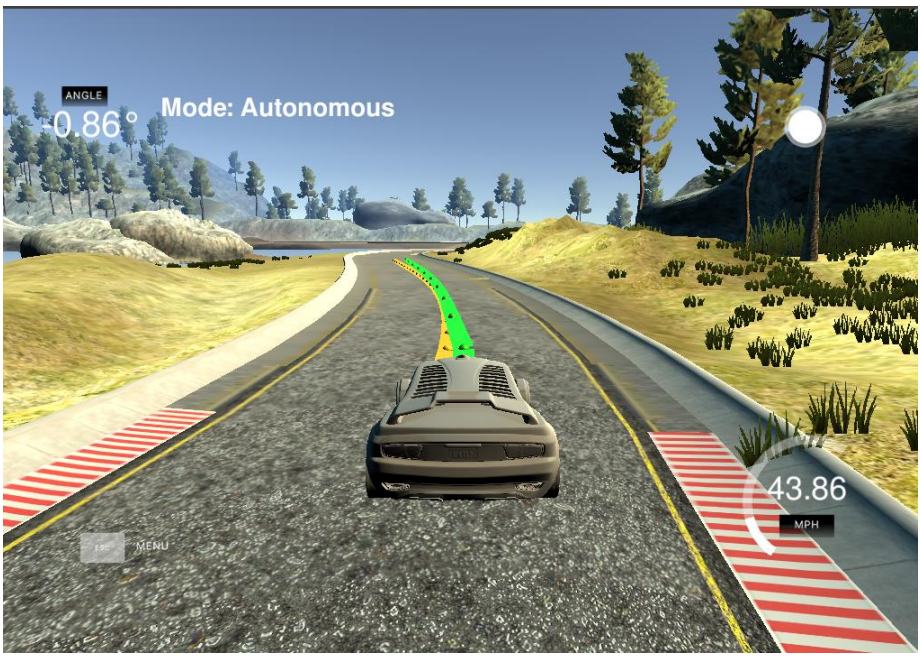
Model Predictive Control

Overview

The purpose of this project is to implement the model predictive controller (MPC) to drive along a path in the Udacity simulator. The simulator offers the x, y positions, speed and orientation of the vehicle along the trajectory.

In the simulator the reference trajectory will be depicted with yellow while the predicted path is illustrated with green, as exemplified in the image below.

This project is part of the series of projects exploring different methods to drive a car autonomously around a given path. Similar implemented projects are: PID Control and Deep learning



Dependencies

- cmake >= 3.5
- 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 `install-ubuntu.sh`.

- Fortran Compiler
    - Linux: `sudo apt-get install gfortran`. Additionall you have also have to install gcc and g++, `sudo apt-get install gcc g++`.
- Ipopt
    - I installed it by running the script using sudo ./install_ipopt.sh
- CppAD
    - I managed to install this dependency only by downloading and building the sources by hand (with cmake .. and make install).
- Eigen. – is available in the repo
- Simulator

To run the project select the corresponding MPC project from the simulator. Build the MPC project using cmake (cmake .. and make) and run it ./mpc.

Model

The model state includes:

- px: X position of the vehicle in the forward direction
- py :Y position of the vehicle in the lateral direction
- psi: orientation of the vehicle
- v: velocity of the vehicle
- cte: cross track error
- epsi : orientation error

The actuators of the vehicle are:

- delta: the steering angle
- a: the acceleration

The future states for our model are predicted using the following equations:

- px(t+1) = p(x) + v(t) * cos(psi(t)) * dt
- py(t+1) = py(t) + v(t) * sin(psi(t)) * dt
- psi(t+1) = psi(t) + v(t) / Lf * deltaPsi * dt
- v(t+1) = v(t) + a * dt
- cte(t+1) = f(x(t)) - y(t) + v(t) * sin(epsi(t)) * dt
- epsi(t+1) = psi(t) – psi_des + v(t) / LF * delta * dt

The timestamp between predictions is denoted by dt and Lf is the distance between the front and the center of gravity of the vehicle, which determines its turning radius.

Time steps and frequency

The value for the time step for the controller is 11 with a frequency of 0.12. Watching the video lectures there was a time step of 9 presented in lecture 6 of the MPC. I started from that value with dt = 0.1, increasing the time step gradually until I saw the best results. If the parameters were set too high the vehicle would oscillate wildly and drive of the track. If the value for N was lower, the vehicle could drive straight of the track.

Cost Function

To maintain a path with the lowest cost at a fast speed several parameters were penalized with different factors. Bellow I list the main parameters and their corresponding weights:

- CrossTrackError^2 * 7000
- OrientationError^2 *7000
- Velocity^2 * 1
- Use of steering actuator^2 * 21000
- Use of acceleration actuator^2 * 1
- Difference of sequential actuations for steering^2 * 2
- Difference of sequential actuations for acceleration^2 * 1

More weight was given to the cross track and orientation errors to keep the vehicle on the road at low speeds. When the speed was increased the level of oscillation would increase eventually kicking the vehicle of the road. If we penalize the use of the steering actuator more, we will get a smoother steering input and a more stable drive.

Polynomial Fitting and MPC Processing

The provided waypoints are transformed into vehicle space and then fitted to a polynomial. After the vector of points is converted to an Eigen vector it can be used as an argument in the polyfit function. I have tried using a 3rd and also a second order polynomial to see which one offers the most robust result. In the end I realized that each solution (using 2nd and 3rd) degree polynomials can have good results. The code section allows me to switch between second and 3rd degree polynomials. The weights in the cost function are adapted for a third degree polynomial. That polynomial is then evaluated using the polyeval function to calculate the cross track error.

Dealing with Latency

In order to simulate a system that is closer to real life, a latency of 0.1 seconds was introduced between a cycle of the MPC and the actual actuation.

This had the following effects:

- It affected the driving speeds towards tight turns – the vehicle would react too late to recognizing a sharp turn and run off-road
- Any existing oscilation was amplified.

Too account for this, I set the initial states to be the states after 100 ms. This allows the vehicle to “look forward” and correct for where it will be in the future, instead where it is currently positioned. This small update in state calculation solved the problem of latency and the car was back on track w.r.t. its desired behavior.