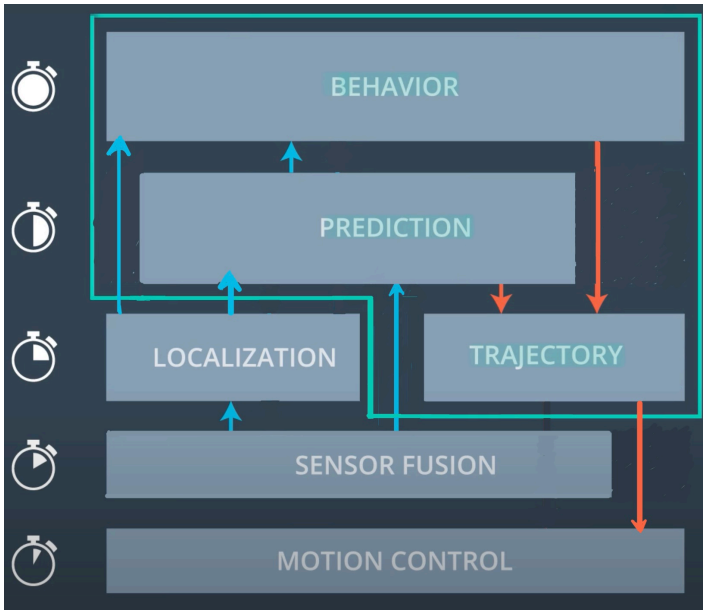
Path Planning - Highway Driving

Goal:

Design a software to plan and continuously update the path of an autonomous vehicle driving on a busy highway.

Autonomous Driving data flow & Path Planing :

Data flow :



Autonomous Vehicle Data pipeline

Data flows within a circular asynchronous pipeline: each module at the level L get inputs from the L-1 and L-2.

Information circulates from one module to the next ones at different frequencies : this asynchronous circular flow keeps running from the Sensor Fusion to the Motion Control module.

Path Planning :

The scope of the project is limited to designing a **Path Planning** module which is composed of 3 interacting modules :

Prediction generation > Behavioral Planning > Trajectory generation

(see these 3 modules and data flows within the whole pipeline in the chart above) .

- **Prediction** : uses data from sensor fusion to generate predictions about the likely behaviors of surrounding moving vehicles.

- **Behavior** : determine which behavior the autonomous vehicle should follow in response to the surrounding vehicles predicted behaviors.
- **Trajectory** : continuously refresh the vehicle trajectory to follow the last output from the Behavior Module.

Technical Set Up Overview:

Drive Simulator :

A provided simulator reproduces the virtual of an autonomous vehicle in motion and its surrounding environment:

- a sensor fusion feed informs our vehicle of all the surrounding vehicles locations and speeds.
- the path planner responds to the sensors flow by continuously recalculating the vehicle trajectory.
- this new trajectory (to be sent to the motion planner) is then sent to the simulator

Data format and programing environment :

The path planning software is implemented in C++ .

The sensor fusion input (other vehicles) and ego vehicle location and speed) flows into the planner via a Json structure.

The highway map is a csv file.

The resulting motion control output is sent to the simulator through a webSocket protocol.

Modules methodology :

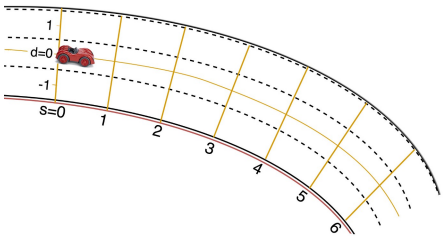
We start with the trajectory module so we' ll introduce the Frenet coordinates concepts used across the modules.

1) Trajectory Planning :

Paper road maps feature linear distances between POI :

Michelin map: linear distances printed in blue.
(eg Dannenberg to Dömitz crossing : **13 km**)

The highway map (highway_map.csv) features Frenet coordinates (a combination of longitudinal distances and distance from the side of the road) , as well as traditional cartesian coordinates .

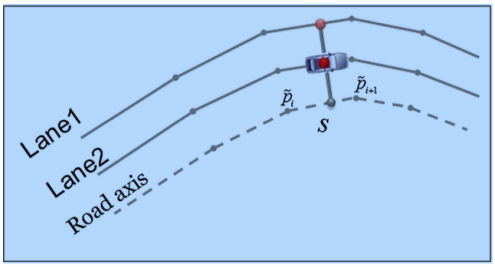


ego vehicle local **Frenet coordinates, "s points "** ranging from 0 to 6 in local Frenet

These Frenet coordinates (longitudinal **s** , transversal **d**) can be easily converted from global to local (ie in the ego vehicle space) , and also to cartesian.

A sequence of increasing Frenet coordinates once converted in Cartesian coordinates already provides a rough path of poses for the car to follow .

In our Project , the ego car will drive from one way point to the next one in 0.02 second (hence the importance of spacing these points adequately to control the ego vehicle speed).



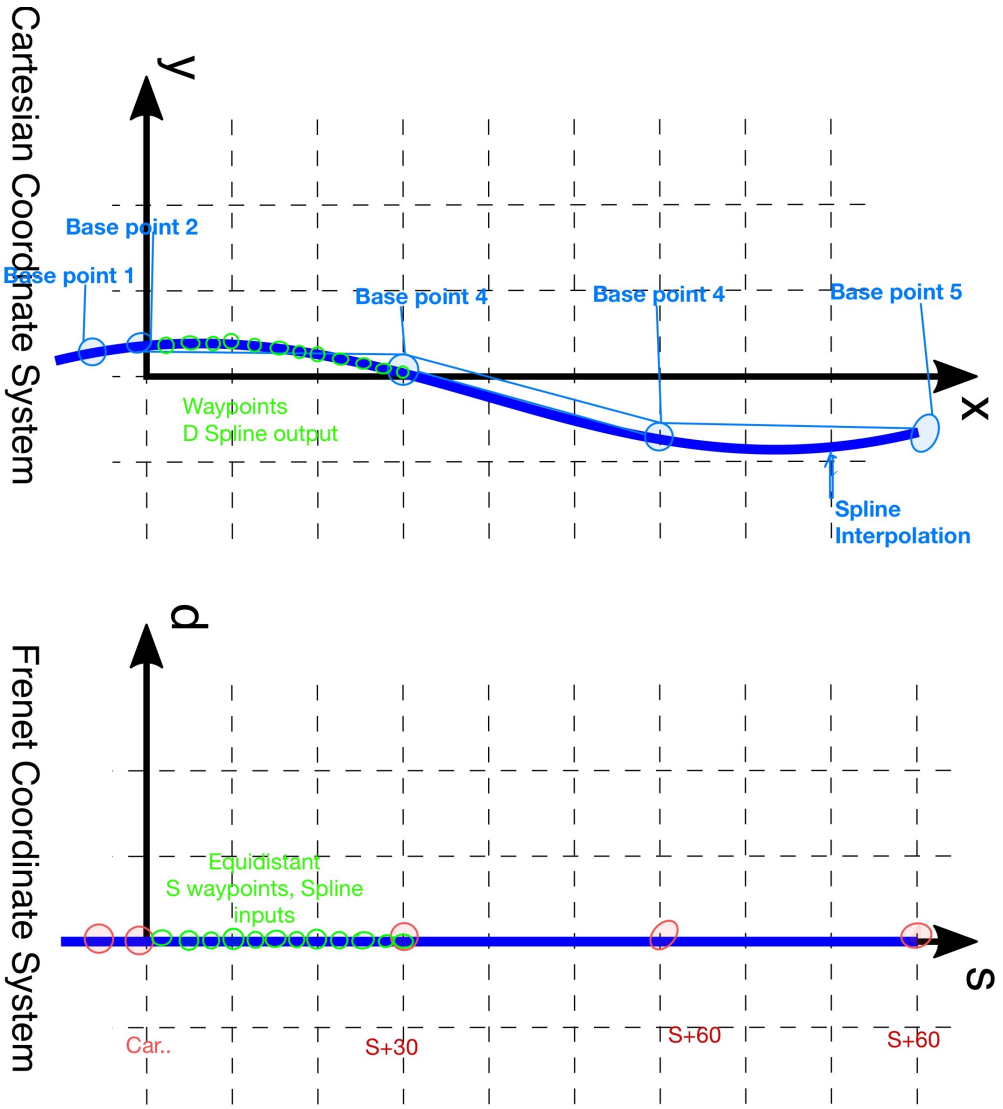
Although inserting the appropriate number of way points in a straight line between the map S-Points will control the speed adequately , the trajectory will be angular between each straight segment while driving through an S-point (particularly while driving on a curved section of the highway) .

Instead the trajectory planner will use the car and map Frenet coordinates to calculate and feed the autonomous vehicle with a smooth **sequence** of points to drive on. This within a limited horizon as the sequence needs to follow the car progression and adapt to the traffic (as the spacing of the waypoints controls the car velocity , they need to be continuously refreshed to avoid collisions) .

We use a Spline polynomial interpolation function to continuously generate a smooth series of 30 points for the car to drive through. A portion of these points will be added to the last

remaining points from the previous generation to insure a smooth transition between en each batch . This way the car always have a fresh horizon of 50 points ahead to drive.

Spline creation :



Spline created from 5 base points : D Frenet coordinates are interpolated from equidistant S coordinates using this Spline . (S,D) coordinates are then converted in (x,y).

The 5 anchor points (aka base points)to instantiate Spline for each sequence generation are :

- the previous sequence last 2 way points (or the car coordinate and a close point on the yaw axis)

- 3 next points ahead at a longitudinal distance of 30 ,60 and 30 meters from the ego car (so with s local coordinates of 30, 60 and 90 meters)

Spline utilisation:

Using some basic geometric calculation we determine what spacing the waypoints should have on the S axis of the local Frenet coordinates . We then use the spline function to calculate their corresponding d-coordinates.

These calculated Frenet waypoint coordinates (s,d) are then converted in global cartesian coordinates (x,y) and added to the vehicle live trajectory to be sent to the Motion control module.

2) Prediction generation :

There are two main approaches to predict the behavior of surrounding vehicles :

- model based using mathematical model (ie probability of a motion derived from a sequence of sensor measurement of an adversarial player's (position, velocity, acceleration) .
- data driven relying on machine learning

For this project basic approach to predict each vehicle pose:

- we use the last way point from the previous batch as a prediction of the ego vehicle pose.
- we predict all vehicle poses using their current velocities , Frenet coordinates and the time to elapse till the ego vehicle will reach the end of the previous waypoint batch.

2) Behavior planning :

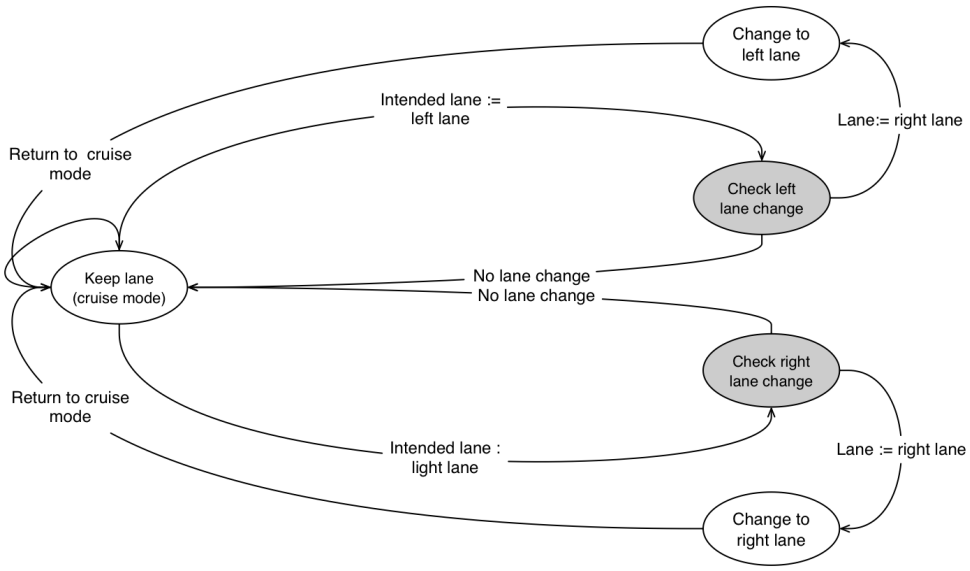
The ego vehicle has to plan its trajectory, speed and acceleration to abide by a set of rules with various priorities. Each rule implies a set of constraints or sub-rules for the vehicle to follow .

These rules can be ranked by cost are as follows :

behavior or deviant behavior / recommended behavior	**deviant behavior cost**
drive over the speed limit (50 mph)/ keep speed under 50 mph	**5**
drive on two lanes when not changing lane/ keep on lane when not intentionally changing	**5**
rear end a vehicle/ respect safe distance ahead	**5**
drive close to the speed limit / drive well below	**3**
create jerk / limit speed change to 0.224 per step	**5**

behavior or deviant behavior / recommended behavior	**deviant behavior cost**
change lane to invading vehicle's lane/ change to right lane when vehicle invades from left (and opposite)	**5**
change to the left lane	**2**
change to the right lane	**3**
cut off in front of a car/check the intended lane rear clearance	**5**
drive below 49.5 mph/ cruise to 49.5 mph	**1**

The ego vehicle behaviors are represented by an ensemble of states :



A function calculates, for the current state, the cost of staying in this state or transitioning to one on the contiguous states : the car will transition to (or stay in) the lowest cost state.

Implementation:

Some rules are to be respected in all states, therefore the motion control module will always be given the set of instructions to follow them .The corresponding controls are hard coded in the the Behavior Planner (eg keep speed under 50 mph: way points are always spaced to follow this rule) . They can be ignored while comparing the cost of one state to the contiguous one (no differential) .

In our code a list of conditional executions (if, if/ else) sorted in increasing transition costs insures that each state transition to (or stay in) the lowest cost contiguous state .

For example while changing lane to pass a slow vehicle :
- the cost to pass from the left is lower then from the right so we'll only pass as a second choice (and eventually just decelerate when passing from the right is also not possible) .

But in the case of avoiding an invading vehicle :
- we first check if it possible to mimics the invader's sway to avoid the collision then our second choice is to decelerate . (*see safe_lane_change_check, intended lane*)

Results :

See our two 10 miles (no incident) test drive videos attached .
Our model comply with all project requirements .

Conclusion :

In real life some constraints (maximum jerk , no emergency lane driving) will be over ridden in case of emergency to avoid collisions. In this case the deceleration calculation need to be a little more involved to still preserve the passenger safety and comfort (the ABS ultimately having the final word) . Once the collision predicted , the waypoints left from the previous batch purged and so the path ahead integrate the new spacing sooner to reduce the speed (basically discarding the last batch residual to reduce the latency) . Knowing that our waypoint buffer contains 50 points , this will reduce the latency by one second (car moves between each point in 0.02 seconds) . In comparison human fasten perception to reaction time is .15 second (https://www.visualexpert.com/Resources/reactiontime.html) . A vehicle traveling at 50 mph, drives 22 meters in one second . Hence the necessity of refreshing the waypoint buffer sooner to decelerate sooner while dealing with a lane invasion.

Credits :

"An efficient lane model for complex traffic simulation"

COMPUTER ANIMATION AND VIRTUAL WORLDS Comp. Anim. Virtual Worlds 2015; 26:397–403 Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cav.1642

Authors : Tianlu Mao , Hua Wang, Zhigang Deng, Zhaoqi Wang

Self-driving car: Path planning to maneuver the traffic. Jonathan Hui

https://jonathan-hui.medium.com/self-driving-car-path-planning-to-maneuver-the-traffic-ac63f5a620e2

Trajectory Planning Frenet Space: Udacity Self Driving Car ND