a function defined piecewise by polynomials. In interpolating problems, spline interpolation is often preferred to polynomial interpolation because it yields similar results, even when using low degree polynomials, while avoiding Runge's phenomenon for higher degrees. In the computer science subfields of computer-aided design and computer graphics, the term spline more frequently refers to a piecewise polynomial parametric curve[*citation needed*]. Splines are popular curves in these subfields because of the simplicity of their construction, their ease and accuracy of evaluation, and their capacity to approximate complex shapes through curve fitting and interactive curve design[*citation needed*].

Reference - https://en.wikipedia.org/wiki/Spline_interpolation for spline overview and library

Image-2 : A spline of three cubic plonomial (example)



Single knots at 1/3 and 2/3 establish a spline of three cubic polynomials meeting with C^2 continuity. Triple knots at both ends of the interval ensure that the curve interpolates the end points

Json framework – used for data exchange (Simulator & project – Standard interface)

In computing, **JavaScript Object Notation** or **JSON** (/ˈdʒeɪsən/ "jay-son", /dʒeɪˈsɒn/)[1] is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). It is a very common data format used for asynchronous browser–server communication, including as a replacement for XML in some AJAX-style systems.[2]

JSON is a language-independent data format. It was derived from JavaScript, but as of 2017 many programming languages include code to generate and parse JSON-format data. The official Internet media type for JSON is `application/json`. JSON filenames use the extension `.json`.

- **Hasdef – constants parameter definitions**

All standard parameters for algorithms, trajectory, waypoints ....etc are defined at one location so that change can be managed easily.

Location: $(Project_Path)/src/hasdef.h

Glimpse of file is given below;

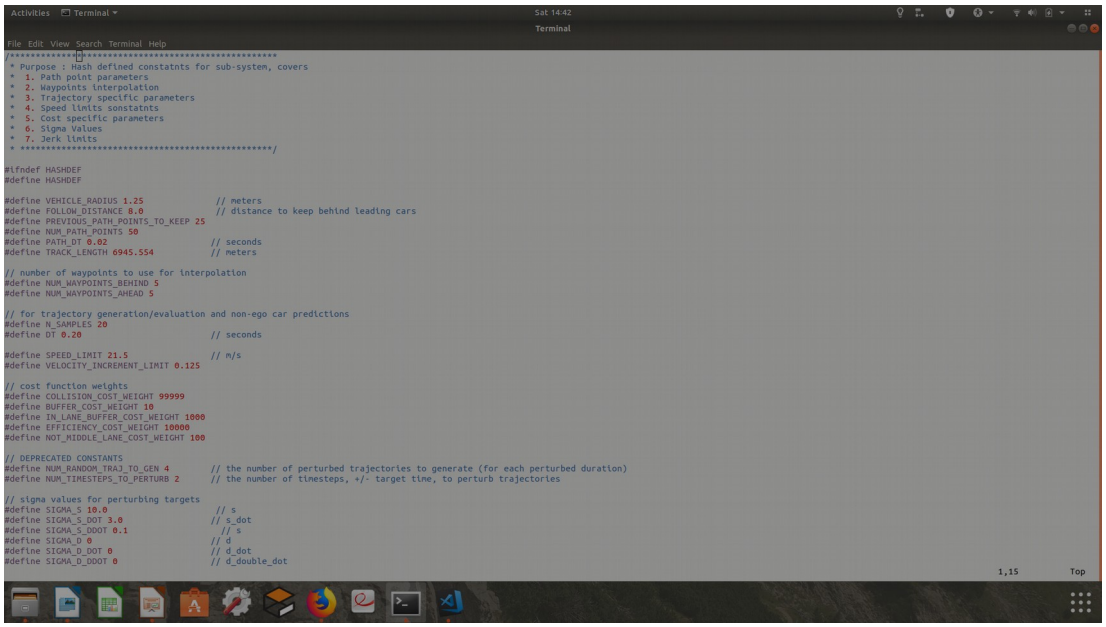


Image-3 : Hashdef file parameters

- **Miscellenous functions** – cost, jerk mesurement technique, vehicle ...etc

Incidents include exceeding acceleration/jerk/speed, collision, and driving outside of the lanes are implemented by leveraging cost, jmt and spline techniques.

Utilities functions implement miscellenous algorithm to evaluate cost in different situations. **Cost functions** leverage sigmoid function to evaluate cost in different situation of motion, trajectory, location, lane and path planning scenario's. A sigmoid function returns a value in between 0 to 1. Overall and average cost is important artifact for deciding motion steps as explained in course lectures.

A glimpse of various ***cost functions*** and definition is as follows :

```
double logistic(double x)     //logistics function
double nearest_approach(vector<double> s_traj, vector<double> d_traj, vector<vector<double>>
prediction)  //nearest vehicles
double nearest_approach_to_any_vehicle(vector<double> s_traj, vector<double> d_traj,
map<int,vector<vector<double>>> predictions)

// Determines the nearest the vehicle comes to any other vehicle throughout a trajectory
double nearest_approach_to_any_vehicle_in_lane(vector<double> s_traj, vector<double> d_traj,
map<int,vector<vector<double>>> predictions)

// Determines the nearest the vehicle comes to any other vehicle throughout a trajectory
vector<double> velocities_for_trajectory(vector<double> traj)
// COST FUNCTIONS

double time_diff_cost(double target_time, double actual_time)
double traj_diff_cost(vector<double> s_traj, vector<double> target_s)
double collision_cost(vector<double> s_traj, vector<double> d_traj,
map<int,vector<vector<double>>> predictions)
double buffer_cost(vector<double> s_traj, vector<double> d_traj,
map<int,vector<vector<double>>> predictions)
double in_lane_buffer_cost(vector<double> s_traj, vector<double> d_traj,
map<int,vector<vector<double>>> predictions)
double exceeds_speed_limit_cost(vector<double> s_traj)
double efficiency_cost(vector<double> s_traj)
double max_accel_cost(vector<double> s_traj)
double avg_accel_cost(vector<double> s_traj)
double max_jerk_cost(vector<double> s_traj)
double avg_jerk_cost(vector<double> s_traj)
double not_middle_lane_cost(vector<double> d_traj)
double calculate_total_cost(vector<double> s_traj, vector<double> d_traj,
map<int,vector<vector<double>>> predictions)
```

Jerk minimization technique primarily usage a vector function defined as,

```
vector<double> get_traj_coeffs(vector<double> start, vector<double> end, double T);
```

JMT actually implements a polinomial;

Calculate the Jerk Minimizing Trajectory that connects the initial state
to the final state in time T.

INPUTS
start - the vehicles start location given as a length three array corresponding to initial values of [s, s_dot, s_double_dot]
end - the desired end state for vehicle. Like "start" this is alength three array.
T - The duration, in seconds, over which this maneuver should occur.

OUTPUT
An array of length 6, each value corresponding to a coefficent in the polynomial
s(t) = a_0 + a_1 * t + a_2 * t**2 + a_3 * t**3 + a_4 * t**4 + a_5 * t**5

Spline Smotther interpolate points using SPLINE library. Function is defined as;

```
vector<double> interpolate_points(vector<double> pts_x, vector<double> pts_y, double interval, int output_size)
```

Input : It uses the spline library to interpolate points connecting a series of x and y values
Output : is output_size number of y values beginning at y[0] with specified fixed interval

Vehicle section defines a class, constructure, destructure and function to read vehicle states in different position of motion trajectories. States machine implementation updates the available "states" based on the current state:

"KL" - Keep Lane
The vehicle will attempt to drive its target speed, unless there is traffic in front of it, in which case it will slow down.

"LCL" or "LCR" - Lane Change Left / Right
The vehicle will change lanes and then follow longitudinal behavior for the "KL" state in the new lane.

➢ Project execution & Video outcome

Purpose of this section is to run project and analyze outcome w.r.t goals set. A video file of simulation and execution recorded (*.mp4) through desktop record utility.

Recorded file name – **path_planning.mp4**

A few observation points moments:

- Vehicle (in left) crossing from right side vehicle

Image-4 : Smooth crossing of left side vehicle

Herein jerk and accelaration parameters remains in permissible limit

- Vehicle (in right) crossing from left side vehicle

  Herein in this scenario vehicle crosses from right while side by lane vehicle is in left side, overtake happens with no jerk (no limit cross) and accelaration parameters remains within permissible limit

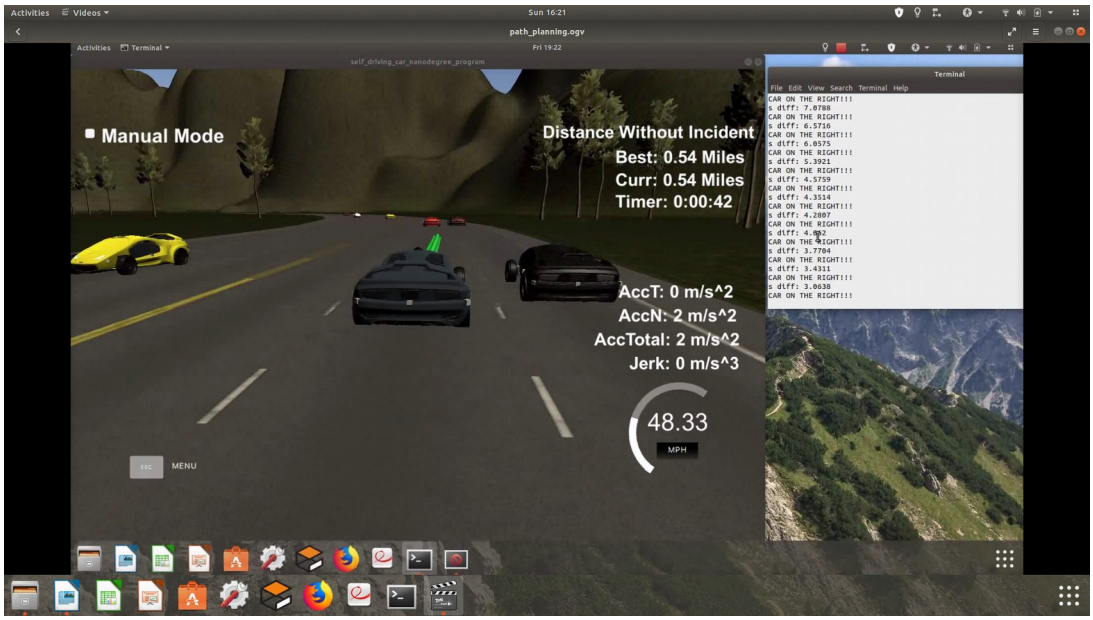


Image-5 : Smooth crossing of right side vehicle

- Lane change – in left

  Herein in this scenario vehicle changes a lane in right side. It happens smoothly in few seconds with no jerk (limit cross) and accelaration parameters remains within permissible limit

Image-6 : Lane change in left

- Multi vehicle scenario – right lane selection and change

Herein in this scenario 2-vehicle are running side by side. Wherein in on vehicle is in same lane and need to avoid collision however other one in side by running in right lane so no scope to change lane in right side. However left side lane is available to make a quick lane change

In this scenario, lane change happens abruptly in few seconds with little jerk voilation and accelaration parameters remains within permissible limit.

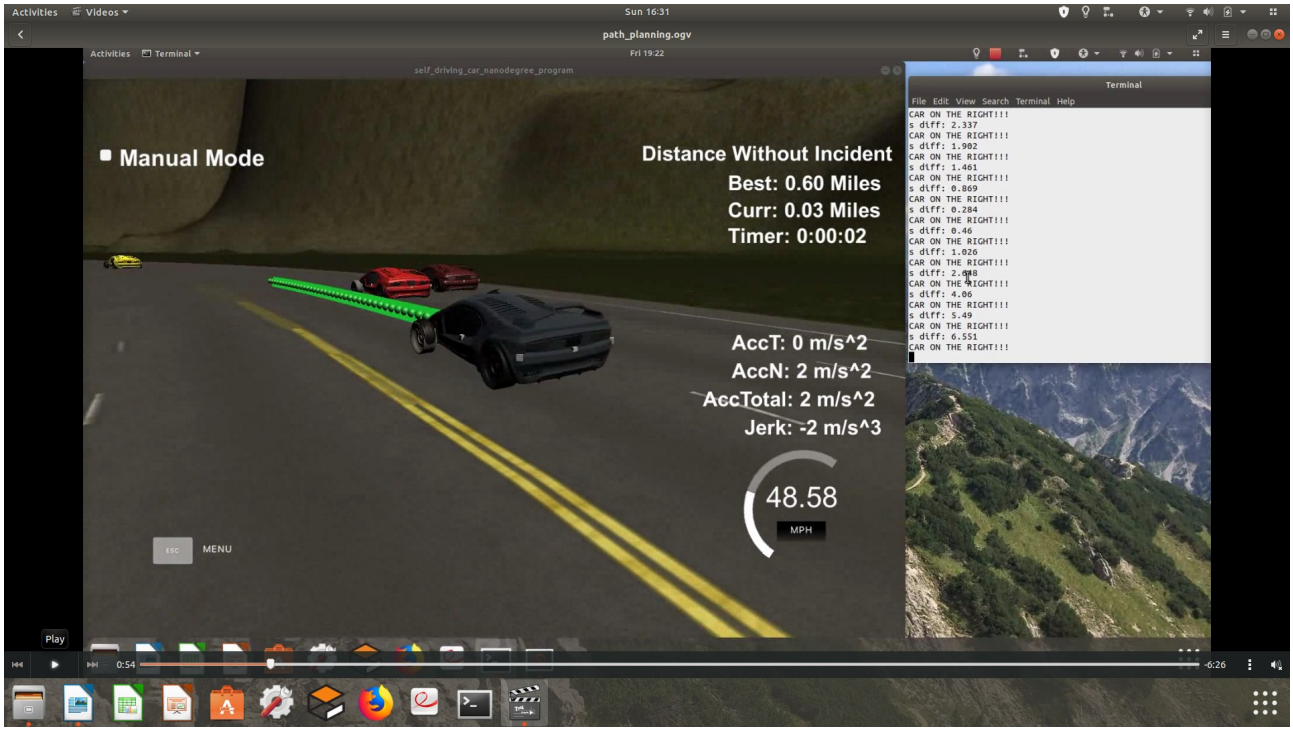


Image-7 : Lane change while 2-lanes occupied by vehicle in parrellel

Post tuning results:

While interpolation waypoint number's increase to 15 for previous and future interpolation than incidences voilation improved drastically.

Earlier while waypoints number was 5 (for previous/future interpolation) – vehicle was sometimes encountering with some incident. Jerk, Accelaration exceed, Speed voilation was happening in specific situations of journey.

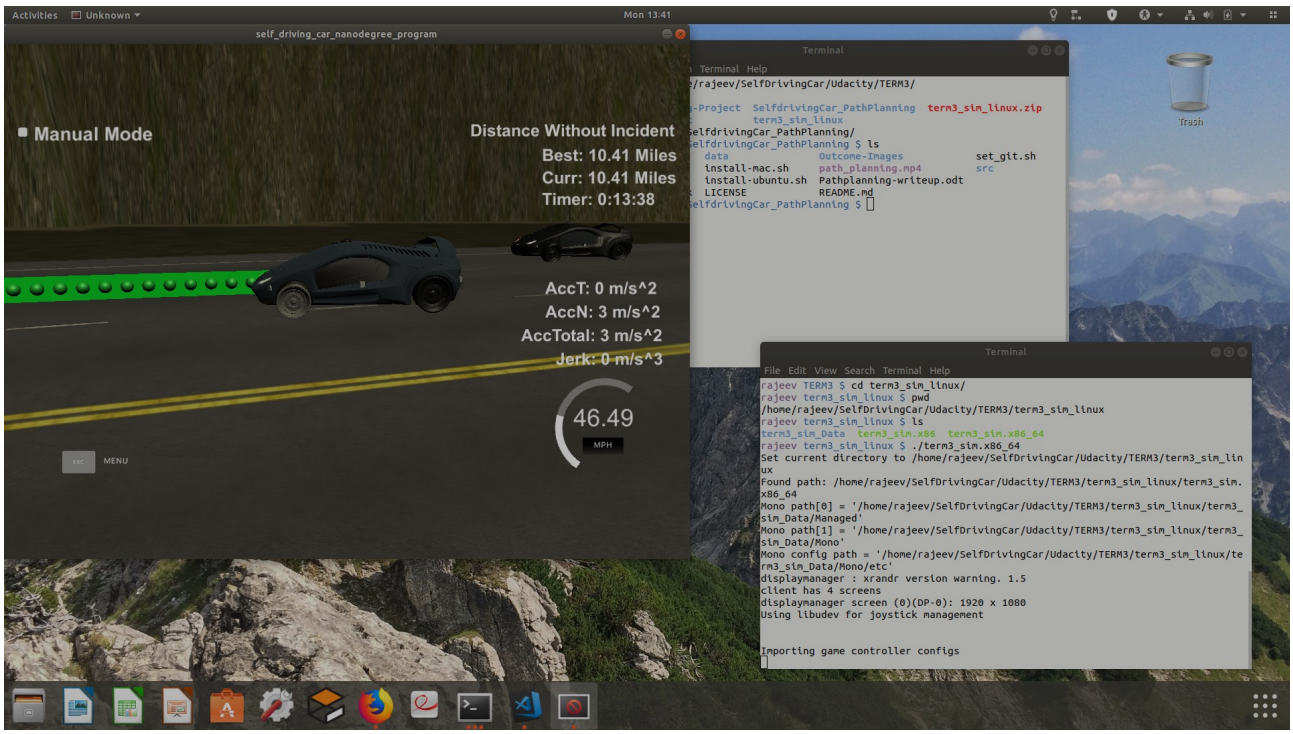


Image-8 : Sucessfull long drive without any incidence (10.41 miles)

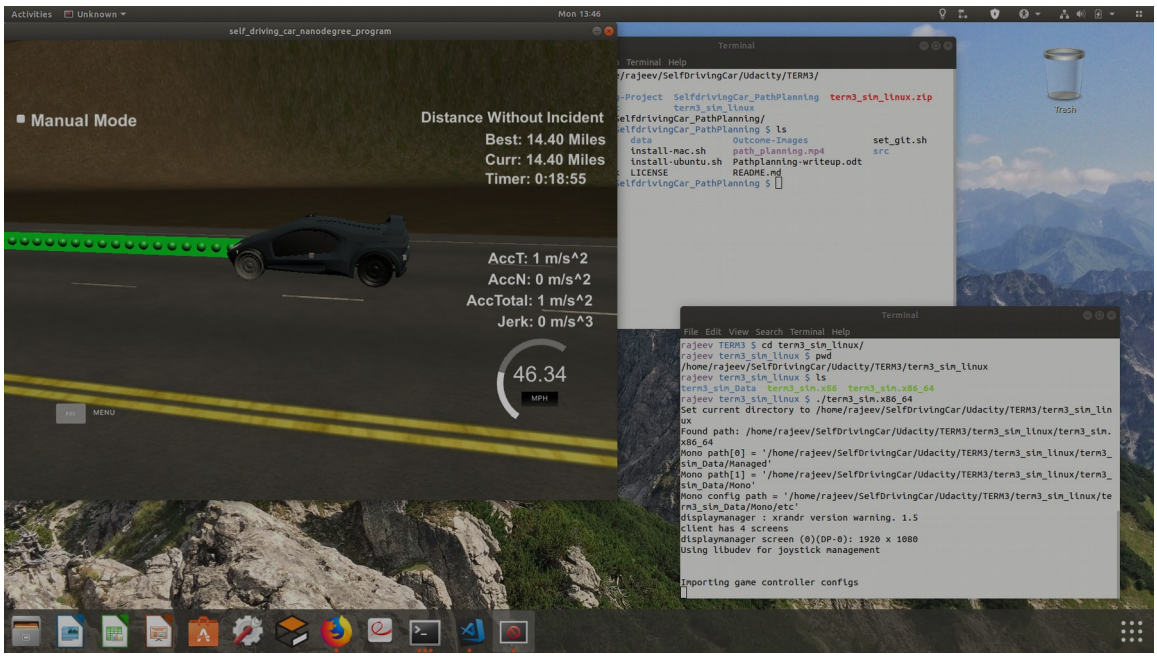


Image-9 : Sucessfull long drive without any incidence (14.40 miles)

Summary:

PathPlanning project repository contains working code, functions definition and outcome visualisation. All sections give desired result. Code has modularity, no hardcoding and readability. Code successfully compiled and build a path planning executable.

Execution of path planning and simulation incidences include exceeding acceleration/jerk/speed, collision, and driving outside of the lanes. Each incident case is also listed below in more detail -

1. *The car rarely drive faster than the speed limit. Also the car isn't driving much slower than speed limit unless obstructed by traffic.*
2. *The car mostly does not exceed a total acceleration of 10 m/s^2 and a jerk of 10 m/s^3*
3. *The car must not come into contact with any of the other cars on the road*

4. *The car doesn't spend more than a 3 second length out side the lane lanes during changing lanes, and every other time the car stays inside one of the 3 lanes on the right hand side of the road.*
5. *The car is able to smoothly change lanes when it makes sense to do so, such as when behind a slower moving car and an adjacent lane is clear of other traffic.*

The video outcome is available with submission for review. All execution done Ubuntu 18.10, GPU (GeForce GTX 1060)/CPU grade machine and compute execution being fairly fast.

While project code outcome recorded in video's at very few occassion (rarely) and momentarily observed – slight jerk and speed violation.