Model Predictive Controller Solution for Udacity Self-driving-car Nanodegree Project

Saurabh Ravindra Badenkal
Computer Engineering Department, College of Engineering
San Jose State University, San Jose, CA 94303
E-mail: saurabhravindra.badenkal@sjsu.edu

Abstract

The goal of this paper is to design a Model Predictive Controller for Udacity Self-driving-car Nanodegree program as solution to Project number 10. The Solution takes inputs from the previous projects solution (given directly with the problem statement) in the form of road center coordinates, using which this model was created to control autonomous driving of a car in Udacity's Unity3D car simulator with reasonable speeds.

Keywords- Self-driving-car, Udacity Nanodegree Program, Model Predictive Controller, Eigen solver

1. Introduction

The core functionality typically provided by a model predictive controller includes solving of given equations made based on the car's physical specifications (such as maximum speed, maximum turning angle, turn radius etc.) in order to predict the next set of (x,y) coordinates of the car for current speed and orientation angle. At the same time using this predictions and by knowing the car's constraints, the car's throttle and turning angle is also predicted in order to follow the predicted (x,y) coordinates. This was achieved by getting the road coordinates from the Unity3D simulator over a localhost web server via get/post of JSON data. This data is then read and used to predict future values, and then sent back to Simulator to drive the car, with new throttle and turn angle values.

2. Design Methodology

In this section, the design methodology is explained which describes the overall model layout as well as the design methodology.

The model used is a kinematic model for the vehicle in this project. The states for the model are: [x,y,psi,v] which stand for the position in x and y, the yaw angle, and the speed. The actuators are [delta,a] which stand for the steering angle and the acceleration. The state update equations are:

$$\boxed{\begin{aligned} x_{t+1} &= x_t + v_t * cos(\psi_t) * dt \\ y_{t+1} &= y_t + v_t * sin(\psi_t) * dt \\ \psi_{t+1} &= \psi_t + \frac{v_t}{L_f} * \delta * dt \\ v_{t+1} &= v_t + a_t * dt \end{aligned}}$$

Figure 1. State Update Equations

and the error equations are:

$$\boxed{\begin{gathered} cte_{t+1} = f(x_t) - y_t + (v_t * sin(e\psi_t) * dt) \\ e\psi_{t+1} = \psi_t - \psi des_t + (\frac{v_t}{L_f} * \delta_t * dt) \end{gathered}}$$

Figure 2. System Error Equations

2.1. Objectives and Technical Challenges

Following Objectives were covered during the Implementation:

1. Creating a model predictive controller which takes x,y, speed, orientation as input and predicts xnext, ynext, throttle and steering angle values.
2. Optimizing the model by modifying equations using different set of contraints values, to achieve higher speeds.
3. To create constraints based on correlation of turning angle and throttle for efficient turning at high speeds.

Technical Challenges faced during Implementation:

1. Compiling and building Ipopt library which was fixed by using new MUMPs wget url. ***Otherwise, the solver will give unknown solutions.***
2. Installing and running Unity3D with Monopole on Ubuntu 16.04.
3. Getting the correct Udacity Self-driving-car Simulator for the given git repo source.
4. Understanding the Simulator coordinates and orientation values which are not real-world values.

3. Implementation

This section describes the design and implementation of the model. This project implements a Model Predictive Control (MPC) algorithm in C++ using Ipopt optimizer and CppAD automatic differentiation libraries to control steering and throttle while driving a simulated vehicle around a track following reference waypoints.

At each timestep, the vehicle's state (global position, heading, speed, steering, and throttle) and the closest six waypoints are received by the controller.

Before applying the MPC control, the vehicle's state is estimated after some expected latency (processing time and ~100ms actuator delay) using the modeled motion equations (bicycle model).

This estimated state position is then used to convert the waypoints from global to vehicle coordinate systems. The waypoints are then preprocessed with waypoint interpolation and weighting before applying a 3rd order polyfit to use as the reference driving path.

The polyfit coefficients and estimated vehicle state after latency are then used by the MPC controller to optimize a planned driving path over ~1 sec horizon. The MPC minimizes a cost function based on a target reference speed, the cross-track error (CTE), the heading error (EPSI), minimal use of actuators and smoothness between actuation steps. Constraints are applied to the model for it to satisfy the motion equations.

The steering and throttle actuation from the MPC's first step of the planned driving path are used to drive the actual car at each timestep including the expected latency delay. The MPC's planned path coordinates are visualized as a green line with the waypoints shown as a yellow line.

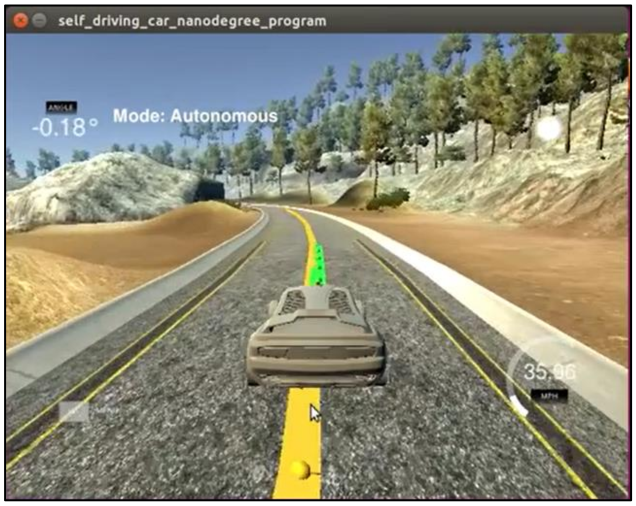


Figure 3. Road Coordinates(yellow), predicted coordinates(green)

3. Setting up Development Environment

The project was developed in Ubuntu 16.04 LTS 64-bit as few dependencies of the project are readily available in Ubuntu.

4.1 Dependencies

The dependencies are described for Ubuntu, for other OS please check the source git repo @ https://github.com/ndrplz/self-driving-car/tree/master/project_10_MPC_control. First, clone this repo in folder and then install the dependencies.

The project has following library dependencies:

- cmake >= 3.5, already present in Ubuntu
- make >= 4.1
  - Linux: make is installed by default on most Linux distros
- gcc/g++ >= 5.4
  - Linux: gcc / g++ is installed by default on most Linux distros
- uWebSockets
  - Run either install-ubuntu.sh (available in the repo).
  - If you install from source, checkout to commit e94b6e1, i.e.

```
git clone
https://github.com/uWebSockets/uWebSockets
cd uWebSockets
git checkout e94b6e1
```

- Fortran Compiler
  - Linux: sudo apt-get install gfortran
- Ipopt [https://projects.coin-or.org/Ipopt]
  - You will need a version of Ipopt 3.12.1 or higher. The version available through apt-get is 3.11.x. If you can get that version to work great but if not there's a script install_ipopt.sh that will install Ipopt. You just need to download the source from the Ipopt releases page or the Github releases page.
  - Then call install_ipopt.sh with the source directory as the first argument, ex: bash install_ipopt.sh Ipopt-3.12.1

IMPORTANT NOTE FOR IPOPT:

- MUMPS Wget url is broken, follow these instructions to fix it
- Download the Ipopt source into a separate folder
- Do the following changes
- filename: /Ipopt-3.12.8/ThirdParty/Mumps/get.mumps
- changes:
- echo "Downloading the source code from ..."
- #REMOVE OLD WGET URL as follows
- #$wgetcmd http://mumps.enseeiht.fr/MUMPS_${mumps_ver}.tar.gz
- #ADD THIS BELOW WGET URL as follows
- $wgetcmd http://ftp.mcs.anl.gov/pub/petsc/externalpackages/MUMPS_${mumps_ver}.tar.gz
- then run the install_ipopt.sh provided in the source repo

- CppAD
  - Linux sudo apt-get install cppad
- Eigen.
  - This is already part of the repo so you shouldn't have to worry about it.
- Unity3D for ubuntu (to run the udacity simulator)
  - Download the Deb file from http://download.unity3d.com/download_unity/linux/unity-editor-5.3.6f1+20160720_amd64.deb
  - install using Ubuntu software installer
  - Install monopole using following instructions:

- ➢ Go to this link: http://www.monodevelop.com/download/linux/
- ➢ Now, click on: Debian, Ubuntu, and derivatives
- ➢ Once in the: Install Mono on Linux page, click in: download page

- ➢ Copy these codes in the Terminal (The whole thing in this order):
- ➢ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF

- ➢ Then this other code (the whole thing):
- ➢ echo "deb http://download.mono-project.com/repo/ubuntu xenial main" | sudo tee /etc/apt/sources.list.d/mono-official.list

- ➢ Then:
- ➢ sudo apt-get update

- ➢ Then:
- ➢ sudo apt-get install mono-devel

  - o Signup in unity, login in unity and create a new project
  - o On the top menu bar goto edit, Preferences, External tools & External script editor. Click browse then go to this path [/opt/Unity/MonoDevelop/bin & choose MonoDevelop.exe.
- Udacity Self-driving-car simulator
  - o Download the executable from https://github.com/udacity/self-driving-car-sim/releases
  - o Download the version Term 2 Simulator v1.45 (the output format of newer ones are changed)

Now this repo can be built. Follow below instructions to do that:

- Make a build directory in the root of the repo: mkdir build && cd build
- Compile: cmake .. && make
- Run it: ./mpc

For easy access you can create s shell script executable to re-build and execute as follows and create a desktop shortcut:

```
#!/bin/bash
cd "/<your project folder>/project_10_MPC_control/src/"
cmake ..
echo ""
make
echo""
echo "REBUILT ALL now executing..."
echo ""
./mpc
$SHELL
```

Now run the simulator and goto Project 5 and click select to start the simulator:

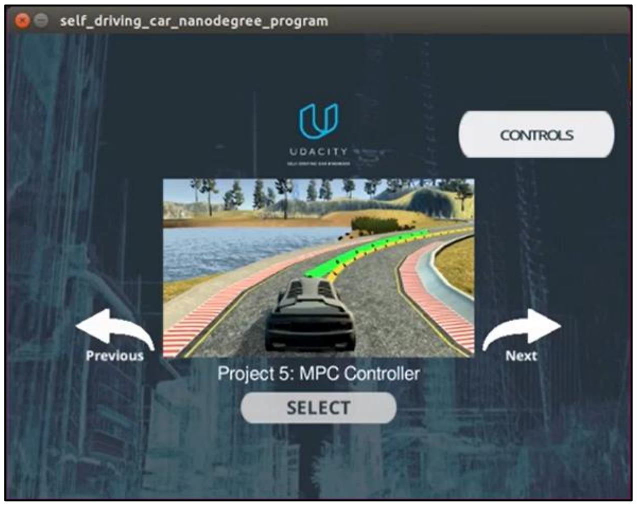

self_driving_car_nanodegree_program
UDACITY
CONTROLS
Previous
Next
Project 5: MPC Controller
SELECT

Figure 4. Simulator Project screen

3.2. Software Design

The source code for the Model is present in MPC.cpp which is used in main.cpp inside the repo's root folder's src folder. The MPC.cpp is where the model should go and in main for debugging messages can be printed.

3.2.1. Communication protocol between uWebSocketIO and Simulator

INPUT to main.cpp: values provided by the simulator to the C++ program:

- ["ptsx"] => Closest waypoints global x position (m)
- ["ptsy"] => Closest waypoints global y position (m)
- ["psi"] => Vehicle's global heading (rad)
- ["psi_unity"] => Vehicle's global heading in Unity coordinates (rad), not used
- ["x"] => Vehicle's global x position (m)
- ["y"] => Vehicle's global y position (m)
- ["steering_angle"] => Vehicle's last steering angle (rad)
- ["throttle"] => Vehicle's last throttle (-1 to +1)
- ["speed"] => Vehicle's last speed (mph)

OUTPUT from main.cpp: values provided by the C++ program to the simulator:

- ["steering_angle"] <= Control value for steering angle (-1 to +1)
- ["throttle"] <= Control value for throttle (-1 to +1)

- ["mpc_x"] <= MPC's planned path x coordinates for visualization (green line)
- ["mpc_y"] <= MPC's planned path y coordinates for visualization (green line)
- ["next_x"] <= Waypoint x coordinates for visualization (yellow line)
- ["next_y"] <= Waypoint y coordinates for visualization (yellow line)

3.2.2. MPC source code

This file describes the MPC model which will be used to do computation to find predicted x,y and new throttle and steering angle values.

Global contraints variables used:

```
// Param length from front wheels to CG to approximate vehicle's turning radius.
constexpr double kLf = 2.67; // m

// MPC solver number of timesteps and duration of each step
constexpr size_t kNSteps = 8; // # of state timesteps
constexpr double kDeltaT = 0.15; // time per step (sec)

// Target velocity for cost function (boosted to 115mph to have some margin
// for cost function to achieve actual 110mph peak speed on lake track)
constexpr double kRefVMax = 115; // mph
constexpr double kRefVTurnFactor = 0.2; // tuned factor to reduce speed in turns

// Set index points for variable starting locations in the combined vector
const size_t x_start = 0;
const size_t y_start = x_start + kNSteps;
const size_t psi_start = y_start + kNSteps;
const size_t v_start = psi_start + kNSteps;
const size_t cte_start = v_start + kNSteps;
const size_t epsi_start = cte_start + kNSteps;
const size_t delta_start = epsi_start + kNSteps;
const size_t a_start = delta_start + (kNSteps - 1); // N-1 delta actuator values
```

The code is self-explanatory with proper comments, which can be referred from the appendix.

3.2.3. Main Source Code

The main.cpp is where the websocket object reads/sends JSON data to & from the simulator. This the main file which gets executed while executing ./mpir.

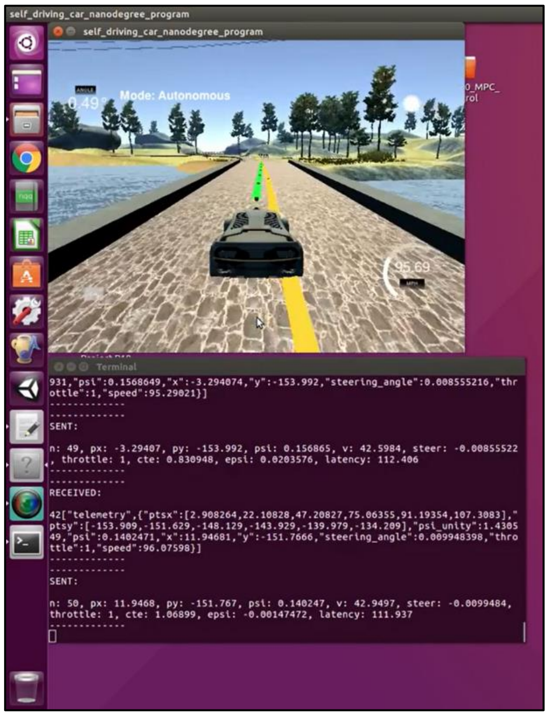
4. Results

The model was simulated with maximum speed as 60mph, 80mph and 100mph with appropriate turning constraints.

It was observed that at high speeds to achieve the required turn the car would drift if the given turning factor is not set according to the top speed.

The car's driving performance deteriorated as the maximum speed was set high, but performed efficiently up to 80mph, but the car always drove on the road even at high speeds.

The demonstration video is available at https://www.youtube.com/watch?v=21EtxGe_lTU&t=2s



```
931,"psi":0.1568649,"x":-3.294074,"y":-153.992,"steering_angle":0.008555216,"thr
ottle":1,"speed":95.29021}]
-------------
-------------
SENT:

n: 49, px: -3.29407, py: -153.992, psi: 0.156865, v: 42.5984, steer: -0.00855522
, throttle: 1, cte: 0.830948, epsi: 0.0203576, latency: 112.406
-------------
-------------
RECEIVED:

42["telemetry",{"ptsx":[2.908264,22.10828,47.20827,75.06355,91.19354,107.3083],"
ptsy":[-153.909,-151.629,-148.129,-143.929,-139.979,-134.209],"psi_unity":1.4305
49,"psi":0.1402471,"x":11.94681,"y":-151.7666,"steering_angle":0.009948398,"thro
ttle":1,"speed":96.07598}]
-------------
-------------
SENT:

n: 50, px: 11.9468, py: -151.767, psi: 0.140247, v: 42.9497, steer: -0.0099484,
throttle: 1, cte: 1.06899, epsi: -0.00147472, latency: 111.937
-------------
```

Figure 5. Simulation in action

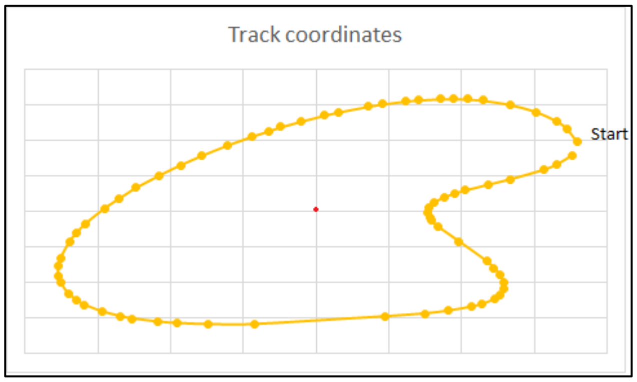


Figure 6. Real Track coordinates (the yellow line in the simulator)

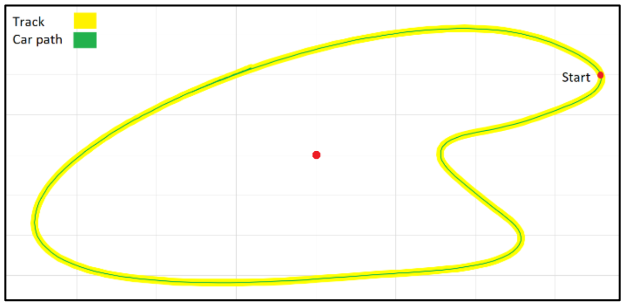


Figure 7. Real Track vs Car predicted track coordinates (the yellow line is road and green line is predicted car coordinates which is always inside the path)

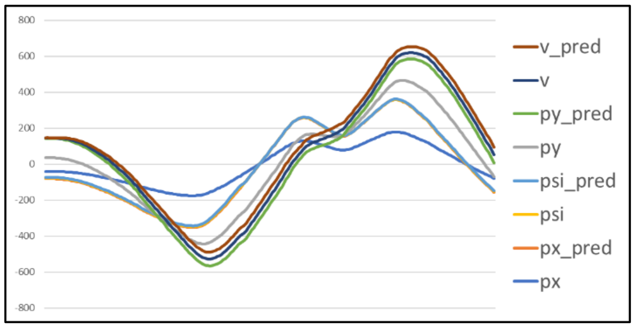


Figure 8. Actual vs Predicted values of x,y, orientation(psi), velocity(v)

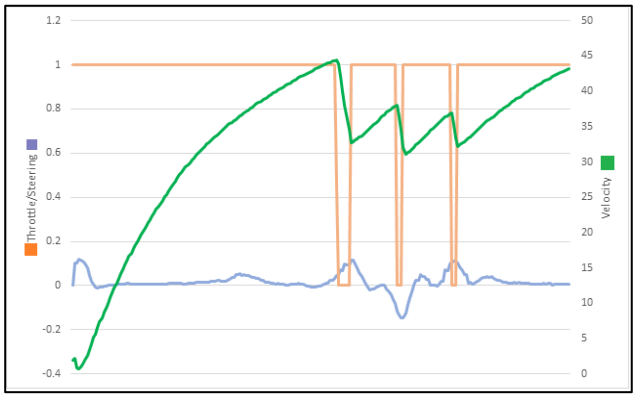


Figure 9. Velocity vs Throttle and Steering values

Speed	RMSE_Velocity	RMSE_Psi
100	0.512064	0.407195
80	0.527706	0.383019
60	0.609655	0.340279

Figure 10. Root Mean Squared error for predicted vs actual velocity and orientation values

5. Conclusion

For simple model predictive controller, it was successfully implemented, and the car could travel autonomously with fair accuracy on the road along the whole track. The RMSE was found to be

6. Acknowledgement

I would like to thank Udacity for providing all the sources to work on the project and for the support provided by them.

7. References

[1] Udacity Self-Driving-Car Nanodegree,
[2] Udacity Self-driving-car resources [https://github.com/ndrplz/self-driving-car/tree/master/project_10_MPC_control]

8. Appendix

8.1. Source Code

https://github.com/saurabhrb/MPC_Project10