cvilas report completed e630395 just now

1 contributor

58 lines (38 sloc) 8.81 KB

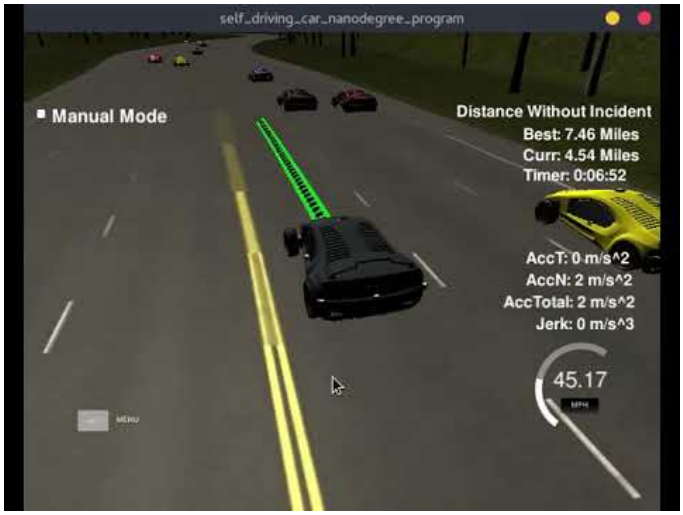
Project 11: Path Planning

Self-Driving Car Engineer Nanodegree Program (Term 3)

Executive Summary

The submitted source code implements an autonomous path planner for a simulated car on a 3-lane highway. As shown in the image below, the car is able to continuously drive without incident multiple loops around the track. Incidents to consider included collisions, departing lanes inadvertently, exceeding acceleration and jerk limits (10m/s^2 and 10 m/s^3, respectively), and exceeding speed limit of 50 miles per hour.

The following video (link: https://youtu.be/-0TvdNeugxE) shows a short segment of a run where the car demonstrates autonomous lane changes upon encountering traffic on its current lane



The rest of this report describes my approach to implementing the path planning solution.

Components and Approach

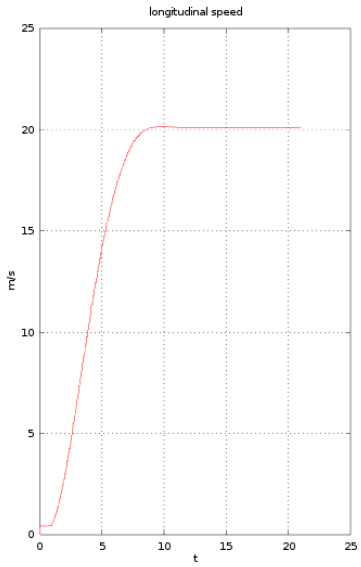
Udacity provided some starter code consisting of helper functions and code to interface with the simulator. Also provided was data file consisting of waypoints spaced roughly 30 meters apart around a simulated track about 6945 meters long. The assigned speed limit was 50 miles/hour. These system wide constants are encoded in `constants.h`.

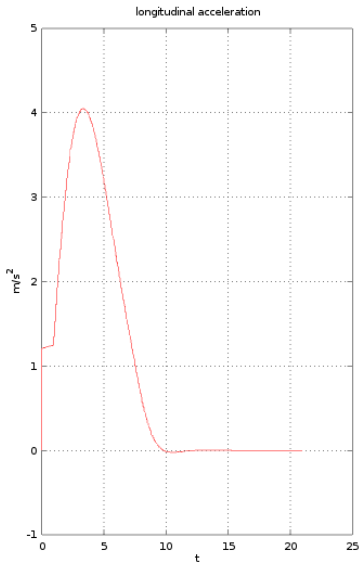
The key component which took most time to implement was the trajectory generator, which I describe first. A bicycle model provides car kinematics, and steering angle and acceleration are set by a PID controller. Lane changes are commanded from a behaviour planning unit, which relies on a prediction unit that estimates the location of other vehicles around the autonomous car.

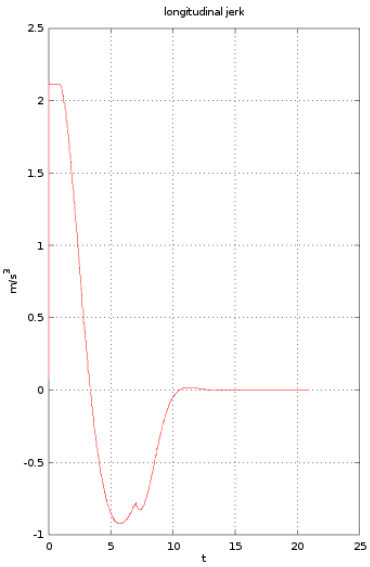
Trajectory Planner

Implemented in `trajectory_planner.cpp`, the primary function of the trajectory generator is to create a reference trajectory that keeps the vehicle safe from *incidents* (mentioned previously) when in its *own lane*. Departing from the approach proposed in the project walkthough, rather than directly using a spline, I implemented the reference trajectory as a quintic polynomial on the Frenet coordinates. Since the objective is to maintain the vehicle as close to the speed limit as possible, the polynomial generates reference *speed* rather than position along the track, which is numerically integrated to generate reference positions in Frenet coordinates. Another quintic polynomial generates reference lateral position (one of three lanes) of the vehicle. At this point, I realised the need for a locally smooth set of waypoints around the current location of the car, which was generated by spline fitting originally provided track waypoints (see `TrajectoryPlanner::updateLocalWaypoints`). This ensured that the reference trajectory when converted from Frenet into Cartesian coordinates did not cause sudden large changes in heading at waypoints. Sudden heading changes could otherwise cause lateral acceleration limits to be exceeded.

The following graph shows evolution of speed, acceleration and jerk on the Frenet 's' coordinate during initial start of the vehicle, showing a smooth ramp up of speed until target speed of about 20 m/s (90% of speed limit). The tiny bump in the jerk curve happens when the trajectory generator exits the start-up mode at `INITIAL_SPEED_UP_TIME` of 7 seconds.







The trajectory generator responds to other vehicles on the lane by lowering reference speed such that the autonomous car maintains a separation of `MIN_RESPONSE_TIME` seconds (3 - 5 seconds is good, according to safe motoring websites).

Vehicle Model

The reference trajectory from trajectory planner is fed into a bicycle model to model the car kinematics. The control inputs are steering angle and acceleration. The steering signal is generated by PID control over cross-track error between reference trajectory and vehicle position in global frame. The acceleration signal is generated by PID control over 'on-track error' (probably my term) which is the error in vehicle position along reference trajectory heading. Both vehicle model and PID controllers are implemented in `vehicle_model.cpp`. Control gains were set by trial and error to avoid exceeding acceleration limits, and vehicle model parameters came from a previous project on model predictive control.

Prediction

Prediction logic is unsophisticated, because I had to get something going quickly. (I had spent way too much time on this project already, over Christmas, and my 3-year old was beginning to complain about me not being available to play). Anyway, it does the job. Not perfectly, but reasonably well. It models other vehicles as constant-velocity constant-lane objects, and provides three sets of predictions, one for each lane. Each prediction includes free space available for lane change ahead of the subject car, and maximum possible lane speed. This is implemented in `behaviour_predictor.cpp`. Due to the unsopisticated model used, the nature of predictions had to be highly conservative, and therefore, the available free space is computed by considering that a vehicle up ahead may brake suddenly while a while behind might speed up. Free space is computed for time horizon defined by `LANE_CHANGE_DURATION`.

Behaviour Planning

The predictions feed into behaviour planning, which is again quite simple, and implemented in `behaviour_planning.cpp`. The implementation somewhat follows the lessons, in that three states are defined: `LANE_FOLLOW`, `LANE_CHANGE_RIGHT`, and `LANE_CHANGE_LEFT`; and a state transition function is implemented. Multiple cost functions control the behaviour. Briefly, a lane change is triggered only if the vehicle is not able to maintain speed in its current lane due to other vehicles in front, and if sufficient free space is available in an adjacent lane to successfully complete a lane change. Lane changes happen only between adjacent lanes (so, a direct change from 1 to 3 is not possible, for instance). The following cost functions are implemented:

Cost function	Weight	Description
Separation cost	200	High cost if separation and relative speed between vehicle and obstacles is such that the vehicle has to respond in less than `MIN_RESPOSE_TIME`. Cost falls off linearly for longer distances
Collision cost	100	Binary function - high cost if vehicle has to respond in less than `MIN_RESPOSE_TIME` to changes in obstacle trajectory
Speed deviation cost	90	Penalises deviation of lane speed from `MAX_SPEED`

Cost function	Weight	Description
Frequent lane changes cost	80	Penalises changing lanes too frequently. See `MIN_LANE_KEEP_TIMESTEPS`.
Manoeuverability cost	50	All else being equal, this priorities vehicle to be in the middle lane. This allows for two lane change options rather than one.

The behaviour planner feeds back into the trajectory planner with target lane and target speed.

Reflection

This project was by no means trivial, and took the longest so far in the SDCND programme to implement (about 2 weeks for me). Many components had to come together and play well with each other in order for the simulation to work well. A slight change in the PID control gains or limits such as max speed would change the behaviour and potentially lead to unwanted *incidents*. Debugging was hard because every occurance of an incident was not easily reproduceable. This meant, running the simulator and watching it continuously waiting for events to happen, and hitting the debugger to trace back what led to the event. Many early mornings and late nights were spent analysing why a jerk limit was exceeded here and why the vehicle went out of lane there. I finally decided to submit the project for review after sufficient tweaking to allow for long runs without incidents (30 minutes or more). It works, but it is not perfect. The following are some limitations of my implementation

- Requires 3 seconds response time. If vehicle barges into path with less than 3 sec to collision, collision may not be averted. This happened rarely, however.
- Only considers changing into the next nearest lane, while the farther lane may actually be free of vehicles.
- Conservative unspohisticated motion model for predictions. Cannot catch nearby vehicles making sudden lane changes. This also seems to happen rarely.
- Conservative lane change strategy. First priorty is to stay safe within the current lane.

In all, I was able to get multiple runs without incident lasting more than half hour and 4 - 5 loops of the track. I would like to keep improving the code, but I am afraid time's up and I should be moving on to my next project.