MPC Controller

MPC Controller Project

The goals / steps of this project are the following:

- Drive a self driving car on lake side track (using udacity simulator)
- Implementation of MPC crontroller
- Hyper parameter tuning
- Simulation of car running on lake side track and system behaviour debug

Image References:

Image-1 : A block diagram of MPC contoller in feedback loop (generic)
Image-2: Mathematical equation of MPC contoller in feedback loop (generic)
Image-3: Image- 3: MPC Equation (Udacity vehicle motion)
Image-4 : Simulator (udacity) for autonomous drive on a track
Image-5 : Simulation outcome - N=5 & dt=.05 (a very small value)
Image-6 : Simulation outcome - N=7 & dt=.07 increased
Image-7 : Simulation outcome - N=9 & dt=.09 increased stabilityk
Image-8 : Simulation outcome - N=10 & dt=.11 (Tunned) motion parameters

Table - 1: Latency MPC vs PID

Files Submitted & Code Quality

Submission includes all required files for build and compile:

1. src (folder) : C++ source code for MPC implementation i.e MPC.cpp, MPC.h & main.cpp
2. src/Eigen-3.3 : Eigen utility source, library and header files
3. build (folder) : project compilation script and executable
4. Simulator Interface : JSON.hpp file in src provides a mechanism to exchange messages on a port
5. Scripts :
   - install-ubuntu.sh - to create WebInterface setup
   - CmakeLists.txt – to build project
   - install-ipopt.sh – to install and configure Ipopt setup
6. Output : Video's (*.mp4)
   - Output folder maintains : MPC_ControllerTunned.mp4
   - URL - https://youtu.be/yhyd8npOrow

Submission includes functional code & other files

Sumitted project contain functional code, libraries/utilities, build script, recorded video and outcome images of project execution.

Submission code is usable and readable

YES

(A) MPC Overview

MPC stands for model-predictive-control. The models used in MPC are generally intended to represent the behavior of complex dynamical systems. The additional complexity of the MPC control algorithm is not generally needed to provide adequate control of simple systems, which are often controlled well by generic PID controllers. Common dynamic characteristics that are difficult for PID controllers include large time delays and high-order dynamics[1].

MPC uses the current plant measurements, the current dynamic state of the process, the MPC models, and the process variable targets and limits to calculate future changes in the dependent variables. These changes are calculated to hold the dependent variables close to target while honoring constraints on both independent and dependent variables[1].

i. MPC Applicabilities

While many real processes are not linear, they can often be considered to be approximately linear over a small operating range. Linear MPC approaches are used in the majority of applications with the feedback mechanism of the MPC compensating for prediction errors due to structural mismatch between the model and the process. In model predictive controllers that consist only of linear models, the superposition principle of linear algebra enables the effect of changes in multiple independent variables to be added together to predict the response of the dependent variables. This simplifies the control problem to a series of direct matrix algebra calculations that are fast and robust[1].

When linear models are not sufficiently accurate to represent the real process nonlinearities, several approaches can be used. In some cases, the process variables can be transformed before and/or after the linear MPC model to reduce the nonlinearity. The process can be controlled with nonlinear MPC that uses a nonlinear model directly in the control application. The nonlinear model may be in the form of an empirical data fit (e.g. artificial neural networks) or a high-

fidelity dynamic model based on fundamental mass and energy balances. The nonlinear model may be linearized to derive a Kalman filter or specify a model for linear MPC[1].

ii. MPC Theory & discrete scheme

MPC is based on iterative, finite-horizon optimization of a plant model. At time the current plant state is sampled and a cost minimizing control strategy is computed (via a numerical minimization algorithm) for a relatively short time horizon in the future: . Specifically, an online or on-the-fly calculation is used to explore state trajectories that emanate from the current state and find (via the solution of Euler–Lagrange equations) a cost-minimizing control strategy until time . Only the first step of the control strategy is implemented, then the plant state is sampled again and the calculations are repeated starting from the new current state, yielding a new control and new predicted state path.. [1]

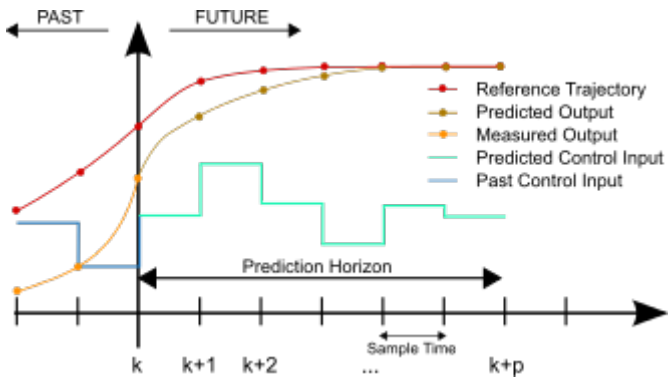


Image-1 : Discrete MPC Scheme (generic)[1]

iii. Principal of MPC

The overall control function can be expressed mathematically as

Model Predictive Control (MPC) is a multivariable control algorithm that uses:

- an internal dynamic model of the process
- a history of past control moves and
- an optimization cost function J over the receding prediction horizon,

to calculate the optimum control moves. An example of a non-linear cost function for optimization is given by [2]:

$$J = \sum_{i=1}^{N} w_{x_i} (r_i - x_i)^2 + \sum_{i=1}^{N} w_{u_i} \Delta u_i{}^2$$

Image- 2: MPC cost function (example)

without violating constraints (low/high limits) with

x_i : i^{th}controlled variable (e.g. measured temperature)
r_i : i^{th} reference variable (e.g. required temperature)
u^i : i^{th} manipulated variable (e.g. control valve)
wx_i : weighting coefficient reflecting the relative importance of i^{th}
wu_i : weighting coefficient penalizing relative big changes in i^{th}

(B) MPC Implementation (Udacity Project)

(1) **MPC Equations (Psuedo Code)**
MPC vehicle model to drive a vehicle estimates six parameters : x-position, y-position, drive angle, velocity and cross talk errors (position and angle). There discretized mathematical equation in terms of time stamp (prev and current) explained in class sessions. Also, summarized in Image-2.

Same set of equation got implemented in C++ by defining class MPC and associated functions. Complex vector computation leverages open source libraries and previously done work by researchers.

$$x_{t+1} = x_t + v_t * cos(\psi_t) * dt$$

$$y_{t+1} = y_t + v_t * sin(\psi_t) * dt$$

$$\psi_{t+1} = \psi_t + \frac{v_t}{L_f} * \delta_t * dt$$

$$v_{t+1} = v_t + a_t * dt$$

$$cte_{t+1} = f(x_t) - y_t + (v_t * sin(e\psi_t) * dt)$$

$$e\psi_{t+1} = \psi_t - \psi des_t + (\frac{v_t}{L_f} * \delta_t * dt)$$

Image- 3: MPC Equation (Udacity vehicle motion)

(2) **MPC application (algorithm) in vehicle motion (Steps)**

We have learned vehicled motion model (MPC) through a quiz assignment to follow the trajectory along a line. Following step taken to implement:

1. Set *N* and *dt*.
2. Fit the polynomial to the waypoints.
3. Calculate initial cross track error and orientation error values.
4. Define the components of the cost function (state, actuators, etc). You may use the methods previously discussed or make up something, up to you!
5. Define the model constraints. These are the state update equations defined in the *Vehicle Models* module.

(3) **Latency (Role in realistic drive)**
In a real car, an actuation command won't execute instantly - there will be a delay as the command propagates through the system. A realistic delay might be in the **order of 100 milliseconds.** This is a problem called **"latency"**, and it's a difficult challenge for some controllers - like a PID controller - to overcome. But a **Model Predictive Controller** can adapt quite well because we can model this latency in the system.

PID Controller	Model Predictive Control
PID controllers will calculate the error with respect to the present state, but the actuation will be performed when the vehicle is in a future (and likely different) state. This can sometimes lead to instability.	A contributing factor to latency is actuator dynamics. For example the time elapsed between when you command a steering angle to when that angle is actually achieved. This could easily be modeled by a simple dynamic system and incorporated into the **vehicle model**. One approach would be running a simulation using the vehicle model starting from the current state for the duration of the latency. The resulting state from the simulation is the new initial state for MPC.
The PID controller could try to compute a control input based on a future error, but without a vehicle model it's unlikely this will be accurate.	Thus, MPC can deal with latency much more effectively, by explicitly taking it into account, than a PID controller.

Table - 1: Latency - MPC vs PID

```
/****************** 100 ms latency introduction in main.cpp from co-ordinates standpoint ************
        msgJson["next_x"] = next_x_vals;
        msgJson["next_y"] = next_y_vals;


        auto msg = "42[\"steer\"," + msgJson.dump() + "]";
        //std::cout << msg << std::endl;

        // Latency
        // The purpose is to mimic real driving conditions where
        // the car does actuate the commands instantly.
        //
        // Feel free to play around with this value but should be to drive
        // around the track with 100ms latency.
        //
        // NOTE: REMEMBER TO SET THIS TO 100 MILLISECONDS BEFORE
        // SUBMITTING.
        this_thread::sleep_for(chrono::milliseconds(100));
        ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
/*************************************************************************************************/
```

iv. MPC C++ Implementation

main.cpp – provides a standard main() function which recieves simulator message on JSON interface and process vehicle motion parameters (position) with MPC controller and managing latency values in drive.
MPC.h – defines class MPC and its destructor/constructors definitions
MPC.cpp – defines MPC class & various functions of MPC class such as;

- Solve() - function to implement Ipopt solver for states i.e interior point optimizer/
- FG_eval – object definition to calculate objective and constraints. Basically to calculate fitted polynomial coefficients for subsequent states.

Libraries :

- Eigen-3.3 : Eigen is a high-level C++ library of template headers for linear algebra, matrix and vector operations, geometrical transformations, numerical solvers and related algorithms[5].
- Ipopt : IPOPT, short for "Interior Point OPTimizer, pronounced I-P-Opt", is a software library for large scale nonlinear optimization of continuous systems. It is written in Fortran and C and is released under the EPL[4]

Polynomial fitting & MPC preprocessing:

Too much increase in

(C) Drive Simulation & MPC interface outcome (Udacity Project)

1) **Simulator, Data exchanged and Visualisation**

- A singular unity unit in the simulator is equivalent to 1 meter.
- For details on the data sent back from the server, read this file.

ψ Updates

In the classroom we've referred to the ψ update equation as:

$$\psi_{t+1} = \psi_t + (v_t / L_f) * \delta_t * dt$$

Note if δ is positive we rotate counter-clockwise, or turn left. In the simulator however, a positive value implies a right turn and a negative value implies a left turn. Two possible ways to get around this are:

1. Change the update equation to $\psi_{t+1} = \psi_t - (v_t / L_f) * \delta_t * dt$
2. Leave the update equation as is and multiply the steering value by -1 before sending it back to the server.

Visualization

When working on the MPC project it helps to visualize both your reference path and the MPC trajectory path.

You can display these connected point paths in the simulator by sending a list of optional x and y values to the mpc_x, mpc_y, next_x, and next_y fields in the C++ main script. If these fields are left untouched then simply no path will be displayed.

The mpc_x and mpc_y variables display a line projection in green. The next_x and next_y variables display a line projection in yellow. You can display these both at the same time, as seen in the image above.

These (x,y) points are displayed in reference to the vehicle's coordinate system. Recall that the x axis always points in the direction of the car's heading and the y axis points to the left of the car. So if you wanted to display a point 10 units directly in front of the car, you could set next_x = {10.0} and next_y = {0.0}.

Remember that the server returns waypoints using the map's coordinate system, which is different than the car's coordinate system. Transforming these waypoints will make it easier to both display them and to calculate the CTE and Epsi values for the model predictive controller [3].

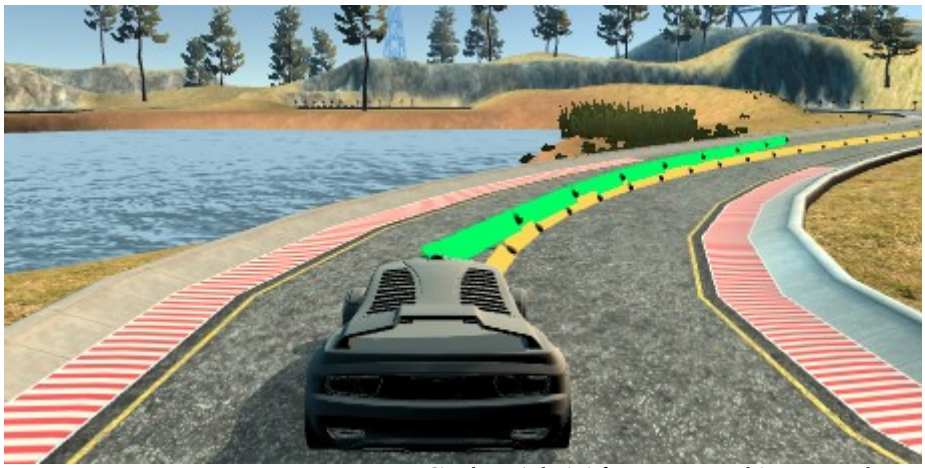


Image-4 : Simulator (udacity) for autonomous drive on a track

2) Drive Simulation & MPC responses

Timestamp Length & Elapshed duration (N & dt):
Initially tried for a very small value of N=5 and dt=.05 time samples, in result very soon leave the track and shift in one direction only.

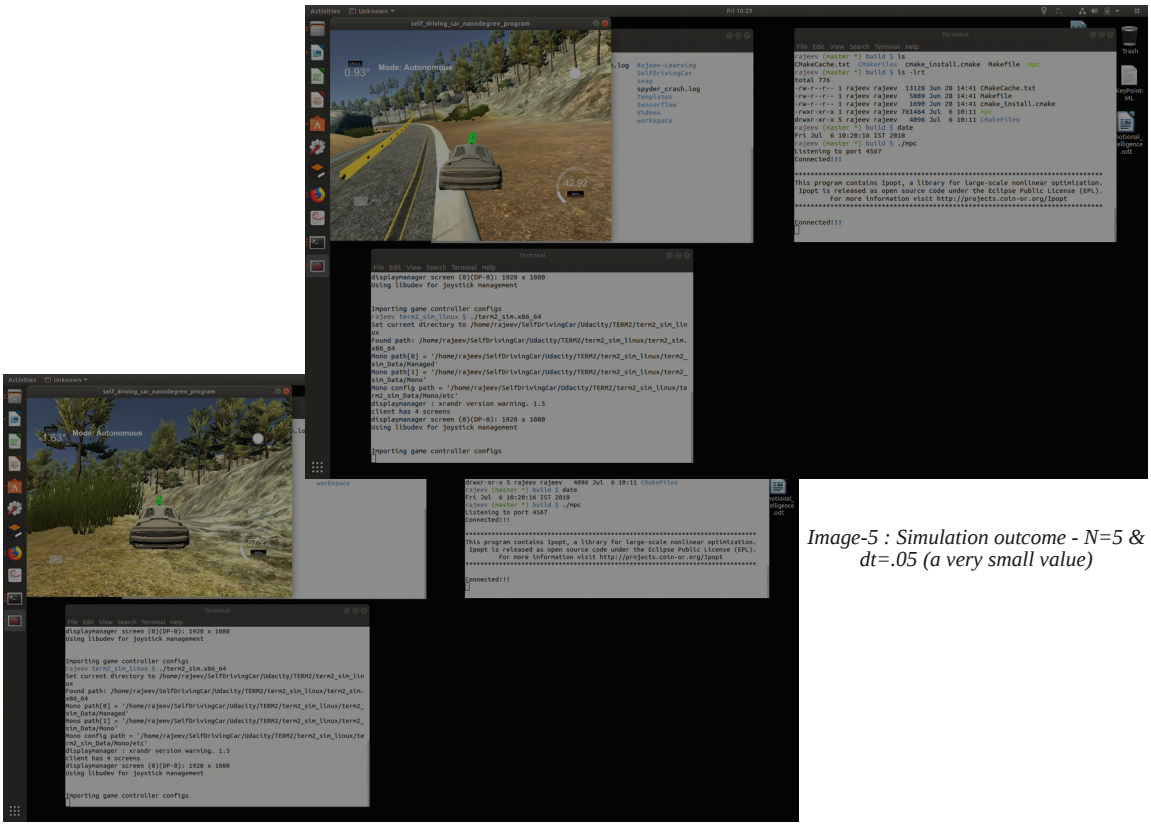


Image-5 : Simulation outcome - N=5 & dt=.05 (a very small value)

When we tunned a value of N=7 and dt=.07, vehicle remain on track mostly however at sharp turns prediction trajectory take vehicle out of track.

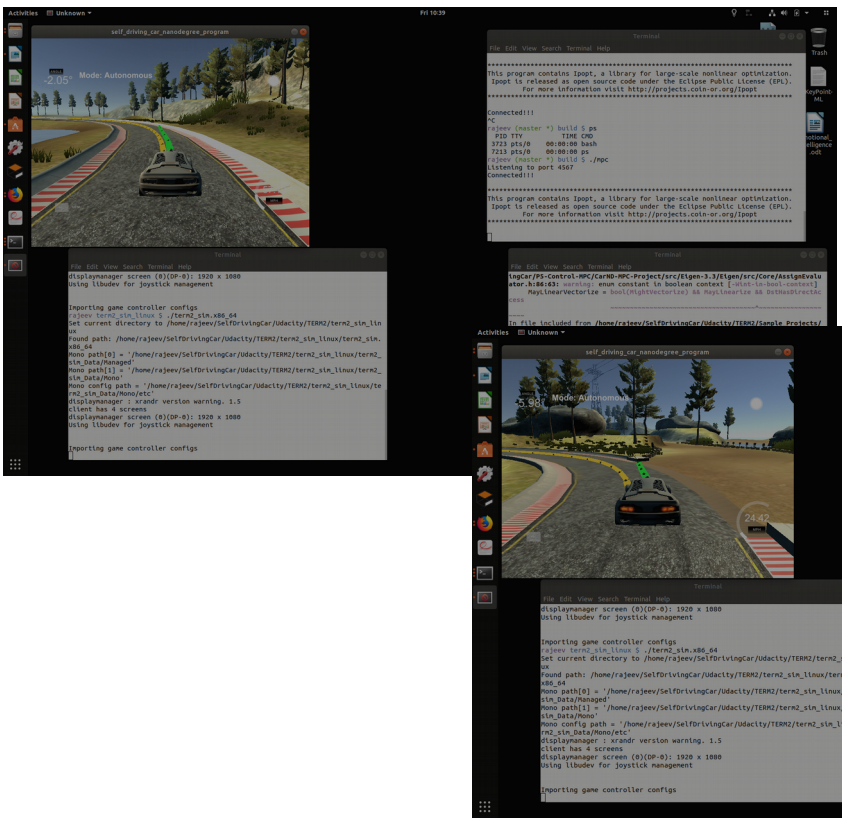
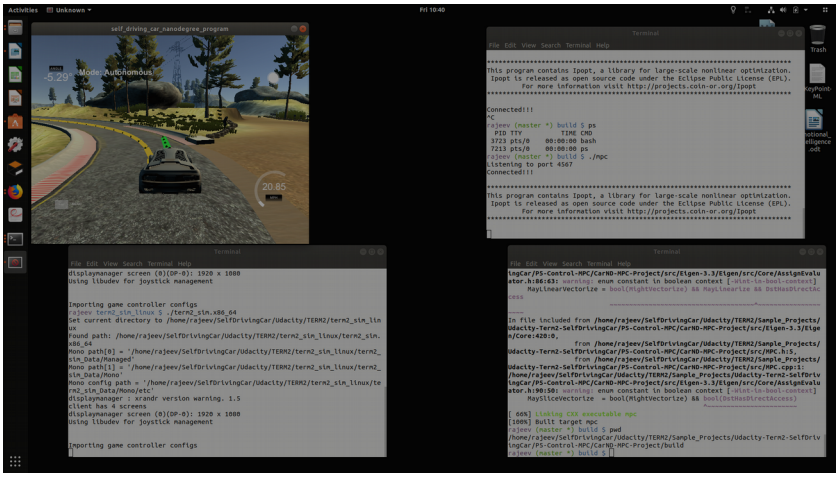


Image-6 : Simulation outcome - N=7 & dt=.07 increased



Further increase up till N=9 and dt=.09, vehicle remain on track mostly and bring stability in motion.

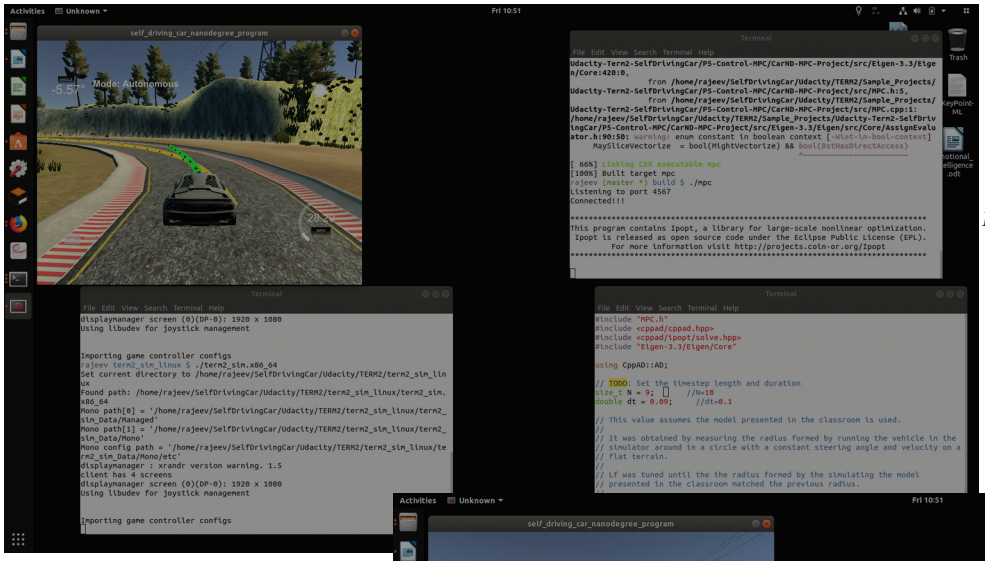


Image-7 : Simulation outcome - N=9 & dt=.09 increased stability

At a tunned value N=10 and dt=.11, vehicle remain on track mostly and move great stability. This seems to be a very stable motion behaviour.

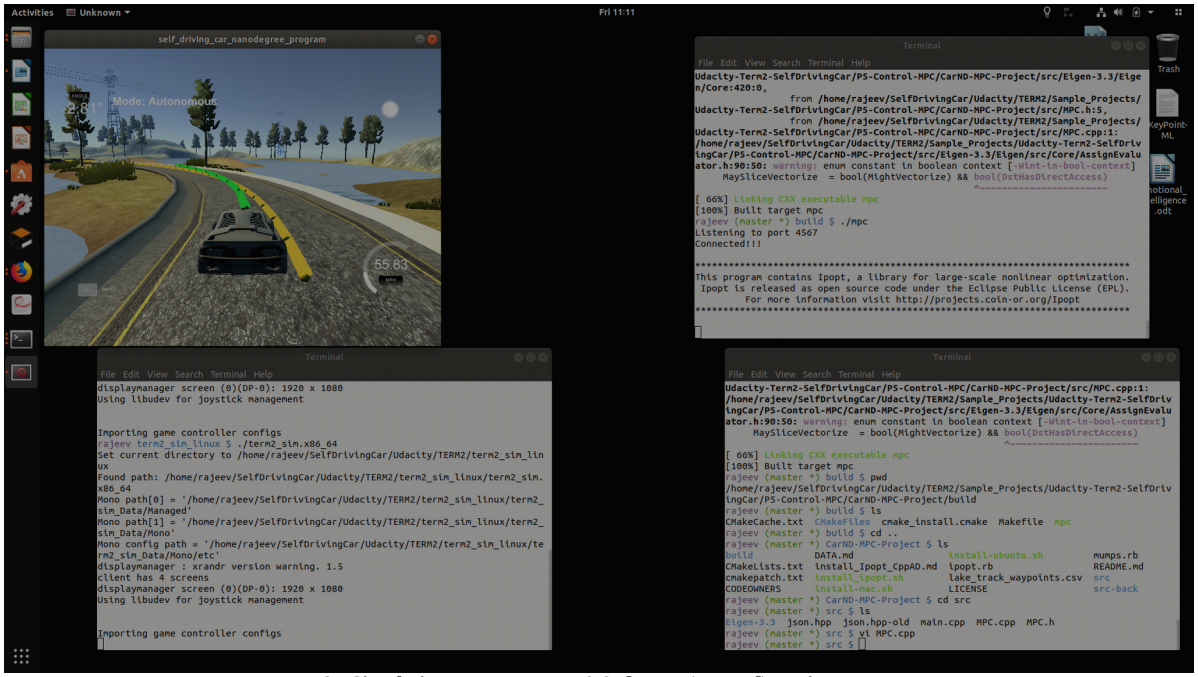


Image-8 : Simulation outcome - N=10 & dt=.11 (Tunned) motion parameters

Interface between a simulator (drive) and MPC controller:
Type – JSON
Port – 4567
Message – Standard fields for kinematics i.e. define in simulator and decoded in MPC controller (position, velocity, angle ...etc)

v. Video Outcomes

Simulation execution get recorded in mp4 format, here is recorded outcomes;

- *MPC_ControllerTunned.mp4 (in Output folder)*
- *or https://youtu.be/yhyd8npOrow*

Summary:

MPC repository src contains functional code & build had executables. All sections give desired result. Code has modularity, no hardcoding and readability. Outcome get recorded in a mp4 file (available in Output folder). Snaps of vehicle motion with various set of parameters (tunning) depicts outcome and stability behaviour. Simulation video available at https://youtu.be/yhyd8npOrow

All execution done Ubuntu 17.10, GPU (GeForce GTX 1060)/CPU grade machine and compute execution being fairly fast.

Reference:

[1][2] – wikipedia MPC
[3] – Udacity class material
[4] - wikipedia Ipopt
[5] - wikipedia Eigen