Trajectory Generation Notes

Some note of mine to the Chapter Trajectory Generation in preparation for Project 2 – Highway Driving.

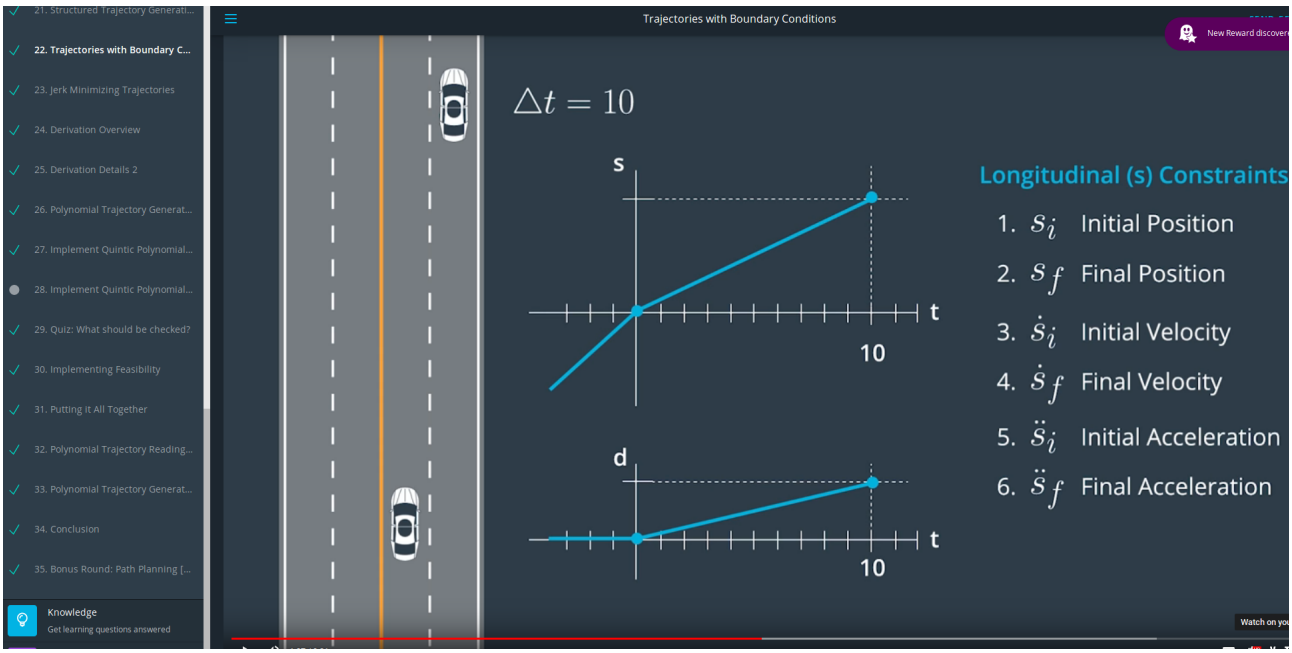


Illustration 1: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

These linear approximated trajectories physically cannot be executed by the car. Instantaneous change in the slope of the trajectories of s (change in longitudinal speed) and d (change lateral speed) mathematically result in a infinitely high acceleration (first derivative) which cannot be executed by any physical system.

Thats why we require to have continuity and smoothness in our trajectories.
→ Continuity in Position → our car cannot teleport or instantaneously change speed
→ Continuity in Velocity → Continuity in Acceleration → Continuity in Jerk

Our function of the trajectory needs to be Ck smooth.

Position → velocity → acceleration → **jerk** → Snap → Crackle → Pop

The jerk is directly related to human perception of comfort. Booth lateral and longitudinal jerk (acceleration changes very quickly) are perceived as uncomfortable if they are high.

So the aim is to calculate a jerk minimizing trajectory.

It is relatively easy to compute a jerk minimizing trajectory in one dimension.

Consider a function s of t in the interval 0 to tf
The Jerk is the third derivative of position = s(t)

Total (squared) Jerk accumulated over the duration of this trajectory is given by integral ….
We want to penalize booth positive as negative jerk therefore we look at total squared jerk.

We need a function that minimizes this integral → going through math that required to find the minimium, we find that all time derivatives of s of order 6 and more have to be zero in order for s to be jerk minimum.

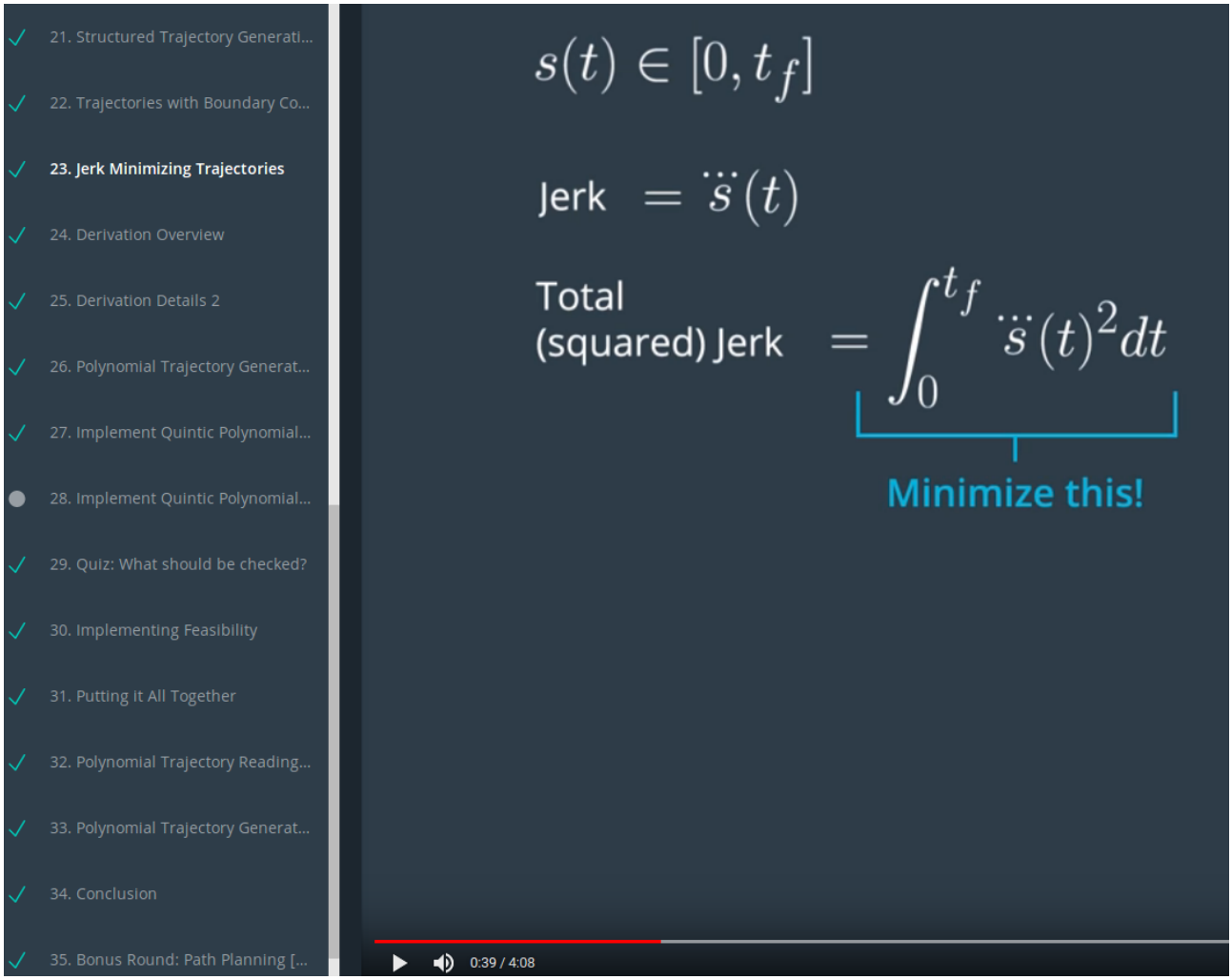


Illustration 2: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

Helpful fact of functions that they can be rewritten as

Parametric continuity (Barsky) ? Wikipedia

All coefficients bigger than 5 are zero. All minimum Jerk trajectories can be represented as a fith order polynomial.

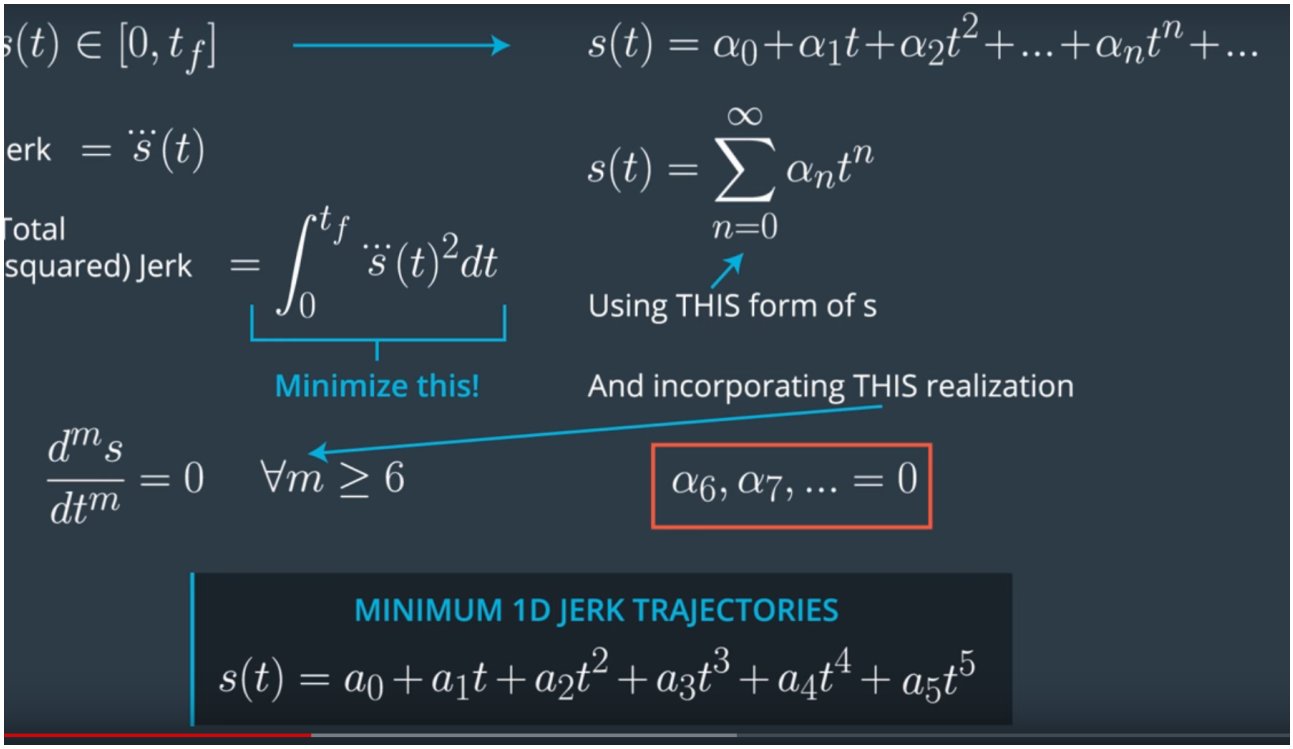


Illustration 3: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

6 coefficients a0 – a5 can be tuned to define the shape of the 1D trajectories.
→ By defining boundary conditions, we can force our trajectory to fulfill this conditions
→ Things we like to constrain on the trajectory are
- The initial position, velocity and acceleration
- The final position, velocity and acceleration

This applies for longitudinal displacment s and lateral displacement d → this gives us 12 variables to pick in order to fully define the motion of our vehicle in s and d over time. The initial state of our vehicle gives us the longitudinal and lateral boundary conditions at t = 0 and we get to pick the end boundary conditions.

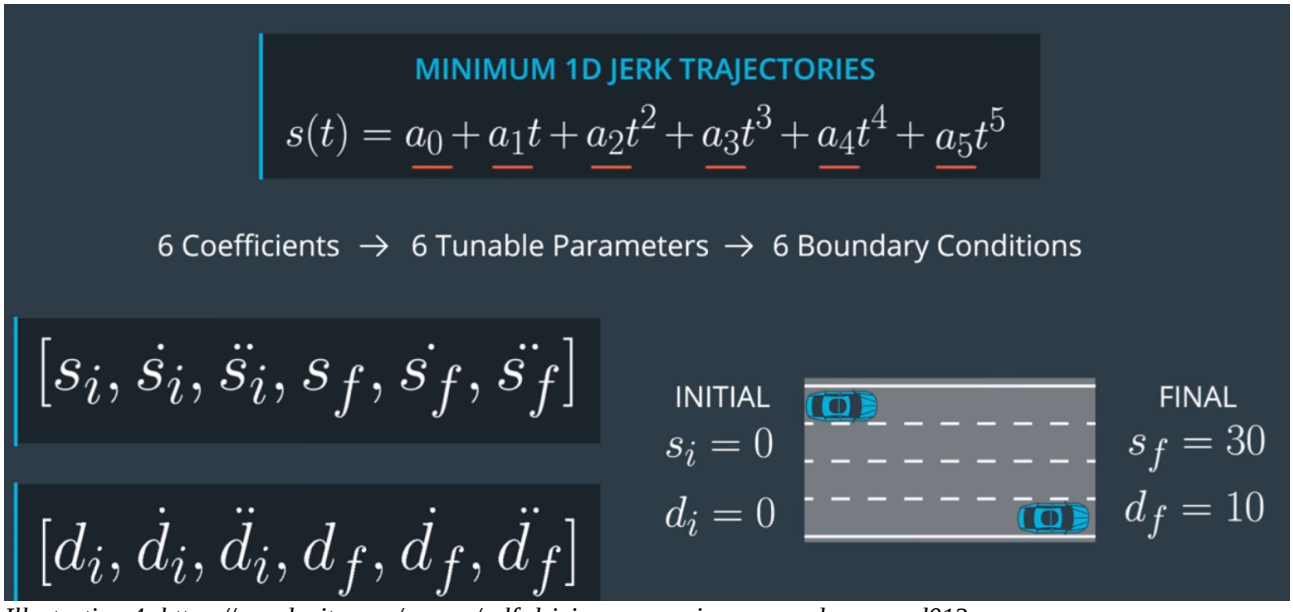


Illustration 4: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

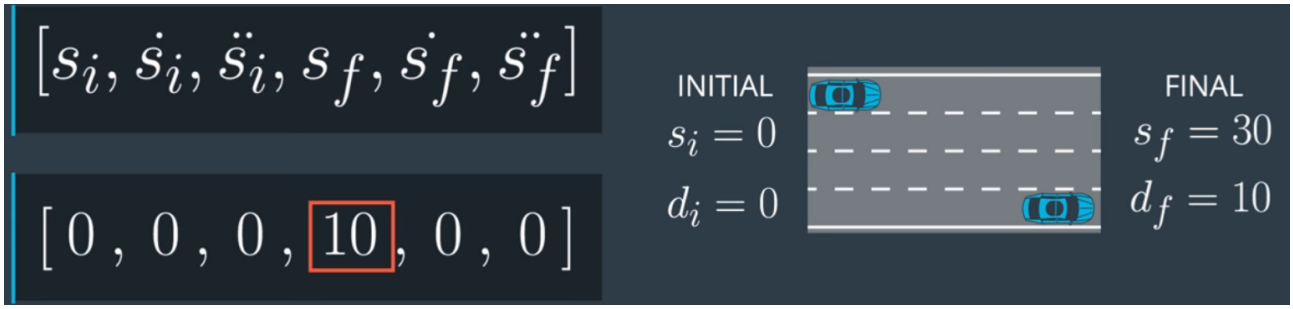


Illustration 5: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

Solving quintic polynomial

1) Differentiate the equation of the position to get the euqation of the velocity s dot
2) Differentiate it again to get the equation for acceleration s dot dot

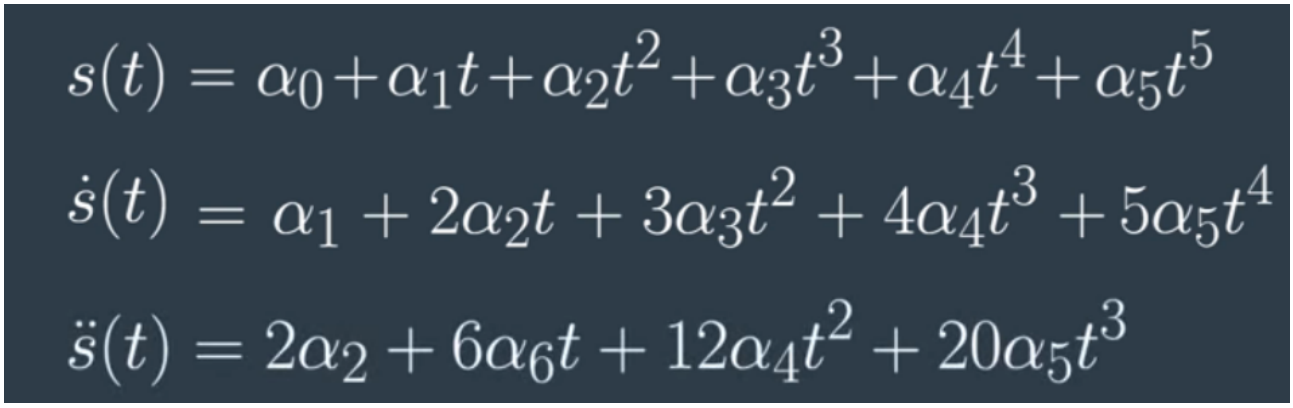


Illustration 6: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

We could just plug in our 6 boundary conditions to get the 6 equations, bur first we are going to do something to make our life a little bit easier.

Set the initial time to 0 → then we find that 3 of our unknowns do not have to be identified anymore. Which reduces our problem from 6 unknowns to 3.

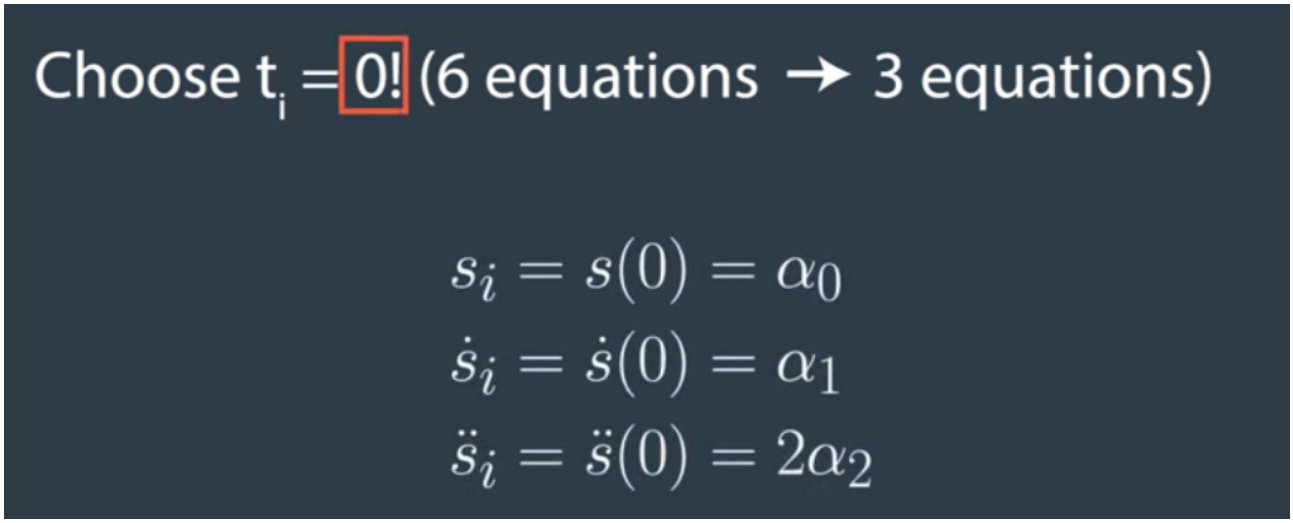


Illustration 7: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

It simplifies the three trajectories to the following equations. These terms are now known form the beginning by inserting s(0) sdot(0) sdotdot(0) and respectively d(0) …. into the terms.

$$s(t) = s_i + \dot{s}_i t + \frac{\ddot{s}_i}{2} t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5$$
$$\dot{s}(t) = \dot{s}_i + \frac{\ddot{s}_i}{2} t^2 + 3\alpha_3 t^2 + 4\alpha_4 t^3 + 5\alpha_5 t^4$$
$$\ddot{s}(t) = \ddot{s}_i + 6\alpha_3 t + 12\alpha_4 t^2 + 20\alpha_5 t^3$$

Illustration 8: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

To make it look cleaner we can gather the known terms into functions of the start boundary conditions. C1, C2, and C3 terms are functions of the initial conditions.

$$s(t) = \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5 + C_1$$
$$\dot{s}(t) = 3\alpha_3 t^2 + 4\alpha_4 t^3 + 5\alpha_5 t^4 + C_2$$
$$\ddot{s}(t) = 6\alpha_6 t + 12\alpha_4 t^2 + 20\alpha_5 t^3 + C_3$$

Illustration 9: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

What remains is that we only have to identify 3 parameters that only depend on our end boundary condition. We know the final position, velocity and accelaration at time t, because since those are all the quantities we picked.

$$s(t_f) = s_f = \alpha_3 t_f^3 + \alpha_4 t_f^4 + \alpha_5 t_f^5 + C_1$$
$$\dot{s}(t_f) = \dot{s}_f = 3\alpha_3 t_f^2 + 4\alpha_4 t_f^3 + 5\alpha_5 t_f^4 + C_2$$
$$\ddot{s}(t_f) = \ddot{s}_f = 6\alpha_3 t_f + 12\alpha_4 t_f^2 + 20\alpha_5 t_f^3 + C_3$$

We know…
- s_f
- $\dot{s}_f$
- $\ddot{s}_f$
- t_f

Illustration 10: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

Since we know them a more useful way to represent the equations is to seperate the known quantities from the unknown quantities and represent the equations in matrix representation.

This solvable system of 3 equations can be best solved in matrix form by any matrix mathematical library.

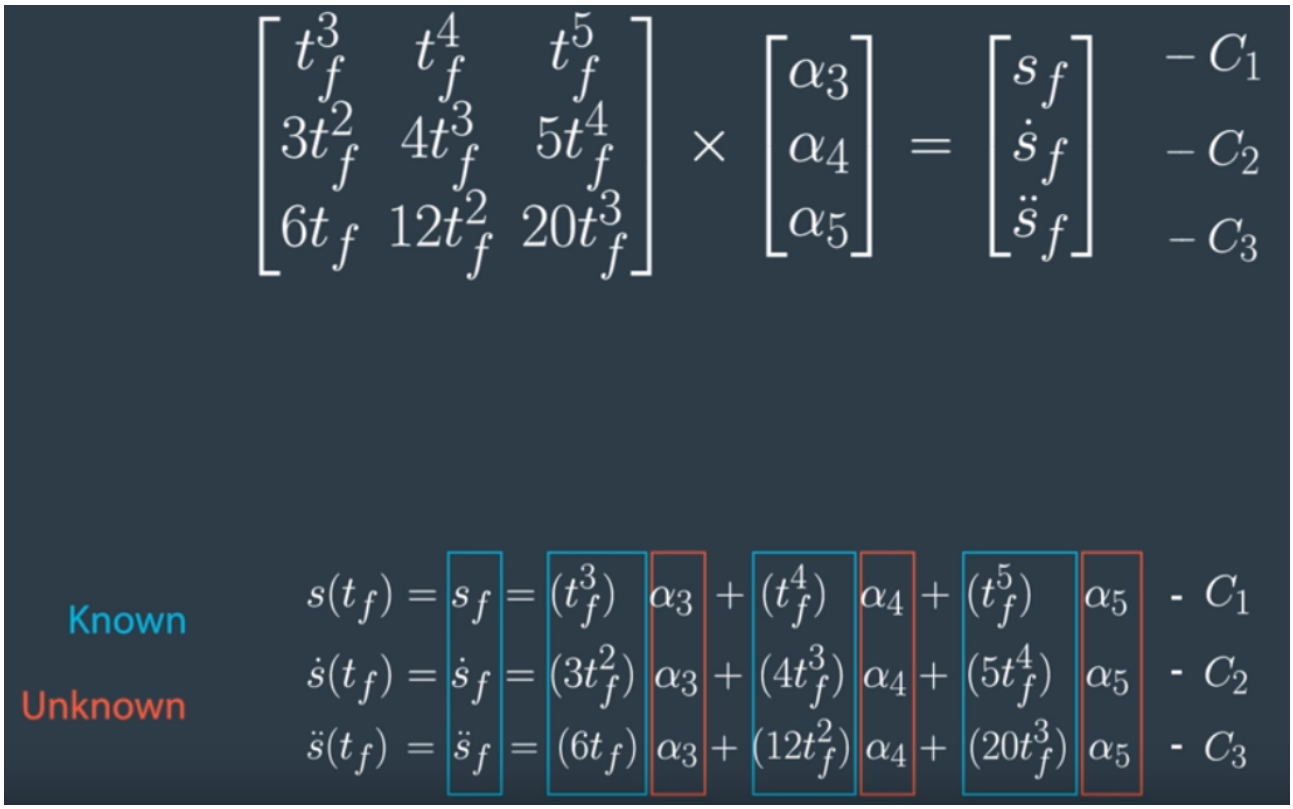


Illustration 11: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

Using Eigen Library matrix https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html
https://eigen.tuxfamily.org/dox/group__QuickRefPage.html

```
#include <iostream>
#include <Eigen/Dense>

using namespace Eigen;

int main()
{
  MatrixXd m(2,2);
  m(0,0) = 3;
  m(1,0) = 2.5;
  m(0,1) = -1;
  m(1,1) = m(1,0) + m(0,1);
  std::cout << "Here is the matrix m:\n" << m << std::endl;
  VectorXd v(2);
  v(0) = 4;
  v(1) = v(0) - 1;
  std::cout << "Here is the vector v:\n" << v << std::endl;
}
```

```
Here is the matrix m:
  3  -1
2.5 1.5
Here is the vector v:
4
3
```

```
Matrix3f m;
m << 1, 2, 3,
     4, 5, 6,
     7, 8, 9;
std::cout << m;
```

```
1 2 3
4 5 6
7 8 9
```

Solve matrix equation with eigen:

https://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html

```
 double si = start[0];
   double si_dot = start[1];
   double si_double_dot = start[2];

    // alpha3, alpha4 and alpha5 need to be calculated by solving matrix
representation of
    // the 3 remaining equations

    // 1) Define matrices

    Eigen::Matrix3f T_mat;
    T_mat << T*T*T, T*T*T*T, T*T*T*T*T,
             3*T*T, 4*T*T*T, 5*T*T*T*T,
             6*T, 12*T*T, 20*T*T*T;


    // end = [sf, sf_dot, sf_double_dot]
    // sf

    double sf = end[0];
    double sf_dot = end[1];
    double sf_double_dot = end[2];

    // m(rows,  cols)

    Eigen::Vector3f terms_mat;
    terms_mat << sf - (si + si_dot*T + 0.5*si_double_dot*T*T),
                 sf_dot - (si_dot + si_double_dot*T),
                 sf_double_dot - si_double_dot;

    // Multiplying by the inverser from left to solve the equation

    // Solve matrix equation Ax = b ... A = T_mat, b = terms_mat

     Eigen::Vector3f alpha_vector =
T_mat.colPivHouseholderQr().solve(terms_mat);

    //std::cout << alpha_vector << std::endl;

    std::vector<double> coeff;
    coeff.push_back(si);
    coeff.push_back(si_dot);
    coeff.push_back(0.5*si_double_dot);
    coeff.push_back(alpha_vector(0,0));
    coeff.push_back(alpha_vector(1,0));
    coeff.push_back(alpha_vector(2,0));

    return coeff;
```

Checking the feasibility of the trajectory:

Initial validation for trajectory
→ Neglect curvature of the road and assume it is locally straight

Longitudinal acceleration

Longitudinal acceleration, we make additional assumtion that our heading is pretty much aligned with the road. This allows us to say that s_double_dot is the longitudinal acceleration of the car.
→ We need to check if at any point of the trajectory the acceleration is less than the maximum acceleration the car should supply and bigger than the maximum deceleration the car can do by braking. This information in reality should incorporate information about the friction of the road.

Lateral Accelearation

→ Simmilary for lateral accelearation we can check that all d_double_dot are less than a fixed lateral acceleration value that can be set for comfort or safety value (risk of rollover).
→ Regarding steering angle: The bicycle model tells us that there is a nice relationship between the steering angle and the radius of circle of curvature.
→ If you remember in the reading assignment of hybrid A*
- Delta phi is the heading difference between two point of a trajectory
- Delta x is the distance between them

Longitudinal Speed
- The velocity is checked against values defined by the map or the behavioural layer
→ Speed limit road for min or max velocity

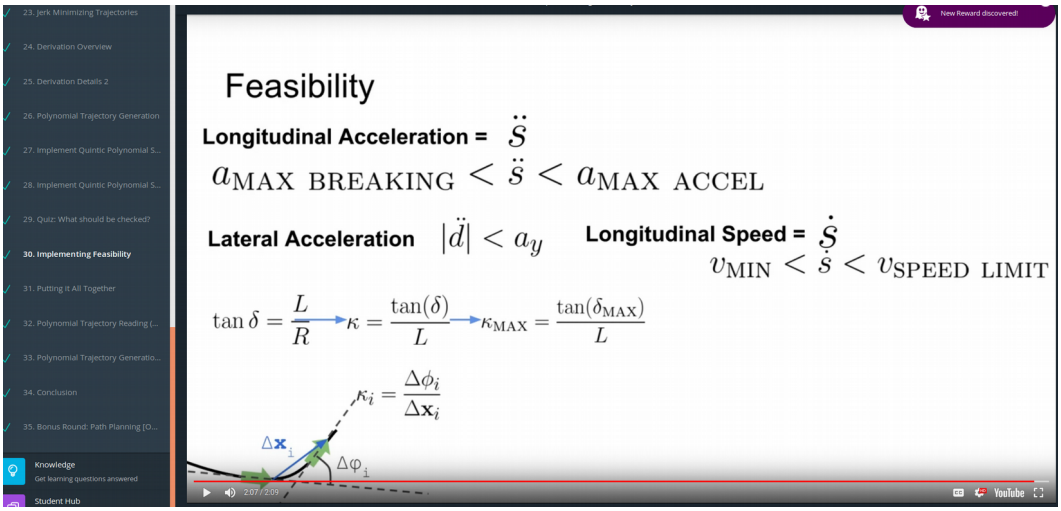


Illustration 12: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

Sampling based method
→ Behavioral layer usually not give an exact point to reach, rather an approximate one

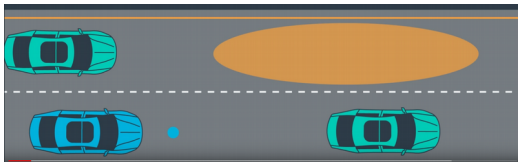
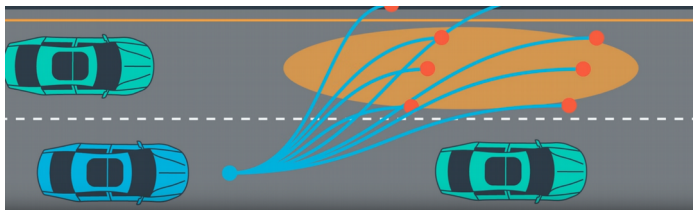


Illustration 13: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

→ We want to identify a valid goal state, that leads our vehicle in the approximate end configuration
→ We do not know what the best solution is to get int o this area
→ With a sampling based approach a large number of outcomes can be generated
→ Non feasible trajectories are deleted (e.g. the ones ending outside of the road or in a collision), → The best solution of the remaining is picked by cost functions (ranking)



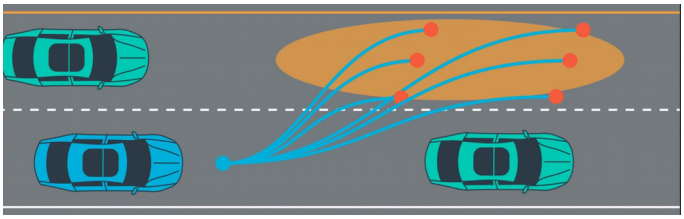


Illustration 14: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

Cost functions:
→ Jerk: Each remaining trajectories are jerk optimal, but maybe some are better than others
- Consider Longitudinal vs Lateral Jerk
- Side to side jerk (lateral) is more uncomfortable than longitudinal → prioritize minimizing

→ Distance to obstacles: The distance to other vehicles should not be too small

→ Distance to center line: If no other indication forces the car to get off the center line, we prefer to drive in the center of the lane

→ Time to goal: We like to arrive at the destination rather further than later

There are plenty of cost functions available, the problem is to balance the weights of the cost functions. Often goals conflict.

All cost functions need to be combined an weighted in a overall cost function.

Example Python code:

This code calculates an acceleration and jerk optimal path to a goal point given by a moving other car. The goal point can be shifted, so it is possible to calculate a trajectory that ends infront or behind the target vehicle and/or beside left/right of the target vehicle.

So this example shows a little bit another scenario as “choosing the best lane according to costs”.
It is the basis to handle scenarios where there are cars in faster lanes and we might be able to position our ego car infront or behind one of these faster cars.

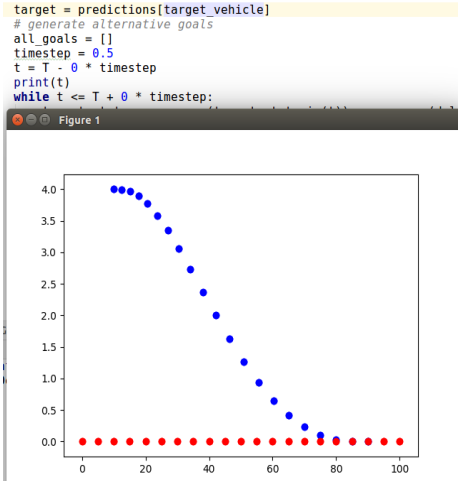
It is possible to configure the s start, s velocity start, s acceleration start, d start, d velocity start and d acceleration start of the target vehicle.
The position of the target vehicle is then predicted according to the defined velocity and acceleration values.
By altering the cost function weights (or/and maximum values for acceleration and velocity for the ego vehicle), the ego vehicle might able to reach the target position at all in the given time and without collision or not.

Multiple trajectories are generated for the ego vehicle using multiple time increments of the target cars predicted position.

Namely in the interval `T - 4 * timestep to T + 4 * timestep`

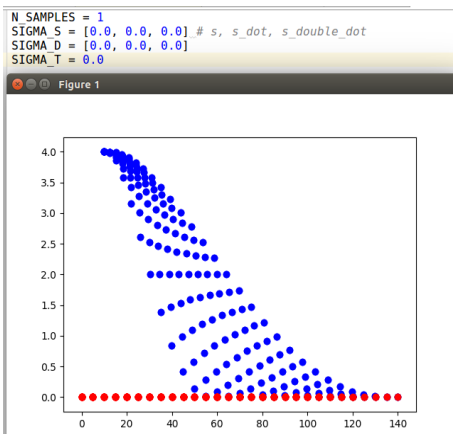
```
target = predictions[target_vehicle]
# generate alternative goals
all_goals = []
timestep = 0.5
t = T - 4 * timestep
print(t)
while t <= T + 4 * timestep:
    target_state = np.array(target.state_in(t)) + np.array(delta)
    goal_s = target_state[:3]
    goal_d = target_state[3:]
    goals = [(goal_s, goal_d, t)]
    for _ in range(N_SAMPLES):
        perturbed = perturb_goal(goal_s, goal_d) # convert to cartesian coordinates
        goals.append((perturbed[0], perturbed[1], t))
    all_goals += goals
    t += timestep
```

Without noise, number of samples ==1 and T interval is [T,T] the code predicts only the polynomial trajctory to T.



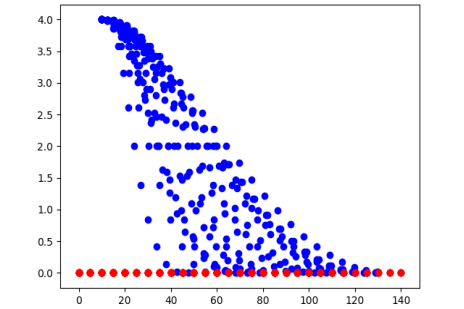
Noise parameters allow to model the uncertainty of the predicted position.
constands.py
N_SAMPLES denote the number of generated trajectories for each time time step.
SIGMA values describe the noise for the predicted vehicle position. If they are set to zero, the target vehicles position is predicted exactly according to the motion model.

```
N_SAMPLES = 1
SIGMA_S = [0.0, 0.0, 0.0] # s, s_dot, s_double_dot
SIGMA_D = [0.0, 0.0, 0.0]
SIGMA_T = 0.0
```



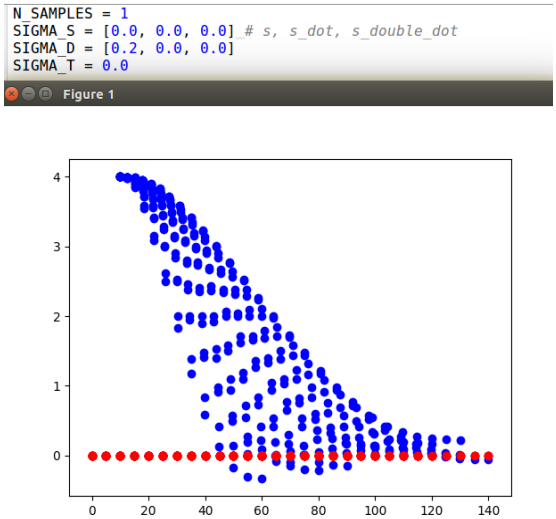
SIGMA_S results in a position noise affecting the longitudinal position

```
N_SAMPLES = 1
SIGMA_S = [10.0, 0.0, 0.0] # s, s_dot, s_double_dot
SIGMA_D = [0.0, 0.0, 0.0]
SIGMA_T = 0.0
```



SIGMA_D results in a position noise affecting the lateral position of the car:

```
N_SAMPLES = 1
SIGMA_S = [0.0, 0.0, 0.0] # s, s_dot, s_double_dot
SIGMA_D = [0.2, 0.0, 0.0]
SIGMA_T = 0.0
```



Used cost functions:

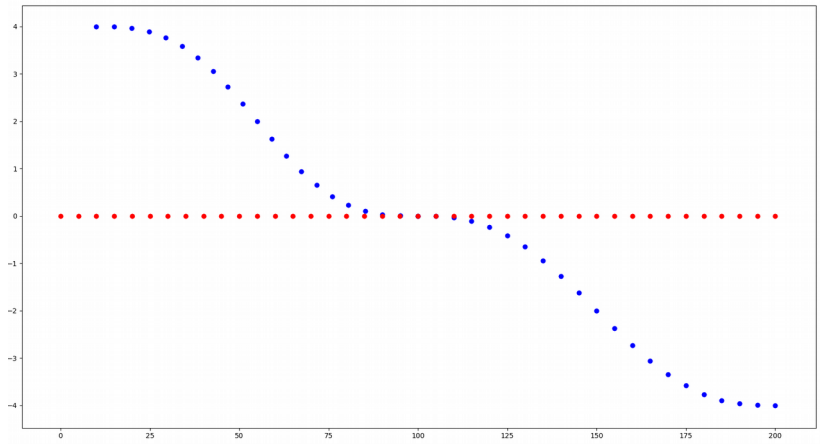
- t_difference: Penalizes trajectories that span duration which is longer or shorter than the duration requested.
- s_difference: Penalizes trajectories whose s coordinate (and derivatives) differ from the goal
- d_difference: Penalizes trajectories whose d coordinate (and derivatives) differ from the goal
- collision_cost: Binary cost function which penalizes collisions → 2* VEHICLE_RADIUS threshold
- buffer_cost: Penalizes getting close to other vehicles
- stay_on_road_cost: not implemented, but should penalize leaving road boundaries (with car body)
- exceeds_speed_limit_cost: not implemented, but should penalize trajectories which exceed speed limit
- Efficiency_cost: Rewards high average speeds
- total_accel_cost: penalizes high acceleration costs
- max_acceleration_cost: Penalizes trajectories which exceed max allowed acceleration

Additional cost functions I can think of:

- eccentrical_cost: Should penalize trajectories which are far from the middle of the lane, this might be combined with/or the same as the “stay on road cost”
-

nearest_approach_to_any_vehicle: calculate nearest distance to all vehicles of a trajectory

Adding another cycle to the code with start = end of the first iteration,



Project: Highway Driving Protocol

Goals:
- Crate path planner for 3 lane highway traffic
- Smooth and safe path have to be created.
- Criteria for “smooth path”:
    - Maximum Acceleration
    - Maximum Jerk
- Criteria for “safe path”:
    - Obey maximum velocity allowed
    - Car have to stay in its lane
    - Car does not have collisions with other highway traffic
- The car is able to change lanes (while also obeying all the given criteria)

→ We move our car in a known map environment, no live lane detection is performed in this project
→ The localization of the car is given at every time step, with an accuracy and precision sufficient to laterally control the car on the street
→ Only dynamic data we need to use is the object detection of other cars moving along the highway with us in the same direction, with approximately in the same velocity.

Map Data:
- 181 Way points
- linearly interpolated between the way points
- Last way points – points to the first
- Way points are given at the middle of the double-yellow dividing line in the center of the highway
- Each lane is 4m wide
- 6 lanes – 3 in every direction
- 3 lanes on the right handed side allowed to drive

Frenet coordinates of the map:
s distance along the direction of the road == 0 at start point
d interval[0,1] – perpendicular to the road in the direction of the right-hand side of the road

```
---> +d unit vector perpendicular to road
|| ----------- | ----------- |  ----------- |
         2m      4m     6m       8m       10m     12m
--------> d*distance to middle lane
```

lane*4 + 2 == middle of the lane; lane [0,3] while 0 = most far left lane

Object Data (sensor fusion):
- Contains only all cars on the right hand side of the road (no objects of oncoming traffic)
- Each car has a unique identifier → No ID jumps e.g. due to wrong associations of sensor fusion
- Position, velocity are given relative to map coordinate system
- Frenet coordinates s and d are given for each car
- There are always 12 cars in the sensor fusion → not changing ID#s

Implementing the project:

After I looked into the Q&A Video, and also read a lot of post in the student hub that Frenet conversion have some issues, I am going to use the Spline library to approximate trajectories. The Q&A video gives a good start for the project.

Implemented classes

Trajectory class

Object Class

EgoVehicle Class

EgoEnvironment Class

Implemented costs
All implemented costs where only about position of the ego car and the vehicles around. I did not manage to incorporate costs for speed (all lanes have the same speed limits) or other proposed cost functions in the course. Also the costs where normalized in the course, which I did not implement too because I that way, I somehow had a better feeling how the costs participate to the overall cost of choosing the right lane.

Penalize lane changes
Penalize double lane changes
Penalize proximity of objects at car height
Penalize lanes with cars in front
Penalize too close following distance

Car Speed
I was not satisfied with the approach to set the speed of the ego car due to the proximity of the preceding car. It led to a not very constant speed of the ego car when following a car. I tried to overcome this issue by setting the ego cars reference speed according to the speed of the car it is following to get rid of the oscillation while following, but the speed calculated by vx vy of the object sensor data was not stable enough. Using the objects position and calculating the objects speed according to its last and current position also not very stable.

So I sticked with the proposed approach in the question and answer video.

Because the approach to decrease the speed only according to the proximity of the preceding car, I had to inhibit lane changes during decreasing speed because it happened that the car got very slow.

Stabilizing the states
If a lane change is safely possible and started, changes in state are inhibited during the execution of the lane change.

I implemented a counter which inhibits another state change shortly after the car has performed a lane change. This was necessary because sometimes the states where switching to fast and the state got lost of the cars position on the track.

Testing
I found it pretty hard to test the state machine, without the possibility to reproduce certain situations, where problems occurred. Especially critical situations occur only randomly in the simulator, which makes it really hard to retest the same critical situation with the changes are made to the costs or the state machine.

I ended up restarting the simulator frequently to provoke some difficult situations which appear to happen shortly after the beginning of the simulation. But I still can not say, if I tackled all situations. I think it still has issues with very fast approaching cars from behind.

It would be great, if there where some standard situations to choose from at the beginning of the simulation. Like start lane of the car, blocked lanes by preceding cars, fast approaching cars from behind, cars changing on short notice in ego lane, etc.. or if there where the possibility to “trace” a situation and being able to replay it.

Some results of test runs on local machine

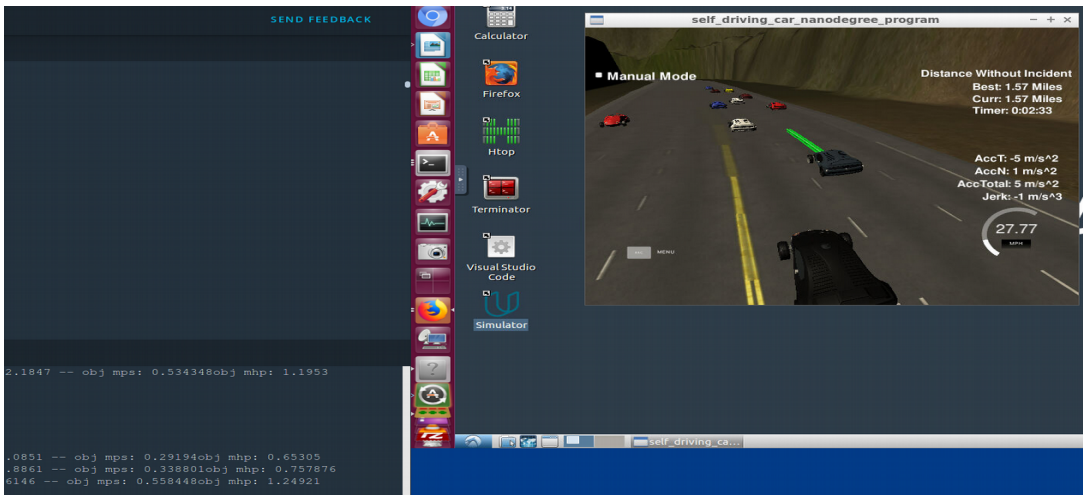
Checking performance on the server

Shortly before I decided to submit my project I decided to double check, if my planner is behaving properly also on the workspace server.

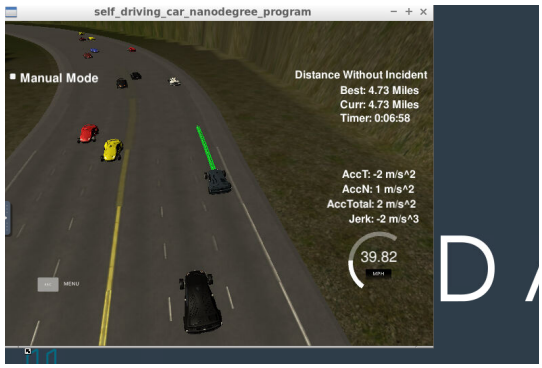
Apparently I have an old Laptop and it hat its difficulties to cope with the simulation. Instead of taking 13 cycles for one planning iteration it takes only up to 3 on the server. This leads to an entirely different behavior. So I checked out my git project on the server and started to do all the tuning work from scratch ending up rewriting the whole cost calculation section.

Object speeds on the server

Maybe I am doing something wrong, but the object speeds are unplausible, also on the server simulation. I hoped that this also was caused by the lack of performance of my local simulation, but it is not. In the left lower corner of the next screenshot, the object speeds of some cars infront of my ego are printed. These speeds are much much to low to be the right ones.



Retesting the new implementation on workspace simulation
After rewriting the cost functions, I did some tests on the on line simulation. After a row of collision free runs, I decided to end my work on this project because I am way past due.



Conclusion

On my local machine one Iteration of the planner took about 8-14 cycles (prev path was always about 43-36 points long), which lead to a quite laggy behavior of the simulation.

Running the same code on the web simulation resulted in entirely different results due to much shorter simulation cycles. On my local machine it took the simulator/code 8-14 cycles to finish an iteration, in the web environment simulation only 3. Calculated Accelerations and Speeds are very different from the two simulations, so I encountered Acceleration and Jerk violations with my tunings on my local machine.

I rewrote all the cost calculations and had to tune the weights newly running the simulation on the server. This was very time consuming and I submitted the project way past due.

All in all this project was not very fun to implement.

- I could not really apply the course content in the project
- The simulation appears to be super laggy and apparently has different time behavior on different machines.
- In the course we saw that the frenet coordinates represent the exact geometry of the road, when transforming back to the xy space. In the simulation, the Frenet coordinates where representing only linear interpolations between xy map waypoints. So the main advantage of the Frenet coordinates was not given, except that we could track our objects approximately in s direction. Maybe I missed the points here.
- I did not manage to get proper speeds from the objects, so it was not very clear whats exactly happening, when planning into the future. Also it was not possible for me to set a right following speed and I was stuck on the proximity on/off faster/slower mechanism which resulted in oscillating distances to the preceding vehicle.
- I spent a lot of time on tuning first on my local machine, then on the sever simulation
- With the simulation running randomly difficult and scenarios, it was not possible to retest a specific scenario which has caused a problem, leaving me reseting the simulation over and over again until the situation might appear again. This was not fun. The simulation should have some standard test scenarios for different state transitions like change left, change right, follow, other car suddenly changes lanes, car fast approaching form behind, car in front suddenly stops, ….. etc. Only with these reproducible scenarios the state transitions can be tested. A stochastic test scenario then can be used to test the whole state machine. There certainly going to be scenarios where my state machine is failing, but I simply cannot provoke them to check if my changes to the state machine are working.